Exploiting Functional Dependence in Query Optimization

BY

GLENN NORMAN PAULLEY

A THESIS
PRESENTED TO THE UNIVERSITY OF WATERLOO
IN FULFILMENT OF THE
THESIS REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

WATERLOO, ONTARIO, CANADA, 2000

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Functional dependency analysis can be applied to various problems in query optimization: selectivity estimation, estimation of (intermediate) result sizes, order optimization (in particular sort avoidance), cost estimation, and various problems in the area of semantic query optimization. Dependency analysis in an ANSI SQL relational model, however, is made complex due to the existence of null values, three-valued logic, outer joins, and duplicate rows. In this thesis we define the notions of strict and lax functional dependencies, strict and lax equivalence constraints, and null constraints, which capture both a large set of the constraints implied by ANSI SQL query expressions, including outer joins, and a useful set of declarative constraints for ANSI SQL base tables, including unique, table, and referential integrity constraints. We develop and prove a sound set of inference axioms for this set of combined constraints, and formalize the set of constraints that hold in the result of each SQL algebraic operator. We define an extended functional dependency graph model (FD-graph) to represent these constraints, and present and prove correct a detailed polynomial-time algorithm to maintain this FD-graph for each algebraic operator. We illustrate the utility of this analysis with examples and additional theoretical results from two problem domains in query optimization: query rewrite optimizations that exploit uniqueness properties, and order optimization that exploits both functional dependencies and attribute equivalence. We show that the theory behind these two applications of dependency analysis is not only useful in relational database systems, but in non-relational database environments as well.

Acknowledgements

For the last two weeks I have written scarcely anything. I have been idle. I have failed.

Katherine Mansfield, diary, 13 November 1921

Determination not to give in, and the sense of an impending shape keep one at it more than anything.

Virginia Woolf, diary, 11 May 1920

Thesis? What thesis?

Motto of the UW Graduate House

Upon completing my Master's degree in the spring of 1990, I applied to Waterloo to pursue further study in information retrieval, hopefully with Frank Wm. Tompa. And now, nearly a decade later, I've completed a doctoral dissertation under Frank's supervision. However, as is clear from its title, this thesis has nothing to do with information retrieval. Instead, Paul Larson took me on to study query optimization in an SQL-to-IMS gateway, which re-kindled my interest in database systems and provided the motivation for much, if not all, of the work herein. After Paul left Waterloo for a career at Microsoft Research, I was paired with Frank. And good thing, too—I could not have hoped for two better mentors. I am indebted to both of them for their guidance, encouragement, and friendship, and for giving me the tools with which to start a new career. I will greatly miss the regular opportunities we had to work together. I would also like to thank my examining committee: Christoph Freytag, David Toman, Edward P. F. Chan, and Ajit Singh. Their comments on earlier drafts have been incorporated into the final text. My particular thanks to Christoph, whose enthusiasm is as contagious as his advice is edifying.

During my tenure at Waterloo I had the privilege of working with several faculty, staff, and students who not only buoyed my spirits but inspired me to follow in their collective footsteps. Ken Salem, Jo Atlee, Wm. Cowan, Darrell Raymond, Alfredo Viola, Igor Benko, Ian Bell, Gord Vreugdenhil, Dave Mason, Peter Buhr, Lauri Brown, David Clark, Andrej Brodnik, Naji Mouawad, Anne Pidduck, Gopi Krishna Attaluri, Martin Van Bommel, Weipeng Yan, Dexter Bradshaw, and Qiang Zhu all offered their friendship, encouragement, ideas, and advice. I am grateful to them all, especially Darrell, who contributed many useful comments to earlier drafts, and to Dave, who developed several of the complex LaTeX macros used to typeset this thesis.

My colleagues and managers at Sybase, notably Dave Neudoerffer, Peter Bumbulis, Anil Goel, Mark Culp, Anisoara Nica, and Ivan Bowman, were a constant source of good

ideas and encouragement. Dave never admonished me for taking too long. Without his support this thesis would have never been completed.

Funding for my studies came from several sources. Most important were scholarships awarded by the Information Technology Research Centre (now Communications and Information Technology Ontario, or CITO), NSERC, IODE, and the Canadian Advanced Technology Association (CATA). Irene Mellick believed enough in me to arrange an unprecedented, private-sector three-year scholarship from the Great-West Life Assurance Company—with no strings attached. I sincerely thank all of these agencies for their financial assistance.

I must also mention the contributions of two other individuals. Helen Tompa has been nothing short of a surrogate aunt to our twin boys, Andrew and Ryan, since they were born in April 1998. From trips to the doctor to swimming lessons, 'Aunt' Helen has always been ready to lend a hand. Thank you, Helen!

Barb Stevens RN coached us through some difficult periods in the past five years. On different occasions Barb has played the roles of coach, counselor, and friend, and always with a touch of laughter. Her contribution to the completion of this thesis is far from small.

Finally, and most of all, I thank my wife Leslie for being there with me at every step of this long adventure. Despite being displaced from her family, selling our home in Winnipeg, changing employers, the drop (!) in income, the working vacations, and the all-too-numerous lonely evenings, she knew how important this was to me.

It wasn't supposed to take nearly this long. But as our boys so often pronounce, with the enthusiasm only a two-year-old can muster: "all done!"

GNP
14 April 2000

Dedication

for Leslie, Andrew, and Ryan

Contents

Tables

Figures

1 Introduction

Although much of the extensive literature on functional dependencies pertains to schema design (decomposition and normalization), functional dependency analysis has a decisive impact on query optimization. For example, during the past decade most, if not all, of the work related to semantic query optimization is based on functional dependency analysis [33, 34, 53, 161, 181, 209, 210, 228, 230, 254, 258, 281, 295]. These techniques can often improve overall query performance by at least an order of magnitude, depending on the particular optimizations involved [230].

Functional dependency analysis also enables other query optimization possibilities. Darwen [70] offered several examples that have been discussed by other authors. His list included group-by column elimination (also studied by Yan [294]), redundant **Distinct** elimination (also studied by Pirahesh et al. [230] and Paulley and Larson [228]), and scalar subquery processing. Our original motivation for studying dependencies was to exploit the ordering of tuples when retrieved through an index, which Selinger et al. termed 'interesting orders' [247]. This very topic was the subject of a recent paper by Simmen et al. [261] that heavily references Darwen's work, though the precise algorithms for performing the functional dependency analysis are unspecified. We will take a much more detailed look at order optimization in Chapter 5, but to exemplify the possibilities we illustrate two ways of exploiting functional dependencies in query optimization.

EXAMPLE 1
Our example schema represents a manufacturing application and contains information about employees, parts, part suppliers (vendors), and so on (see Appendix A). Suppose we wish to determine the average unit price quoted by all suppliers for each individual part:

```
Select P.PartID, P.Description, Avg(Q.UnitPrice)
From   Part P, Quote Q
Where  Q.PartID = P.PartID
Group by P.PartID, P.Description
```

In this example the specification of **P.Description** in the **Group by** list is necessary, as otherwise most database systems will *reject* this query on the grounds that it contains a column in the **Select** list that is not also in the **Group by** list [70]. However, since

P.PartID is the key of the PART table there exists the functional dependency P.PartID $\longrightarrow$ P.Description. Consequently grouping the intermediate result by both columns is unnecessary—grouping the rows by P.PartID alone is sufficient[1].

EXAMPLE 2
Consider the nested query

```
Select P.PartID, V.Name
From   Part P, Supply S, Vendor V
Where  P.PartID = S.PartID and
       S.VendorID = V.VendorID and
       P.Price ≥ ( Select 1.20 × Avg(Q.QtyPrice)
                   From   Quote Q
                   Where  Q.PartID = P.PartID and
                          Q.UnitPrice ≤ 0.9 × P.Cost )
```

which gives the parts, and their suppliers, for those parts that can be acquired through at least one supplier at a reasonable discount but whose markup is, on average, at least 20%.

A naive access plan for this query involves evaluating the subquery for each row in the derived intermediate result formed by the outer query block, a procedure termed 'tuple substitution' in the literature [158]. Because such an execution strategy can result in much wasted recomputation, various researchers have proposed other semantically equivalent access strategies using query rewrite optimization techniques [158, 210, 253]. On the other hand, another possibility is to cache (or *memoize* [127, 202]) the subquery results during query execution to avoid recomputation. That is, if we think of the subquery as a function whose range is Q.QtyPrice and whose domain is the set of correlation attributes { P.PartID, P.Cost }, then it is easy to see how one can cache subquery results as they are computed to avoid subsequent subquery computations on the same inputs. IBM S DB2/MVS[2] and Sybase's SQL Anywhere and Adaptive Server Enterprise are examples of commercial database systems that memoize the previously computed results of subqueries in this manner.

We can make several observations about this nested query with regard to the memoization of its results. First, it is clear that the only correlation attribute that matters is P.PartID, since the functional dependency P.PartID $\longrightarrow$ P.Cost holds in the outer

---

1 The (redundant) specification of columns in the Group-by clause is so common that the ANSI SQL standards committee is to consider permitting functionally-determined columns to be omitted from the Group by clause. Hugh Darwen, personal communication, 17 October 1996.

2 Guy M. Lohman, IBM Almaden Laboratory, personal communication, 30 June 1996.

query block. Exploiting this fact can make memoization less costly, as there is one less attribute to consider while caching the subquery's result. Second, an elaborate caching scheme for subquery results can only pay off if the subquery will be computed multiple times for the same input parameters. If, for example, the join strategy in the outer block began with a sequential scan of the PARTS table, then the cache need only be of size one, since once a new part number is considered the subquery will never be invoked with part numbers encountered previously. Third, suppose we modify the nested query slightly to consider each vendor in the average price calculation, as follows:

```
Select P.PartID, V.Name
From   Part P, Supply S, Vendor V
Where  P.PartID = S.PartID and
       S.VendorID = V.VendorID and
       P.Price ≥ ( Select 1.20 × Avg(Q.QtyPrice)
                   From   Quote Q
                   Where  Q.PartID = P.PartID and
                          Q.VendorID = V.VendorID
                          Q.UnitPrice ≤ 0.9 × P.Cost ),
```

so that parts are considered on a vendor-price basis. In this case there are three correlation attributes (though once again `P.Cost` is functionally determined by `P.PartID`). What is interesting here is that the two attributes `P.PartID` and `V.VendorID`, together with the join conditions in the outer block, form the key of the outer query block. Consequently memoization of the subquery results is unnecessary, as the subquery will be invoked only once for each distinct set of correlation attributes.

In this thesis we present algorithms for determining which *interesting* functional dependencies hold in derived relations (we discuss what defines an *interesting* dependency in Chapter 3). Our interests lie in not only determining the functional dependencies that hold in the final result, but in any intermediate results as well, so their analysis can lead to additional optimization opportunities. In two subsequent chapters we discuss applications of this dependency analysis: semantic (rewrite) query optimization and order optimization. Our research contributions are:

1. a detailed algorithm to develop the set of interesting functional dependencies that hold in a final or intermediate result, and a description of how this framework can be integrated into an existing query optimizer. The underlying data model supported is based on the ANSI SQL standard [136, 137], which includes multiset semantics, null values, and three-valued logic. The query syntax we support encompasses a large subset of ANSI SQL and includes query expressions, query specifications involving grouping, and nested queries.

2. Theorems (with proofs of correctness) for exploiting functional dependencies in semantic query optimization, specifically illustrating the equivalence of nested queries with a canonical form [158] consisting of only the projection, restriction, and join operations. Portions of this work have been previously published by Paulley and Larson in a conference paper [228].

3. A formal description, axioms, and theorems for describing *order properties* (what Selinger et al. originally described as 'interesting orders') and how order properties interact with functional dependencies. We explore how we can exploit functional dependencies to simplify order properties and hence discover more opportunities for eliminating unnecessary sorting during query processing.

The rest of the thesis is organized as follows. We begin with a description of the algebra used to represent SQL queries and definitions of constraints and functional dependencies in an ANSI relational data model. We follow this with an overview of query processing in a relational database system and present a brief survey of query optimization literature, with a focus towards query rewrite optimization and access plan generation techniques that utilize functional dependencies.

Chapter 3 presents detailed algorithms for determining derived functional dependencies, using a customized graph [19] to represent a set of functional dependencies. Of particular note is the development of algorithms to determine the set of functional dependencies that hold in the result of an outer join.

In Chapter 4 we describe semantic query optimization techniques that can exploit our knowledge of derived functional dependencies. In particular we concentrate on determining whether a (final or intermediate) result contains a key. If so, we can then determine if an unnecessary `Distinct` clause can be eliminated, which can significantly reduce the overall cost of computing the result. While the hypergraph framework described in Chapter 3 would result in better optimization of complex queries (particularly those involving grouped views), we present a simplified algorithm that can handle a large subclass of queries without the need for the complete hypergraph implementation. We go on to describe other applications of duplicate analysis, including the transformation of subqueries to joins and intersections (and vice-versa).

Chapter 5 describes the relationship between functional dependencies and order properties. We define an order property as a form of dependency on a tuple sequence and develop axioms for order properties in combination with functional dependencies. Our focus is on determining order properties that hold in a derived result, specifically one that includes joins, outer joins, or a mixture of the two. This work formalizes several con-

cepts presented in two earlier papers by Simmen et al. [261, 262] and extends it to consider queries over more than one table.

Finally, we conclude the thesis in Chapter 6 with an overview of the major contributions of the thesis, present some possible extensions to the work given herein, and add some ideas for future research. Appendix A outlines the example schema used throughout the thesis.

2 Preliminaries

2.1 Class of SQL queries considered

In this thesis, we consider a subset of ANSI SQL2 [136] queries for which the query optimization techniques discussed in subsequent chapters may be beneficial. Following the SQL2 standard, queries corresponding to *query specifications* consist of the algebraic operators selection, projection, inner join, left-, right-, and full- outer joins, Cartesian product, and grouping. The selection condition in a **Where** clause involving tables R and S is expressed as $C_R \wedge C_S \wedge C_{R,S}$ where each condition is in conjunctive normal form. For grouped queries we denote the grouping attributes as A^G and any aggregation columns as A^A. $F(A^A)$ denotes a series of arithmetic expressions F on aggregation columns A^A. More precise definitions of these operators and attributes are given below.

Without loss of generality, for query specifications consisting of only inner joins and Cartesian products we assume that the **From** clause consists of only two tables, R and S, since we can rewrite a query involving three or more tables in terms of two (we ignore here the recently-added ANSI syntax for inner joins, which in all cases can be rewritten as a set of restriction conditions over a Cartesian product).

Because outer joins do not commute with other algebraic operators, outer joins require much more detailed analysis, in particular with nested outer joins [30, 33, 98]. In the simple case involving two (possibly derived) tables, the result of R **Left Outer Join** S **On** P is a table where each row of R is guaranteed to be in the result (thus R is termed the *preserved relation*). Tables R and S are joined over predicate P as for inner join, but with a major difference: if any row r_0 of R fails to join with any row from S—that is, there is no row s_0 of S which, in combination with r_0 satisfies P—then r_0 appears in the result with null values appended for each column of S (S is termed the *null-supplying* relation for this reason). Such a resulting tuple, projected over the columns of S, is termed the *all-null row* of S.

ANSI SQL query specifications can contain *scalar functions*. We denote a scalar function with input parameters X with the notation $\lambda(X)$. Scalar functions are permitted in a **Select** list, and are also permitted in any condition C in a **Where** or **Having** clause, or in the **On** condition of an outer join.

EXAMPLE 3 (SCALAR FUNCTIONS)
Suppose we have the query

```
Select P.*, (P.Price - P.Cost) as Margin
From   Part P
Where  P.Status = 'InStock'.
```

Then we interpret 'margin' as a scalar function λ with two input parameters `P.Price` and `P.Cost`. In this thesis we assume that scalar functions (1) reference only constants κ and input attributes; (2) are free of side-effects; (3) are idempotent—they return the same result for the same input values in every case; and (4) for the purposes of functional dependency analysis the function's parameter list can be treated as a set, that is, the order of the function's parameters is unimportant.

Subqueries, involving existential or universal quantification, are permitted in most selection predicates, including θ-comparisons, but not in a **Select** list. Set containment predicates that utilize subqueries, such as **In**, **Some**, **Any**, or **All**, are converted to their semantically equivalent canonical form (see Section 2.3.1.2 below). Query specifications may contain a **Group By** or **Having** clause, involve aggregation operators, or contain arithmetic expressions[3]. Thus the SQL query specifications we consider have the following familiar syntax:

```
Select [Distinct/All] [A^G] [, A] [, F(A^A)]
From   R, S or
       R [Left Right Full] Outer Join S On P_R ∧ P_S ∧ P_R,S
Where  C_R ∧ C_S ∧ C_R,S
Group by A^G
Having C
```

The semantics of ANSI SQL query specifications are roughly as follows. The derived table defined by the table expression in the **From** clause is constructed first; only those rows satisfying the condition in the query's **Where** clause are retained. If the query includes a Group by clause, then the derived table is *partitioned* into distinct sets based on the values of the columns and/or expressions in the **Group by** clause. A **Having** clause restricts the result of the partitioning to include only those groups that satisfy its condition. Finally, the result is *projected* over the columns in the **Select** list, and in the case of **Select** Distinct, duplicate rows in the final result are eliminated.

In addition to query specifications, we also consider a subset of SQL2 *query expressions*. These expressions involve two query specifications related by one of the following algebraic

---

3 We alert the reader that each section in the sequel may restrict the set of allowed SQL syntax to focus the analysis on particular optimization techniques.

operators: **Union**, **Union All**, **Intersect**, **Intersect All**, **Except**, and **Except All**. We assume the two query specifications produce derived tables that are *union-compatible* (see Definition 19 below). Similar to group-by expressions, we identify those attributes specified in an **Order by** clause by A^O, which can be specified for a single query specification or for a query expression. In summary, we consider both query specifications—of the familiar **Select-From-Where** variety—and query expressions that match the following basic syntax:

Select [Distinct/All] [A^G] [, A] [, $F(A^A)$]
From R, S *or*
R [Left Right Full] Outer Join S On $P_R \wedge P_S \wedge P_{R,S}$
Where $C_R \wedge C_S \wedge C_{R,S}$
Group by A^G
Having C
Union *or* **Union All** *or*
Intersect *or* **Intersect All** *or*
Except *or* **Except All**
Select [Distinct/All] [A^G] [, A] [, $F(A^A)$]
From R, S *or*
R [Left Right Full] Outer Join S On $P_R \wedge P_S \wedge P_{R,S}$
Where $C_R \wedge C_S \wedge C_{R,S}$
Group by A^G
Having C
Order by A^O

These query expressions will range over a multiset relational model described by the ANSI SQL standard [136] that supports duplicate rows, null values, three-valued logic, and multiset semantics for algebraic operators, as described below.

2.2 Extended relational model

To reason conveniently about derived functional dependencies in a multiset relational algebra such as ANSI SQL, we require extensions to the ANSI SQL relational model that distinguishes between *real* and *virtual* attributes. Real attributes correspond to those available for manipulation by SQL statements—that is, those attributes that form the set of schema attributes in the 'traditional' relational algebra. On the other hand, virtual attributes are used solely by the DBMS; a typical use of a virtual attribute is to represent a unique tuple identifier [30, 74, 228, 230] that serves as a surrogate primary key.

Following an approach similar to that of Bhargava, Goel, and Iyer [30], we define an extended relational model that includes 'virtual attributes' to enable the analysis of functional dependencies and equivalence constraints that hold in the results of algebraic ex-

pressions. Furthermore, in Section 2.3, we define an algebra over this extended relational model and show its equivalence to standard SQL expressions as defined by ANSI [136].

We note that in any DBMS implementation it is unnecessary for any 'virtual attribute' to actually exist; they serve only as a bookkeeping mechanism. In particular, the tuple identifier of a derived table does not imply that the intermediate result must itself be materialized.

DEFINITION 1 (TUPLE)
A *tuple* t is a mapping from a set of finite attributes $\alpha \cup \iota \cup \kappa \cup \rho$ to a set of atomic or set-valued values (see Definition 5 below) where α is a non-empty set of *real attributes*, ι is a solitary *virtual attribute* consisting of a unique *tuple identifier*, κ is a set, possibly empty, of constant atomic values, and ρ is a set, possibly empty, of additional *virtual attributes* constrained such that the sets α, ι, κ, and ρ are mutually disjoint and t maps ι to a non-`Null` value.

Notation. We use the notation $t[A]$ to represent the values of a nonempty set of attributes $A = \{a_1, a_2, \ldots, a_n\}$, where $A \subseteq \alpha \cup \iota \cup \kappa \cup \rho$, of tuple t.

DEFINITION 2 (TUPLE IDENTIFIER)
The *tuple identifier* ι of an extended table R, written $\iota(R)$, is an attribute in the schema of R whose values uniquely identify each tuple in any instance of R. Its values are taken from a boundless domain D_ι whose values are atomic, definite, and comparable (see Definitions 4 and 5 below); the domain of natural numbers N has the required characteristics to substitute for tuple identifiers. We assume the existence of a generating function that, when required, produces a new tuple identifier that is unique over the entire schema.

DEFINITION 3 (EXTENDED TABLE)
An *extended table* R is a five-tuple $\langle \alpha, \iota, \kappa, \rho, I \rangle$ where α is a non-empty set of real attributes, ι represents a solitary virtual attribute representing a tuple identifier, κ is a set, possibly empty, of constants, ρ is a set, possibly empty, of virtual attributes, and I is the *extension* of R containing a set, possibly empty, of tuples over $\alpha \cup \iota \cup \kappa \cup \rho$ such that

$$\forall\, t_1, t_2 \in I : t_1 \neq t_2 \Longrightarrow t_1[\iota] \neq t_2[\iota]. \tag{2.1}$$

The combined set of attributes $\alpha \cup \iota \cup \kappa \cup \rho$ is termed the *schema* of R and abbreviated $sch(R)$.

Notation. We use the notation $\alpha(R)$ to represent the nonempty set of real attributes of a table R, and similarly use the notation $\iota(R)$, $\kappa(R)$, and $\rho(R)$ to denote the respective virtual columns of table R. We follow convention by calling the extension I of table R an *instance* of R, written $I(R)$. In order to keep our algebraic notation more readable, we adopt the shorthand convention of simply writing $S \times T$ instead of $I(S) \times I(T)$.

DEFINITION 4 (DEFINITE AND NULLABLE ATTRIBUTES)
Each attribute in a table has an associated domain of values. To simplify the exposition of the theorems in this thesis, we assume without loss of generality that all domain values are taken from the set of natural numbers, denoted N. Such a simplification is typical in the research literature; cf. references [65, 163, 223].

Using Maier's terminology [193, pp. 373] we define a *definite attribute* as one that cannot be `Null`, that is, its domain is simply N. If a set of attributes X in table R are each definite, then we say that R is X-definite[4]. A *nullable attribute* takes its domain from the set $\overline{N} = N \cup \texttt{Null}$.

We are, in almost all cases, concerned with nullability as defined by the schema or a query, independent of any particular database instance. As with base table attributes, derived attributes are also either definite or nullable. Nevertheless, we occasionally need to consider specific instances of tables, in which case we extend the notions of definite and nullable to apply to instances $I(R)$ of an extended table R and to specific tuples in such an instance; for example, 'X is nullable' means 'the value of attribute X of a tuple $r \in I(R)$ may be `Null` .

DEFINITION 5 (ATOMIC AND SET-VALUED ATTRIBUTES)
An attribute is *atomic* or *single-valued* if its domain is a subset of $\overline{N}$. An attribute is *set-valued* if its domain is a subset of the power set of $\overline{N}$.

Notation. A *theta-comparison*, or θ-comparison, is an atomic condition comparing two values using any of the operators $\{<, \leq, >, \geq, =, \neq\}$. Following standard notational conventions, we represent the logical operators implication, equivalence, negation, conjunction and disjunction with the standard notation $\Longrightarrow$, $\Longleftrightarrow$, $\neg$, $\wedge$, and $\vee$, respectively. We write X instead of $\{X\}$ when X is understood to be a set of attributes, and we write XY to denote the set union of X and Y, X and Y not necessarily disjoint. $\|X\|$ represents the number of attributes in X; the logical operator $\setminus$ denotes the set difference

4 Other researchers, for example Lien [184], use the term *total* instead of definite; the semantics are equivalent.

of two sets X and Y, written $X \setminus Y$. We normally omit specifying the universe of attributes as it is usually obvious from the context. Table 2.1 summarizes additional notation used throughout this thesis.

In SQL2 the comparison of `Null` with any non-null value always evaluates to *unknown*. However, the result of the comparison between two null values depends on the context: within `Where` and `Having` clauses, the comparison evaluates to *unknown;* within `Group By`, `Order By`, and particularly duplicate elimination via `Select Distinct`, the comparison evaluates to *true*. To accommodate this latter interpretation, we adopt the *null comparison operator* of Negri et al. [214, 216]:

DEFINITION 6 (NULL COMPARISON OPERATOR)
The *null comparison operator* $\stackrel{\omega}{=}$ evaluates to *true* if both its operands are `Null` or if both operands have the same value, and *false* otherwise.

Using the null comparison operator, we can formally state that two tuples t_0 and t'_0 from instance $I(R)$ of an extended table R are equivalent if

$$\forall\, t_0, t'_0 \in I(R) : \bigwedge_{u_i \in U} t_0[u_i] \stackrel{\omega}{=} t'_0[u_i] \tag{2.2}$$

where $U = sch(R) \setminus \iota(R)$.

Note that Definition 3 does not preclude $t_1[\alpha(R)] \stackrel{\omega}{=} t_2[\alpha(R)]$. Hence, when considering real attributes only, our definition of table represents a multiset, and not a 'classical' *relation*. In the remainder of this thesis we will use the term *extended table* to denote a 'relation' as defined in Definition 3, use the term *table* to denote a multiset 'relation' as defined in ANSI SQL [136], and use the term *relation* when we mean a relation in the 'classical' relational model [65].

2.3 An algebra for SQL queries

Because our relational model includes both real and virtual attributes, we cannot simply base our algebraic operators on their 'equivalent' SQL statements alone. Instead, we define a set of algebraic operators that manipulate the tables in our extended relational model, and subsequently we show the semantic equivalence between this algebra and SQL expressions. For each operator, assume that R, S, and T denote extended tables and the sets $\alpha(R), \iota(R), \rho(R), \alpha(S), \iota(S), \rho(S), \alpha(T), \iota(T), \rho(T)$ are mutually disjoint.

Symbol	*Definition*
$sch(R)$	Real and virtual attributes of extended table R
$\alpha(R)$	Real attributes of extended table R
$\alpha(C)$	Attributes referenced in predicate C
$\alpha(e)$	Attributes in the result of a relational expression e
$\alpha_R(C)$	Attributes of table R referenced in predicate C
$\kappa(R)$	constant attributes of extended table R
$\kappa(C)$	Constants, host variables, and outer references present in predicate C
Key(R)	A key of table R
$\iota(R)$	Tuple identifier attribute of extended table R
$\rho(R)$	Virtual attributes of extended table R
A_R	Attributes specifically from extended table R
a_i	ith attribute from the set A_R (or its variations below)
A_R^G	Grouping attributes on R
A_R^O	Ordering attributes on R
A_R^A	Set-valued aggregation columns of grouped table R
$F(A_R^A)$	Set $F = \{f_1, f_2, \ldots f_n\}$ of arithmetic aggregation expressions over set-valued aggregation columns A^A of grouped table R
C_R	Predicate on attributes of R in conjunctive normal form
$C_{R,S}$	Predicate over attributes of both R and S in conjunctive normal form
h	Set $\{h_1, h_2, \ldots, h_n\}$ of host variables in a query predicate
$I(R)$	An instance I of extended table R
T_R	Table constraints on table R in CNF
$K_i(R)$	Attributes of candidate key i on table R (primary key or unique index)
$U_i(R)$	Attributes of unique constraint i (candidate key) on table R

TABLE 2.1: Summary of symbolic notation.

2.3.1 Query specifications

In this section, we define a relational algebra over extended tables that mirrors the definition of a query specification in the ANSI SQL standard [136]. In SQL, a query specification includes the algebraic operators projection, distinct projection, selection, Cartesian product, and inner and outer join, which we describe below. We describe our algebraic operators that implement grouping and aggregation, which are also contained within query specifications, in Section 2.3.1.4.

2.3.1.1 Select-project-join expressions

DEFINITION 7 (PROJECTION)
The *projection* $\pi_{All}[A](R)$ of an extended table R onto attributes A forms an extended table R'. The set $A = A_R \cup \Lambda$, where $A_R \subseteq \alpha(R)$ and Λ represents a set of m scalar functions $\{\lambda_1(X_1), \lambda_2(X_2), \ldots, \lambda_m(X_m)\}$ with each $X_k \subseteq \alpha(R) \cup \kappa$. The scheme of R' is:

- $\alpha(R') = A$;
- $\iota(R') = \iota(R)$;
- $\kappa(R') = \kappa(R) \cup \kappa(\Lambda)$;
- $\rho(R') = \rho(R) \cup \{\alpha(R) \setminus A\}$.

The instance $I(R')$ is constructed as follows:

$$I(R') = \{r' \mid \exists r \in I(R) : r'[sch(R)] \stackrel{\omega}{=} r[sch(R)] \wedge (\forall \lambda(X) \in \Lambda : r'[\lambda] \stackrel{\omega}{=} \lambda(r[X])) \wedge (\forall \kappa \in \kappa(\Lambda) : r'[\kappa] \stackrel{\omega}{=} \kappa)\}. \quad (2.3)$$

As a shorthand notation, we denote the 'view' of an extended table over which each ANSI SQL algebraic operator [136] is defined with a *table constructor.*

DEFINITION 8 (TABLE CONSTRUCTOR)
A *row* r in an ANSI table is a mapping of real attributes α to a set of atomic values. The *table construction operator* $\mathcal{R}_\alpha$, applied to an extended table R having atomic attributes $\alpha(R)$, is written $\mathcal{R}_\alpha(R)$ and produces an (ANSI SQL) table R' with attributes $\alpha(R)$ and one row r_i corresponding to each tuple $t_i \in R$ such that $r_i[\alpha(R)] \stackrel{\omega}{=} t_i[\alpha(R)]$.

Definition 8 permits us to show the equivalence of the semantics of our algebraic operators with the ANSI-defined behaviour of the corresponding operators in ANSI SQL [136].

CLAIM 1
The expression

$$Q = \mathcal{R}_\alpha(\pi_{All}[A](R))$$

correctly models the ANSI SQL statement **Select All A From** $\mathcal{R}_\alpha(R)$ where the set of attributes $A \subseteq \alpha(R) \cup \Lambda$ and Λ is a set of scalar functions as defined above.

In the rest of the thesis we reserve the term *table* to describe a base or derived table in the ANSI relational model, unless we are describing the semantics of operations over extended tables and there is no chance of ambiguity. Similarly, we use the term *tuple* to denote an element in the set $I(R)$, where R is an extended table, and the term *row* to denote the corresponding object in an ANSI SQL table.

DEFINITION 9 (EXTENDED TABLE CONSTRUCTOR)
To define instances of extended tables that result from various operators, we will occasionally need to create a collection of tuples to which we attach new, unique tuple identifiers. We will use the notation $\mathcal{R}$ to denote an *extended table constructor*, written

$$I(R) = \mathcal{R}\{t \mid P(R)\}, \tag{2.4}$$

as a shorthand for the following three-step construction:

1. Let $T(R) = \{t \mid P(R)\}$ be a set of tuples defined over $sch(R) \setminus \iota(R)$.
2. Let $\mid T(R) \mid = n$ and let $\mathcal{I}$ be a set of n newly generated, unique tuple identifiers. Form the ordered sets $\widehat{T}$ and $\widehat{\mathcal{I}}$ by arbitrarily ordering $T(R)$ and $\mathcal{I}$ respectively.
3. Let

$$\begin{aligned}\mathcal{R}\{t \mid P(R)\} = \{\, r' \;\mid\; & \exists i : 1 \le i \le \mid T(R) \mid \wedge\, r'[\iota(R')] = \widehat{\mathcal{I}}_i \wedge \\ & r'[sch(R') \setminus \iota(R')] \stackrel{\omega}{=} \widehat{T}_i[sch(R') \setminus \iota(R')] \,\}\end{aligned} \tag{2.5}$$

where $\widehat{\mathcal{I}}_i$ and $\widehat{T}_i$ denote the i^{th} members of $\widehat{\mathcal{I}}$ and $\widehat{T}$ respectively.

DEFINITION 10 (DISTINCT PROJECTION)
The *distinct projection* $\pi_{Dist}[A](R)$ of an extended table R onto attributes A is the extended table R' where $A \subseteq \alpha(R) \cup \Lambda$ and Λ represents a set of m scalar functions, as above, such that:

- $\alpha(R') = A$;
- $\iota(R')$ = a new tuple identifier attribute;

- $\kappa(R') = \kappa(R) \cup \kappa(\Lambda)$;
- $\rho(R') = \rho(R) \cup \iota(R) \cup \{\alpha(R) \setminus A\}$.

Each tuple $r' \in I(R')$ is constructed as follows. For each set of 'duplicate' tuples in $I(R)$, nondeterministically select any tuple r and include all the values of r and any scalar function results based on tuple r in the result. Hence, without loss of generality, we select the tuple with the smallest tuple identifier and define the instance as follows:

$$I(R') = \mathcal{R}\{\, r' \mid \exists r \in I(R) : (\forall r_k \in I(R) : r_k[A] \stackrel{\omega}{=} r'[A] \Longrightarrow r[\iota(R)] \leq r_k[\iota(R)]) \wedge r'[sch(R)] \stackrel{\omega}{=} r[sch(R)] \wedge (\forall \lambda(X) \in \Lambda : r'[\lambda] \stackrel{\omega}{=} \lambda(r[X])) \wedge (\forall \kappa \in \Lambda : r'[\kappa] \stackrel{\omega}{=} \kappa)\}. \quad (2.6)$$

CLAIM 2
The expression

$$Q = \mathcal{R}_\alpha(\pi_{Dist}[A](R))$$

correctly models the ANSI SQL statement **Select Distinct** A **From** $\mathcal{R}_\alpha(R)$ where the set of attributes $A \subseteq \alpha(R) \cup \Lambda$ and Λ is a set of scalar functions as defined above.

To handle the three-valued logic of ANSI SQL comparison conditions properly, we adopt the *null interpretation operator* from Negri et al. [214, 216], which defines the interpretation of an SQL predicate when it evaluates to *unknown*. In two-valued predicate calculus, the expression

$$\{x \; in \; R : P(x)\} \quad (2.7)$$

is well-defined. As a consequence of the use of three-valued logic, however, this expression is undefined since it is not known whether a tuple x', such that the predicate $P(x')$ evaluates to *unknown*, belongs to the result or not.

DEFINITION 11 (NULL INTERPRETATION OPERATOR)
The null interpretation operator is defined as follows: let $P(x)$ be a three-valued predicate formula (with range *true*, *false*, and *unknown*) and $Q(x)$ be a two-valued predicate formula. Then $Q(x)$ is a true-interpreted two-valued equivalent of $P(x)$ if

$$\begin{aligned} P(x) = \mathrm{T} &\Rightarrow Q(x) = \mathrm{T} \\ P(x) = \mathrm{F} &\Rightarrow Q(x) = \mathrm{F} \\ P(x) = \mathrm{U} &\Rightarrow Q(x) = \mathrm{T} \end{aligned}$$

Notation	*Interpretation of* Null	SQL *Semantics*
$P(x)$	undefined	x Is Not Null $\Longrightarrow P(x)$
$\lceil P(x) \rceil$	true-interpreted	$P(x)$ Is Not False
$\lfloor P(x) \rfloor$	false-interpreted	$P(x)$ Is True
$X \stackrel{\omega}{=} Y$	equivalent	(X Is Null And Y Is Null) Or X = Y

TABLE 2.2: Interpretation and Null comparison operator semantics. $P(x)$ represents a predicate P on an attribute x.

for all x. In this case, we may write $Q(x) \equiv \| P(x) \|^T$. Similarly, $Q(x)$ is a false-interpreted two-valued equivalent of $P(x)$, written $Q(x) \equiv \| P(x) \|^F$ if

$$P(x) = \mathrm{T} \Rightarrow Q(x) = \mathrm{T}$$
$$P(x) = \mathrm{F} \Rightarrow Q(x) = \mathrm{F}$$
$$P(x) = \mathrm{U} \Rightarrow Q(x) = \mathrm{F}$$

for all x. We use as a shorthand notation the form $\lceil P(x) \rceil$ to represent $\| P(x) \|^T$ and $\lfloor P(x) \rfloor$ to represent $\| P(x) \|^F$. Table 2.2 summarizes the semantics of the null interpretation operators and the null comparison operator defined previously.

DEFINITION 12 (RESTRICTION)
The *restriction* $\sigma[C](R)$ of an extended table R selects all tuples in $I(R)$ that satisfy condition C where $\alpha(C) \subseteq \alpha(R) \cup \Lambda$, and Λ represents a set of m scalar functions $\{\lambda_1(X_1), \lambda_2(X_2), \ldots, \lambda_m(X_m)\}$. Atomic conditions in C may contain *host variables* or *outer references* whose values are available only at execution time, but for the purposes of evaluation of C are treated as constants. Condition C can also contain **Exists** subquery predicates that evaluate to *true* or *false* (see Section 2.3.1.2 below). Selection conditions may be combined with other conditions using the logical binary operations *and*, *or*, and *not*. Selection does not eliminate duplicate tuples in R. By default, if condition C evaluates to unknown, then we interpret C as *false*.

The restriction operator $\sigma[C](R)$ constructs as its result an extended table R' where

- $\alpha(R') = \alpha(R)$;
- $\iota(R') = \iota(R)$;
- $\kappa(R') = \kappa(R) \cup \kappa(C) \cup \kappa(\Lambda)$;

- $\rho(R') = \rho(R) \cup \Lambda$;

and

$$I(R') = \{r' \mid \exists r \in I(R) : \lfloor C(r) \rfloor \wedge r'[sch(R)] \stackrel{\omega}{=} r[sch(R)] \wedge (\forall \lambda(X) \in \Lambda : r'[\lambda] \stackrel{\omega}{=} \lambda(r[X])) \wedge (\forall \kappa \in \kappa(C) \cup \kappa(\Lambda) : r'[\kappa] \stackrel{\omega}{=} \kappa) \}. \quad (2.8)$$

CLAIM 3
The expression

$$Q = \mathcal{R}_\alpha(\sigma[C](R))$$

correctly models the ANSI SQL statement **Select All * From** $\mathcal{R}_\alpha(R)$ **Where** C.

DEFINITION 13 (CARTESIAN PRODUCT)
The *Cartesian product* $S \times T$ of two extended tables S and T is a result R' consisting of all possible pairs of any tuple $s_0 \in S$ with any tuple $t_0 \in T$. The schema of the result R' is defined as follows:

- $\alpha(R') = \alpha(S) \cup \alpha(T)$;
- $\iota(R')$ = a new tuple identifier attribute;
- $\kappa(R') = \kappa(S) \cup \kappa(T)$;
- $\rho(R') = \rho(S) \cup \rho(T) \cup \iota(S) \cup \iota(T)$.

The instance $I(R')$ is constructed as follows:

$$I(R') = \mathcal{R}\{r' \mid \exists s \in I(S), \exists t \in I(T) : r'[sch(S)] \stackrel{\omega}{=} s[sch(S)] \wedge r'[sch(T)] \stackrel{\omega}{=} t[sch(T)] \}. \quad (2.9)$$

CLAIM 4
The expression

$$Q = \mathcal{R}_\alpha(S \times T)$$

correctly models the ANSI SQL statement **Select All * From** $\mathcal{R}_\alpha(S), \mathcal{R}_\alpha(T)$.

2.3.1.2 Translation of complex predicates

In their syntax-directed translation of SQL queries to *Extended three-valued predicate calculus* (E3VPC), Negri, Pelagatti, and Sbattella [214–216] constructed predicate calculus formulae for subqueries that directly correspond to a syntax-directed translation of ANSI SQL in that syntactic constructs involving universal quantification were translated to predicate formulas also involving universal quantification. In contrast, our canonical form for subquery evaluation uses only existential quantification, the reason being that most, if not all, implementations of SQL do not support universal quantification directly [109][5].

In this thesis we assume that all complex predicates containing **In**, **Some**, **Any**, and **All** quantifiers over nested subquery blocks have been converted into an equivalent canonical form that utilizes only **Exists** and **Not Exists**, hence transforming the original nested query into a correlated nested query. The proper transformation of universally-quantified subquery predicates relies on careful consideration of how both the original subquery predicate and the newly-formed correlation predicate must be interpreted using Negri's null-interpretation operator (see Table 2.3). In particular, to produce the correct result from an original universally-quantified subquery predicate, we must typically true-interpret the generated correlation predicate so that it evaluates to *true* when its operands consist of one or more null values.

We illustrate several of these transformations using nested queries which correspond to Kim's [158] Type N classification (no aggregation and the original subquery does not contain a correlation predicate). The queries are over the example schema defined in Appendix A.

EXAMPLE 4 (STANDARDIZED **In** PREDICATE)
For positive existentially quantified subqueries a straightforward standardization of an In predicate to an **Exists** predicate is as follows. The original query

---

5 We are concerned here with specifying the formal semantics of complex predicates, and not their optimization. Under various circumstances it may be advantageous to convert universally-quantified comparison predicates into **Exists** predicates so that evaluation of the subquery can be halted immediately once a qualifying tuple has been found [188, pp. 413]. Furthermore, we can reduce the number of permutations of predicates to simplify optimization. However, in most cases non-correlated subqueries offer better possibilities for efficient access plans.

1.	$\lceil P(x) \vee Q(x) \rceil \Longleftrightarrow \lceil P(x) \rceil \vee \lceil Q(x) \rceil$
2.	$\lfloor P(x) \vee Q(x) \rfloor \Longleftrightarrow \lfloor P(x) \rfloor \vee \lfloor Q(x) \rfloor$
3.	$\lceil P(x) \wedge Q(x) \rceil \Longleftrightarrow \lceil P(x) \rceil \wedge \lceil Q(x) \rceil$
4.	$\lfloor P(x) \wedge Q(x) \rfloor \Longleftrightarrow \lfloor P(x) \rfloor \wedge \lfloor Q(x) \rfloor$
5.	$\lceil \neg P(x) \rceil \Longleftrightarrow \neg \lfloor P(x) \rfloor$
6.	$\lfloor \neg P(x) \rfloor \Longleftrightarrow \neg \lceil P(x) \rceil$
7.	$\lceil \lfloor P(x) \rfloor \rceil \Longleftrightarrow \lfloor P(x) \rfloor$
8.	$\lfloor \lceil P(x) \rceil \rfloor \Longleftrightarrow \lceil P(x) \rceil$

TABLE 2.3: Axioms for the null interpretation operator [216, pp. 528].

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  ⌊ S.PartID In ( Select P.PartID
                       From   Part P
                       Where  ⌊ Q(P) ⌋ )  ⌋
```

may be standardized to:

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  Exists( Select *
               From   Part P
               Where  ⌊ Q(P) ⌋ and
                      ⌊ S.PartID = P.PartID ⌋ ).
```

EXAMPLE 5 (STANDARDIZED NEGATED **In** PREDICATE)
To standardize a negated, existentially quantified **In** subquery, it is necessary to consider the outcome of the comparison of the correlated values, which may be *unknown*. With a negated In predicate we must interpret an *unknown* result as *true*, so that **Not Exists** will evaluate to *false* and the current row in SUPPLY will not appear in the result:

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  ⌊ S.PartID Not In ( Select P.PartID
                           From   Part P
                           Where  ⌊Q(P)⌋) ⌋.
```

The above nested query can be standardized to:

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  not Exists( Select *
                   From   Part P
                   Where  ⌊Q(P)⌋ and
                          ⌈ S.PartID = P.PartID ⌉ ).
```

We make the following observations concerning the standardization of **In** predicates:

- An **In** predicate is equivalent to a *quantified comparison predicate* containing **Any** or **Some** combined with the arithmetic comparison '='.
- We have altered the originally non-correlated subquery by adding a correlation predicate to the subquery's **Where** clause. This transformation can also be applied to correlated subqueries that already contain one or more correlation predicates, since the existence of additional correlation predicates does not affect the correctness of the transformation.
- A **Not** may be straightforwardly applied to both the **In** and **Exists** predicates.

EXAMPLE 6 (STANDARDIZED **All** PREDICATE)
Consider the following nested query containing an **All** predicate (where θ is one of the standard SQL arithmetic comparison operators):

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  ⌊ S.PartID θ All ( Select P.PartID
                          From   Part P
                          Where  ⌊Q(P)⌋ ) ⌋.
```

To standardize this nested query to use **Exists** we must (1) invert the comparison operator used in the **All** predicate, and (2) again account for the possible result of the comparison to be *unknown*. The latter situation also requires the true interpretation of the correlation predicate, so that the **Not Exists** predicate returns *false:*

```
Select S.VendorID, S.SupplyCode, S.Lagtime
From   Supply S
Where  not Exists( Select *
                   From   Part P
                   Where  ⌊Q(P)⌋ and
                          ⌈ ¬ S.PartID θ P.PartID ⌉).
```

We claim, without proof, that all other forms of subquery predicates that occur in a `Where` clause can be converted in the same manner, including those whose subqueries contain aggregation, **`Group by`**, **`Distinct`**, or consist of query expressions involving **`Union`**, `Intersect`, or **`Except`**.

2.3.1.3 Outer joins

DEFINITION 14 (LEFT OUTER JOIN)
The *left outer join* $S \xrightarrow{p} T$ of the extended table S (preserved) and the extended table T (null-supplying) forms an extended table R' whose result $I(R')$ consists of the union of those tuples that result from the inner join of S and T, and those tuples in S, padded with null values, that fail to join with any tuples of T. The outer join predicate p is such that $\alpha(p) \subseteq \alpha(S) \cup \alpha(T) \cup \kappa \cup \Lambda$ and, like restriction conditions, may contain outer references to attributes of super queries and host variables (both treated as constant values in κ), or scalar functions Λ. The schema of the result R' is as follows:

- $\alpha(R') = \alpha(S) \cup \alpha(T)$;
- $\iota(R')$ = a new tuple identifier attribute;
- $\kappa(R') = \kappa(S) \cup \kappa(T) \cup \kappa(p)$;
- $\rho(R') = \rho(S) \cup \rho(T) \cup \iota(S) \cup \iota(T) \cup \Lambda$.

The instance $I(R')$ is constructed as follows. First, we construct a single tuple t_{Null} defined over $sch(T)$ where $t_{\text{Null}}[\iota(T)]$ is a newly-generated, unique tuple identifier and $t_{\text{Null}}[sch(T) \setminus \iota(T)] =$ **`Null`** to represent the all-**`Null`** row of T. Then $I(R')$ is:

$$
\begin{aligned}
I(R') = \mathcal{R}\{r' \mid\ & (\exists s \in I(S), \exists t \in I(T) : \lfloor p(s,t) \rfloor \wedge r'[sch(S)] \stackrel{\omega}{=} s[sch(S)] \wedge \\
& \quad r'[sch(T)] \stackrel{\omega}{=} t[sch(T)]) \\
\vee\ & (\exists s \in I(S) : (\nexists t \in I(T) : \lfloor p(s,t) \rfloor) \wedge r'[sch(S)] \stackrel{\omega}{=} s[sch(S)] \wedge \\
& \quad r'[sch(T)] \stackrel{\omega}{=} t_{\text{Null}}[sch(T)])\}.
\end{aligned}
\tag{2.10}
$$

CLAIM 5
The expression

$$Q = \mathcal{R}_\alpha(S \xrightarrow{p} T)$$

correctly models the ANSI SQL statement
`Select All * From` $\mathcal{R}_\alpha(S)$ **`Left Outer Join`** $\mathcal{R}_\alpha(T)$ **`On`** p.

DEFINITION 15 (RIGHT OUTER JOIN)
The *right outer join* of extended tables S and T on a predicate p, written $S \xleftarrow{p} T$, is semantically equivalent to the left outer join $T \xrightarrow{p} S$.

DEFINITION 16 (FULL OUTER JOIN)
The *full outer join* $S \xleftrightarrow{p} T$ of extended tables S and T forms an extended table R' whose result $I(R')$ consists of the union of (1) those tuples that result from the inner join of S and T, (2) those tuples in S that fail to join with any tuples of T, and (3) those tuples in T that fail to join with any tuples of S. Hence S and T are both preserved and null-supplying. As with left outer join, the outer join predicate p is such that $\alpha(p) \subseteq \alpha(S) \cup \alpha(T) \cup \kappa \cup \Lambda$ and may contain outer references to attributes of super queries and host variables (both treated as constant values κ) or scalar functions Λ. The schema of R' is as follows:

- $\alpha(R') = \alpha(S) \cup \alpha(T)$;
- $\iota(R')$ = a new tuple identifier;
- $\kappa(R') = \kappa(S) \cup \kappa(T) \cup \kappa(p)$;
- $\rho(R') = \rho(S) \cup \rho(T) \cup \iota(S) \cup \iota(T) \cup \Lambda$.

The instance $I(R')$ is constructed similarly to left outer joins. We first construct the two single tuples s_{Null} and t_{Null} defined over the schemes $sch(S)$ and $sch(T)$, respectively, to represent the all-`Null` row from each, such that $s_{\text{Null}}[\iota(S)]$ is a newly-generated, unique tuple identifier and $s_{\text{Null}}[sch(S) \setminus \iota(S)] = \texttt{Null}$ and $t_{\text{Null}}[\iota(T)]$ is a newly-generated, unique tuple identifier and $t_{\text{Null}}[sch(T) \setminus \iota(T)] = \texttt{Null}$. Then we construct the instance $I(R')$ as follows:

$$
\begin{aligned}
I(R') = \mathcal{R}\{r' \mid\ & (\exists s \in I(S), \exists t \in I(T) : \lfloor p(s,t) \rfloor \wedge r'[sch(S)] \stackrel{\omega}{=} s[sch(S)] \wedge \\
& \quad r'[sch(T)] \stackrel{\omega}{=} t[sch(T)]) \\
\vee\ & (\exists s \in I(S) : (\nexists t \in I(T) : \lfloor p(s,t) \rfloor) \wedge r'[sch(S)] \stackrel{\omega}{=} s[sch(S)] \wedge \\
& \quad r'[sch(T)] \stackrel{\omega}{=} t_{\text{Null}}[sch(T)]) \\
\vee\ & (\exists t \in I(T) : (\nexists s \in I(S) : \lfloor p(s,t) \rfloor) \wedge r'[sch(T)] \stackrel{\omega}{=} t[sch(T)] \wedge \\
& \quad r'[sch(S)] \stackrel{\omega}{=} s_{\text{Null}}[sch(S)]) \}.
\end{aligned}
\tag{2.11}
$$

CLAIM 6
The expression

$$Q = \mathcal{R}_{\alpha}(S \xleftrightarrow{p} T)$$

correctly models the ANSI SQL statement
`Select All * From` $\mathcal{R}_\alpha(S)$ `Full Outer Join` $\mathcal{R}_\alpha(T)$ `On` p.

2.3.1.4 Grouping and aggregation

A *grouped query* in SQL is a query that contains aggregate functions, the `Group by` clause, or both. The idea is to *partition* the input table(s) by the distinct values of the *grouping columns*, namely those columns specified in the `Group by` clause. Each partition forms a row of the result; the values of the aggregate functions are computed over the rows in each partition. If there does not exist a `Group by` clause then each aggregation function treats the input relation(s) as a single 'group'.

Precisely defining the semantics of grouped queries in terms of SQL is problematic, since SQL does not define an operator to create a *grouped table* in isolation of the computation of aggregate functions. Two approaches to the problem have appeared in the literature. Yan and Larson [294, 296] chose to use the `Order by` clause to capture the semantics of 'partitioning' a table into groups. Darwen [70], on the other hand, defined a grouped table using nested relations, something yet to be supported in ANSI SQL. In fact, the expressive power of arbitrary nested relations is unnecessary; simply defining aggregate functions over set-valued attributes (see Definition 5), as in reference [223], is sufficient to capture the semantics required. We separate the definition of a grouped table from aggregation using set-valued attributes as in Darwen's approach. Our formalisms, however, are a simplified version of the formalisms defined by Yan [294]. That is, we separate the concepts of grouping and aggregation by treating a `Having` clause as a restriction over a projection of a grouped extended table, as described below.

EXAMPLE 7 (CONVERSION OF HAVING PREDICATES)
Suppose we are given the query

```
Select Q.PartID, Avg(MinOrder)
From   Quote Q
Where  Q.ExpiryDate > '10-10-1997'
Group by Q.PartID
Having Avg(UnitPrice) > 25.00
```

that computes the average minimum order for recent orders of each part, so long as that part's average unit price exceeds $25.00. We can syntactically transform this query into an equivalent query

```
Select QG.PartID, QG.AvgOrder
From   QG
Where  QG.AvgPrice > 25.00
```

over a materialized intermediate result QG whose definition is the query

```
Select Q.PartID as PartID, Avg(MinOrder) as AvgOrder, Avg(UnitPrice) as AvgPrice
From   Quote Q
Where  Q.ExpiryDate > '10-10-1997'
Group by Q.PartID.
```

Notice that the intermediate result contains the average unit price, since the **Where** clause in the query over QG must be able to restrict its result by using this value.

DEFINITION 17 (PARTITION)
The *partition* of an extended table R, written $\mathcal{G}[A_R^G, A_R^A](R)$, partitions R on n grouping columns $A_R^G \equiv \{a_1^G, a_2^G, \ldots, a_n^G\}$, n possibly 0, and $A_R^G \subseteq \alpha(R) \cup \kappa \cup \Lambda$ where Λ represents a set of m scalar functions $\{\lambda_1(X_1), \lambda_2(X_2), \ldots, \lambda_m(X_m)\}$. The result is a *grouped extended table,* which we denote as R', that contains one tuple per partition. We note that any of the grouping columns can be one of (a) a base table column, (b) a derived column from an intermediate result, or (c) the result of a scalar function application λ in the Group by clause. Each tuple in R' contains as real attributes in $\alpha(R)$ the n grouping columns A_R^G and m set-valued columns, $A_R^A \equiv \{a_1^A, a_2^A, \ldots, a_m^A\}$, where each $a_k^A \in A_R^A$ contains the values of that column for each tuple in the partition. If $n > 0$ and $I(R)$ is empty then $I(R') = \emptyset$. If $n = 0$ then $I(R')$ consists of a single tuple where $\alpha(R)$ consists of only the set-valued attributes A_R^A, which in turn contain all the values of that attribute in $I(R)$. Note that if $I(R) = \emptyset$ and $n = 0$ then $I(R')$ still contains one tuple, but each of the m set-valued attributes A_R^A consists of the empty set.

More formally, the schema of R' is as follows:

- $\alpha(R') = A_R^G \cup A_R^A$;
- $\iota(R')$ = a new tuple identifier attribute;
- $\kappa(R') = \kappa(R) \cup \kappa(\Lambda)$;
- $\rho(R') = \rho(R) \cup \iota(R) \cup \Lambda$.

Note that, after partitioning, the only atomic attributes in $sch(R')$ are the grouping columns A^G. Furthermore, if A^G is empty—that is, there is no **Group by** clause—then the set Λ is also empty. The instance $I(R')$ is constructed as follows.

- *Case (1),* $A^G \neq \emptyset$. Each tuple $r'_0 \in I(R')$ is constructed as follows. For each set of tuples in $I(R)$ that form a partition with respect to A^G, nondeterministically select any tuple of that set, say tuple r, and include all the values of r and any

scalar function results based on tuple r, in the result as r'. Then extend r' with the necessary set-valued attributes derived from each tuple in the set. Hence

$$I(R') = \mathcal{R}\{r' \mid \exists r \in I(R) : (\forall r_k \in I(R) : r[A^G] \stackrel{\omega}{=} r_k[A^G] \Longrightarrow r[\iota(R)] \leq r_k[\iota(R)]) \wedge r'[sch(R)] \stackrel{\omega}{=} r[sch(R)] \wedge (\forall \lambda(X) \in \Lambda : r'[\lambda] \stackrel{\omega}{=} \lambda(r[X])) \wedge (\forall \kappa \in \Lambda : r'[\kappa] \stackrel{\omega}{=} \kappa) \wedge (\forall a_i^A \in A^A : r'[a_i^A] = \{t[a_i^A \cup \iota(R)] \mid t \in I(R) \wedge r[A^G] \stackrel{\omega}{=} t[A^G]\})\}. \quad (2.12)$$

- *Case (2)*, $A^G = \emptyset$: Construct a single tuple $r'_0 \in I(R')$ such that:
  - $r'_0[\iota(R')]$ is a newly-generated, unique tuple identifier;
  - $\forall a_i^A \in A^A : r'_0[a_i^A] = \{r[a_i^A \cup \iota(R)] \mid r \in I(R)\}$; and
  - for $r \in I(R)$ such that $\forall r_k \in I(R) : r[\iota(R)] \leq r_k[\iota(R)]$, $r'_0[sch(R)] \stackrel{\omega}{=} r[sch(R)]$.

DEFINITION 18 (GROUPED TABLE PROJECTION)
The *grouped table projection* of a grouped extended table R, written $\mathcal{P}[A_R^G, F[A_R^A]](R)$, where $F \equiv \{f_1, f_2, \ldots, f_k\}$, $A_R^G \equiv \{a_1^G, a_2^G, \ldots, a_n^G\}$, $A_R^A \equiv \{a_1^A, a_2^A, \ldots, a_m^A\}$, and $F[A_R^A] \equiv (f_1(A_R^A), f_2(A_R^A), \ldots, f_k(A_R^A))$, projects the grouped extended table R over the n grouping columns A_R^G in R and over the aggregation expressions contained in F, retaining duplicate tuples in the result. More formally, the schema of R' is defined as:

- $\alpha(R') = A_R^G \cup F[A_R^A]$;
- $\iota(R') = \iota(R)$;
- $\kappa(R') = \kappa(R)$;
- $\rho(R') = \rho(R) \cup \{\alpha(R) \setminus A_R^G\}$.

The result of the grouped table projection operator is an extended table R' where

$$I(R') = \{r' \mid \exists r \in I(R) : r'[sch(R)] \stackrel{\omega}{=} r[sch(R)] \wedge (\forall f(A^A) \in F : r'[f] \stackrel{\omega}{=} f(r[A^A]))\}. \quad (2.13)$$

The input to $\mathcal{P}$ must be a grouped extended table R, partitioned by grouping columns A^G. Furthermore, $\mathcal{P}$ is the only operator defined over a grouped extended table, as we

have not modified any other algebraic operator to support set-valued attributes[6]. Each f_i is an arithmetic expression, e.g. **Sum, Avg, Min, Max, Count**, applied to one or more set-valued columns in A_R^A and yields a single arithmetic value (possibly null). Specifically, if the value of the aggregation column a_i^A is the empty set, then in the case of **Count** the result is the single value 0, otherwise it is **Null**. If the value set is not empty then each aggregate function computes its single value in the obvious way[7]. In most cases, each f_i will simply be a single expression consisting of one of the above built-in aggregation functions, but we can also quite easily support (1) arithmetic expressions, such as **Count(X) + Count(Y)**, and (2) aggregation functions over distinct values, such as **Count(Distinct X)**. However, this formalism is insufficient to handle aggregate functions with more than one attribute parameter, which requires pair-wise correspondence of the function's input values. Full support of nested relations would be required to effectively model such functions.

CLAIM 7

We claim, without proof, that the above definitions of our algebraic forms of the grouping and aggregation operators follows the semantics of ANSI SQL, namely

$$Q = \mathcal{R}_\alpha(\mathcal{P}[A^G, F[A^A]](\mathcal{G}[A^G, A^A](R)))$$

correctly models the ANSI SQL statement

Select All $A^G, F[A^A]$ **From** $\mathcal{R}_\alpha(R)$ **Group by** A^G.

EXAMPLE 8

To illustrate the algebraic formalism for a grouped query, suppose we are given the query

```
Select D.Name, Sum(E.Salary / 52) + Sum(E.Wage * 35.0)
From Division D, Employee E
Where  E.DivName = D.Name and
       D.Location in ('Chicago', 'Toronto')
Group by D.Name
Having Sum(E.Salary) > 60000
```

which computes the weekly department payroll for all employees who are assigned to departments located in Chicago or Toronto and where the total salaries in that department

---

6 This is in contrast to the work of Ozsoyoğlu, Özsoyoğlu, and Matos [223] where set-valued attributes are supported across all algebraic operators.

7 Readers familiar with query optimization practice will realize that our definition does not correspond to the standard technique of early aggregation: pipelining aggregate computation with the grouping process itself [107]. However, at this point we are interested in defining the correct *semantics* for SQL queries, not optimization details.

must be greater than \$60.000. In terms of our formalisms for SQL semantics we express this query as

$$\pi_{All}[a_1^G, f_1](\sigma[C_h](\mathcal{P}[A^G, F[A^A]](\mathcal{G}[A^G, A^A](\sigma[C](\widehat{E} \times \widehat{D}))))) \tag{2.14}$$

where

- $\widehat{D}$ and $\widehat{E}$ are extended tables corresponding to the EMPLOYEE and DIVISION tables respectively.
- A^G are the grouping attributes; specifically a_1^G is the attribute `D.Name`.
- $\mathcal{P}$ is of degree 3 and consists of the grouping attribute `D.Name` and the two aggregation function expressions f_1 and f_2 in F. Expression f_1 computes the sum of the sums defined over the aggregation columns `E.Salary` and `E.Wage`. Expression f_2 computes the sum of `E.Salary` required for the evaluation of the `Having` clause.
- C_h represents the `Having` predicate which compares the result of applying the aggregation function f_2 (the grouped sum of `E.Salary`) to the aggregation attribute `E.Salary` to the constant value 60,000.
- $\mathcal{G}$ represents the partitioning of the join of DIVISION and EMPLOYEE over the grouping attribute `D.Name` and forming the three set-valued columns a_1^A, a_2^A, and a_3^A from the base attributes `E.Salary` (twice) and `E.Wage`, respectively;
- $C = C_{D.E} \wedge C_D$ represents the two predicates in the query's `Where` clause, the first being the join predicate and the second representing the restriction on DIVISION.

2.3.2 Query expressions

DEFINITION 19 (UNION-COMPATIBLE TABLES)
Two tables $T_1 = \langle \{a_1, a_2, \ldots, a_n\}, \iota_1, \kappa_1, \rho_1, E_1 \rangle$ and $T_2 = \langle \{b_1, b_2, \ldots, b_n\}, \iota_2, \kappa_2, \rho_2, E_2 \rangle$ are said to be *union-compatible* if and only if the domains of their corresponding real attributes are identical, that is $\text{Domain}(a_i) = \text{Domain}(b_i)$ for $1 \leq i \leq n$. We represent this correspondence by the function $b_i = \text{corr}(a_i)$.

DEFINITION 20 (UNION)
The *union* of two union-compatible extended tables S and T, written $S \cup_{All} T$ produces an extended table R' as its result with schema attributes as follows:

- $\alpha(R') = \alpha(S)$;
- $\iota(R')$ = a new tuple identifier attribute;

- $\kappa(R') = \kappa(S) \cup \kappa(T)$;
- $\rho(R') = \rho(S) \cup \rho(T) \cup \iota(S) \cup \iota(T) \cup \alpha(T)$.

Note that we have arbitrarily chosen to model the real set of attributes in R' using those real attributes from S.

The instance $I(R')$ is constructed as follows. Similarly to full outer join (see Definition 16 above) we construct two tuples s_{Null} and t_{Null} as 'placeholders' for missing attribute values. Then $I(R')$ is:

$$\begin{aligned} I(R') = \mathcal{R}\{r' \mid\ & (\exists s \in I(S) : r'[sch(S)] \stackrel{\omega}{=} s[\alpha(S)] \wedge r'[sch(T)] \stackrel{\omega}{=} t_{\text{Null}}[sch(T)]) \\ \vee\ & (\exists t \in I(T) : (\forall a \in \alpha(S) : r'[a] \stackrel{\omega}{=} t[\text{corr}(a)]) \wedge \\ & r'[sch(T)] \stackrel{\omega}{=} t[sch(T)] \wedge r'[sch(S) \setminus \alpha(S)] \stackrel{\omega}{=} s_{\text{Null}}[sch(S) \setminus \alpha(S)]) \}. \end{aligned} \tag{2.15}$$

CLAIM 8
The expression

$$Q = \mathcal{R}_\alpha(S \cup_{All} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Union All
Select * From R_α(T).
```

DEFINITION 21 (DISTINCT UNION)
The *distinct union* of two union-compatible extended tables S and T, written $S \cup_{Dist} T$ produces an extended table R' equivalent to the expression $\pi_{Dist}[\alpha(S \cup_{All} T)](S \cup_{All} T)$.

CLAIM 9
The expression

$$Q = \mathcal{R}_\alpha(S \cup_{Dist} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Union
Select * From R_α(T).
```

DEFINITION 22 (DIFFERENCE)
The *difference* of two union-compatible extended tables S and T, written $S -_{All} T$ produces an extended table R' with $sch(R') = sch(S)$. The semantics of difference are as follows. Let s_0 denote a tuple in $I(S)$ and t_0 a tuple in $I(T)$ such that $s_0[\alpha(S)] \stackrel{\omega}{=} t_0[\alpha(T)]$. Let $j \geq 1$ be the number of occurrences of tuples in $I(S)$ such that $s[\alpha(S)] \stackrel{\omega}{=} s_0[\alpha(S)]$, and let k similarly be the number of occurrences (possibly 0) of t_0 in $I(T)$. Then the number of instances of s_0 that occur in the result $I(R')$ is the maximum of $j - k$ and zero; if $j > k \geq 1$ then we select $j - k$ tuples of $I(S)$ nondeterministically.

CLAIM 10
The expression

$$Q = \mathcal{R}_\alpha(S -_{All} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Except All
Select * From R_α(T).
```

DEFINITION 23 (DISTINCT DIFFERENCE)
The *distinct difference*, or difference with duplicate elimination, of two union-compatible extended tables S and T, written $S -_{Dist} T$, produces an extended table R' equivalent to the expression $\pi_{Dist}[\alpha(S -_{All} T)](S -_{All} T)$.

CLAIM 11
The expression

$$Q = \mathcal{R}_\alpha(S -_{Dist} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Except
Select * From R_α(T).
```

DEFINITION 24 (INTERSECTION)
The *intersection* of two union-compatible extended tables S and T, written $S \cup_{All} T$ produces an extended table R' with schema:

- $\alpha(R') = \alpha(S)$;

- $\iota(R') = \iota(S)$;
- $\kappa(R') = \kappa(S) \cup \kappa(T)$;
- $\rho(R') = \rho(S) \cup \rho(T) \cup \iota(T)$.

The semantics of intersection are as follows. Let s_0 denote a tuple in $I(S)$ and t_0 a tuple in $I(T)$ such that $s_0[a] \stackrel{\omega}{=} t_0[\text{corr}(a)]$ for all $a \in \alpha(S)$. Let $j \geq 1$ be the number of occurrences of tuples in $I(S)$ such that $s[\alpha(S)] \stackrel{\omega}{=} s_0[\alpha(S)]$, and let $k \geq 1$ similarly be the number of occurrences of $t_0[\alpha(T)]$ in $I(T)$. Then the number of occurrences of s_0 in the result $I(R')$ of the intersection of these two subsets of tuples is the minimum of j and k. Let m be the absolute value of $j - k$. We construct the m tuples $r' \in I(R')$ as follows. Nondeterministically select m tuples $s_1, s_2, \ldots, s_m$ from the j occurrences matching $s_0[\alpha(S)]$ in $I(S)$. Similarly, nondeterministically select m tuples $t_1, t_2, \ldots, t_m$ from the k occurrences matching $t_0[\alpha(T)] \in I(T)$. Now construct the m tuples $r'_1, r'_2, \ldots, r'_m$ such that

$$r'_i[sch(S)] \stackrel{\omega}{=} s_i[sch(S)] \wedge r'_i[\rho(T) \cup \iota(T) \cup \kappa(T)] \stackrel{\omega}{=} t_i[\rho(T) \cup \iota(T) \cup \kappa(T)]$$

for $1 \leq i \leq m$. Constructing $I(R')$ in this manner for each set of matching tuples in S and T gives the entire result.

CLAIM 12
The expression

$$Q = \mathcal{R}_\alpha(S \cap_{All} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Intersect All
Select * From R_α(T).
```

DEFINITION 25 (DISTINCT INTERSECTION)
The *distinct intersection* of two union-compatible extended tables S and T, written $S \cap_{Dist} T$, produces an extended table R' consisting of the intersection of S and T with duplicates removed, and is equivalent to the expression $\pi_{Dist}[\alpha(S \cap_{All} T)](S \cap_{All} T)$.

CLAIM 13
The expression

$$Q = \mathcal{R}_\alpha(S \cap_{Dist} T)$$

correctly models the ANSI SQL statement

```
Select * From R_α(S)
Intersect
Select * From R_α(T).
```

2.4 Functional dependencies as constraints

Intensional data such as integrity constraints offer an important form of metadata that can be exploited during query optimization [50, 105, 114, 161, 205, 258, 260, 282, 283, 293]. Ullman [277], Fagin [87], Casanova et al. [45], Sadri and Ullman [243], and Missaoui and Godin [204] offer surveys of various classes of integrity constraints. These constraints form two major classes: inclusion dependencies and functional dependencies.

Functional dependencies are a broad class of data dependencies that have been widely studied in the relational database literature (cf. references [13, 14, 23, 45, 77, 83, 85, 86, 88, 128, 185, 206, 281]). A 'classical' formal definition of a functional dependency is as follows [45]. Consider the relation scheme $R(A)$ with attributes $A = \{a_1, \ldots, a_n\}$. Let $A_1 \subseteq A$ and $A_2 \subseteq A$ denote two subsets of A (not necessarily disjoint). Then we call the dependency

$$R : A_1 \longrightarrow A_2 \tag{2.16}$$

a functional dependency of A_2 on A_1 and say that R satisfies the functional dependency if whenever tuples $r_1, r_2 \in R$ and $r_1[A_1] = r_2[A_1]$, then $r_1[A_2] = r_2[A_2]$.

Exploiting inclusion and functional dependencies in query optimization relies upon their discovery. ANSI SQL defines several mechanisms for declaring constraints, which we now describe in some detail. The main difference between 'classical' constraint definitions, such as that for functional dependencies, and those permitted in ANSI SQL is that the classical definitions utilize two-valued logic, whereas the semantics of SQL constraints have to take into account the existence of null values.

2.4.1 Constraints in ANSI SQL

The schema definition mechanisms in ANSI SQL permit the specification of a wide variety of constraints on database instances. For example, a `Not Null` constraint on a column definition imposes the obvious restriction that no tuple can contain a null value for that attribute. Integrity constraints in ANSI SQL consist of two major classes, *column constraint definitions* and *table constraint definitions.*

A column constraint may reference either a specific domain or a **Check** clause, which defines a *search condition* that cannot be *false* (thus the predicate is true-interpreted). For example, the *check constraint definition* for column **EmpID** in EMPLOYEE could be `Check (EmpID Between 1 and 100)`. A tuple in EMPLOYEE violates this constraint if its EmpID value is not **Null** and lies outside this range. **Check** constraints on domains are identical to **Check** constraints on columns and typically specify ranges of possible values.

There are several different forms of table constraints. A **Check** clause that is specified as part of a table constraint in ANSI SQL can subsume any column or domain constraint: furthermore the condition can specify conditions that must hold between multiple attributes. Other forms of table constraints include referential integrity constraints (primary key and foreign key definitions) and **Unique** constraint definitions that define candidate keys. In each form of table constraint there is an implicit range variable referencing the table over which the constraint is defined. More general constraints, termed **Assertions**, relax this requirement and permit the specification of any SQL expression (hence range variables are explicit).

In this thesis we consider **Not Null** column constraints and two forms of table constraints. **Check** constraints on base tables in SQL2 identify conditions for columns in a table that must always evaluate to true or unknown. For example, our DIVISION table is defined as:

```
Create Table Division (
  Name ..., Location ..., ManagerID ...
  Primary Key (Name),
  Check (Location in ('Chicago', 'New York', 'Toronto')))
```

which specifies a **Check** condition on **Location**. Since this condition cannot be false, then the query

```
Select * From Division
Where  Location in
       ('Chicago', 'New York', 'Toronto') or
       Location is null
```

must return all tuples of DIVISION. What this means is that *we can add any table constraint to a query (suitably adjusted to account for null values) without changing the query result.*

The second type of table constraint we consider is a *unique specification* that identifies a primary or candidate key. Our interest in keys is because they explicitly define a functional dependency between the key columns and the other attributes in the table. There are three sources of unique specifications:

- primary key specifications,

- unique indexes, and
- unique constraints.

The semantics of primary keys are straightforward; no two rows in the table can have the same primary key value, and each column of a primary key identified by the **Primary Key** clause must be definite.

In terms of the ANSI standard, indexes are implementation-defined schema objects, outside the scope of the multiset relational model. However, both unique and nonunique indexes are ubiquitous in commercial database systems, and hence deserve consideration. Unique indexes provide an additional means of defining an integrity constraint on a base table. A unique index defined on a table $R(A)$ over a set of attributes $U \subseteq A$ offers similar properties to those of a primary key; no two rows in the table can have the same values for U. Unlike primary keys, however, attributes in U can be nullable. In this thesis we adopt the semantics of unique indexes taken in some commercial database systems such as Sybase SQL Anywhere, in which when comparing the values of U for any two rows, null values are considered equivalent (see Section 2.5). This definition mirrors the interpretation of null values with the SQL set operators (**Union**, **Intersect**, and **Except**) and the algebraic operators partition and projection discussed previously (see Section 2.3).

The **Unique** clause defines a unique constraint on a base table. As with both primary key specifications and unique indexes, a unique clause is another mechanism with which to define a candidate key. Like unique indexes the columns specified in a **Unique** constraint may contain null values; however, the ANSI standard interprets the equivalence of null values with respect to a unique constraint differently than for SQL's algebraic operators. In ANSI SQL a constraint is satisfied if it *is not known* that it is violated; therefore there can be multiple candidate keys with null values, since it is not known that one or more null values actually represent a duplicate key [72, pp. 248]. Hence any constraint predicate that evaluates to *unknown* is interpreted as *true*.

The table definition for the EMPLOYEE table:

```
Create Table Employee (
  EmpID ..., Surname ..., GivenName ...,
  Title ..., Phone ..., Salary ...,
  Wage ..., DivName ...
  Primary Key (EmpID),
  Unique (Surname, GivenName),
  Check (EmpID Between 1 and 30000),
  Check (Salary ≠ 0 Or Wage ≠ 0),
  Foreign Key (DivName) references Division)
```

defines a Check constraint on salary and hourly wage, along with the composite candidate key (Surname, GivenName), in addition to the specified primary and foreign key constraints.

2.5 SQL and functional dependencies

Because the ANSI SQL standard permits a Unique constraint over nullable attributes, null values may exist on both the left- and right-hand sides of a functional dependency in both base and derived tables. To show that such a functional dependency holds for any two tuples t_0 and t_1, we must be able to compare both the determinant and dependent values of the two tuples. Such comparisons involving null values must follow ANSI SQL semantics for three-valued logic. Using the null interpretation operator (Definition 11) and the null comparison operator $\stackrel{\omega}{=}$ (Definition 6) we formally define functional dependencies in the presence of nulls as follows:

DEFINITION 26 (FUNCTIONAL DEPENDENCY)
Consider an extended table R and sets of attributes $A_1 \subseteq sch(R)$ and $A_2 \subseteq sch(R)$ where A_1 and A_2 are not necessarily distinct. Let $I(R)$ denote a specific instance of R. Then A_1 *functionally determines* A_2 in $I(R)$ (written $A_1 \stackrel{I(R)}{\longrightarrow} A_2$) if the following condition holds:

$$\forall\, t_0, t_0' \in I(R) : t_0[A_1] \stackrel{\omega}{=} t_0'[A_1] \Longrightarrow t_0[A_2] \stackrel{\omega}{=} t_0'[A_2].$$

In other words, if the functional dependency holds and two tuples agree on the set of attributes A_1, then the two tuples must agree on the value of the attributes in A_2. Note the treatment of null values implicit in this definition: corresponding attributes in A_1 and A_2 must either agree in value, or both be Null.

Table definitions serve to define constraints (nullability, primary keys, table and column constraints) that must hold for every instance of a table. Consequently we assume that any constraint in an extended table definition automatically applies to every instance of that table. Hence we write $A_1 \longrightarrow A_2$ when $I(R)$ is clear from the context.

From Definition 26 we can now formally define a key dependency.

DEFINITION 27 (KEY DEPENDENCY)
Consider an extended table R. Let $I(R)$ denote an instance of R. Let K denote some subset of $sch(R)$. Then K is a key of R if and only if the following functional dependency holds:

$$K \longrightarrow \iota(R). \tag{2.17}$$

This formalism merely states our intuitive notion of a key: no two distinct tuples may have the same key.

2.5.1 Lax functional dependencies

It is precisely our interpretation of null values as 'special' values in each domain—and correspondingly, our use of Negri's null interpretation operator to test their satisfaction using three-valued logic—that differentiates our approach to the handling of null values with respect to functional dependencies from other schemes that introduce the notion of 'weak dependency' (cf. Section 2.5.3 below). Following convention, our definition of functional dependency, which we term *strict*, only permits *strong satisfaction* in the sense that the constraint defined by Definition 26 must evaluate to *true*. However, due to the existence of (1) nullable attributes in `Unique` constraints, (2) true-interpreted predicates formed from `Check` constraints or through the conversion of a nested query into a canonical form, and (3) the semantics of outer joins, we must also define a weaker form of functional dependency, which we term a *lax* functional dependency[8].

DEFINITION 28 (LAX FUNCTIONAL DEPENDENCY)
Consider an extended table R and sets of attributes $A_1 \subseteq sch(R)$ and $A_2 \subseteq sch(R)$ where A_1 and A_2 are not necessarily distinct. Let $I(R)$ denote a specific instance of R. Then A_1 *laxly determines* A_2 in $I(R)$ (written $A_1 \overset{I(R)}{\longmapsto} A_2$) if the following condition holds:

$$\forall\, t_0, t_0' \in I(R) : \lfloor\, t_0[A_1] = t_0'[A_1] \,\rfloor \Longrightarrow \lceil\, t_0[A_2] = t_0'[A_2] \,\rceil.$$

Unlike strict dependencies, both the antecedent and consequent expressions must be equal only for non-null determinant and dependent values, which corresponds to the classical definition of functional dependency [13]. Again we write $A_1 \longmapsto A_2$ when $I(R)$ is clear from the context. Henceforth when we use the term 'dependency' without qualification we mean *either* a strict or lax functional dependency.

By themselves, lax functional dependencies are not that interesting since they cannot guarantee anything about the relationship between their left- and right-hand sides if either side includes nullable attributes (the conditions in Definitions 26 and 28 are equivalent if every attribute in the determinant and dependent sets cannot be `Null`). However, they are worth capturing because there are circumstances in which we can convert lax dependencies into strict ones.

---

8 We use the term 'lax' to avoid any confusion with other definitions of 'weak' functional dependencies; see Section 2.5.3.

EXAMPLE 9 (LAX DEPENDENCY CONVERSION)
Consider the following query over the SUPPLY and VENDOR tables:

```
Select Distinct V.Name, V.ContactName, V.Address, S.PartID, S.Rating
From   Vendor V, Supply S
Where  V.Name like :Pattern and V.VendorID = S.VendorID
```

where :Pattern is a host variable containing the pattern for desired vendor names. From the Unique constraint declared in the definition of the VENDOR table, the attribute Name constitutes a candidate key, and thus laxly determines each of the other attributes in VENDOR. However, the false-interpreted, *null-intolerant*[9] like predicate will eliminate from the result any rows from VENDOR which have unknown names, ensuring the uniqueness of V.Name attributes. Hence V.Name and S.PartID together form a derived key dependency in the result, and we can infer that duplicate elimination is not necessary.

2.5.2 Axiom system for strict and lax dependencies

Although Definition 26 extends the equivalence relationship between two attributes to include null values, the *inference rules*, known as Armstrong's axioms [13], used to infer additional functional dependencies still hold: all that we have really done is to define an equivalence relationship between null values in each domain. However, we need to augment these inference rules to support lax functional dependencies. In this section, we describe a set of sound inference rules for a combined set of strict and lax dependencies corresponding to Definitions 26 and 28 respectively.

LEMMA 1
The following inference rules, defined over an instance $I(R)$ of an extended table R with subsets of attributes $X, Y, Z, W \subseteq sch(R)$, are sound:

---

9 A null-intolerant predicate is one which cannot evaluate to *true* if any of the predicate's operands are Null. In ANSI SQL, virtually all false-interpreted comparison predicates, Like predicates, and similar search conditions are null-intolerant. A simple example of a null-tolerant predicate is *p* is null.

X	Y	Z
3	Null	5
3	Null	3

FIGURE 2.1: An instance of table $\mathcal{R}_\alpha(R)$.

Reflexivity	FD1	If $Y \subseteq X$ then $X \longrightarrow Y$.
Augmentation	FD2A	If $X \longrightarrow Y$ and $Z \subseteq W$ then $XW \longrightarrow YZ$.
	FD2B	If $X \longmapsto Y$ and $Z \subseteq W$ then $XW \longmapsto YZ$.
Union	FD3A	If $X \longrightarrow Y$ and $X \longrightarrow Z$ then $X \longrightarrow YZ$.
	FD3B	If $X \longmapsto Y$ and $X \longmapsto Z$ then $X \longmapsto YZ$.
Strict decomposition	FD4A	If $X \longrightarrow YZ$ then $X \longrightarrow Y$ and $X \longrightarrow Z$.
Weakening	FD5	If $X \longrightarrow Y$ then $X \longmapsto Y$.
Strengthening	FD6	If $X \longmapsto Y$ and $I(R)$ is XY-definite then $X \longrightarrow Y$.
Strict transitivity	FD7A	If $X \longrightarrow Y$ and $Y \longrightarrow Z$ then $X \longrightarrow Z$.

PROOF. Omitted. □

Note that, in general, transitivity does not hold for lax dependencies in the same way that transitive relationships break down for other definitions of weak dependencies [17] [16, pp. 243]. Consider the table instance shown in Figure 2.1. It is easy to see that the strict functional dependency $X \longrightarrow Y$ and the lax functional dependency $Y \longmapsto Z$ both hold. However, note that neither $X \longrightarrow Z$ nor $X \longmapsto Z$ hold; the problem, of course, is that the definition of a lax functional dependency means that Null attributes cannot lead to a dependency violation. However, transitivity of a lax dependency over definite attributes is sound since eliminating Null determinants and dependents yields a functional dependency with the same semantics as one that is strict.

LEMMA 2 (TRANSITIVITY OF LAX DEPENDENCIES)
The inference rule:

Lax transitivity	FD7B	If $X \longmapsto Y$ and $Y \longmapsto Z$ and $I(R)$ is Y-definite then $X \longmapsto Z$.

defined over an instance $I(R)$ of an extended table R with subsets of attributes $X, Y, Z \subseteq sch(R)$, is sound.

PROOF. Consider an instance $I(R)$ of an extended table R where $XYZ \subseteq sch(R)$ and $I(R)$ is Y-definite. By contradiction, assume that $I(R) \models X \longmapsto Y$ and $Y \longmapsto Z$ but

that $X \longmapsto Z$ does not hold. If $X \not\longmapsto Z$ then there must exist at least two tuples, say r_0 and r_1 in $I(R)$, that have identical non-Null X-values but different Z-values that are not Null. However, since $X \longmapsto Y$ and Y is definite, then r_0 and r_1 must have identical Y-values. Since $Y \longmapsto Z$ holds and Y is definite, then the Z-values for r_0 and r_1 cannot both be definite and not equal; a contradiction. □

Note that Lemma 2 holds even if one of the dependencies is strict, since by inference rule FD5 a strict functional dependency implies a lax functional dependency.

Now consider the lax functional dependency $X \longmapsto YZ$ which clearly holds in the table in Figure 2.1. However, note that the lax dependency $X \longmapsto Z$ does *not* hold in that table. As with transitivity, decomposition of lax functional dependencies also requires definite dependent attributes.

LEMMA 3 (DECOMPOSITION OF LAX DEPENDENCIES)
The inference rule:

Lax decomposition	FD4B	If $X \longmapsto YZ$ and $I(R)$ is Y-definite then $X \longmapsto Z$.

defined over an instance $I(R)$ of an extended table R with subsets of attributes $X, Y, Z \subseteq sch(R)$, is sound.

PROOF. Omitted. □

THEOREM 1
The axiom system comprising inference rules FD1–FD7 is sound for strict and lax dependencies.

PROOF. Follows from Lemmas 1, 2, and 3. □

2.5.3 Previous work regarding weak dependencies

Whereas our interest in functional dependencies is solely for their exploitation in query optimization, much, if not all, of the existing literature which addresses incomplete relations with respect to functional and multivalued dependencies has centered on the related goals of database design (decomposition) with incomplete relations and the verification of constraints when null values are modified to definite ones. Our only interest in maintaining information regarding 'weak' (lax) dependencies is in anticipation of the discovery of additional constraints that render lax dependencies as strict dependencies, in spite of the nullability of the determinant or dependent attributes.

In general, in the literature there are two basic interpretations of null values which have led to slightly different approaches in defining the semantics of data dependencies over incomplete relations.

2.5.3.1 Null values as unknown

The first approach to the problem of defining the semantics of data dependencies in incomplete relations involves the substitution, or possible substitution, of a null value with a definite one. This is usually referred to as the 'value exists but is unknown' interpretation of Null, in that the null value represents some unknown quantity in the real world: i.e. if attribute X in tuple t is Null there exists a value for $t[X]$, but this value is presently unknown. Implicit in this approach is an assumption that all null values are 'independent', in that each null value in the database can be substituted with some definite value from that attribute's domain (subject to other constraints in the database). This interpretation of Null has been previously studied by Codd [66], Biskup [35], Grant [115], and Maier [193]. In general, the satisfaction of a functional dependency in this approach depends on whether or not some definite value can be substituted for a null value such that the dependency holds [113].

Vassiliou [286, 287] pioneered the study of null values with respect to dependency theory. He defined a weak dependency as a constraint having the capacity to substitute null values with some set of arbitrary non-null ones that would fail to render a given dependency patently false (the domains of all attributes are assumed known and finite). Below we reiterate Vassiliou's Proposition 1 [287. pp. 263] that defines his satisfaction criteria for functional dependencies.

PROPOSITION 1 (VASSILIOU'S AXIOMS FOR SATISFACTION OF FDS)
Let $R(U)$ be a relation scheme with $X \cup Y = U$ such that $X \cap Y = \emptyset$, and let $f : X \longrightarrow Y$ be a functional dependency in R. Let t denote a tuple in $I(R)$. Assume that $I(R) - t$ is definite, or alternatively consider all completions of $I(R) - t$ iteratively. Then f holds with respect to t iff one of the following conditions holds:

1. t is XY-definite and there exists no tuple $t' \in I(R)$ such that $t'[X] = t[X]$ and $t'[Y] \neq t[Y]$.
2. t is X-definite and X uniquely occurs in $I(R)$.
3. t is Y-definite and either no completion of $t[X]$ is in $I(R)$, or if a completion of $t[X]$ is in $I(R)$, say $t'[X]$, then $t[Y] = t'[Y]$.

f fails to hold with respect to $t \in I(R)$ iff one of the following conditions holds:

1. t is XY-definite and there exists a tuple $t' \in I(R)$ such that $t'[X] = t[X]$ and $t'[Y] \neq t[Y]$, or
2. t is Y-definite and both

(a) all completions of $t[X]$ appear in $I(R)$, and

(b) $t[Y]$ is unique among all those completions.

Aside. Since Vassiliou deals with *relations*, not multisets, this last rule means that any null substitution within $t[X]$ either cannot be permitted due to a domain constraint on X, or will result in a duplicate row in $I(R)$. This rule can also lead to an inconsistency where f may not be false for each pair of tuples $r_0, r_1 \in I(R)$ independently, but f may be false in the *whole* relation. This problem is often referred to as the *additivity problem* for functional dependencies in incomplete relations [177, 179].

Otherwise, it is *unknown* if f holds with respect to t in $I(R)$.

With these conditions, a functional dependency f strongly holds if f holds for each tuple t in $I(R)$, and weakly holds if f does not fail to hold for any t. Vassiliou went on to show that Armstrong's axioms [13] form a sound and complete set of inference axioms for functional dependencies that strongly hold.

The above work considers a single set of dependencies over a database possibly containing null values, where each dependency can either strongly or weakly hold depending on the particular instance. In a recent paper, Levene and Loizou [179] develop definitions of *strong* and *weak* functional dependencies and an axiomatization for a combined set of the two distinct types. As with Vassiliou, dependency satisfaction relies on the substitution of null values. They use the following definition, which we state here informally, to describe this substitution:

DEFINITION 29 (POSSIBLE WORLDS OF A RELATION R)
The set of all possible worlds relative to an instance $I(R)$ of a relation R, which we denote POSS($I(R)$), is

$$\text{POSS}(I(R)) = \{I(S) \mid I(S) \text{ is a relation over } R \text{ and there exists a total and onto mapping } f : I(R) \rightarrow I(S) \text{ such that } \forall t \in I(R), f(t) \text{ is complete (that is, each attribute in } f(t) \text{ is definite)}. \quad (2.18)$$

With this definition of substitution, the satisfaction of a functional dependency is defined as follows:

- Strong satisfaction: the strong dependency $f : X \longrightarrow Y$ holds in a relation $I(R)$ over R if and only if there exists at least one possible world ($\text{POSS}(I(R)) \neq \emptyset$) and for all such possible worlds $s \in \text{POSS}(I(R))$, $\forall t_0, t_1 \in s$, if $t_0[X] = t_1[X]$ then $t_0[Y] = t_1[Y]$.

Loosely speaking, this definition is less restrictive than our definition of strict functional dependency (Definition 26) since our definition *equates* two null values, which corresponds to the *null equality constraint* defined by Vassiliou [287, Definition 1].

- Weak satisfaction: the weak dependency $f : X \longmapsto Y$ holds if and only if there exists at least one possible world $s \in \text{POSS}(I(R))$ such that $\forall t_0, t_1 \in s$, if $t_0[X] = t_1[X]$ then $t_0[Y] = t_1[Y]$.

  This definition is incomparable to our definition of lax dependency, simply because we do not rely on substitution but instead omit from consideration any tuple $t \in I(R)$ that is not XY-definite.

With these definitions, Levene and Loizou go on to describe a sound and complete axiom system for the combined set of strong and weak functional dependencies. Their definitions permit the inference of strong dependencies from a mixed set of strong and weak dependencies (their inference rule *FD9*).

2.5.3.2 Null values as no information

The second approach, more simplistic than the first, is to interpret `Null` as representing *no information* [300, 302]. In essence this means avoiding any attempt at value substitution as the null value can represent 'unknown', 'undefined', 'nonexistent', or 'inapplicable' values [183].

Independently from Vassiliou[10], Lien [183][184, Section 4.1] considered *multivalued dependencies with null values* and *functional dependencies with null values*, which he abbreviated NMVDs and NFDs respectively. An NFD $f : X \longrightarrow Y$ is satisfied if

$$\forall r_0, r_1 \in I(R) : \lfloor r_0[X] = r_1[X] \rfloor \Longrightarrow r_0[Y] \stackrel{\omega}{=} r_1[Y]. \tag{2.19}$$

With this definition Lien showed that his inference rules

Reflexivity	F1	If $Y \subseteq X$ then $X \longrightarrow Y$.
Augmentation	F2	If $X \longrightarrow Y$ and $Z \subseteq W$ then $XW \longrightarrow YZ$.
Union	F3	If $X \longrightarrow Y$ and $X \longrightarrow Z$ then $X \longrightarrow YZ$.
Decomposition	F4	If $X \longrightarrow YZ$ then $X \longrightarrow Y$ and $X \longrightarrow Z$.

were all sound with respect to NFDs, but notably transitivity (or pseudotransitivity) was not (see Figure 2.1).

10 There are no corresponding references from Lien's work [183, 184] to either of Vassiliou's early papers [286, 287] on null values in relational databases, or vice-versa.

Atzeni and Morfuni [17] also defined NFDs on the basis of definite determinants (2.19) and the 'no information' interpretation of null values. In this short paper they introduced a modified version of Armstrong's transitivity axiom, which they termed *null-transitivity*, that relied on definite dependent attributes. Atzeni and Morfuni go on to show that their null-transitivity axiom, which is quite similar to the lax-transitivity inference rules in Lemma 2 above, and Lien's inference rules F1 through F4 form a sound and complete set of inference rules for NFDs; a more detailed version of the proof can be found in reference [18].

Related work on functional dependencies over incomplete databases have been addressed by Maier [193, pp. 377–86], Imielinski and Lipski [134], Abiteboul et al. [4, pp. 497–8], Zaniolo [300, 302], Libkin [182], Levene [176], Levene and Loizou [177, 178, 180], and Atzeni and De Antonellis [16, pp. 239–48]).

2.6 Overview of query processing

Several excellent references regarding the complete framework of query optimization exist in the literature [51, 71, 107, 143, 193, 203, 278, 299]. We define *centralized*, as opposed to distributed, query optimization as the following sequence of steps:

1. Find an internal representation into which user queries[11] can be cast. This representation must typically be richer than either classical relational calculus or algebra, due to language extensions such as scalar and aggregate functions, outer joins, duplicate rows, and null values. One example of an internal representation is the *Query Graph Model* used in STARBURST [124].

2. Apply rewriting transformations to the query to *standardize* it by rewriting it into a *canonical form*, *simplify* it by eliminating redundancy, and improve (*ameliorate*) it if possible. This process also combines the query with the view definitions of the schema [71, 186, 267]. The point is that the performance of a query should not depend on how the query was cast originally by the user [71, pp. 459]. Many query languages, including SQL, allow queries to be expressed in several equivalent, though syntactically different, forms.

3. Generate access plans for each transformed query by mapping each of them into sequences of lower-level operations, and augment these plans with information about the physical characteristics of the database.

---

11 In this context, the term 'query' not only refers to retrieval operations, but also (and perhaps more importantly) to database updates.

4. Choose the best access plan alternative, depending on the cost of each plan and the performance goals of the optimizer (resource or response time minimization).

5. Generate a detailed query execution plan that captures all necessary information to execute the plan.

6. At run time, execute the detailed plan.

2.6.1 Internal representation

The first stage of query processing is the conversion of the syntactic statement into an internal representation that constitutes the statement's canonical form—where extraneous syntax has been eliminated and view definitions have been expanded—that is suitable for analysis by an optimizer. A typical representation for an SQL request, and the one used herein, is an *algebraic expression tree* [193, pp. 296–314] [278]. One can think of this expression tree as an annotated parse tree that models the semantics of the query using unary (projection, restriction) and binary (inner join, outer join) operators with structures representing base tables at the leaves.

EXAMPLE 10
Consider the query

```
Select P.PartID, P.Description, S.SupplyCode, V.VendorID, V.Name
From   Parts P Left Outer Join
               (Supply S Join Vendor V On ( S.VendorID = V.VendorID )
               On ( P.PartID = S.PartID and V.Address like '%Canada%' )
Where  Exists  ( Select *
               From    Quote Q
               Where   Q.PartID = P.PartID and
                       Q.Date ≥ '1993-10-01' )
```

which retrieves those parts that have received quotes at any time since 1 October 1993, along with the vendors of any of those parts that have Canadian addresses. Figure 2.3 illustrates a straightforward mapping of this query into a relational algebra tree. Note that the operators used for the nested subquery are distinct from the operators that represent the main query block.

The algebraic expression tree is a restricted form of an acyclic[12] directed graph. Vertices in the tree represent unary or binary algebraic operators and have one outgoing edge

12 We remind the reader that in this thesis we restrict the class of queries considered to nonrecursive queries.

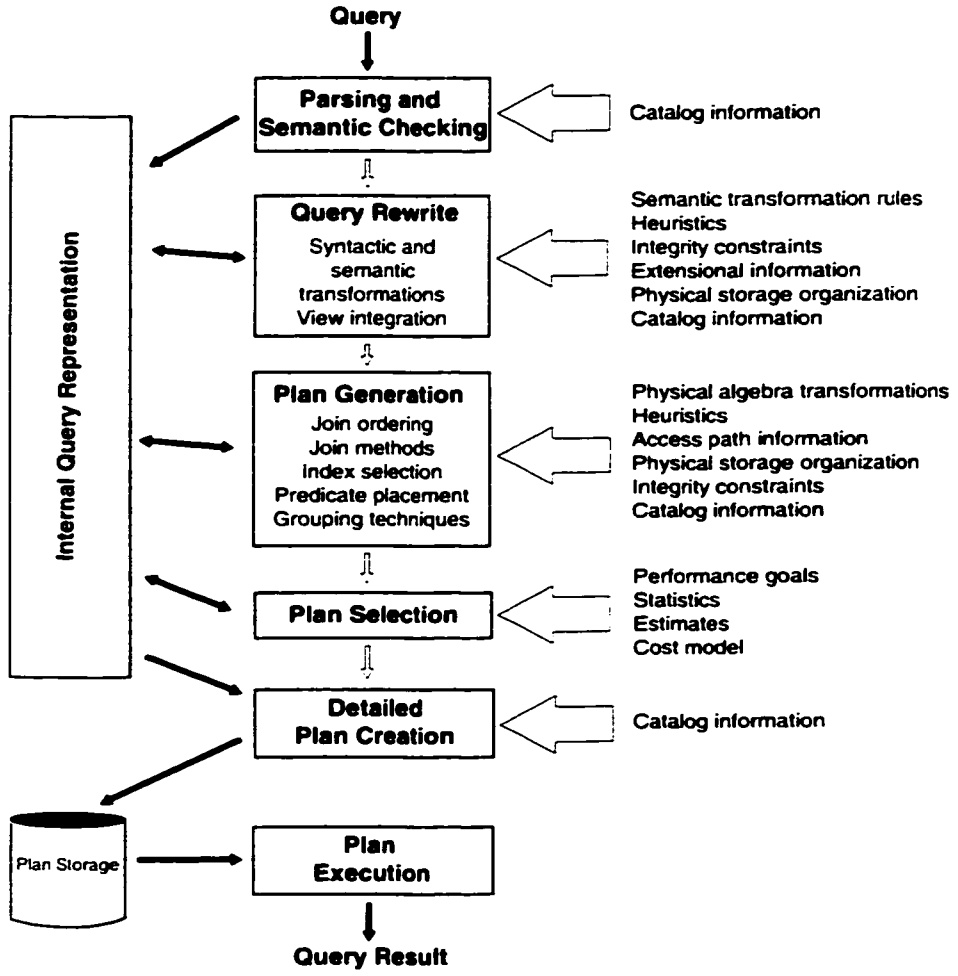


FIGURE 2.2: Phases of query processing. For simplicity, each phase is shown as a independent step; however, some processing overlap is inevitable, especially in limiting the number of alternative execution strategies generated in the query rewrite and plan generation. Inputs to each phase are shown on the right.

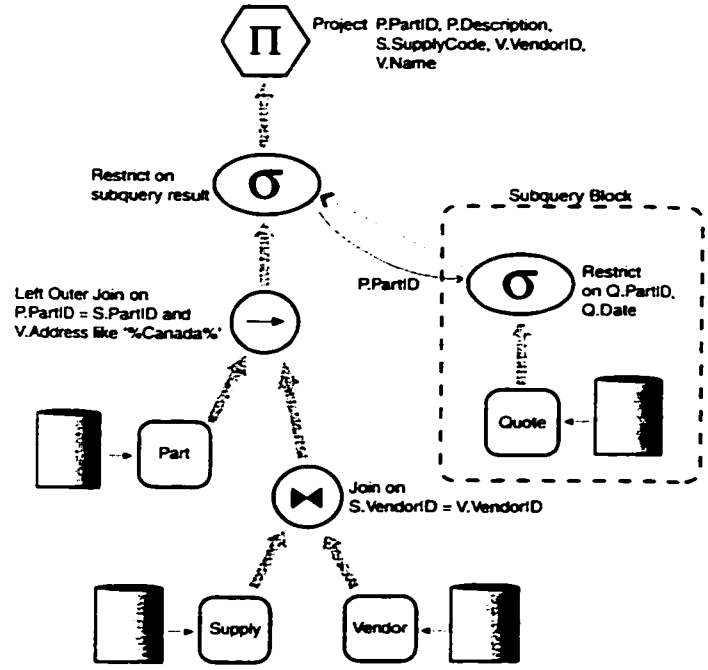


FIGURE 2.3: Example relational algebra tree for the query in Example 10.

and either one (for the unary operators projection and restriction) or two (for the binary operators join, outer join, set union, etc.) incoming edges. The directed edges in the graph represent data flowing up the tree from the leaves to the root; the outgoing edge at the tree's root represents tuples returned as part of the query's result. Note that a unary operator can appear anywhere in the tree. For example, an equivalent form of the expression tree in Figure 2.3 is one where the `Exists` predicate is placed immediately above the node representing the `Part` base table. Placing the subquery at that point corresponds to a naive predicate pushdown approach [263] and is possible because the range of the subquery only consists of the single attribute `PartID` from the PART table.

Each node in the tree is typically *annotated* in that the output edge of each node is labelled with the attributes that are returned as part of that tuple stream, along with their data types. View information is often retained, even if the views are completely merged into the query (see Section 2.6.2.2), because subsequent operations—for example, an `Update ... Where Current of Cursor` statement—can (and must) refer to the original objects specified in the query.

Subqueries in a **Where** or **Having** clause are modelled by constructing a separate, independent expression tree, but are connected through an alternate form of edge to a restriction node in the subquery's parent query block[13]. In Figure 2.3, the nested query block representing the **Exists** predicate in the example does not contain a projection since a projection is irrelevant in the context of an **Exists** predicate.

In several commercial systems, including DB2, SYBASE SQL Anywhere, and SYBASE IQ, algebraic expression trees form the basis for preliminary query optimization analysis and rewrite optimization [90, 187, 230] though their implementations may differ somewhat from the expression trees described herein. For example, STARBURST [121, 187] models queries using *Logical Plan Operators* (LOLEPOPs) that represent a richer set of algebraic operators than that described here. Like STARBURST, Graefe's Volcano execution model [108, 110, 112] also represents query plans using *executable algebraic operators,* but permits the operators to be organized into any tree structure so long as it adheres to Volcano's demand-driven data flow paradigm. Volcano's extensible architecture permits one to implement any algebraic operation that supports Volcano's three standardized iteration operations, namely **Open**, **Next**, and **Close**. Hence specialized algebraic operations such as Dayal's [74] Generalized Join or existence-semijoin can constitute portions of the expression tree. We do not consider these types of executable algebraic operations in this thesis, though the techniques presented here could be readily adapted to encompass these specialized operators.

In the following sections we provide a brief overview of the optimization phases of query rewrite, plan generation, and plan selection, beginning with query rewrite.

2.6.2 Query rewrite optimization

Query rewriting, often termed *semantic query optimization*, is the process of generating semantically equivalent queries from the original to give the optimizer a wider range of access plans from which to choose. Often, but not always, the rewritten query will itself be expressible in SQL. More complex transformations, on the other hand, may involve the use of specialized algebraic operators, such as Dayal's existence-semijoin mentioned above, or they may involve system-generated elements, such as the generation of row-identifiers, necessary to retain semantic equivalence. In addition to algebraic manipulations and operator re-sequencing, semantic query optimization techniques often exploit

---

13 In this thesis we do not consider subqueries that occur in a projection list, supported in some commercial systems such as SYBASE SQL Anywhere.

any available metadata, such as domain and integrity constraints, to generate semantically equivalent requests.

Equivalent queries may differ greatly in execution performance; a difficult problem is how to determine if a particular semantically equivalent query is 'promising'. A brute force approach is to generate *all* possible semantically equivalent queries, and estimate the performance of each using the optimizer's cost model. Several authors [49, 50, 160, 258] claim that in many cases generating an exhaustive list of equivalent queries is warranted. Jarke and Koch [143] more realistically state that the success of semantic query optimization depends on the efficient selection of the many possible transformation that the optimizer might generate. This is especially true for ad-hoc queries, where the cost of optimization directly affects the database user [255].

The tradeoff in query rewrite optimization, then, is expanding the search space of possible execution strategies versus the additional optimization cost of finding equivalent expressions. Many query rewrite implementations rely on heuristics to 'prune' the list of equivalent expressions to a reasonable number. For example, IBM s DB2 attempts to rewrite nested queries as joins whenever possible, even though they may introduce an expensive duplicate elimination step [230]. The idea is to simplify the query into a canonical form using joins, to rely on join strategy enumeration to select the most efficient access plan. With other transformations, such as lazy and eager aggregation [294, 296] or the use of magic sets [209, 210] the set of tradeoffs is not so clear, and the rewritten query may result in a much poorer execution strategy, sometimes by one or two orders of magnitude [294]. Hence cost-based selection of rewritten alternatives is necessary [253].

Semantic query optimization techniques can be classified into two general categories: simple transformations that deal primarily with the addition or removal of predicates, and more complex algebraic transformations that can result in a query significantly different in overall structure from the original. We next briefly outline various approaches in both these categories.

2.6.2.1 Predicate inference and subsumption

Chakravarthy, Grant, and Minker [50] categorized five types of semantic transformations that employed various types of integrity constraints to generate semantically equivalent queries. Their categorization, originally used to categorize transformations in nonrecursive deductive databases, is still useful as a classification for simple semantic query optimization techniques in relational databases. We have augmented their classification with additional techniques from Jarke and Koch [142] and King [160, 161] to arrive at the following seven types of simple rewrite transformations:

Literal Enhancement. The idea behind literal enhancement [142] is straightforward: a query's evaluable predicates may be made more powerful by substituting more restrictive clauses, which may be inferred from any relevant integrity constraints. For example, suppose that a query includes attributes a_R^1 and a_R^2, such that $a_R^1 > 100$ and $a_R^2 = 4$. If the integrity constraints imply that $(a_R^2 = 4) \Longrightarrow a_R^1 > 400$ then we can replace the clause $a_R^1 > 100$ with $a_R^1 > 400$. Depending on how the database is organized such a transformation may lead to a more efficient access plan. For example, if a_R^1 is an indexed attribute, then we may retrieve fewer tuples with $a_R^1 > 400$ than with $a_R^1 > 100$, but the savings are dependent on the distribution of the values of a^1 in relation R. Note that if one predicate is subsumed (possibly transitively) by another, then that predicate can simply be eliminated from the query without altering the result.

Literal Elimination. If an integrity constraint can be used to eliminate a literal clause in the query, we may be able to eliminate a join operation as well. To do so would necessitate that the relation being dropped from the query does not contribute any attributes in the result. King [161], Sagiv [244], Xu [293], Shenoy and Ozsoyoğlu [257, 258], Missaoui and Godin [205] and Sun and Yu [269] term this heuristic *join elimination.*

Outer joins provide another possible context for join elimination. If the query specifies that only `Distinct` elements of a preserved table are desired in the result, then the outer join is unnecessary since (1) the semantics of a left- or right-outer join are that every preserved row is a candidate row in the final result and (2) any additional (duplicate) preserved tuples generated by a successful outer join condition will be eliminated by the projection [33].

EXAMPLE 11 (OUTER JOIN ELIMINATION)
Consider the query

```
Select Distinct P.Description, P.Cost
From   Part P Left Outer Join Supply S On (P.PartID = S.PartID)
Where  P.Status = 'InStock'.
```

This query is equivalent to

```
Select Distinct P.Description, P.Cost
From   Part P
Where  P.Status = 'InStock'
```

as the query's characteristics match the algebraic identity [33]

$$\mathcal{R}_\alpha(\pi_{Dist}[A_1^R, \ldots, A_m^R](R \xrightarrow{p} S)) \equiv \mathcal{R}_\alpha(\pi_{Dist}[A_1^R, \ldots, A_m^R](R)). \tag{2.20}$$

Restriction Introduction. The idea behind this heuristic is to reduce the number of tuples that require retrieval by introducing additional (conjunctive) predicates which the query optimizer can exploit as matching index predicates. This technique is also referred to as *scan reduction* by King [160] and Shenoy and Ozsoyoğlu [258].

Generating the transitive closure of any equality conditions specified in the original query is one way of introducing additional predicates [165, 221]. Care must be taken to ensure that the query optimizer takes into account the fact that the additional predicates are not independent from the others and are redundant in the sense that they do not affect the query's overall result. Hence the selectivities of these redundant predicates must not be 'double counted'. Integrity constraints provide another source of additional predicates. though as mentioned previously **Check** constraints in ANSI SQL must be suitably modified to take into account the existence of null values.

Restriction introduction also encompasses the technique of *predicate move-around* [181] which generalizes classical predicate pushdown [263] by utilizing functional dependencies to move predicates up, down, and 'sideways' in an expression tree.

Join Introduction. Here, the heuristic attempts to reduce the number of tuples involved overall by introducing another relation into the query that contributes no attributes to the result. If the new relation's relative size is substantially smaller than the other relation(s) involved, executing the join may be less costly than proceeding with the original query. Chakravarthy, Grant, and Minker [50] call the technique *literal introduction* since a predicate must be introduced into the query to represent the join.

Index Introduction. Index introduction [161] tries to use an integrity constraint that refers to both a query-restricted attribute, and another attribute in the same relation that is indexed. With this transformation the optimizer can reduce the query cost from a possible sequential scan to a series of probes using the index. If the index is *clustered* then the final cost will be further reduced. Note the linking of this heuristic to the physical implementation of the supporting data structure: it is not clear that query transformations can be made entirely independent from the choice of the underlying physical system.

Result by Transformations. This approach, by Chakravarthy et al. [50], is a hybrid of the heuristics discussed above. The idea is as follows. The set of integrity constraints for the database may include implication constraints, such as 'Chicago ships only red parts'. A query which asks 'What color parts are shipped from Chicago?' may then be answered solely on the knowledge contained within the constraints. In this case, no database access is required.

Another situation may involve referential integrity constraints. It may be possible to determine that if the database contains a tuple, or set of tuples, meeting certain constraints, then the answer to the query *must* correspond to the existence, or non-existence, of a particular tuple in the database. Although this result must be verified by a lookup to the actual database, such a lookup is probably much preferable to executing the original query.

Result by Contradiction. This method is not a heuristic *per se*. During the query transformation stage we may arrive at a contradiction between the integrity constraints of the database and the query predicates (though in general this problem is undecidable). Such a contradiction implies an empty result, and therefore we require no database access to answer the query.

2.6.2.2 Algebraic transformations

View expansion and merging. Typically, an initial phase of algebraic rewriting involves *view expansion* and *view merging*. In view expansion, any views referenced in a **Select** block are expanded from their definition stored in the database catalog. View merging attempts, where possible, to *merge* the view directly into the query so as to standardize the query's internal representation in a canonical form that minimizes any differences between a query that references a view and one that directly references the view's underlying base tables [71, 186].

EXAMPLE 12
Consider the query

```
Select C.Name, C.Address
From   Canadian-Suppliers C, Supplier-Summary S
Where  C.VendorID = S.VendorID and
       S.AverageCost > 50.0
```

which retrieves those Canadian vendors who, on average, supply relatively expensive parts. Suppose that the view CANADIAN-SUPPLIERS is defined as

```
Select V.VendorID, V.Name, V.Address
From   Vendor V
Where  V.Address like '%Canada%'
```

and the grouped view SUPPLIER-SUMMARY is defined as

```
Select S.VendorID, Count(*), Avg(P.Cost) as AverageCost
From   Parts P Join Supply S On ( P.PartID = S.PartID)
Group by S.VendorID.
```

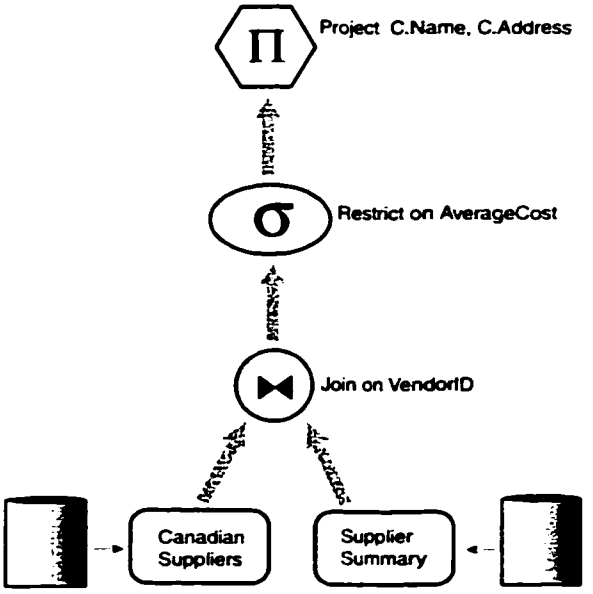


FIGURE 2.4: An expression tree embodying a syntactic mapping for the query in Example 12.

Figure 2.4 illustrates a syntactic mapping of this query into a relational algebra expression tree.

During the view expansion step, the view definitions stored in the system's catalog replace the view references in the original query (see Figure 2.5). For the query in Example 12, the expression tree now contains three projection nodes that correspond to the three `Select` blocks present in the original query and the two referenced views. A critical part of the view expansion process is keeping track of the aliases and correlation names used in the query or any referenced view. For example, both the query and one or more referenced views may refer to the same or different schema objects by the same name; during view expansion care must be taken not to confuse the different instances of the referenced objects.

Once the views have been expanded, a subsequent step is to merge (where possible) the view definition with its referencing query block (see Figure 2.6). The goal of this rewriting is to produce a tree in canonical form with superfluous projection operations

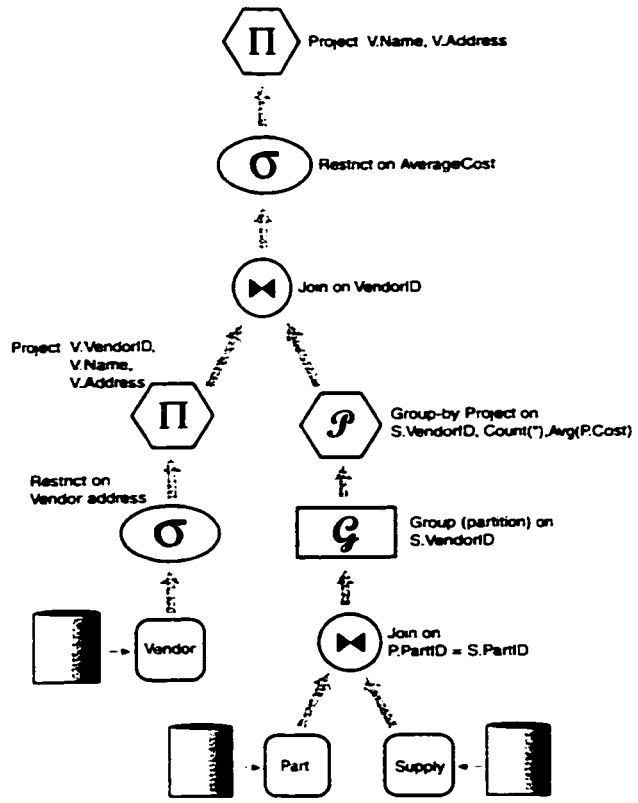


FIGURE 2.5: An expression tree containing expanded view definitions for the query in Example 12.

removed. As mentioned previously, view reference information is often kept as 'annotations' to the expression tree in case subsequent cursor operations refer to a view.

In most commercial systems an initial rewriting is restricted to select-project-join views (often termed SPJ-expressions)[14]. Views that contain **Distinct**, **Group by**, or any of the set operators **Union**, **Intersect**, or **Except** remain as is until other more complex rewritings are performed (see below). In general, SPJ views can be merged because their algebraic operations (restriction, projection, join, and Cartesian product) both com-

14 Other rewritings of more complex expressions are possible; for example, several systems utilize magic sets rewriting [209, 210, 253, 254, 278] of grouping, aggregation, and distinct projection operations. These other semantic optimizations usually take place after view expansion and merging.

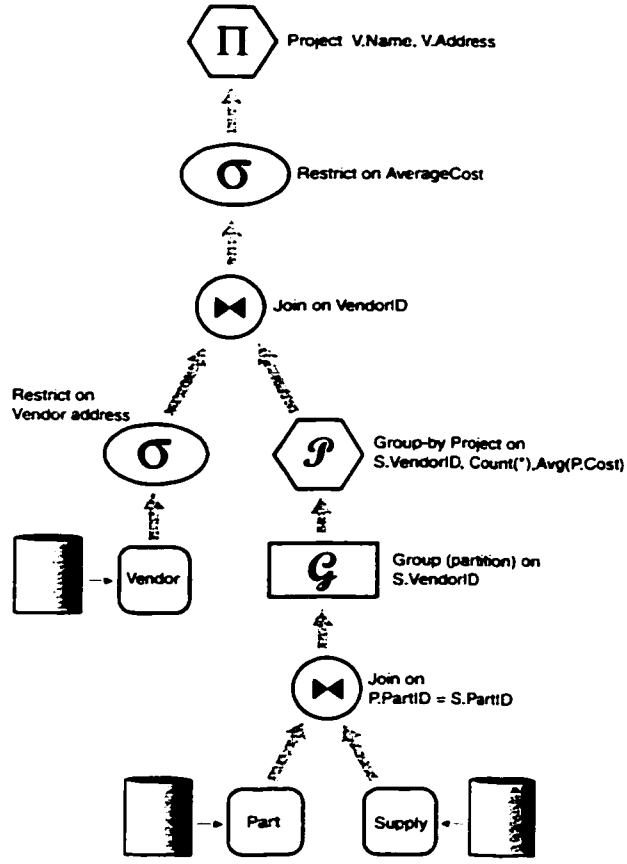


FIGURE 2.6: An expression tree containing expanded view definitions, with one merged SPJ view, for the query in Example 12.

mute and associate with the expressions in the referencing query block (cf. Maier [193, pp. 302–4] and Ullman [278]). In particular, it is the following algebraic identity [278, pp. 665]

$$\mathcal{R}_\alpha(\pi_{All}[A_R^1, \ldots, A_R^n](\sigma[C](e))) \tag{2.21}$$

is equivalent to

$$\mathcal{R}_\alpha(\pi_{All}[A_R^1 \ldots, A_R^n](\sigma[C](\pi_{All}[A_R^1, \ldots, A_R^n](e)))),$$

where e is any algebraic expression, that enables the merging of SPJ views.

Many systems include the various outer join operators as part of the class of operations that constitute SPJ expressions. Well-known axioms for the associativity and commutativity of outer joins have been previously published (cf. Galindo-Legaria and Rosenthal [98]). However, almost without exception these axioms fail to consider the impact of an outer join on the projection operator. Identity (2.21) above, originally defined for classical relational algebra, fails to hold when e can contain any form of outer join. The problem lies with the semantics of outer join which generates an all-`Null` row for the null-supplying side should the join condition not evaluate to `True` for at least one preserved row.

EXAMPLE 13 (PROJECTION AND OUTER JOIN)
Consider the query

```
Select P.PartID, P.Description, V.Rating'
From   Part P Left Outer Join
            ( Select S.PartID, f( S.Rating ) as Rating' From Supply S ) as V
                 On ( P.PartID = V.PartID )
```

which will generate a `Null` value for `Rating'` for a part that is not supplied by any supplier. This query is *not* equivalent to the rewritten query

```
Select P.PartID, P.Description, f( S.Rating )
From   Part P Left Outer Join Supply S
          On ( P.PartID = S.PartID )
```

if the scalar function f can evaluate to a definite value when its argument is `Null`. ANSI SQL functions that have this property include `Nullif`, `Case` and `Coalesce`. More formally,

$$\mathcal{R}_\alpha(\pi_{All}[f(A_T^m)](R \bowtie (\pi_{All}[f(A_T^m)](S \xrightarrow{C_1} T)))) \tag{2.22}$$

is not equivalent to

$$\mathcal{R}_\alpha(\pi_{All}[f(A_T^m)](R \bowtie (S \xrightarrow{C_1} T)))$$

if $f()$ can evaluate to a definite value when any of its arguments are `Null`.

More complex algebraic transformations, such as the rewriting of nested queries as joins [157], involve detailed analysis of the query's semantics, any integrity constraints imposed by the schema, and the application of axioms that hold for the various algebraic operations. It is beyond the scope of this thesis to exhaustively document the rewrite transformations that have been proposed. Herein we present a sampling of semantic transformations that have appeared in the literature, with a focus towards their exploitation of functional dependencies (if any).

Elimination of unnecessary Distinct processing. Since duplicate elimination is typically implemented through either sorting or hashing the entire input, it pays to eliminate unnecessary duplicate elimination whenever possible [228, 230, 281]. Because duplicate elimination is unnecessary when the projection contains a key, the sufficient condition to avoid a redundant sort is to discover if the derived table given by the query contains a key. This is done by exploiting schema constraints (e.g. primary key declarations) and utilizing functional dependencies implied by conjunctive conditions in the query's **Where** clause.

Transformations that exploit associativity and commutativity of operators. Join order enumeration relies on the associativity of Cartesian product and inner joins and the commutativity of restriction with both these operators to arrive at an optimal access plan. However, one can transform a query during query rewriting by exploiting axioms that hold for each specific operator. For example, it is common for an optimizer to rewrite an outer join as an inner join when there exists a conjunctive, *null-intolerant* predicate on a null-supplying table in a **Where** or **Having** clause [94, 98]. Galindo-Legaria [98] offers additional outer join transformations that can assist an optimizer by giving it more flexibility in choosing the query's join strategy. To this end, various researchers [31, 74, 235, 237] have proposed a *generalized join* operator that is 're-orderable'; input queries are then recast using this generalized join operator, whose semantics are fully understood by the rest of the optimizer.

Subquery unnesting and magic sets. Kim [157, 158] originally suggested rewriting correlated, nested queries as joins to avoid nested-loop execution strategies; his desired 'canonical form' was $n - 1$ joins for a query over n relations. Subsequently, several researchers corrected and extended Kim's work, particularly in the aspects of grouping and aggregation [47, 74, 101, 155, 188, 212, 213, 228, 230, 288]. Pirahesh, Hellerstein, and Hasan [230] document the implementation of these transformations in STARBURST. As with unnecessary duplicate elimination, several of the rewriting techniques for nested queries rely on the discovery of derived key dependencies, exploiting any functional dependencies that can be inferred from query predicates.

STARBURST and its production version, DB2 Common Server, implement these transformations using a rule-based implementation where the transformation is expressed as a condition-action pair [231]. Magic set optimization techniques [209, 210, 253, 254] are more complex methods to unnest subqueries that contain grouping and aggregation by first materializing intermediate results that are subsequently joined with components of the original query to produce an equivalent result.

Common subexpression elimination. Hall [122] proposed simplifying common subexpressions in the PRTV system to eliminate the unnecessary query processing cost of evaluating

redundant expressions. Jarke [144] pursues this idea in the context of multiple query optimization; Aho et al. [7] and Sagiv [244] exploit simplification rules for conjunctive queries to minimize the number of rows in tableaux, thereby minimizing the number of joins.

Eager and lazy aggregation. Yan and Larson [294–296], Chaudhuri and Shim [54, 55, 57], and Gupta et al. [117] have independently studied the problem of group-by pullup/pushdown: that is, rewrite transformations that 'pull' a group-by operation past a join in an algebraic expression tree, or its converse, group-by pushdown. In both cases, the optimization is based upon the discovery of derived key dependencies; this discovery utilizes declared key dependencies, functional dependencies inferred from predicates, and other schema constraints. This is similar to the situation in discovering unnecessary duplicate elimination for SPJ queries, but made more complex due to the introduction of the group-by and aggregation operators.

A difficult problem with group-by pullup/pushdown is that it can exponentially increase the size of the optimization problem. Moreover, as Yan has shown [294], not all of the various possible rewritings for a given query may offer improved performance, which Chaudhuri and Shim [55] claim cannot be analyzed by comparing execution plan subtrees in isolation. One reason for this is that the placement of a `Group by` node can affect the plan's *interesting orders* [247, 261] and can thus affect the optimization and performance of the other plan operators. These complexities have led to research on cost-based comparison of rewrite alternatives [55, 57, 253] and/or some gross restrictions on the strategy space considered [55, 57].

Materialized views. Adiba and Lindsay [5, 6] originally proposed the use of materialized views to speed query processing by storing precomputed intermediate results redundantly. Semantic query optimization involving such views involves rewriting portions of the query to reference a materialized view, rather than one or more base tables [52, 53, 174, 175, 297]. Larson and Yang [175] separate the complexities of rewriting from optimization; that is, one should only consider rewritten queries that are semantically equivalent to the original. The main aspect of this semantic equivalence is *query containment* [8, 139, 145, 234, 245] which Larson and Yang specify as the conditions of *tuple coverage*, *tuple selectability*, and *attribute coverage*. Whether or not these conditions hold for one or more materialized views is in general a PSPACE-complete problem, since it involves the analysis of quantified Boolean expressions [102, pp. 171–2]. Hence the consideration of strategies that involve materialized views significantly increases the overall complexity of the optimization problem [1, 52, 53], which, as in the case of multiple query optimization (cf. references [9, 38, 248, 251, 252, 259]) requires common subexpression analysis [10, 59, 89, 144, 227].

Functional dependencies are critical to the optimization of queries over materialized views [26]. For example, consider a materialized view definition consisting of the grouped query given in Example 1:

```
Select P.PartID, P.Description, Avg(Q.UnitPrice)
From   Part P, Quote Q
Where  Q.PartID = P.PartID
Group by P.PartID, P.Description
```

Now suppose a query over the database is similar in structure to the above, but fails to reference the column `P.Description`. Such a query is answerable using the view alone, since (1) the view covers all the necessary attributes and (2) the functional dependency `P.PartID` $\longrightarrow$ `P.Description` holds. In the converse case—where the query refers to additional attributes of the PART table that are not contained in the view—the existence of the functional dependency (in this case, a key dependency) enables a *back join* [175] to the PART table to retrieve any attributes missing in the view [26].

In a data warehouse environment, the existence of snapshots [6] that summarize multiple base tables means that not all functional dependencies are derivable from schema constraints and query or view predicates alone. ORACLE 8i, which contains extensive materialized view support [26], recognizes this situation and permits a DBA to *explicitly* declare the existence of functional dependencies through the specification of `Dimension` objects. 'Dimensions' were originally introduced to identify dependencies across time-series data types (ie. `date` $\longrightarrow$ `month`), but they can be used to declare dependencies between attributes that ORACLE's optimizer can try to utilize when determining the attribute coverage (in ORACLE terminology, *data sufficiency*) of a materialized view. No mechanism exists in ORACLE to enforce these functional dependency constraints, although a class of declared inter-relational dependencies—specifically an inclusion dependency—could be enforced through a referential integrity constraint.

In other contexts, materialized views have been implemented as 'indexes' whose existence can be exploited to reduce the number of page accesses for a given access plan. Various implementations of materialized views have been described as *view indexes* [239, 241, 242], *join indexes* [190, 220, 238, 279, 280, 292], both in relational and object-oriented contexts, *view caches* [240], *index caches* [249, 250], and *access support relations* [150–154].

An important related problem to the exploitation of materialized views in query optimization is the problem of *view maintenance*—that is, strategies to keep the derived result and its base tables synchronized in the face of updates [37, 39, 40, 67, 68, 118–120, 123, 189, 240, 274]. In their survey paper from 1995, Gupta and Mumick [119] state that comprehensive view maintenance techniques have yet be developed that fully exploit functional dependencies and other forms of data constraints. The support for ex-

plicit declaration of functional dependencies in ORACLE 8i [26], through its use of dimensions, is a step in this direction. ORACLE 8i also offers the DBA both incremental and bulk maintenance policies, along with specific controls so that the DBA can specify whether or not back-joins to a view's underlying base tables are permitted in any resulting access plan. This feature is important as the database instances represented by the view and the base table(s) may be different: the base tables may have been updated since the materialized view was last refreshed.

2.6.3 Plan generation

The query's original internal representation (the algebraic expression tree) generally reflects the original input—that is, the expression tree is an arbitrary shape. Assuming the query rewrite phase has converted each semantically equivalent query into an intermediate representation (see Figure 2.2), the next step is to generate an optimal execution strategy for each of them. Plan generation involves both the sequencing of operations and the choice of their physical implementation (e.g. sort-merge join, hybrid-hash join, nested-loop join, etc.). Hence it is typical for an optimizer to transform the algebraic expression tree into one that represents the physical operations themselves. Conceptually, the intermediate representation commonly used for execution strategies is an *executable algebraic expression* [51, 226] where each leaf node is a relation in the database, and a nonleaf node is an intermediate relation produced by a physical algebraic operation.

Moreover, a transformation from the original 'bushy' algebraic representation to one more amenable to optimization is often necessary. It is typical for query optimizers—such as SYSTEM R [247]—to consider only *left-deep processing trees* as the solution space for join strategy enumeration. For SPJ queries a left-deep processing tree is one where the right child of any join can only be a base table. For more complex queries, a left-deep tree means that the right child of any binary operator cannot be a join—though it could be the (materialized) result of a view containing `Union`, `Group by`, or aggregation. Left-deep trees are desirable because (1) such a tree reduces the need to materialize intermediate results and (2) the space of 'bushy' plans is considerably larger, and hence more expensive to search [284]. Ono and Lohman [221] show that the complexity of optimization also depends on the number of *feasible joins*. For example, for a *linear* query consisting of n tables, a dynamic programming enumeration algorithm must consider $(n-1)^2$ feasible joins when considering left-deep trees, and $(n^3-n)/6$ when considering bushy trees. However, for *star* queries or arbitrary queries containing Cartesian products, the complexity is $O(3^n)$ or $O(4^n)$, depending upon the implementation of the join enumeration algorithm [229, 284].

The objective of plan generation, then, is to [143]:

- generate all reasonable logical access plans corresponding to the desired solution space for evaluating the query, and
- augment each access plan with optimization details, such as join methods, physical access paths, and database statistics.

Once the plan generator creates this set of access plans, the plan selection phase will choose one access plan as the 'optimal' plan, using the optimizer's cost model. Excellent surveys of join strategy optimization can be found in references [107, 203, 221, 229, 266, 284].

Generating an optimal strategy is an NP-hard problem [58, 64, 129, 221, 226, 266]; to discover all possible strategies requires an exhaustive search. In the worst case, a completely-connected join graph for a query with n relations has $n!$ alternative strategies with left-deep trees, and $\frac{(2n-2)!}{(n-1)!}$ alternatives when considering bushy processing trees [229]. Consequently, optimizers often use heuristics [221, 224, 225, 266] to reduce the number of strategies that the plan selection phase must consider. A common heuristic used in most commercial optimizers is to restrict the strategy space by performing unary operations (particularly restriction) first, thus reducing the size of intermediate results [263, 278]. Another common optimization heuristic, and one used by STARBURST, is to defer the evaluation of any Cartesian products [208] to as late in the strategy as possible [221].

There are several ways to perform join enumeration; a recent paper by Steinbrunn, Moerkotte, and Kemper [266] classifies them into four categories: *randomized algorithms, genetic algorithms, deterministic algorithms,* and *hybrid algorithms.* Randomized algorithms view solutions as points in a solution space, and the algorithms randomly 'walk' through this solution space from one point to another using a pre-defined set of *moves.* Galindo-Legaria, Pellenkoft, and Kersten [96, 99] have recently proposed a probabilistic approach to optimization that randomly 'probes' the space of all valid join strategies in an attempt to quickly find a 'reasonable' plan, whose cost can then be used to limit a deterministic search of the entire strategy space. Other well-known examples of randomized approaches include iterative improvement [138, 270, 271] and simulated annealing [140, 271]. Genetic algorithms for join enumeration, such as those described by Bennett et al. [27], are very experimental and are derived from algorithms used to analyze genetic sequences. For example, a left-deep join strategy can be modelled as a chromosome with an ordered set of genes that represent each table in the join. Join enumeration is performed through randomly 'mutating' the genes, swapping the order of two adjacent

genes, and applying a 'crossover' operator, that interchanges two genes in one chromosome with the corresponding genes in another, retaining their relative order (and, in this case, the join implementation method). The latter operator is often described as 'breeding' since it generates a new chromosome from its two 'parents'.

Several deterministic join enumeration algorithms have appeared in the literature. INGRES uses a dynamic optimization algorithm [165, 291] that recursively breaks up a calculus (QUEL) query into smaller pieces by decomposing queries over multiple relations into a sequence of queries having one relation (tuple variable) in common, using as a basis the estimated cardinality of each. Each single-relation query is optimized by assessing the access paths and statistical information for that relation in isolation. Ibaraki and Kameda [129] showed that it is possible to compute the optimal join strategy in polynomial time, given certain restrictions on the query graph and properties of the cost model. Krishnamurthy et al. [166] proposed a polynomial-time algorithm that provides an optimal solution, though it can handle only a simplified cost model and is restricted to nested-loop joins. Swami and Iyer [272] subsequently extended their work in an attempt to remove some of its restrictions, and to also consider access plans containing sort-merge joins.

The best example of a deterministic algorithm is *dynamic programming*, the 'classical' join enumeration algorithm used by SYSTEM R and described by Selinger et al. in their seminal paper [247]. It performs static query optimization based on exhaustive search of the solution space using a modified dynamic programming approach [186, 247]. Originally developed to enumerate only join order, the algorithm has been adapted to handle other operators as well: aggregation [57], outer joins [30, 31, 106], and expensive predicates [56, 125–127]. The optimizer assigns a cost to every candidate access plan, and retains the one with the lowest cost. In addition, the algorithm keeps track of the 'sorted-ness' of each intermediate result, which are termed *interesting orders* [247, 261]. Analysis of these interesting orders can lead to less expensive strategies through the avoidance of (usually expensive) sorts on intermediate results.

2.6.3.1 Physical properties of the storage model

To augment equivalent access plans, the plan generator takes into account the physical characteristics of the database. For a join, the optimizer can choose from several *join methods*, such as block nested loop [156], index nested loop, sort-merge, hashed-loop, hybrid-hash, and PID-partitioning [203, 256]. If a query's selection predicate refers to an indexed attribute, the plan generator may choose to use an indexed retrieval of tuples instead of a table scan. It is possible that an index alone may cover the necessary attributes required, and hence access to the underlying base table can be avoided [275]. If multi-

ple indexes exist, then the generator may choose among them, or create a more sophisticated strategy utilizing index intersection and/or index union [207].

For grouped queries, the generator must decide how to best implement the grouping operation. This is typically done either through sorting or hashing; precisely which techniques are used over a given query and data distribution can have a marked effect on query performance [173]. However, a sort can be avoided if the ordering of tuples from a previous operation is preserved, e.g. if a table was retrieved using an index [261].

2.6.4 Plan Selection

Selection of an access plan is usually based on a cost model of storage structures and access operations [143]. A survey of selectivity and cost estimation is beyond the scope of this introduction; we refer interested readers to other literature surveys [116, 196, 203].

In centralized query optimizers, some combination of three measures are the basis of the cost function that the plan selection phase attempts to minimize:

- working storage requirements; for example, the size of intermediate relations [157, 247];
- CPU costs [247]; and
- secondary storage accesses [298].

Most cost models in centralized query optimizers focus primarily on the cost of secondary storage access, on the basis of estimates of the cardinalities of the operands in the algebra tree [73, 247] and a general assumption that tuples are randomly assigned pages in secondary storage [298].

Figure 2.2 illustrates plan selection as a separate phase from plan generation. With this approach, an estimated cost is computed for each access plan and the choice of strategy is based simply on the one with minimum cost [186, 247, 298]. There are, however, several alternative and complementary approaches. An optimizer can *incrementally* compute the cost of access plans in parallel to their generation. For example, Rosenthal and Reiner [236] propose that an optimizer retain only the least expensive strategy to obtain an intermediate result, discarding any other approach as soon as its cost exceeds the cheapest one found so far. This technique is used by both ORACLE [26] and DB2 [221] to quickly reduce the number of alternatives so as to minimize the overhead of optimization. *Dynamic query optimization* is the process of generating strategies only as needed during execution time, when the *exact* sizes of intermediate results are known [11, 12, 143].

The tradeoff in this approach is the generation of a more optimal strategy on the basis of real costs (not estimates) versus the optimization overhead, which now occurs at run time.

Knowledge of functional dependencies can be useful in cost estimation, but exploiting them fully has yet to be studied in detail. For the most part, database systems treat attributes and predicates as independent variables to minimize the complexity of estimation [62, 63]. However, several examples of query rewrite optimization described above, such as literal enhancement, clearly can have an impact on the selectivity of a given set of predicates, and hence affect the cost of the overall query.

Several ways to exploit attribute correlations—possibly defined by the existence of one or more functional dependencies—in cost or selectivity estimation exist in the literature. Bell, Ling, and McClean [25] study techniques to estimate join sizes where known correlations exist. Similarly, Vander Zanden et al. [285] explore estimation formulae for block accesses when attributes are highly correlated. Wang et al. [289] study several classes of predicates whose selectivity are largely affected by correlated attributes. Christodoulakis [60–63] provides additional background on the problems of cost estimation in the face of attribute correlation.

2.6.5 Summary

Our brief overview of query processing in centralized relational environments is intended to highlight areas where knowledge of functional dependencies can be exploited. As we have seen, semantic query optimization, as exemplified by ORACLE's support for rewrite optimization over materialized views, offers significant potential for dramatic reductions in query execution cost. Nonetheless, specific areas in the plan generation and plan selection phases also can benefit from the knowledge of dependencies. In the next chapter, we present an algorithm that computes the set of functional dependencies that hold for a given algebraic expression tree. In Chapters 4 and 5 we look at two ways to exploit these dependencies: techniques for query rewrite optimization, and the interaction of functional dependencies with interesting orders.

3 Functional dependencies and query decomposition

In this chapter we present algorithms for determining which *interesting* functional dependencies hold in derived relations (we discuss what defines an *interesting* dependency in the first section). In particular, for each relational algebra operator (projection, selection, etc.) we give an algorithm to compute the set of interesting dependencies that hold on output, given the set of dependencies that hold for its inputs. Our contributions are:

1. we analyze a wider set of algebraic operators (including left- and full outer join) than did Darwen [70] or Klug [162], and in addition consider the implications of null values and SQL2 semantics;

2. the algorithm handles the specification of unique constraints, primary and candidate keys, nullable columns, table and column constraints, and complex search conditions to support the computation of derived functional dependencies for a reasonably large class of SQL queries; and

3. we formally prove the correctness of our results.

Our representation of functional dependencies, described in Section 3.3, uses an extended form of FD-graphs previously introduced by Ausiello, D'Atri , and Saccà [19]. FD-graphs are a specialized form of directed hypergraph. In an FD-graph, a vertex represents a singleton attribute, and directed edges represent functional dependencies. A hypervertex represents composite determinants which consist of more than one attribute. Ausiello et al. claim that FD-graphs offer a more natural basis for formal analysis of functional dependencies than other approaches, such as Bernstein's [28] *derivation trees*. Our extensions to FD-graphs are required to represent derived dependencies in ANSI SQL expressions, which can include outer joins, null values and three-valued logic, and multiset semantics.

The rest of the chapter is organized as follows. After some preliminary discussion of sources of dependency information inherent in relational expressions, in Section 3.2 we provide the theoretical foundation for the determination of strict functional dependencies, lax functional dependencies, and strict and lax equivalence constraints for an arbitrary relational expression e consisting of the relational algebra operators introduced in

Section 2.3. We will utilize the virtual attributes of extended tables to enable the derivation of transitive dependencies over attributes that have been projected out of an intermediate or final result. We reiterate that our definition of extended table provides only a proof mechanism; their implementation is unnecessary. Section 3.4, which contains this chapter's main contributions, presents algorithms to develop an FD-graph for an arbitrary relational expression e that represents those derived functional dependencies that hold in e. Once constructed, the FD-graph can be analyzed for dependencies that can affect the outcomes of semantic query optimization algorithms, as described in Chapter 4, or can affect the outcome of sort avoidance analysis, as described in Chapter 5. In Section 3.5 we formally prove the correctness of the FD-graph construction algorithms. Section 3.6 describes and proves algorithms to compute dependency and equivalence closures from an FD-graph. Section 3.7 briefly summarizes known work in exploiting functional dependencies in query optimization, and finally Section 3.8 concludes with some ideas for further research.

3.1 Sources of dependency information

As described in Chapter 2, relational database systems typically represent a query, or relational expression e, as an algebraic expression tree. Derived functional dependencies result from the semantics of the various unary and binary relational algebra operators such as projection, selection, and join that make up the expression (see Figure 2.3). In Section 3.2 below we will describe in detail the functional dependencies implied by each of the algebraic operators. But before we do so, we outline the broad categories of semantic information inherent in schema definitions and algebraic expressions that can be analyzed to infer derived functional dependencies.

3.1.1 Axiom system for strict and lax dependencies

Sources of additional dependencies include the axiom system for strict and lax dependencies defined in Section 2.5.2. A trivial example of a *strict transitive dependency* is the logical implication of $A \longrightarrow C$ from the two strict functional dependencies $A \longrightarrow B$ and $B \longrightarrow C$ (through the application of inference rule FD7A, strict transitivity). More formally, a set of strict functional dependencies $\mathcal{F}$ implies a strict dependency f if f holds in every extended table in which $\mathcal{F}$ holds.

DEFINITION 30 (STRICT DEPENDENCY CLOSURE)
The *strict closure* of $\mathcal{F}$, denoted $\mathcal{F}^+$, is the set of all strict functional dependencies logically implied by $\mathcal{F}$ through application of the axioms defined in Section 2.5.2. The strict

closure of a *set of attributes* X with respect to $\mathcal{F}$, denoted $\overrightarrow{X^+_{\mathcal{F}}}$, is the set of attributes functionally determined by X through direct or transitive strict dependencies defined by the strict closure of $\mathcal{F}$. That is, $\overrightarrow{X^+_{\mathcal{F}}} = \{A \mid X \longrightarrow A \in \overrightarrow{\mathcal{F}^+}\}$.

It is easy to see that the number of dependencies in $\overrightarrow{\mathcal{F}^+}$ is exponential in the size of the universe of attributes and the given set $\mathcal{F}$ [24], due in part to the reflexivity axiom (e.g. if $Y \subseteq X$ we have $X \longrightarrow Y$). These dependencies are 'uninteresting' in the sense that they convey no useful information about the constraints that hold in either a derived or a base table. We explicitly avoid the exponential explosion of representing such transitive dependencies by keeping $\mathcal{F}$ in a *simplified form.*

DEFINITION 31 (SIMPLIFIED FORM)
A set of dependencies $\mathcal{F}$ is in *simplified form* [79] if the following conditions hold:

1. for all dependencies $f \in \mathcal{F}$, $X \longrightarrow Y \Longrightarrow (X \cap Y) = \emptyset$.

2. $\mathcal{F}$ contains no duplicate dependencies with identical left- and right-hand sides.

Furthermore, we assume that all strict and lax functional dependencies in $\mathcal{F}$ have single attribute dependents (right-hand sides)[15].

DEFINITION 32 (LAX DEPENDENCY CLOSURE)
Similarly to strict closure, the *lax closure* of $\mathcal{F}$, denoted $\overrightarrow{\mathcal{F}^+}$, is the set of all lax functional dependencies logically implied by $\mathcal{F}$ through application of the axioms defined in Section 2.5.2. The lax closure of a *set of attributes* X with respect to $\mathcal{F}$, denoted $\overrightarrow{X^+_{\mathcal{F}}}$, is the set of attributes functionally determined by X through direct or transitive lax dependencies defined by the lax closure of $\mathcal{F}$. That is, $\overrightarrow{X^+_{\mathcal{F}}} = \{A \mid X \longmapsto A \in \overrightarrow{\mathcal{F}^+}\}$.

For convenience we denote the union of the closures of the set of strict and lax functional dependencies in $\mathcal{F}$ with the notation $\mathcal{F}^+ = \overrightarrow{\mathcal{F}^+} \cup \overrightarrow{\mathcal{F}^+}$. We avoid the exponential explosion of lax transitive dependencies by also maintaining the lax functional dependencies in $\mathcal{F}$ in simplified form.

---

15 Note that lax decomposition only holds in the case of definite attributes (rule FD4B).

3.1.2 Primary keys and other table constraints

Schema constraints, such as primary keys, unique indexes, and unique constraints form the basis of a known set of strict or lax functional dependencies that are guaranteed to hold for every instance of the database. However, because we intend to maintain only non-trivial dependencies in simplified form, $\mathcal{F}$ will not contain any dependencies between an attribute and itself. Consider, however, an SQL table T consisting of the single column $A_T = \{a_1\}$ that cannot contain duplicate rows—that is, a_1 is a primary key of T. This uniqueness cannot be represented in $\mathcal{F}$ if we restrict dependencies to attributes in T.

To solve this problem we utilize the unique tuple identifier attribute $\iota(R)$ in the extended table R (see Section 2.2 above). $\iota(R)$ is the dependent attribute of each key dependency, and is the determinant of a set of strict dependencies whose dependent attributes are in the set $\alpha(R) \cup \rho(R)$. For SQL base tables, this mechanism provides a source of functional dependencies even for those tables that have various forms of unique constraints but lack a primary key.

3.1.3 Equality conditions

Observe that if a particular attribute $X \in sch(R)$ is guaranteed to have the same value for each tuple in $I(R)$, then in $\mathcal{F}^+$ all attributes in $sch(R)$ functionally determine X. This is typical for derived tables formed by restriction, when the query includes a **Where** clause that contains an equality condition between a column and a constant. For query optimization purposes it would be a mistake to lose the circumstances behind the generation of this new set of dependencies; knowing that an attribute is equated to a constant can be exploited in a myriad of ways during query processing, in particular the satisfaction of order properties [261] (see Chapter 5).

Further observe that an equality condition in a false-interpreted **Where** clause predicate, e.g. $A = B$, will correspond to the two strict functional dependencies $A \longrightarrow B$ and $B \longrightarrow A$ in $\mathcal{F}$. However, in this case, as in the one above, it is important to retain the fact that these dependencies stem from equalities, and that not only do the dependencies hold but that one attribute (or constant) can be substituted for the other because their *values* are the same. Henceforth we denote the set of strict attribute equivalence constraints that exist in an expression e using the symbol $\mathcal{E}$.

DEFINITION 33 (STRICT EQUIVALENCE CONSTRAINT)
Consider an extended table R and singleton attributes $A_1 \subseteq sch(R)$ and $A_2 \subseteq sch(R)$ where $A_1 \neq A_2$. Let $I(R)$ denote a specific instance of R. Then A_1 is *strictly equivalent to* A_2 in $I(R)$ (written $A_1 \overset{I(R)}{\cong} A_2$) if the following condition holds:

$$\forall\, t_0 \in I(R) : t_0[A_1] \cong t_0[A_2]. \tag{3.1}$$

CLAIM 14 (INFERENCE AXIOMS FOR STRICT EQUIVALENCE CONSTRAINTS)
The inference rules:

Identity	EQ1	$X \cong X$.
Strict commutativity	EQ2	If $X \cong Y$ then $Y \cong X$.
Strict transitivity	EQ3	If $X \cong Y$ and $Y \cong Z$ then $X \cong Z$.
Strict implication	EQ4	If $X \cong Y$ then $X \longrightarrow Y$ and $Y \longrightarrow X$.

defined over an instance $I(R)$ of an extended table R with singleton attributes $X, Y, Z \subseteq sch(R)$ are sound.

Conjunctive equality conditions between attribute pairs in an outer join's `On` condition can, in some circumstances, produce a result where the attribute pair has identical values except for a result row that contains an all-`Null` row generated for the null-supplying side. We term this weak form of attribute equivalence a *lax equivalence constraint.*

DEFINITION 34 (LAX EQUIVALENCE CONSTRAINT)
Consider an extended table R and singleton attributes $A_1 \subseteq sch(R)$ and $A_2 \subseteq sch(R)$ where $A_1 \neq A_2$. Let $I(R)$ denote a specific instance of R. Then A_1 is *laxly equivalent to* A_2 in $I(R)$ (written $A_1 \overset{I(R)}{\simeq} A_2$) if the following condition holds:

$$\forall\, t_0 \in I(R) : \lceil t_0[A_1] = t_0[A_2] \rceil. \tag{3.2}$$

Again we write $A_1 \simeq A_2$ when $I(R)$ is clear from the context. Henceforth when we use the term 'equivalence constraint' without qualification we mean *either* a strict or lax equivalence constraint.

LEMMA 4 (INFERENCE AXIOMS FOR LAX EQUIVALENCE CONSTRAINTS)
The inference rules:

Lax commutativity	EQ5	If $X \simeq Y$ then $Y \simeq X$.
Weakening	EQ6	If $X \cong Y$ then $X \simeq Y$.
Strengthening	EQ7	If $X \simeq Y$ and $I(R)$ is XY-definite then $X \cong Y$.
Lax implication	EQ8	If $X \simeq Y$ then $X \longmapsto Y$ and $Y \longmapsto X$.

defined over an instance $I(R)$ of an extended table R with singleton attributes $X, Y \subseteq sch(R)$ are sound for a combined set of strict and lax equivalence constraints.

PROOF. Omitted. □

By Claim 14, strict equivalence constraints are transitive: if $X \stackrel{\omega}{=} Y$ and $Y \stackrel{\omega}{=} Z$ then $X \stackrel{\omega}{=} Z$, which corresponds to our definition of functional dependencies (Definition 26). However, like lax functional dependencies (Definition 28 on page 36), lax equivalence constraints are transitive only over definite attributes.

LEMMA 5 (TRANSITIVITY OF LAX EQUIVALENCE)
The inference rules

Lax transitivity	EQ9A	If $X \stackrel{\omega}{=} Y$ and $Y \simeq Z$ then $X \simeq Z$.
	EQ9B	If $X \simeq Y$ and $Y \simeq Z$ and $I(R)$ is Y-definite then $X \simeq Z$.

defined over an instance $I(R)$ of an extended table R with singleton attributes $X, Y, Z \subseteq sch(R)$ are sound for a combined set of strict and lax equivalence constraints.

PROOF (RULE EQ9A). We first consider rule EQ9A. Consider an instance $I(R)$ of extended table R. By contradiction, assume that rule EQ9A is not sound. Then we must have $X \stackrel{\omega}{=} Y$ and $Y \simeq Z$, but $X \not\simeq Z$. If $X \not\simeq Z$ then there must exist at least one tuple, say r_0 in $I(R)$, that has different X- and Z-values that are each not `Null`. However, since $X \stackrel{\omega}{=} Y$, r_0 must have identical definite X- and Y-values. Since $Y \simeq Z$ holds and r_0 must have definite Y- and Z-values, then we must have $r_0[X] = r_0[Y] = r_0[Z]$, a contradiction. □

PROOF (RULE EQ9B). Consider an instance $I(R)$ of an extended table R where attribute $Y \subset sch(R)$ is definite. By contradiction, assume that rule EQ9B is not sound. Then we must have $X \simeq Y$ and $Y \simeq Z$, but $X \not\simeq Z$. If $X \not\simeq Z$ then there must exist at least one tuple, say r_0 in $I(R)$, that has different X- and Z-values that are each not `Null`. However, since $X \simeq Y$ and $r_0[Y]$ is definite, then we must have $r_0[X] = r_0[Y]$. Similarly, $r_0[Y] = r_0[Z]$, which implies that $r_0[X] = r_0[Z]$; a contradiction. Hence rule EQ9B is sound. □

THEOREM 2 (INFERENCE RULES FOR EQUIVALENCE CONSTRAINTS)
The inference rules EQ1 through EQ9 are sound for a combined set of strict and lax equivalence constraints.

PROOF. Follows from Claim 14, Lemma 4, and Lemma 5. □

DEFINITION 35 (STRICT EQUIVALENCE CLOSURE)
The *strict equivalence closure* of $\mathcal{E}$, denoted $\mathcal{E}^+$, is the set of all equivalence constraints logically implied by $\mathcal{E}$ through the application of the axioms defined in Claim 14. The *strict equivalence closure* of an attribute X with respect to $\mathcal{E}$, denoted $X^+_{\mathcal{E}}$, is the equivalence class of X such that all elements in the set are guaranteed to have the same value for any tuple t in an instance $I(R)$. That is, $X^+_{\mathcal{E}} = \{A \mid X \stackrel{w}{=} A \in \mathcal{E}^+\}$.

DEFINITION 36 (LAX EQUIVALENCE CLOSURE)
The *lax equivalence closure* of $\mathcal{E}$, denoted $\mathcal{E}^+$, is the set of all equivalence constraints logically implied by $\mathcal{E}$ through the application of the axioms defined in Claim 14 and Lemmas 4 and 5. The *lax equivalence closure* of an attribute X with respect to $\mathcal{E}$, denoted $X^+_{\mathcal{E}}$, is the equivalence class of X such that all elements in the set are guaranteed to have the same value for any tuple t in an instance $I(R)$, or may be `Null`. That is, $X^+_{\mathcal{E}} = \{A \mid X \simeq A \in \mathcal{E}^+\}$.

For convenience, we denote the union of the strict and lax equivalence closures of a set of equivalence constraints $\mathcal{E}$ with the notation $\mathcal{E}^+ = \mathcal{E}^+ \cup \mathcal{E}^+$.

3.1.4 Scalar functions

A source of additional dependencies stems from the presence of scalar functions and arithmetic expressions (which we assume to be deterministic and free of side-effects) in a query's `Select` list or one of its clauses. In Example 3 on page 8 we introduced scalar functions using the following example query:

```
Select P.*, (P.Price - P.Cost) as Margin
From   Part P
Where  P.Status = 'InStock'.
```

One can consider `Margin` as the result of a function whose input parameters are `P.Price` and `P.Cost`. In this case the derived strict functional dependency { `P.Price, P.Cost` } $\longrightarrow$ `Margin` holds in the result. We arbitrarily assume that the result of any such computation is possibly `Null`.

3.2 Dependencies implied by SQL expressions

While determining the dependencies that hold in the final query result can be beneficial, clearly such information can be exploited during the query optimization process for each subtree (including nested query blocks) of the complete algebraic expression tree. Below we describe a large class of strict and lax functional dependencies that are implied by each algebraic operator. To determine the set of dependencies that hold in an

entire expression e, one can simply recursively traverse the expression tree in a postfix manner and compute the dependencies that hold for a given operator once the dependencies of its inputs have been determined.

As most database systems directly implement existential quantification (and do not implement relational division—see Graefe and Cole [109]) the algebraic expression tree e that represents an ANSI SQL query expression also includes (possibly negated) **Exists** predicates that refer to correlated or non-correlated [188] subqueries. We use this combination calculus-algebra expression tree as the basis for forming an internal representation of a query (or sub-query) and manipulate this internal representation during query optimization to derive a more efficient computation of the desired result [90, 110, 111, 247].

If we assume that all forms of nested query predicates in SQL (**Any**, **All**, **In**, etc.) have been rewritten as **Exists** predicates (see Section 2.3.1.2) then a bottom-up analysis of the subqueries can treat any correlation attributes from super-queries as constant values. As **Exists** predicates restrict the result set of a derived table, the handling of these predicates is explained as part of the restriction operator in Section 3.2.4 below.

CLAIM 15 (DEPENDENCIES AND CONSTRAINTS IN SQL TABLES)
The following statements regarding functional dependencies and equivalence constraints that hold for any extended table R over attributes $Xyz \subseteq sch(R)$, where X denotes a set and y and z denote atomic attributes, are true for the corresponding SQL table $\mathcal{R}_\alpha(R)$:

- If $f : X \longrightarrow y$ holds in $I(R)$ and $Xy \subseteq \alpha(R)$ then f holds in the corresponding instance of $\mathcal{R}_\alpha(R)$.
- If $f : X \longmapsto y$ holds in $I(R)$ and $Xy \subseteq \alpha(R)$ then $X \longmapsto y$ in the subset of the corresponding instance of $\mathcal{R}_\alpha(R)$ where each of the values of X and y are not **Null**.
- If $f : X \longrightarrow y$ holds in $I(R)$, $X \subseteq \alpha(R)$, and $\iota(R) = y$ then X forms a candidate superkey of $\mathcal{R}_\alpha(R)$: there cannot be two rows in the corresponding instance of $\mathcal{R}_\alpha(R)$ that have identical X-values.
- If $f : X \longmapsto y$ holds in $I(R)$, $X \subseteq \alpha(R)$, and $\iota(R) = y$ then there cannot be two rows in the corresponding instance of $\mathcal{R}_\alpha(R)$ that have identical non-**Null** X-values.
- If $f : X \longrightarrow y$ holds, $y \in \alpha(R)$, and $X \subseteq \kappa(R)$, then each row of the corresponding instance of $\mathcal{R}_\alpha(R)$ has an identical Y-value (possibly **Null**).
- If $f : X \longmapsto y$ holds, $y \subseteq \alpha(R)$, and $X \subseteq \kappa(R)$, then in each row of the corresponding instance of $\mathcal{R}_\alpha(R)$ either Y is **Null** or Y is the identical non-**Null** value.

- If $e : y \stackrel{\omega}{=} z$ holds in $I(R)$ and $yz \subseteq \alpha(R)$ then each row in $\mathcal{R}_\alpha(R)$ contains identical (possibly `Null`) values for y and z. If either y or z are constants in $\kappa(R)$, then each row in $\mathcal{R}_\alpha(R)$ will have a value identical to that of the constant.

- If $e : y \simeq z$ holds in $I(R)$ and $yz \subseteq \alpha(R)$ then each row in $\mathcal{R}_\alpha(R)$ either contains identical non-`Null` values for y and z, or at least one of y or z is `Null`. Similarly, if y is a constant in $\kappa(R)$ then each row of $\mathcal{R}_\alpha(R)$ either contains identical values of z, equivalent to the value of the constant y, or z is `Null`.

3.2.1 Base tables

At the leaves of the expression tree are nodes representing quantified range variables over base tables. The **`Create Table`** statement for these tables—see the examples in Appendix A—include declarations of one or more keys (see Section 2.4.1). We do not attempt to determine a minimal key, if it exists, for each operator in the tree. In part we do this because of the complexity of finding one or all of the minimal keys for any algebraic expression e (cf. Fadous and Forsyth [82], Lucchesi and Osborn [191], and Saiedian and Spencer [246]). We also refrain from computing sets of minimal keys throughout the tree due to the realization that much of the computation will likely be wasted. Often it is sufficient to determine the *closure* of a set of attributes—for example, finding if the closure of a set of attributes *includes* a key of a base table (see Chapter 4).

Other arbitrary constraints on base tables can be handled as if they constitute a restriction condition on R (see Section 3.2.4). Since table constraints are true-interpreted (if their value is *unknown* then the constraint still holds), then those **`Check`** constraints over non-null attributes can imply strict dependencies and/or equivalence constraints if the `Check` constraint includes an equality condition that can be recognized during constraint analysis (see Section 3.2.4). Otherwise, such constraints may infer a lax dependency or equivalence constraint that should still be captured in case the existence of any null-intolerant restriction predicates can later be used to transform a lax dependency or equivalence constraint into a strict one.

CLAIM 16 (DEPENDENCIES AND CONSTRAINTS IN BASE TABLES)
The set of attributes $K_i(R)$ of the primary key and each unique index on a base table $\mathcal{R}_\alpha(R)$ imply strict functional dependencies of the form $K_i \longrightarrow \iota(R)$ hold in $I(R)$. Similarly, the set of attributes $U_i(R)$ of each unique constraint on a base table $\mathcal{R}_\alpha(R)$ imply lax functional dependencies of the form $U_i \longmapsto \iota(R)$ hold in $I(R)$.

3.2.2 Projection

The primary purpose of the projection operator π_{All} and the distinct projection operator π_{Dist} is to project an extended table R over attributes A, thereby eliminating attributes from the result. In the case of π_{Dist}, the auxiliary purpose is to eliminate 'duplicate' tuples from the result with respect to A. Clearly any functional dependency that includes an attribute not in A is rendered meaningless; however, we must be careful not to lose dependencies implied through transitivity. Unlike Darwen's approach [70], which relies on the recomputation of the closure $\mathcal{F}^+$ at each step, we intend to compute $\mathcal{F}^+$ as seldom as possible. This means that we must maintain dependency information even for a table's virtual columns (see Section 2.2).

The projection and distinct projection operators can both add and remove functional dependencies to the set of dependencies that hold in the result. If projection includes the application of scalar functions, then R is *extended* with the result of the scalar function to form R'. Moreover, new strict functional dependencies now exist between the function result λ and its input(s) (see Section 3.1.4 above). If the projection operator removes an attribute, say C, then $\mathcal{F}^+$ consists of the closure of the dependencies that hold in the input, less any dependencies that directly include C as a determinant or a dependent. In other words, if we have the strict dependencies $A \longrightarrow \iota(R)$, $\iota(R) \longrightarrow ABCDE$, and $BC \longrightarrow F$ and attribute C is projected out, then the dependencies $\iota(R) \longrightarrow C$ and $BC \longrightarrow F$ can no longer hold, since attribute C is no longer present. However, by the inference axioms presented in Section 2.5.2 the transitive dependency $A \longrightarrow F$ still holds in the extended table formed by e.

CLAIM 17 (DEPENDENCIES IMPLIED BY PROJECTION)
Let R' denote the result of the algebraic expression $\pi_{All}[A](R)$ denoting the projection (retaining duplicates) of the extended table R over A. Suppose that $I(R)$ satisfies a set of functional dependencies $\mathcal{F}$ and a set of equivalence constraints $\mathcal{E}$. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The functional dependencies and equivalence constraints that hold in $I(R)$ continue to hold in $I(R')$.
- For each scalar function $\lambda(X) \in A$ the strict functional dependencies $X \longrightarrow \lambda(X)$ and $\iota(R') \longrightarrow \lambda(X)$ hold in $\mathcal{F}_{R'}$.

PROOF. Omitted. □

CLAIM 18 (DEPENDENCIES IMPLIED BY DISTINCT PROJECTION)
Let R' denote the result of the algebraic expression $\pi_{Dist}[A](R)$ denoting the distinct projection of R over A. Suppose that $I(R)$ satisfies a set of functional dependencies $\mathcal{F}$ and a set of equivalence constraints $\mathcal{E}$. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The functional dependencies and equivalence constraints that hold in $I(R)$ continue to hold in $I(R')$.
- For each scalar function $\lambda(X) \in A$ the strict functional dependency $X \longrightarrow \lambda(X)$ holds in $\mathcal{F}_{R'}$.
- The strict functional dependencies $A \longrightarrow \iota(R')$ and $\iota(R') \longrightarrow \alpha(R')$ hold in $I(R')$.

PROOF. Recall that by the definition of distinct projection, we nondeterministically select a representative tuple r for each set of tuples in $I(R)$ with matching values of A. Simply removing a tuple from a set has no effect on the functional dependencies or equivalence constraints satisfied by that instance; hence if f holds in R then it must follow that f holds in R'. □

3.2.3 Cartesian product

The set of dependencies that hold in the result of a Cartesian product is the union of those dependencies that hold in the inputs. Klug [162, pp. 266] stated that the Cartesian product of two tables S and T does not add any additional dependencies. However, in addition to dependencies involving attributes, we would like to retain knowledge of key dependencies that hold in either input, and if they exist derive a key dependency in the result. The maintenance of key dependencies is critical to the success of the overall algorithm, since we are following both Darwen and Klug by modelling inner join as a restriction condition over a Cartesian product.

We observe that if we can guarantee that either of the inputs, say $I(S)$, can have at most one tuple then $K_T \longrightarrow sch(S) \cup sch(T)$. This optimization is omitted from the algorithm for computing derived dependencies implied by a Cartesian product, which is described in Section 3.4.4 below.

CLAIM 19 (DEPENDENCIES IMPLIED BY CARTESIAN PRODUCT)
Let R' denote the result of the algebraic expression $R' = S \times T$ denoting the Cartesian product of extended tables S and T. Suppose $I(S)$ satisfies the set $\mathcal{F}_S$ of functional dependencies and the set $\mathcal{E}_S$ of equivalence constraints, and similarly $I(T)$ satisfies the set $\mathcal{F}_T$ of functional dependencies and the set $\mathcal{E}_T$ of equivalence constraints. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The set of functional dependencies that hold in $I(S)$, and those that hold in $I(T)$, continue to hold in $I(R')$.
- The set of equivalence constraints that hold in $I(S)$, and those that hold in $I(T)$, continue to hold in $I(R')$.
- The strict functional dependency $(\iota(S) \cup \iota(T)) \longrightarrow \iota(R')$ holds in $I(R')$.
- The strict functional dependency $\iota(R') \longrightarrow \iota(S) \cup \iota(T)$ holds in $I(R')$.

PROOF. Omitted. □

3.2.4 Restriction

The algebraic restriction operator is used for both selection and having clauses; the semantics are identical since we model a **Having** clause as a restriction operator over the result of a grouped table projection, possibly followed by another projection to remove extraneous results of aggregate functions (see Example 7). Restriction is one operator that can only *add* strict functional dependencies to $\mathcal{F}$; it cannot remove any existing strict dependencies.

Both Fagin [83] and Nicolas [218] showed that functional dependencies are equivalent to statements in propositional logic; thus if one can transform a **Where** or **Having** predicate into an implication, then one can derive an additional dependency that will hold in the result. For example, the constraint "if $A < 5$ then $B = 6$ else $B = 7$" implies the functional dependency $A \longrightarrow B$ even though no direct relationship between A and B is expressed in the constraint. Consequently, the problem of inferring additional functional dependencies from predicates in a **Where** or **Having** clause depends entirely on the sophistication of the algorithm that translates predicates into their equivalent logic sentences.

A comprehensive study of this problem is beyond the scope of this thesis. Instead, we consider a simplified set of conditions. In two earlier papers Klug [162, 164] considered only conjunctive atomic conditions with equivalence operators, that is conditions of the form $(v = c)$ (which he terms *selection*) and conditions of the form $(v_1 = v_2)$ (which Klug terms *restriction*) where v, v_1, and v_2 are columns and c is a constant. For ease of reference we term a condition of the form $(v = c)$ a *Type 1* condition, and a condition of the form $(v_1 = v_2)$ a *Type 2* condition. Each false-interpreted equivalence condition of Type 1 or 2 implies both an strict equivalence constraint and two symmetric strict functional dependencies.

Darwen [70] argued that one need consider only conjunctive θ-comparisons because if a search condition is more complex it can be reformulated using the algebraic set operations union, intersection, and difference. However, with ANSI SQL semantics such query

reformulation will not work in general due to the possible existence of null values, complex predicates, and duplicate tuples. Consequently, for completeness one must consider disjunctive conditions as an integral part of handling restriction. However, in this thesis we do not exploit disjunctive conditions for inferring constraints and functional dependencies. We also assume that negated conditions have been transformed where possible (cf. Larson and Yang [175]); in particular, that inequalities (e.g. $X \neq 5$) have been transformed to the semantically equivalent $X < 5$ or $X > 5$. We also assume that the restriction conditions can be simplified through their conversion into conjunctive normal form [264, 290] so as to recognize and eliminate redundant expressions and unnecessary disjunctions. Once these transformations have been performed, we restrict our analysis of atomic conditions in a **Where** clause to conjunctive conditions of Type 1 or Type 2.

We note, however, that the algorithms described can be easily extended to capture and maintain additional equivalence constraints and functional dependencies through a more complete analysis of ANSI SQL semantics. In addition to θ-comparisons between attributes and constants there are several additional atomic conditions that can be expressed in ANSI SQL: `is [not] null`, `like`, and θ-comparisons between an attribute or constant and a subquery. For the purposes of dependency analysis we could exploit these other conditions as follows:

- *is null predicates.* We could interpret an `is null` predicate as equivalent to a Type 1 equality condition with a null value which can either return *true* or *false* (and not *unknown*). Negated `is null` predicates, e.g. `X is not null`, could also be exploited—in this case X is guaranteed to be definite in the result.

- *like predicates.* In general `Like` predicates are useless for functional dependency analysis due to the presence of wildcards. However, if no wildcards are specified in the pattern then the `Like` predicate is equivalent to an equality predicate with the pattern string.

- *equality comparison with a subquery.* If the subquery is non-correlated it can be evaluated independently of its outer query block; thus if attribute X is equated to a subquery result then X can be equated to a (yet unknown) constant value, possibly `null`.

Extension. If we detect a scalar function $\lambda(X)$ during the analysis of a **Where** clause, then as with both projection and distinct projection we add the result of the function to the extended table produced by restriction as a virtual attribute, to retain the (strict) functional dependency $X \longrightarrow \lambda(X)$ in $\mathcal{F}$.

Inferring lax equivalences and dependencies. As a result of the conversion of nested queries to their canonical form, correlation predicates in a subquery's **Where** clause may require true-interpretation. Since true interpretation commutes over both disjunction and conjunction (axioms 1 and 3 in Table 2.3), we can infer lax equivalence constraints and lax functional dependencies from each Type 1 and Type 2 condition in these predicates.

Conversion of lax equivalences and dependencies. As per ANSI SQL semantics, by default we assume that each conjunct of the restriction predicate is false-interpreted. Any null-intolerant predicate referencing an attribute X will automatically eliminate any tuples from the result where X is the null value. Hence for any algebraic operator higher in the expression tree, it can be guaranteed that X cannot be **Null**, and this can be extended to any other attribute Y that is (transitively) equated to X. Hence any lax equivalence constraint involving X can be strengthened, using inference rule EQ7, into a strict equivalence constraint if X is (transitively) equated to another non-**Null** attribute or constant. Similarly, we can convert any lax dependencies into strict dependencies once we can determine that neither the dependent attributes, nor any of its determinant attributes, can be **Null**, satisfying inference axiom FD6 (strengthening). In the case of composite determinants, we must be able to show that each individual component cannot be **Null**.

CLAIM 20 (DEPENDENCIES IMPLIED BY RESTRICTION)
Let R' denote the result of the algebraic expression $R' = \sigma[C](R)$ denoting the restriction of R by false-interpreted predicate C. Suppose $I(R)$ satisfies the set $\mathcal{F}$ of functional dependencies and the set $\mathcal{E}$ of equivalence constraints. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The set of strict functional dependencies that hold in $I(R)$ continue to hold in $I(R')$. Similarly, the set of strict equivalence constraints that hold in $I(R)$ continue to hold in $I(R')$.
- For each lax functional dependency $f : X \longmapsto Y$ that holds in $I(R)$, if $I(R')$ is XY-definite then f holds as a strict dependency in $I(R')$; otherwise f continues to hold as a lax dependency in $I(R')$.
- For each lax equivalence constraint $e : X \simeq Y$ that holds in $I(R)$, if $I(R')$ is XY-definite then e holds as a strict equivalence constraint in $I(R')$; otherwise e continues to hold as a lax equivalence constraint in $I(R')$.
- For each scalar function $\lambda(X) \in \alpha(C)$ the strict functional dependencies $X \longrightarrow \lambda(X)$ and $\iota(R') \longrightarrow \lambda(X)$ hold in $I(R')$.

- Each false-interpreted Type 1 or Type 2 condition of the form $X = Y$ in C implies the strict equivalence constraint $X \stackrel{\omega}{=} Y$ and the strict functional dependencies $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$ hold in $I(R')$.
- Each true-interpreted Type 1 or Type 2 condition of the form $X = Y$ in C implies the lax equivalence constraint $X \simeq Y$ and the lax functional dependencies $f : X \longmapsto Y$ and $g : Y \longmapsto X$ hold in $I(R')$.

PROOF. Omitted. □

3.2.5 Intersection

The result R' of the intersection of two inputs S and T contains either unique instances of each tuple in $I(R')$ with respect to the real attributes in $sch(R)$ (in the case of the `Intersect` operator) or some number of duplicate tuples corresponding to the definition of `Intersect All`. In either case, by our definition of the intersection operators $\cap_{Dist}$ and $\cap_{All}$ (see Section 2.3.2) a tuple q_0 can only exist in the result of $R' = S \cap T$ if a corresponding image of $q_0[\alpha(R)]$ exists in both $I(S)$ and $I(T)$. Hence $I(R')$ must satisfy the set of dependencies that hold with respect to the real attributes in both S and T.

LEMMA 6 (STRICT DEPENDENCIES IMPLIED BY INTERSECTION)
Consider a query Q consisting of the intersection of two tables S and T

$$Q = R \cap_{All} S$$

where each attribute $A_i^S \in \alpha(S)$ is union-compatible with its corresponding attribute $A_i^T \in \alpha(T)$. Suppose $I(S)$ satisfies the strict functional dependency $f : A_1^S \longrightarrow A_2^S \in \mathcal{F}_S$, where $A_1^S \cup A_2^S \subseteq \alpha(S)$. Then the functional dependency $f : A_1^Q \longrightarrow A_2^Q$ holds in $I(Q)$, where A_1^Q and A_2^Q correspond to the input attributes A_1^S and A_2^S.

PROOF. By contradiction, assume the dependency $f : A_1^Q \longrightarrow A_2^Q$ does *not* hold in Q, but its corresponding dependency $A_1^S \longrightarrow A_2^S$ holds in S. This means that there exist at least two tuples q_0, q_0' in Q such that $q_0[A_i^Q] \stackrel{\omega}{=} q_0'[A_i^Q]$ but $q_0[A_j^Q] \stackrel{\omega}{\neq} q_0'[A_j^Q]$. By the definition of intersection, any tuple component $q_0[\alpha(Q)]$ in Q must exist in both $I(S)$ and $I(T)$ (the null comparison operator is used so that null value comparisons resulting in *unknown* are interpreted as *true*). Hence the result is both a subset of the tuples in $I(S)$, and a subset of the tuples in $I(T)$. By our initial assumption, this means that there must exist tuples in S where the strict functional dependency $A_1^S \longrightarrow A_2^S$, where $(A_1^S \cup A_2^S) \subseteq \alpha(S)$, cannot hold in S; a contradiction. A similar situation occurs if the dependency is assumed to hold in T. Hence we conclude that if a dependency holds for real attributes in either S or T then it must also hold in Q. □

Since the set of dependencies that hold in $\mathcal{R}_\alpha(Q)$ is at least the union of those that hold in $\mathcal{R}_\alpha(S)$ and $\mathcal{R}_\alpha(T)$ (with attributes appropriately renamed), then the superkeys that hold in either S or T also hold in Q, as we formally state below.

COROLLARY 1 (SUPERKEYS IMPLIED BY INTERSECTION)
Consider a query Q consisting of the intersection of two tables S and T

$Q = S \cap_{All} T.$

All members of the union of the sets of superkeys in $\alpha(S)$ and $\alpha(T)$ that hold in S and T respectively hold as superkeys in Q.

PROOF. Follows directly from Lemma 6 as the set of functional dependencies that hold in $\mathcal{R}_\alpha(Q)$ is at least the union of those that hold in $\mathcal{R}_\alpha(S)$ and $\mathcal{R}_\alpha(T)$. □

Conversion of lax equivalence constraints and lax functional dependencies. We observe that, similarly to strict functional dependencies, any lax functional dependencies, lax equivalence constraints, and strict equivalence constraints that hold in $I(S)$ or $I(T)$ will continue to hold in Q. However, lax dependencies (equivalence constraints) from one of the two inputs may be converted to strict dependencies (equivalence constraints) if, by being 'paired' with the other extended table, it can now be guaranteed that both the determinant and dependent attributes cannot be `Null` in the result—exactly as was the case for the restriction operator.

EXAMPLE 14 (LAX DEPENDENCIES AND INTERSECTION)
Consider the following query expression involving intersection:

```
Select S.VendorID, S.PartID, S.Lagtime, S.Rating
From   Supplier S
Where  S.Rating = 'B'
Intersect All
Select S.VendorID, S.PartID, S.Lagtime, S.Rating
From   Supplier S
Where  S.Lagtime > :Lagtime
```

where :`Lagtime` denotes a host variable. In the result of the intersection, neither **Lagtime** nor `Rating` can be `Null` since the null-intolerant predicates in each query specification's `Where` clause will prevent null values in the result. Hence a hypothetical lax functional dependency **Lagtime** $\longmapsto$ **Rating** in either input will hold as a strict dependency in the result.

CLAIM 21 (DEPENDENCIES IMPLIED BY INTERSECTION)
Let R' denote the result of the algebraic expression $R' = S \cap_{All} T$ denoting the intersection of extended tables S and T. Suppose $I(S)$ satisfies the set $\mathcal{F}_S$ of functional dependencies and the set $\mathcal{E}_S$ of equivalence constraints, and similarly $I(T)$ satisfies the set $\mathcal{F}_T$ of functional dependencies and the set $\mathcal{E}_T$ of equivalence constraints. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The set of strict functional dependencies that hold in $I(S)$, and those that hold in $I(T)$, continue to hold in $I(R')$.
- The set of strict equivalence constraints that hold in $I(S)$, and those that hold in $I(T)$, continue to hold in $I(R')$.
- For each lax functional dependency $f : X \longmapsto Y$ in $\mathcal{F}_S \cup \mathcal{F}_T$, if $I(R')$ is XY-definite then f holds as a strict dependency in $I(R')$; otherwise f continues to hold as a lax dependency in $I(R')$.
- For each lax equivalence constraint $e : X \simeq Y$ in $\mathcal{E}_S \cup \mathcal{E}_T$, if $I(R')$ is XY-definite then e holds as a strict equivalence constraint in $I(R')$; otherwise e continues to hold as a lax equivalence constraint in $I(R')$.
- For each pair X and Y of corresponding union-compatible attributes, where $X \subseteq \alpha(S)$ and $Y \subseteq \alpha(T)$, the strict equivalence constraint $X \stackrel{\omega}{=} Y$ and the strict functional dependencies $X \longrightarrow Y$ and $Y \longrightarrow X$ hold in $I(R')$.
- The strict functional dependencies $\iota(S) \longrightarrow \iota(T)$ and $\iota(T) \longrightarrow \iota(S)$ hold in $I(R')$.

PROOF. Omitted. □

3.2.6 Union

If considering only dependencies, there is no way in general to determine the dependencies that hold in $Q = S \cup_{All} T$ [70]. Both Darwen and Klug offer one additional possibility: if it can be determined that S and T are two distinct subsets of the same expression (typically a base table R) then the dependencies that hold in R also hold in Q. However, determining if two subexpressions return distinct sets of tuples is undecidable [162]. Consequently we take the conservative approach and assume that none of the dependencies that hold in the inputs also hold in the result.

However, by considering strict attribute equivalence constraints in either input, it is possible to retain these constraints in the result, and the strict functional dependencies they imply.

EXAMPLE 15 (DERIVED DEPENDENCIES WITH UNION)
Consider the following query expression involving union over the SUPPLIER, VENDOR, and QUOTE tables in the example schema:

```
Select S.VendorID, S.PartID, S.Lagtime, S.Rating, V.VendorID
From   Supplier S, Vendor V, Quote Q
Where  S.Rating = 'B' and S.VendorID = V.VendorID and
       Q.VendorID = S.VendorID and Q.PartID = S.PartID and
       Q.EffectiveDate Between '10-01-1999' and '12-31-1999' and
       V.VendorID = :VendorID
Union All
Select S.VendorID, S.PartID, S.Lagtime, S.Rating, V.VendorID
From   Supplier S, Vendor V
Where  S.Rating = 'A' and S.VendorID = V.VendorID and
       V.VendorID = :VendorID
```

where :VendorID denotes a host variable.

By analyzing the strict equivalence constraints in each query specification, we can see that S.VendorID and V.VendorID must be equivalent in the result, since unlike dependencies, which imply a relationship amongst a set of tuples, equivalence constraints must hold for each tuple in the instance. Moreover, since the host variable :VendorID is used in both query specifications, we can also determine that each tuple in the result has an *identical* VendorID. This information can be exploited by other algebraic operators if the union forms part or all of an intermediate result.

If the Union query expression eliminates duplicates then we convert the expression tree into one with a distinct projection over a union. In this case, the application of distinct projection can add one dependency: all of the (renamed) attributes in the result, by definition, now form a superkey of Q.

3.2.7 Difference

If $Q = S - T$ then Q simply contains a subset of the tuples of S, regardless if the set difference operator eliminates duplicate tuples or not. As with restriction and intersection, eliminating tuples from a result does not affect existing dependencies: in our extended relational model (and other relational models) tuples are independent and removing a tuple s_0 from $I(S)$ has no affect on any other tuple in $I(S)$, including satisfaction of strict or lax functional dependencies. Hence it is easy to see that the same set of dependencies and equivalence constraints that hold in $I(S)$ also hold in $I(Q)$, and furthermore any superkey that holds in $I(S)$ also holds in the result.

3.2.8 Grouping and Aggregation

As described in Section 2.3 we model SQL's group-by operator with two algebraic operators. The *partition* operator, denoted $\mathcal{G}$, produces a *grouped table* as its result with one tuple per distinct set of group-by attributes. Each set of values required to compute any aggregate function is modelled as a set-valued attribute. The *grouped table projection* operator projects a grouped table over a `Select` list. Projection of a grouped table differs from an 'ordinary' projection in that it must deal not only with atomic attributes (those in the `Group by` list) but also the set-valued attributes used to compute aggregate functions.

3.2.8.1 Partition

We first describe our procedure for computing the derived functional dependencies that hold for partition, which is quite similar to that for distinct projection (see Section 3.2.2 above). If the set of n grouping attributes A^G is not empty then A^G forms a superkey of the grouped table R' that constitutes the intermediate result. Hence $A^G \longrightarrow A^A$ in R'. Other dependencies that hold in the input extended table R are maintained, as there may exist transitive dependencies that relate attributes in A^G, in which case they too hold in R'. Otherwise, if the set A^G is empty, then the result (by definition) consists of a single tuple. The result R' contains as many set-valued attributes as required to compute the aggregate functions F, modelled by the grouped table projection operator $\mathcal{P}$ below, which range over the set-valued attributes in each tuple of R'.

CLAIM 22 (DEPENDENCIES IMPLIED BY PARTITION)
Let R' denote the result of the algebraic expression $\mathcal{G}[A^G, A^A](R)$ denoting the partition of R over n grouping attributes A^G. Suppose $I(R)$ satisfies the set $\mathcal{F}$ of functional dependencies and the set $\mathcal{E}$ of equivalence constraints. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The functional dependencies and equivalence constraints that hold in $I(R)$ continue to hold in $I(R')$.
- For each scalar function $\lambda(X) \in A^G$ the strict functional dependency $X \longrightarrow \lambda(X)$ holds in $I(R')$.
- The strict functional dependency $A^G \longrightarrow \iota(R')$ holds in $I(R')$. Note that if A^G is empty still holds, since $I(R')$ consists of a single tuple.
- The strict functional dependency $\iota(R') \longrightarrow \alpha(R')$ holds in $I(R')$.

3.2.8.2 Projection of a grouped table

The projection of a grouped table, denoted $\mathcal{P}$, projects a grouped table R over the set of grouping columns and computes any aggregate functions F over one or more set-valued attributes A^A. Recall from our definition of the partition operator (Definition 17 on page 25) that the projection of a grouped table retains duplicates in the result. Any further projection over this intermediate result, either through (1) eliminating attributes from the input, (2) extending the result by the use of scalar functions, or (3) eliminating duplicates through the specification of `Select Distinct` is modelled by an additional projection or distinct projection operator that takes as its input the result of $\mathcal{P}$.

CLAIM 23 (DEPENDENCIES IMPLIED BY GROUPED TABLE PROJECTION)
Let R' denote the result of the algebraic expression $\mathcal{P}[A^G, F[A^A]](R)$ denoting the grouped table projection of the grouped extended table R over n grouping attributes A^G and the aggregation expressions contained in F. Suppose $I(R)$ satisfies the set $\mathcal{F}$ of strict and lax functional dependencies, and the set $\mathcal{E}$ of strict and lax equivalence constraints. The set of functional dependencies and equivalence constraints that hold in $I(R')$ are as follows:

- The functional dependencies and equivalence constraints that hold in $I(R)$ continue to hold in $I(R')$.
- For each aggregate function $f_i(A_j^A) \in F[A^A]$ the strict functional dependencies $A_j^A \longrightarrow f_i$ and $\iota(R') \longrightarrow f_i$ hold in $I(R')$.

3.2.9 Left outer join

In Section 2.3 we referred to the three types of outer joins we consider in this thesis: left, right, and full outer joins. As left and right outer joins can be made semantically equivalent by commuting their operands, without loss of generality we heretofore consider only left and full outer joins. The bulk of the discussion below refers to left outer joins. We will qualify these remarks as necessary when addressing the issues that pertain to full outer joins.

Outer joins introduce several problems in computing derived functional dependencies because of the possibility of the generation of an all-`Null` row from the null-supplying side of the outer join. The following example illustrates a typical case with left outer joins.

EXAMPLE 16 (LEFT OUTER JOIN)
Suppose we have the algebraic expression

$$Q = \mathcal{R}_\alpha(\pi_{All}[a_1^P, a_2^P, a_1^S, a_3^S, a_4^S](P \xrightarrow{p} S))$$

that represents the query

```
Select P.PartID, P.Description, S.VendorID, S.Rating, S.SupplyCode
From   R_a(Part) P Left Outer Join R_a(Supply) S
       on ( P.PartID = S.PartID )
```

which lists all parts and their suppliers' ratings and supply codes. If a part lacks a corresponding supplier then for that part the result contains Null for those attributes of the SUPPLY table. In this case p represents the atomic equivalence predicate contained in the On condition. Now suppose that we have a database instance where the PART and SUPPLY tables are as follows (for brevity only the relevant real attributes have been included):

PART

Part ID	*Description*	*Price*
100	'Bolt'	0.09
200	'Flange'	0.37
300	'Tapcon'	0.23
400	'Switch'	3.49

SUPPLY

Vendor ID	*Part* ID	*Rating*	*SupplyCode*
002	100	Null	'03AC'
011	200	'A'	Null

With this database instance the result Q consists of the four rows

Q

P.Part ID	*Description*	*Vendor* ID	*S.Part* ID	*Rating*	*SupplyCode*
100	'Bolt'	002	100	Null	'03AC'
200	'Flange'	011	200	'A'	Null
300	'Tapcon'	Null	Null	Null	Null
400	'Switch'	Null	Null	Null	Null.

3.2.9.1 Input dependencies and left outer joins

From Example 16 above we can make several observations about derived dependencies that hold in the result of left outer joins.

First, we note that the functional dependency **PartID** $\longrightarrow$ **Description** holds in Q, as do any other dependencies that hold in the PART table (which in this case is termed the preserved table[16]). Clearly, lax functional dependencies from the null-supplying side of a

16 See Section 2.1 on page 7 for an explanation of the components of an outer join.

left outer join will continue to hold in the result. If we project the result of a left outer join over the null-supplying real attributes, we get either those null-supplying tuples or the all-Null row, which by definition cannot violate a lax functional dependency. In general, however, strict functional dependencies that hold in the *null-supplying* side of a left outer join do *not* hold in the result. For example, suppose the strict functional dependency f = Rating $\longrightarrow$ SupplyCode is guaranteed to hold in SUPPLY. In the example above, we see that f does *not* hold in Q (by our definition of functional dependency—see Definition 26). f does not hold due to the generation of at least one all-Null row in the result from the null-supplying table SUPPLY.

Second, note that while strict dependencies from a null-supplying table may not hold in Q, these dependencies still hold for all tuples in the result that *do not* contain an all-Null row. We can model these dependencies as lax functional dependencies as their characteristics are identical to those implied by the existence of a **Unique** constraint.

Third, any strict dependencies that hold in the null-supplying table (SUPPLY in the example) whose determinants contain at least one definite attribute will continue to hold in the result. In Example 16 the strict dependency { **VendorID, S.PartID** } $\longrightarrow$ **Supply-Code** continues to hold in Q because the only way in which either **VendorID** or **PartID** can be Null is if they are part of a generated all-Null row, in which case all of the other attributes from SUPPLY will also be **Null**. Therefore, the generation of an all-Null row in the result will not violate the dependency. This also means that any strict functional dependency that is a result of a superkey with at least one definite attribute in the null-supplying table will continue to hold in the result.

Fourth, we may be able to exploit one or more conditions in the left outer join's **On** condition to retain strict dependencies from the null-supplying table in the result, as the following example illustrates.

EXAMPLE 17 (EXPLOITING NULL-INTOLERANT ON CONDITIONS)
Consider a slightly modified version of the query in Example 16 that includes a conjunctive null-intolerant predicate on the SUPPLY attribute **Rating**:

```
Select P.PartID, P.Description, S.VendorID, S.Rating, S.SupplyCode
From   R_α(Part) P Left Outer Join R_α(Supply) S
       on ( P.PartID = S.PartID and S.Rating = 'A' ).
```

In this case, the **On** condition's second conjunct will eliminate from the result any row from SUPPLY that fails to join with PART or contains a null value for **Rating** (or, for that matter, any value other than 'A'). The result Q will now be

Q

P.Part ID	*Description*	*Vendor* ID	*S.Part* ID	*Rating*	*SupplyCode*
100	'Bolt'	Null	Null	Null	Null
200	'Flange'	011	200	'A'	Null
300	'Tapcon'	Null	Null	Null	Null
400	'Switch'	Null	Null	Null	Null.

Earlier we presumed that the strict functional dependency f = **Rating** $\longrightarrow$ **SupplyCode** holds in SUPPLY. The left outer join's **On** condition guarantees that the only way for a **Rating** of **Null** to appear in the result is as part of a generated all-**Null** row, because p is null-intolerant on **Rating**. This means that a null value for **Rating** in the result implies that the dependent attribute in f (**SupplyCode**) must also be **Null**. Hence f can remain a strict dependency in the result.

In summary, any strict dependency f that holds in the null-supplying side of a left outer join will continue to hold in the result if either (a) any of the determinant attributes of f are definite, or (b) the **On** condition p cannot evaluate to *true* for at least one of f s determinant values which are nullable. These two conditions represent a generalization of Bhargava, Goel, and Iyer's rule for removing attributes from the key of a derived table formed by a left outer join[34, Lemma 1, pp. 445]. If neither case (a) nor (b) hold, a strict or lax dependency f that holds in the null-supplying side of a left outer join can only be modelled as a lax dependency in the result. For brevity, we assume the existence of a nullability function $\eta(p, X)$ that determines if either case (a) or (b) holds for the determinant of any strict dependency f.

DEFINITION 37 (NULLABILITY FUNCTION)
The function $\eta(p, X)$ over attributes $\alpha(p) \cup X$ evaluates to *true* if and only if at least one attribute $x \in X$ meets one of the following conditions:

1. x is guaranteed to be definite, or
2. $x \in \alpha(p)$ and p evaluates to *false* or *unknown* whenever x is **Null**.

Otherwise, $\eta(p, X)$ evaluates to *false*.

We state the rule for the propagation of strict dependencies from a null-supplying table more formally as follows.

LEMMA 7 (STRICT DEPENDENCIES FROM A NULL-SUPPLYING TABLE)
Consider an outer join query Q over two extended tables R and S

$$Q = R \overset{C_{R,S} \wedge C_S}{\longrightarrow} S$$

where the On condition $p = C_{R,S} \wedge C_S$ consists of a conjunct of C_S containing predicates solely referencing real attributes in the null-supplying table S, and $C_{R,S}$ containing those predicates referencing real attributes from both the preserved and null-supplying sides of the left outer join. Then if the strict functional dependency $f = a_i^S \longrightarrow a_j^S$ (a_i^S and a_m^S not necessarily distinct, and a_i^S possibly composite) holds in S for every instance of the database and $\eta(p, a_i^S)$ evaluates to *true*, then f also holds in $I(Q)$.

PROOF. By contradiction, assume that f holds in S and $\eta(p, a_i^S)$ evaluates to *true*, but f does not hold in Q. Then there must exist at least two tuples q_0, q_0' in $I(Q)$ such that $q_0[a_i^S] \stackrel{\omega}{=} q_0'[a_i^S]$ but $q_0[a_j^S] \stackrel{\omega}{\neq} q_0'[a_j^S]$. There are three possible ways in which the two tuples q_0 and q_0' could be formed:

- *Case 1:* both tuples $q_0[sch(S)]$ and $q_0[sch(S)]$ are projections of their corresponding tuples s_0 and s_0' in S; hence the values of each corresponding attribute in q_0 and q_0' are identical. If the values of $q_0[a_j^S]$ and $q_0'[a_j^S]$ are different, however, then f cannot hold in $I(S)$, a contradiction.

- *Case 2:* both tuples $q_0[sch(S)]$ and $q_0'[sch(S)]$ are formed using the all-Null row s_{Null}, that is there are no tuples in S that satisfy p for both tuples $q_0[\alpha(R)]$ and $q_0'[\alpha(R)]$. However, this scenario is impossible, since the two tuples q_0 and q_0' must contain different values for a_j^S if f does not hold, and hence at least one of the values of a_j^S cannot be **Null**.

- *Case 3:* One tuple, arbitrarily $q_0[sch(S)]$, is a projection of its corresponding tuple in S, and the other $q_0'[sch(S)]$ is formed from the all-Null row s_{Null}. Since $q_0[a_i^S] \stackrel{\omega}{=} q_0'[a_i^S]$, then each of the determinant values for attributes a_i^S for both tuples q_0 and q_0' must be **Null**. However, $\eta(p, a_i^S)$ evaluated to *true*, meaning that at least one attribute in a_i^S cannot be **Null**, a contradiction. An identical situation which leads to the same contradiction occurs if q_0 and q_0' are interchanged.

Hence we conclude that f holds in $I(Q)$ if f holds in $I(S)$ and $\eta(p, a_i^S)$ evaluates to *true*. □

COROLLARY 2 (STRICT DEPENDENCIES AND EQUIVALENCE CONSTRAINTS)
The condition in Lemma 7 stated above is sufficient but not necessary; we can utilize an existing strict equivalence constraint to draw the proper inferences. As in the above Lemma, consider an outer join query Q over two extended tables R and S

$$Q = R \overset{C_{R,S} \wedge C_S}{\longrightarrow} S$$

with On condition $p = C_{R,S} \wedge C_S$. Assume the strict functional dependency $f = a_i^S \longrightarrow a_j^S$ holds in S for every instance of the database, and a_i^S and a_j^S are distinct singleton attributes in $sch(S)$. Then if the strict equivalence constraint $e : a_i^S \stackrel{\omega}{=} a_j^S$ holds in S then f also holds as a strict dependency in $I(Q)$.

PROOF. The strict equivalence constraint ensures that if the determinant a_i^S is Null then the dependent attribute a_j^S must also be Null. However, if this is true then the generation of any all-Null row cannot produce a dependency violation, since in that tuple q_0 both $q_0[a_i^S]$ and $q_0[a_j^S]$ will be Null. □

As well as preserving strict dependencies, we can also exploit null-intolerant predicates in a left outer join's On condition to transform lax dependencies from the null-supplying side into strict dependencies. As the following Lemma illustrates, if the On condition p is such that each null-supplying tuple containing a null value for an attribute X is eliminated from the join, then a lax dependency $X \longmapsto Y$ can be strengthened (by inference rule FD6), as the generation of any all-Null row cannot violate the dependency.

LEMMA 8 (LAX DEPENDENCIES FROM A NULL-SUPPLYING TABLE)
Consider an outer join query Q over two extended tables R and S

$$Q = R \overset{C_{R,S} \wedge C_S}{\longrightarrow} S$$

where the On condition $p = C_{R,S} \wedge C_S$ consists of a conjunct C_S containing predicates solely referencing real attributes in the null-supplying table S, and $C_{R,S}$ containing those predicates referencing real attributes from both the preserved and null-supplying sides of the left outer join. Then if the lax functional dependency $g = a_i^S \longmapsto a_j^S$ (a_i^S and a_m^S not necessarily distinct, and a_i^S possibly composite) holds in S for every instance of the database and $\eta(p, a_k)$ evaluates to *true for each* $a_k \in (a_i^S \cup a_j^S)$ then g also holds in $I(Q)$ as a strict functional dependency, that is $g' = a_i^S \longrightarrow a_j^S$ holds in $I(Q)$.

PROOF. The proof is similar to the proof of Lemma 7, in that the requirement that $\eta(p, a_k)$ is true for each attribute in $(a_i^S \cup a_j^S)$ guarantees that g holds as a strict dependency since any tuples of $I(S)$ containing null values for any of $(a_i^S \cup a_j^S)$ are eliminated from the result. Elimination of these tuples guarantees that the all-Null row generated by the left outer join cannot violate g. □

3.2.9.2 Analysis of an On condition: left outer joins

In addition to the strict and lax functional dependencies from the left outer join's inputs that may hold in the result, additional dependencies that hold in the result can be

deduced from an analysis of the predicates that constitute the On condition. From Example 16 above we can make several observations about derived dependencies formed from a left outer join's On condition.

In Example 16 above, the equivalence predicate in the On condition p leads to the strict dependency P.PartID $\longrightarrow$ S.PartID and leads to the lax dependency S.PartID $\longmapsto$ P.PartID. The latter is a lax dependency because two or more rows from PART may not join with any rows of SUPPLY, resulting in two result rows with Null values for each attribute of SUPPLY. Such a result would violate a strict functional dependency S.PartID $\longrightarrow$ P.PartID according to Definition 26. Similarly, we cannot define a strict equivalence constraint between these two attributes. However, as with lax dependencies, we can define a lax equivalence constraint between the two part identifiers.

Example 17 offers several additional insights into the generation of additional dependencies. First, note that the constant comparison S.Rating = 'A' can generate only a lax dependency, since in the result Rating could be Null as part of an all-Null generated row. Second, while this null-intolerant condition may fail to evaluate to *true* for any rows of PART and SUPPLY that match on PartID, that failure cannot lead to a violation of the strict dependency P.PartID $\longrightarrow$ S.PartID. This is because an all-Null row is generated in a left outer join only when there are *no* tuples in the null-supplying extended table that can pair with a given tuple from the preserved side. Hence no two null-supplying rows of SUPPLY with different Rating values can join with the same row from PART, and the strict dependency holds. Third, note as well that the On condition implies that the strict dependency P.PartID $\longrightarrow$ S.Rating also holds in $I(Q)$. This is because any row from SUPPLY which successfully joins with a row from PART will have a Rating of 'A'; otherwise, a PART row that fails to join with any row of SUPPLY will generate an all-Null row. Furthermore, any null-intolerant comparison of two (or more) attributes from the null-supplying table also generate a strict equivalence constraint (and hence two strict functional dependencies) because for any tuple in the result either their values are equivalent, or they are both Null. Constant comparisons or other equality conditions involving only preserved attributes fail to generate additional dependencies themselves, since the semantics of a left outer join means that each tuple in the preserved table will appear in the result, *regardless of the comparison's success or failure.*

Aside. Our definitions of left-, right-, and full outer join restrict the join condition p such that $sch(p) \subseteq \alpha(S) \cup \alpha(T) \cup \kappa \cup \Lambda$. In the SQL standard [137], however, outer join conditions can also contain *outer references* to attributes from other tables in the table expression defined by the query's From clause. We assume that in this situation the algebraic expression tree representing the query is modified so that the extended tables (or table expressions) that supply the outer reference attribute(s) are added to the pre-

served side of the outer join without any change to the query semantics. Should this be impossible—as it is for full outer join—then we assume that only the functional dependencies that hold in the preserved extended table hold in the result, and we avoid any attempt to infer additional dependencies by analyzing the outer join's On condition.

Both Galindo-Legaria and Rosenthal [94, 97, 98] and Bhargava et al. [33, 34] exploit null-intolerant predicates to generate semantically equivalent representations of outer join queries. However, neither considered compound predicates in an outer join's On condition, and their effect on functional dependencies. In the following example, we similarly illustrate that null-tolerant predicates affect the inference of derived functional dependencies.

EXAMPLE 18 (NULL-TOLERANT PREDICATES IN AN ON CONDITION)
Consider a left outer join

$$Q = \mathcal{R}_\alpha(T \xrightarrow{p} S)$$

over extended tables T and S with real attributes $WXYZ \subset sch(T)$ and $ABCDE \subset sch(S)$ respectively, and where predicate p consists of the null-tolerant, true-interpreted condition $\lceil T.X = S.B \rceil$ which corresponds to the SQL statement

```
Select *
From   Rα(T) Left Outer Join Rα(S) On ( T.X = S.B is not false ).
```

Given the following instances of $\mathcal{R}_\alpha(T)$ and $\mathcal{R}_\alpha(S)$ (for brevity only the real attributes of each extended table are shown):

$\mathcal{R}_\alpha(S)$

A	B	C	D	E
a	b_1	c	d	e
a	Null	c	d	e

$\mathcal{R}_\alpha(T)$

W	X	Y	Z
w	b_1	y	z
w	b_2	y	z

The result Q of the outer join yields the three rows

Q

W	X	Y	Z	A	B	C	D	E
w	b_1	y	z	a	b_1	c	d	e
w	b_1	y	z	a	Null	c	d	e
w	b_2	y	z	a	Null	c	d	e.

Notice that with this database instance the strict dependency T.X ⟶ S.B does not hold in the result.

EXAMPLE 19

Consider a left outer join whose On condition involves more than one join predicate:

```
Select S.VendorID, S.PartID, Q.VendorID, Q.EffectiveDate, Q.QtyPrice
From   R_α(Supply) S Left Outer Join R_α(Quote) Q
       on ( S.PartID = Q.PartID and S.VendorID = Q.VendorID
            and Q.EffectiveDate > '1999-06-01' )
Where  Exists( Select *
               From   R_α(Vendor) V
               Where  V.VendorID = S.VendorID and
                      V.Address like '%Regina%' )
```

which lists the suppliers located in Regina, along with any quotes on parts supplied by them that are effective after 1 June 1999. In this example, the strict dependency S.VendorID ⟶ Q.VendorID may *not* hold in the result for every instance of the database. Consider the instances of tables SUPPLY and QUOTE below (assume that both SUPPLY tuples refer to a supplier located in Regina):

SUPPLY

Vendor ID	*Part* ID	*Rating*	*SupplyCode*
002	100	Null	'03AC'
002	200	'A'	Null

QUOTE

Vendor ID	*Part* ID	*EffectiveDate*	*QtyPrice*
002	100	1999-08-08	34.56

The result Q of the outer join yields the two rows

Q

S.Vendor ID	*S.Part* ID	*Q.Vendor* ID	*EffectiveDate*	*QtyPrice*
002	100	002	1999-08-08	34.56
002	200	Null	Null	Null.

Notice that with this database instance the strict dependency S.VendorID ⟶ Q.VendorID does not hold in the result.

Why does this example seemingly contradict the observations made in Examples 16 and 17? The root of problem for dependencies implied by an On condition in a left outer join is that the failure of *any* conjunct in the On condition to evaluate to *true* for a pair of tuples from the preserved and null-supplying sides may lead to the generation of an all-Null row. To put it another way, it is each *combination* of preserved attributes that will either successfully join with a null-supplying tuple, or cause the generation of an all-Null row.

LEMMA 9 (DEPENDENCIES IMPLIED BY A LEFT OUTER JOIN'S ON CONDITION)
The set of dependencies formed from the **On** condition p of a left outer join $R \xrightarrow{p} S$ can be derived as follows. First, determine the set of dependencies[17] $\mathcal{F}$ that would be implied if p was a restriction condition (see Section 3.2.4). For each such strict functional dependency $f : X \longrightarrow Y$ in $\mathcal{F}$, proceed as follows:

- *Case 1.* If $XY \subseteq \alpha(R) \cup \kappa(p)$, then eliminate f from $\mathcal{F}$, as the condition may not necessarily hold in the result since R is preserved[18];
- *Case 2.* If $XY \subseteq \alpha(S)$ and $\eta(p, X)$ is *true* then retain f as a strict dependency;
- *Case 3.* If $X \subseteq \alpha_R(p) \cup \kappa(p)$, $\alpha_R(p)$ is not empty, $Y \subseteq \alpha_S(p)$, and $\eta(p, Y)$ is *true* then introduce the strict functional dependency $g : \alpha_R(p) \longrightarrow Y$ and mark f as a lax functional dependency $X \longmapsto Y$.

  Note that each preserved attribute must be included as part of the determinant of g; this would include, for example, any references to these attributes in a conjunctive or disjunctive condition, or an outer reference to one or more preserved attributes embedded in nested **Exists** predicates that are part of the **On** condition.
- *Case 4.* Otherwise, mark f as a lax dependency $X \longmapsto Y$. In practice, the bulk of conjunctive **On** condition predicates, such as those in Examples 16 and 17, fall into this category.

For each lax functional dependency $f : X \longmapsto Y$ in $\mathcal{F}$ as defined above:

- *Case 1.* If $XY \subseteq \alpha(R) \cup \kappa(p)$, then eliminate f from $\mathcal{F}$.
- *Case 2.* Otherwise retain f as a lax dependency.

PROOF. Omitted. □

---

17 Recall that both strict and lax functional dependencies in $\mathcal{F}$ have singleton right-hand sides.

18 In this context we are treating a correlation attribute from a parent query block in the case of a nested query specification as a constant value, i.e. the correlation attribute is an *outer reference*. For further details, see references [72, 137, 200].

EXAMPLE 20
Consider a left outer join

$$Q = \mathcal{R}_\alpha(T \xrightarrow{p} S)$$

over extended tables T and S with real attributes $WXYZ \subset sch(T)$ and $ABCDE \subset sch(S)$), respectively, with the outer join predicate

$$p = \lfloor T.X = S.B \wedge T.Y = S.C \wedge T.Z = 5 \wedge S.A = S.B \wedge S.D = 2 \rfloor$$

which corresponds to the SQL statement

```
Select *
From   Rα(T) Left Outer Join Rα(S)
          On ( T.X = S.B and T.Y = S.C and T.Z = 5 and
               S.A = S.B and S.D = 2 )
```

If treating p as a restriction condition, each equality condition would generate two strict functional dependencies and a strict equivalence condition. By following the construction above for left outer joins, the set of dependencies $\mathcal{F}$ implied by p is:

1. $T.X \longmapsto S.B$
2. $S.B \longmapsto T.X$
3. $T.Y \longmapsto S.C$
4. $S.C \longmapsto T.Y$
5. $S.A \longrightarrow S.B$
6. $S.B \longrightarrow S.A$
7. $2 \longmapsto S.D$
8. $S.D \longmapsto 2$
9. $\{T.X, T.Y, T.Z\} \longrightarrow S.B$
10. $\{T.X, T.Y, T.Z\} \longrightarrow S.C$, and
11. $\{T.X, T.Y, T.Z\} \longrightarrow S.D$.

Lax functional dependencies, lax equivalence constraints, and the nullability of an attribute are all crucial in inferring additional dependencies that hold in the result of a left outer join. Eliminating lax functional dependencies altogether from the analysis loses information that may be pertinent to query optimization, a straightforward example being the conversion of outer joins to inner joins[19]. In addition, a left outer join implies a *null constraint* among the definite attributes from the null-supplying extended table:

DEFINITION 38 (NULL CONSTRAINT)
Consider an extended table R with instance $I(R)$. A null constraint over R is a statement of the form $X \dashv Y$ for $X \cup Y \subset sch(R)$. An instance $I(R)$ satisfies $X \dashv Y$ if for every tuple $r_0 \in I(R)$, if $r_0[X]$ is `Null` then $r_0[Y]$ is `Null`.

Because of our interest in the all-`Null` row, we find the notion of null constraint preferable to its more traditional contrapositive form:

DEFINITION 39 (EXISTENCE CONSTRAINT)
Consider an extended table R and a specific instance $I(R)$. An *existence constraint* [193, pp. 385] over R is a statement of the form $X \vdash Y$ (read 'X' requires 'Y') for $X \cup Y \subset sch(R)$. An instance $I(R)$ satisfies $X \vdash Y$ if for every tuple $r_0 \in I(R)$, if $r_0[X]$ is definite then $r_0[Y]$ is definite.

Consider a left outer join $Q = R \xrightarrow{p} S$ over extended tables R and S with attributes $\{a_1, a_2, \ldots, a_n\} \subset sch(R)$ and similarly $\{b_1, b_2, \ldots, b_n\} \subset sch(S)$. Let $I(R)$ and $I(S)$ denote specific instances of R and S. Now consider any two definite attributes b_i and b_j. The null constraint $b_i \dashv b_j$ holds in $I(Q)$, since only an all-`Null` row of S can contain null values for the corresponding attributes $b_i, b_j \in sch(Q)$.

Null constraints provide an opportunity to convert nullable attributes to definite attributes when a nullable attribute is referenced as part of a conjunctive, null-intolerant restriction predicate. If that attribute forms the head of a null constraint, then any attributes in the body which are definite but for the all-`Null` row can be similarly converted. This conversion may subsequently yield additional strict functional dependencies and equivalence constraints through their strengthening from lax dependencies and lax equivalence constraints.

We summarize all the dependencies and constraints that result from a left outer join in the following theorem:

---

19 Inner joins are typically preferred over outer joins by most query optimizers as they permit a larger space of possible access plans in which to find the 'optimal' join order [33].

THEOREM 3 (SUMMARY OF CONSTRAINTS IN A LEFT OUTER JOIN)
Given a left outer join expression $Q = S \xrightarrow{p} T$ over extended tables S and T with `On` condition p, the following functional dependencies, equivalence constraints, and null constraints hold in Q:

- *Strict functional dependencies:*
  1. Any strict functional dependency $f : X \longrightarrow Y$ that held in S continues to hold in Q.
  2. Any strict functional dependency f that held in T will continue to hold in Q if either $\eta(p, X)$ evaluates to *true* or there exists a strict equivalence constraint $e : X \overset{\omega}{=} Y$ that held in T. (Note that if X is a set, $\eta(p, X)$ will evaluate to *true* if any $x \in X$ cannot be `Null`.)
  3. If p would have produced the strict functional dependency $f : X \longrightarrow Y$ when treated as a restriction condition and $XY \subseteq \alpha(T)$ and $\eta(p, X)$ is *true* then the strict functional dependency $X \longrightarrow Y$ holds in Q.
  4. If p would result in a strict functional dependency $f : X \longrightarrow Y$ when treated as a restriction condition, and $X \subseteq \alpha_S(p) \cup \kappa(p)$, $\alpha_S(p)$ is not empty, $Y \subseteq \alpha_T(p)$, and $\eta(p, Y)$ is *true* then $\alpha_S(p) \longrightarrow Y$ holds in Q.
  5. The newly-constructed tuple identifier $\iota(Q)$ strictly determines both $\iota(S)$ and $\iota(T)$, and $(\iota(S) \cup \iota(T)) \longrightarrow \iota(Q)$.

- *Lax functional dependencies:*
  1. Any lax functional dependency $f : X \longmapsto Y$ that held in S continues to hold in Q.
  2. Any lax functional dependency f that held in T will continue to hold in Q.
  3. Any strict functional dependency $f : X \longrightarrow Y$ that held in T will hold as a lax functional dependency $X \longmapsto Y$ in Q if $\eta(p, X)$ evaluates to *false* and, if both X and Y are singleton attributes, there does not exist a strict equivalence constraint $e : X \overset{\omega}{=} Y$ that held in T.
  4. If p would have produced either the functional dependency $X \longrightarrow Y$ or $X \longmapsto Y$ when treated as a restriction condition and $XY \cap \alpha(T)$ is not empty then $X \longmapsto Y$ in Q.

- *Strict equivalence constraints:*
  1. Any strict equivalence constraint $e : X \overset{\omega}{=} Y$ that held in S continues to hold in Q.

2. Any strict equivalence constraint e that held in T will continue to hold in Q.
3. If $\eta(p, X)$ is *true* and $\eta(p, Y)$ is *true* for singleton attributes $XY \subseteq sch(T)$ then a lax equivalence constraint $e : X \simeq Y$ that held in T will hold as as strict equivalence constraint in Q.
4. If p would have produced the strict equivalence constraint $e : X \stackrel{\omega}{=} Y$ when treated as a restriction condition and $XY \subseteq \alpha(T)$ then $X \stackrel{\omega}{=} Y$ holds in Q.

- *Lax equivalence constraints:*

1. Any lax equivalence constraint $e : X \simeq Y$ that held in S continues to hold in Q.
2. Any lax equivalence constraint e that held in T will continue to hold in Q.
3. If p would have produced either the equivalence constraint $X \stackrel{\omega}{=} Y$ or $X \simeq Y$ when treated as a restriction condition and $XY \cap \alpha(T) \neq \emptyset$ then $X \simeq Y$ holds in Q.

- *Null constraints:*

1. Any null constraint $X \dashv Y$ that held in S continues to hold in Q.
2. Any null constraint $X \dashv Y$ that held in T continues to hold in Q.
3. For each pair of definite attributes X and Y where $XY \subseteq \alpha(T)$ the null constraint $X \dashv Y$ holds in Q.

PROOF. Follows from Lemmas 7, 8, and 9, and Corollary 2 (page 88). □

Null constraints with other algebraic operators. Much like equivalence constraints, it is easy to see that each algebraic operator maintains the null constraints that hold in their input(s), with two notable exceptions. Both restriction and intersection can mark an attribute X as definite. If the null constraint $X \dashv Y$ holds, then Y, along with any other attribute that directly or transitively is part of a null constraint with X, can similarly be made definite.

3.2.10 Full outer join

With full outer joins, each side of the join is both preserved and null-supplying, hence an all-`Null` row can be generated for either input should the `On` condition fail to evaluate to *true* for a row from either input. The following example illustrates the semantics of full outer join.

EXAMPLE 21 (FULL OUTER JOIN)
Suppose we have the algebraic expression

$$Q = \mathcal{R}_\alpha(\pi_{All}[a_1^P, a_2^P, a_1^S, a_3^S, a_4^S](P \overset{P}{\longleftrightarrow} S))$$

that represents the query

```
Select P.PartID, P.Description, S.VendorID, S.Rating, S.SupplyCode
From   Rα(Part) P Full Outer Join Rα(Supply) S
       on ( P.PartID = S.PartID )
```

which is a modified version of the query in Example 16. The query lists all parts and supplier information, joining the two when they agree on the part identifier, and otherwise generating an all-Null row for either input[20]. Given a database instance where the PART and SUPPLY tables are as follows (for brevity only the relevant real attributes have been included):

PART

Part ID	*Description*	*Price*
100	'Bolt'	0.09
200	'Flange'	0.37
300	'Tapcon'	0.23

SUPPLY

Vendor ID	*Part* ID	*Rating*	*SupplyCode*
002	100	Null	'03AC'
011	Null	'A'	Null
011	401	'A'	Null

then the result Q consists of the five rows

Q

P.Part ID	*Description*	*Vendor* ID	*S.Part* ID	*Rating*	*SupplyCode*
100	'Bolt'	002	100	Null	'03AC'
200	'Flange'	Null	Null	Null	Null
300	'Tapcon'	Null	Null	Null	Null
Null	Null	011	Null	'A'	Null
Null	Null	011	401	'A'	Null

20 For this query to really make sense, we would require schema changes to remove the primary key constraint on SUPPLY and to remove the referential integrity constraint between SUPPLY and PART. This would permit the insertion of a SUPPLY tuple with an invalid or Null part identifier.

3.2.10.1 Input dependencies and full outer joins

Because each input table to a full outer join is null-supplying, any strict functional dependency that holds in either input can remain strict only if its determinant cannot be wholly non-Null—that is, for any dependency $f : X \longrightarrow Y$ that holds in either input of an outer join $S \overset{p}{\longleftrightarrow} T$, $\eta(p, X)$ must be *true*. Otherwise, it is possible that the generation of an all-Null row will lead to a dependency violation, and f must be converted to its lax counterpart, $X \longmapsto Y$.

Similarly, a strict candidate superkey of the result can exist only if the generation of any all-Null row cannot violate the key dependency of either input. Hence $\eta(p, K)$ must be *true* for each candidate key K from either input, in order to combine the two keys to a candidate key of the result. Otherwise, a lax key dependency can hold in the result, which by Definition 28 is unaffected by the generation of an all-Null row.

3.2.10.2 Analysis of an On condition: full outer joins

In the case of dependencies implied by the full outer join's On condition, the problem is that both inputs are *both* preserved *and* null-supplying in the result—therefore an arbitrary condition in p will fail to restrict either input[21]. Consequently almost any dependency between attributes of the two inputs, either strict or lax, derived from the clauses in the On condition will not necessarily hold for every instance of the database.

LEMMA 10 (DEPENDENCIES IMPLIED BY A FULL OUTER JOIN)
The set of dependencies formed from the On condition p of a full outer join $R \overset{p}{\longleftrightarrow} S$ can be derived as follows. First, determine the set of dependencies $\mathcal{F}$ that would be implied if p was a restriction condition (see Section 3.2.4). For each such strict functional dependency $f : X \longrightarrow Y$ in $\mathcal{F}$, proceed as follows:

- *Case 1.* If $XY \subseteq \alpha(R) \cup \kappa(p)$, then eliminate f from $\mathcal{F}$, as the condition may not necessarily hold in the result since R is preserved.
- *Case 2.* Similarly, if $XY \subseteq \alpha(S) \cup \kappa(p)$ then eliminate f from $\mathcal{F}$.

---

21 This observation makes it clear that both Theorem 9 and Corollary 10 in a recent ANSI standard change proposal [303, pp. 24] are erroneous. Because both inputs to a full outer join are preserved, any equality comparison in the outer join's On condition that pertains only to either input will not necessarily imply a dependency since the equality condition will not restrict either input.

- *Case 3.* If $X \subseteq \alpha_R(p) \cup \kappa(p)$, $\alpha_R(p)$ is not empty, $Y \subseteq \alpha_S(p)$, $\eta(p, Y)$ is *true*, and $\eta(p, \alpha_S(p))$ is *true* then introduce the strict functional dependency $g : \alpha_R(p) \longrightarrow Y$ and mark f as a lax functional dependency $X \longmapsto Y$.

  As with left outer joins, each attribute in $\alpha_S(p)$ must be included as part of the determinant of g; this would include, for example, any references to these attributes in a conjunctive or disjunctive condition, or an outer reference to one or more preserved attributes embedded in nested **Exists** predicates that are part of the On condition.

- *Case 4.* Otherwise, mark f as a lax dependency $X \longmapsto Y$.

For each lax functional dependency $f : X \longmapsto Y$ in $\mathcal{F}$ as defined above:

- *Case 1.* If $XY \subseteq \alpha(R) \cup \kappa(p)$, then eliminate f from $\mathcal{F}$.
- *Case 2.* If $XY \subseteq \alpha(S) \cup \kappa(p)$, then eliminate f from $\mathcal{F}$.
- *Case 3.* Otherwise retain f as a lax dependency.

PROOF. Omitted. □

THEOREM 4 (SUMMARY OF CONSTRAINTS IN A FULL OUTER JOIN)
Given a full outer join expression $Q = R \overset{p}{\longleftrightarrow} S$ over extended tables R and S with On condition p, the following functional dependencies, equivalence constraints, and null constraints hold in Q:

- *Strict functional dependencies:*

  1. Any strict functional dependency $f : X \longrightarrow Y$ that held in R or S will continue to hold in Q if either (1) both X and Y are singleton attributes and there exists the strict equivalence constraint $X \overset{w}{=} Y \in \mathcal{E}_R$ (or $\mathcal{E}_S$), or (2) $\eta(p, X)$ evaluates to *true*. Once again, note that since X is a set, $\eta(p, X)$ will evaluate to *true* if any $x \in X$ cannot be **Null**.
  2. If p would have produced the strict functional dependency $f : X \longrightarrow Y$ when treated as a restriction condition and $X \subseteq \alpha_R(p) \cup \kappa(p)$, $\alpha_R(p)$ is not empty, $Y \subseteq \alpha_S(p)$, $\eta(p, Y)$ is *true*, and $\eta(p, \alpha_S(p))$ is *true* then the strict functional dependency $\alpha_R(p) \longrightarrow Y$ holds in Q.
  3. The newly-constructed tuple identifier $\iota(Q)$ strictly determines both $\iota(R)$ and $\iota(S)$, and $(\iota(R) \cup \iota(S)) \longrightarrow \iota(Q)$.

- *Lax functional dependencies:*

  1. Any lax functional dependency $f : X \longmapsto Y$ that held in R continues to hold in Q.
  2. Similarly, any lax functional dependency f that held in S will continue to hold in Q.
  3. If p would have produced either the functional dependency $X \longrightarrow Y$ or $X \longmapsto Y$ when treated as a restriction condition and $X \cap \alpha(R)$ is not empty and $Y \cap \alpha(S)$ is not empty then $X \longmapsto Y$ in Q.
  4. If p would have produced either the functional dependency $X \longrightarrow Y$ or $X \longmapsto Y$ when treated as a restriction condition and $X \cap \alpha(S)$ is not empty and $Y \cap \alpha(R)$ is not empty then $X \longmapsto Y$ in Q.

- *Strict equivalence constraints:*

  1. Any strict equivalence constraint $e : X \stackrel{\omega}{=} Y$ that held in R continues to hold in Q.
  2. Similarly, any strict equivalence constraint e that held in S will continue to hold in Q.

- *Lax equivalence constraints:*

  1. Any lax equivalence constraint $e : X \simeq Y$ that held in R continues to hold in Q.
  2. Any lax equivalence constraint e that held in S will continue to hold in Q.
  3. If p would have produced either the equivalence constraint $X \stackrel{\omega}{=} Y$ or $X \simeq Y$ when treated as a restriction condition and $X \in \alpha(R)$ and $Y \in \alpha(S)$ then $X \simeq Y$ holds in Q.

- *Null constraints:*

  1. Any null constraint $X \dashv Y$ that held in R continues to hold in Q.
  2. Any null constraint $X \dashv Y$ that held in S continues to hold in Q.
  3. For each pair of definite attributes X and Y where $XY \subseteq \alpha(R)$ or $XY \subseteq \alpha(S)$ the null constraint $X \dashv Y$ holds in Q.

PROOF. Omitted. □

3.3 Graphical representation of functional dependencies

In order to determine what functional dependencies hold in a derived relation we need to represent a set of dependencies $\mathcal{F}$. Ausiello, D'Atri, and Saccà [19, 20] define an FD-graph as a modified directed hypergraph that models a set of functional dependencies in simplified form (see Figure 3.1). Their definition is as follows. The FD-graph $G = \langle V, E \rangle$ that represents a set of dependencies $\mathcal{F}$ in a relation R with scheme $R(U)$ such that:

1. for every attribute $A \in U$ there is a vertex in $V^0[G]$ labeled A (termed a *simple* vertex);
2. for every dependency $X \longrightarrow A \in F$ where $X \subseteq U$ and $\|X\| > 1$ there is a vertex in $V^1[G]$ labeled X (termed a *compound* vertex);
3. for every dependency $X \longrightarrow Y$ where $Y = \{A_1, A_2, \ldots, A_n\}$ and $X \cup Y \subseteq U$ there are edges in $E[G]$ labeled with '0', termed *full arcs*, from the vertex labeled X to each vertex $A_1, A_2, \ldots, A_n$ (consequently compound vertices can exist only as determinants);
4. for every compound vertex $X \in V[G]$ there are edges in $E[G]$, labeled with '1' and termed *dotted arcs*, from X to each of its component (simple) vertices $A_1, A_2, \ldots, A_n$.

The combination of compound vertices and dotted arcs provide the 'hypergraph' flavor of an FD-graph, as together they constitute a hypervertex. Edges in this hypergraph represent only strict functional dependencies.

For clarity, henceforth we will use slightly different notation for attributes and strict dependencies in an FD-graph than described above. We relabel full arcs with 'F' (to denote a functional dependency) and dotted arcs with 'C' (to denote an edge to a component vertex from its compound 'parent'). Similarly, simple vertices are in the set V^A and compound vertices in the set V^C. Table 3.1 contains the revised notation for FD-graphs.

With this construction of an FD-graph we can not only represent the dependencies in $\mathcal{F}$, but we can also determine those transitive dependencies that hold in $\mathcal{F}^+$. Consider an FD-graph G that contains only simple vertices so that $V^C = E^C = \emptyset$. Starting at an arbitrary vertex X, by following directed edges through G one can easily determine the closure of X (X^+) with respect to the dependencies represented in G. Ausiello et al. term such a path through G an FD-*path*. Once we introduce compound vertices, however, the definition of what constitutes an FD-*path* becomes slightly more complex as we have to take the existence of E^C edges into account. For example, if $\mathcal{F} = \{A \longrightarrow B, A \longrightarrow C, BC \longrightarrow D\}$ we need to be able to infer the transitive dependency $A \longrightarrow D \in F^+$.

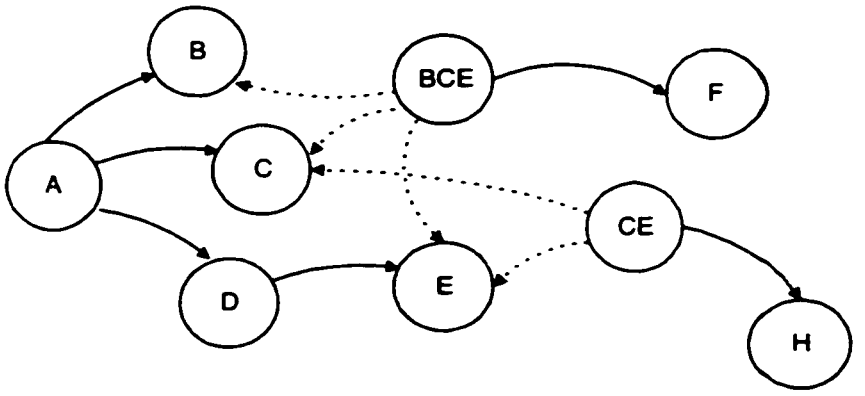
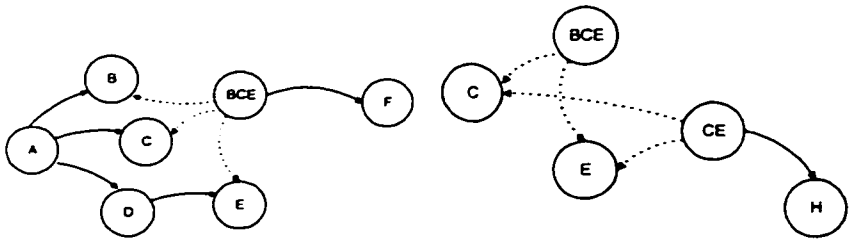


FIGURE 3.1: An example of an FD-graph [19], representing the set of functional dependencies $\mathcal{F} = \{A \longrightarrow BCD, D \longrightarrow E, BCE \longrightarrow F, CE \longrightarrow H\}$. It should be clear from the graph that A functionally determines each attribute in the graph, either directly or transitively.

Symbol	*Definition*
G	an FD-graph, i.e. $G = \langle V, E \rangle$.
V	The set of vertices in G, where $V = V^A \cup V^C$.
V^A	the set of vertices that represent a single attribute.
V^C	the set of vertices that represent a compound attribute.
E	the set of edges in G, where $E = E^F \cup E^C$.
E^F	the set of full (unbroken) edges in E that represent a strict functional dependency.
E^C	the set of dotted edges in E that relate compound vertices to their components (simple vertices).

TABLE 3.1: Notation for an FD-graph, adopted from reference [19].



(a) Full FD-path $\langle A, F \rangle$. (b) Dotted FD-path $\langle BCE, H \rangle$.

FIGURE 3.2: Full and dotted FD-paths [19].

DEFINITION 40 (BASIC FD-PATH FROM AUSIELLO ET AL. [19])
Consider an FD-graph G and any two vertices $i, j \in V^A \cup V^C\}$. An FD-path from i to j is a minimal subgraph $G' = \langle V' \subseteq (V^A \cup V^C), E' \subseteq (E^F \cup E^C) \rangle$ of G such that $i, j \in V'$ and either

1. there exists an edge directly linking i and j, i.e. $((i, j) \in E')$, or
2. j is a simple vertex and there exists a vertex k such that the directed edge $(k, j) \in E'$ and there is an FD-path $\langle i, k \rangle$ in G', or
3. j is a compound vertex with components $a_1, a_2, \ldots, a_n$ and there are corresponding n dotted arcs (in the set E'^C) for each, i.e. $(j, a_1), \ldots, (j, a_n) \in E'$, and n FD-paths $\langle i, a_1 \rangle, \langle i, a_2 \rangle, \ldots, \langle i, a_n \rangle$ in G'.

Ausiello et al. term an FD-path $\langle i, j \rangle$ a *dotted* FD-path if all of the outgoing edges of i are dotted, that is, i is a compound vertex and all of its outgoing arcs are in the set E^C. Otherwise, the FD-path is termed *full* (see Figure 3.2).

In the next section we define a modified version of FD-graph for extended tables that represents a set of functional dependencies and equivalence constraints in simplified form for which we can infer additional dependencies as required.

3.3.1 Extensions to FD-graphs

Unfortunately, the basic form of FD-graph defined by Ausiello, D'Atri, and Saccà [19, 20] falls short of our requirements for representing derived functional dependencies that we can exploit during the optimization of ANSI SQL expressions. Two such requirements were

introduced in Chapter 2: key dependencies and lax dependencies. Other requirements, such as the need to maintain strict and lax equivalence constraints, were introduced in Section 3.1. In this section we briefly describe our extensions to FD-graphs to capture both lax and strict dependencies and lax and strict equivalence constraints for a given algebraic expression over extended tables.

3.3.1.1 Keys

We add tuple identifiers to FD-graphs, mirroring our definition of an extended table (Definition 3) where we assume the existence of a unique tuple identifier for each tuple in a (base or derived) table. For each base table, we add a vertex representing a tuple identifier for that table (see Section 3.4.1 below). This vertex will belong to a new set of vertices denoted V^R. To represent strict superkeys of an extended table, we add strict dependency edges between each single or compound vertex that represent its attributes and the vertex representing the table's tuple identifier.

For complex expressions involving multiple intermediate table subexpressions, the tuple identifier that denotes a tuple in the result of the combined expression will be represented by a hypervertex $v_k \in V^R$, with edges $(v_{e_i}, v_k) \in E^R$ that relate each subexpression's tuple identifier to v_k, in a manner similar to that of edges in E^C for compound vertices in V^C. This construction essentially denotes the Cartesian product of the subexpressions e_i. One important difference between edges in E^C and E^R is that the target of edges in E^C must be simple vertices, but the targets of edges in E^R can be *either* simple or compound tuple identifier vertices. For tuple identifiers this means that the right-hand side of a dependency need not be a 'simple' vertex (although it refers to a singleton tuple identifier attribute).

With the addition of tuple identifiers to the set of vertices maintained in an FD-graph, our definition of FD-path must be modified accordingly.

DEFINITION 41 (STRICT FD-PATH)
Consider an FD-graph G and any two vertices $i, j \in V^A \cup V^C \cup V^R$. A *strict* FD-*path* from i to j is a minimal subgraph $G' = \langle V' \subseteq (V^A \cup V^R \cup V^C), E' \subseteq (E^C \cup E^F \cup E^R) \rangle$ of G such that $i, j \in V'$ and either

1. there exists an edge directly linking i and j, i.e. $((i, j) \in E')$, or
2. $j \in V^A \cup V^R$ and there exists a vertex $k \in V'$ such that the directed edge $(k, j) \in E'$ and there is a strict FD-path $\langle i, k \rangle$ in G', or

3. j is a compound vertex with components $a_1, a_2, \ldots, a_n$ and there are corresponding n dotted arcs (in the set $E^{C'}$) for each, i.e. $(j, a_1), \ldots, (j, a_n) \in E'$, and n strict FD-paths $\langle i, a_1 \rangle, \langle i, a_2 \rangle, \ldots, \langle i, a_n \rangle$ in G', or

4. j is a compound tuple identifier vertex in V^R representing component subexpressions $e_1, e_2, \ldots, e_n$ such that there are corresponding n dotted arcs (in the set $E^{R'}$) for each component tuple identifier, i.e. $(k, e_1), (k, e_2), \ldots, (k, e_n) \in E'$, and n strict FD-paths $\langle i, e_1 \rangle, \langle i, e_2 \rangle, \ldots, \langle i, e_n \rangle$ in G'.

CLAIM 24
A strict FD-path embodies those inference rules that are applicable for strict functional dependencies. In particular,

- Item (1) embodies the inference rule for reflexivity (FD1) on compound determinants, as well as encoding the given dependencies, including the uniqueness of tuple identifiers;

- Item (2) embodies the inference rules for strict transitivity (FD7A) and strict decomposition (FD4A);

- Items (3) and(4) embody rule FD3A (strict union).

By definition, a strict FD-path $G' = \langle X, Y \rangle$ is acyclic since G' must be minimal; hence the edges in G' form a spanning tree.

3.3.1.2 Real and virtual attributes

To correctly represent the schema of an extended table R, simple vertices in V^A that represent real attributes in $\alpha(R)$ are coloured 'white'. Virtual attributes—those in the sets $\iota(R)$, $\kappa(R)$, and $\rho(R)$—are coloured either 'gray' or 'black'. The sole vertex in V^R which denotes the tuple identifier of the algebraic expression ($\iota(R)$) modelled by the FD-graph is coloured gray; vertices in V^R that represent tuple identifiers of subexpressions which are in the set $\rho(R)$ are coloured 'black'. Vertices in V^A which represent constants, denoted V^A_κ, that appear in equality conditions of restriction predicates (see Section 3.3.1.4 below) are coloured 'gray'. Other virtual attributes in $\rho(R)$, which typically result from schema modifications for those algebraic operators (projection, distinct projection, partition, intersection, and difference) that remove attributes from a result, are represented by simple vertices in V^A that are coloured black.

3.3.1.3 Nullable attributes

Each vertex in V^A is marked as either *Definite* or *Nullable*, corresponding to our definitions of definite or nullable attributes. For base tables, we assume that we have access to the table's definition to determine if a `Not Null` constraint exists on any attribute (see Section 3.4.1) . We maintain the 'nullability' property for each attribute as we construct the FD-graph for the algebraic expression representing the SQL query. Our purpose is to eliminate the possibility of null values for as many determinants as possible, which will then lead to the conversion of lax dependencies into strict ones (see below).

3.3.1.4 Equality conditions

We augment FD-graphs for equality conditions as follows. First, instead of maintaining only real attributes in FD-graphs, we also include (simple) vertices of constant values, which can functionally determine any other (simple) attribute. We denote the set of constants referenced in predicates included in e with the notation V_κ^A. These constants, coloured gray, represent the virtual attributes $\kappa(R)$ in an extended table. Second, we introduce a third type of (undirected) edge (the set of which we denote as E^E), which we term an *equivalence edge*, that represents a strict equivalence constraint between two simple vertices. We will continue to maintain strict FD-paths consisting of the edges in E^F and E^C in these cases as well to simplify the dependency maintenance algorithms. We will also require the ability to infer transitive equalities in an FD-graph, as we did for implied functional dependencies:

DEFINITION 42 (STRICT EQUIVALENCE-PATH)
Consider an FD-graph G and any two vertices $i, j \in V^A$. A strict equivalence-path from i to j is a minimal subgraph $G' = \langle V' \subseteq V^A, E' \subseteq E^E \rangle$ of G such that $i, j \in V'$ and either

1. there exists a strict equivalence edge directly linking i and j, i.e. $((i, j) \in E')$, or
2. there exists a vertex $k \in V'$ such that the directed edge $(k, j) \in E'$ and there is an equivalence-path $\langle i, k \rangle$ in G'.

CLAIM 25
A strict equivalence-path embodies those inference rules that are applicable for strict equivalence constraints. In particular, item (2) embodies inference rule EQ3 (strict transitivity).

3.3.1.5 Lax functional dependencies

The original construction for FD-graphs handled only strict functional dependencies. In our extended implementation of FD-graphs, lax dependencies are represented by edges in the set E^f, which have similar characteristics to their strict counterparts in the set E^F. Lax dependencies will appear in diagrams of FD-graphs similarly to their strict dependency edges, only they will be labelled with the letter 'L'.

DEFINITION 43 (LAX FD-PATH)
Consider an FD-graph G and any two vertices $i, j \in V' \equiv (V^A \cup V^C \cup V^R)$. A lax FD-path from i to j is a minimal subgraph $G' = \langle V' \subseteq (V^A \cup V^C \cup V^R), E' \subseteq (E^F \cup E^f \cup E^C \cup E^R) \rangle$ of G such that $i, j \in V'$ and either

1. there exists an edge directly linking i and j, i.e. $((i,j) \in E')$, or
2. $j \in V^A \cup V^R$ and there exists a *strict* FD-path $\langle i, j \rangle$ in G', or
3. $j \in V^A \cup V^R$ and there exists a vertex k, such that either
   (a) $k \in V^A$ and k is definite and the directed edge $(k,j) \in E^F \cup E^f$, or
   (b) $k \in V^R$ such that the directed edge $(k,j) \in E^F$, or
   (c) $k \in V^C$ with definite components $a_1, a_2, \ldots, a_n$ and the directed edge $(k,j) \in E^F \cup E^f$

   and there is a lax FD-path $\langle i, k \rangle$ in G', or
4. j is a compound vertex with definite components $a_1, a_2, \ldots, a_n$ and there are corresponding n dotted arcs (in the set $E^{C'}$) for each, i.e. $(j, a_1), \ldots, (j, a_n) \in E'$, and n lax FD-paths $\langle i, a_1 \rangle, \langle i, a_2 \rangle, \ldots, \langle i, a_n \rangle$ in G', or
5. j is a compound tuple identifier vertex in V^R representing component subexpressions $e_1, e_2, \ldots, e_n$ and there are corresponding n dotted arcs (in the set $E^{R'}$) for each component tuple identifier, i.e. $(k, e_1), (k, e_2), \ldots, (k, e_n) \in E'$, and n lax FD-paths $\langle i, e_1 \rangle, \langle i, e_2 \rangle, \ldots, \langle i, e_n \rangle$ in G'.

As with strict FD-paths, we term a lax FD-path $\langle i, j \rangle$ a *dotted* lax FD-path if all of the outgoing edges of i are dotted, that is, i is a compound vertex and all of its outgoing arcs are in the set E^C. Otherwise, the lax FD-path is termed *full*.

CLAIM 26
A lax FD-path embodies those inference rules that are applicable for lax functional dependencies. In particular,

- As with strict dependencies, item (1) in Definition 43 above embodies the inference rule for reflexivity (FD1) on compound determinants, as well as encoding the given dependencies:
- Item (2) embodies inference rule FD5 (weakening);
- Each item in (3) embodies the inference rule for lax transitivity (FD7B), and in addition Item (3c) embodies the rule for lax union (FD3B).
- Items (4) and (5) embody rule FD3B (lax union).

3.3.1.6 Lax equivalence constraints

In an FD-graph, lax equivalence constraints are represented by edges in the set E^e, whose construction is similar to their strict counterparts in E^E. Like lax functional dependencies, lax equivalence edges will appear in diagrams of FD-graphs similarly to their strict counterparts, only they will be labelled with the letter 'L'.

DEFINITION 44 (LAX EQUIVALENCE-PATH)
Consider an FD-graph G and any two vertices $i, j \in V^A$. A lax equivalence-path from i to j is a minimal subgraph $G' = \langle V' \equiv V^A, E' \equiv (E^E \cup E^e) \rangle$ of G such that $i, j \in V'$ and either

1. there exists an equivalence edge directly linking i and j, i.e. $((i, j) \in E'$, or
2. there exists a vertex $k \in V'$ such that the undirected edge $(k, j) \in E^e$ and there is a strict equivalence-path $\langle i, k \rangle$ in G', or
3. there exists a vertex $k \in V'$ such that the undirected edge $(k, j) \in E^E$ and there is a lax equivalence-path $\langle i, k \rangle$ in G', or
4. there exists a definite vertex $k \in V'$ such that the undirected edge $(k, j) \in E^e$ and there is a lax equivalence-path $\langle i, k \rangle$ in G'.

CLAIM 27
A lax equivalence-path embodies those inference rules that are applicable for lax equivalence constraints. In particular,

- Item (1) embodies inference rule EQ6 (weakening);
- Items (2), (3), and (4) embody the inference rule for lax transitivity (EQ9A).

3.3.1.7 Null constraints

To model outer joins we introduce an additional type of vertex (in the set V^J) and a new set of edges (E^J) to track the *origin* of a lax functional dependency introduced by an outer join (see Section 3.4.8 below). These new FD-graph components are necessary to enable the discovery of additional strict or lax dependencies when, through query analysis, we can determine that an all-`Null` row from the outer join's null-supplying side will be eliminated from the query's result.

To complete the mechanism for converting lax dependencies to strict ones, we introduce another value for the 'nullability' property of each null-supplying vertex in V^A: 'Pseudo-definite'. This value will be used to mark the nullability of a vertex v when $\eta(p, v)$ is *true*, meaning that it is only a generated all-`Null` row that can produce a null value for this attribute; otherwise its values are guaranteed to be definite. Such null-supplying attributes form a null constraint with other pseudo-definite null-supplying attributes (see Definition 38).

Using the null-supplying vertices V^J in an FD-graph, we can define a *null-constraint path* whose existence models a null constraint between any two attribute vertices in an FD-graph implied by an outer join:

DEFINITION 45 (NULL-CONSTRAINT PATH)
Consider an FD-graph G and any two vertices $i, j \in V^A$. A *null-constraint path* from i to j is a minimal subgraph $G' = \langle V' \subseteq (V^A, V^J), E' \subseteq E^J \rangle$ of G such that $i, j \in V^{A'}$ and there exist vertices $m, n \in V^J$ and edges (m, i) and (n, j) in E' and either

1. m and n denote the same vertex in V^J, or
2. there is an edge directly linking m and n, such that the directed edge $(m, n) \in E'$, or
3. there exist vertices $P \in V^{J'}$ and $k \in V^A$ such that the directed edges (P, n) and (P, k) exist in E' and there is a null-constraint path $\langle i, k \rangle$ in G'.

3.3.1.8 Summary of FD-graph notation

The information represented by an extended FD-graph G for an expression e is the 13-tuple

$$\tau[G] = \langle V^A, V^C, V^R, V^J, E^C, E^F, E^I, E^E, E^e, E^R, E^J, \mathit{nullability}(), \mathit{colour}() \rangle \tag{3.3}$$

where

- a vertex in V^A, labeled A and termed a *simple* vertex, represents an attribute A that stems from a base or derived table expression in e;
- a vertex in V^C, labeled X (termed a *compound* vertex), with indegree 0, represents the determinant of a strict or lax functional dependency ($f : X \longrightarrow A$ or $g : X \longmapsto A$, respectively) in e where $\|X\| > 1$;
- a vertex $v_K \in V^R$ represents the tuple identifier of the expression e (if coloured gray) or some subexpression e' (if coloured black). Three invariants for tuple identifier vertices are:
  1. Every FD-graph must have one, and only one, tuple identifier vertex coloured gray.
  2. Each gray tuple identifier vertex, which denotes a tuple of the result of the expression e, functionally determines all white attribute vertices in V^A.
  3. No tuple identifier vertex $v \in V^R$ may be the source of a lax dependency edge, part of any equivalence edge in E^E or E^e, or the source or target of an edge in E^J.
- each vertex in V^J represents a set of attributes that stem from the null-supplying side of an outer join in e;
- for every compound vertex $X \in V^C$ there are edges in E^C, termed *dotted arcs*, from X to each of its component (simple) vertices $A_1, A_2, \ldots, A_n$ representing the set of strict functional dependencies $X \longrightarrow A_k, 1 \leq k \leq n$ in e.
- a directed edge in the set E^F, termed a *strict full arc*, from a vertex with label X to a vertex with label Y where $X \in (V^A \cup V^R \cup V^C)$ and $Y \in (V^A \cup V^R)$ represents the strict functional dependency $X \longrightarrow Y$ in e.
- a directed edge in E^f, termed a *lax full arc*, from the vertex labeled X to the vertex labeled Y where $X \in (V^A \cup V^C)$ and $Y \in (V^A \cup V^R)$ represents the lax functional dependency $X \longmapsto Y$. Note that tuple identifiers cannot form the determinant of a lax functional dependency.
- an undirected edge in E^E, termed a *strict dashed arc*, from the vertex labeled X to the vertex labeled Y where $X \in V^A$ and $Y \in V^A$ represents the strict equivalence constraint $X \stackrel{\omega}{=} Y$ in e.
- an undirected edge in E^e, termed a *lax dashed arc*, from the vertex labeled X to the vertex labeled Y where $X \in V^A$ and $Y \in V^A$ represents the lax equivalence constraint $X \simeq Y$.

- for every compound vertex $K \in V^R$ there are edges in E^R, termed *dotted rowid arcs*, from K to each of its component tuple identifier vertices $k_1, k_2, \ldots, k_n$ representing the set of strict functional dependencies $K \longrightarrow k_i, 1 \leq i \leq n$ in e, as well as the single strict dependency $\{k_1, k_2, \ldots, k_n\} \longrightarrow K$. Note that each vertex k_i can, in turn, be a compound vertex with its own set of edges in E^R.

- there is a directed edge in E^J, termed a *mixed arc*, from the vertex $v_i \in V^J$ to either:

  1. the vertex labeled $v \in V^A$ for each attribute $v \in V^A$ that stems from the null-supplying side of an outer join, or
  2. another vertex $v_j \in V^J$, where v_j represents null-supplying attributes from another table referenced in a nested table expression containing a left- or full-outer join (see Figure 3.8).

  Furthermore, any subgraph G' of G consisting solely of vertices V^J and edges E^J forms a tree.

- the function *nullability*(v) has the range { *definite, nullable, pseudo-definite* } and the domain V^A such that:

  1. the function returns *nullable* if $v \in V^A$ represents a nullable attribute, or
  2. the function returns *definite* if v represents a definite attribute, or
  3. the function returns *pseudo-definite* if v represents an attribute that is definite but for the all-`Null` row of the null-supplying side of an outer join;

- the function *colour*(v) has the range { *white, gray, black* } and the domain $V^A \cup V^R$. For vertices $v \in V^R$ the colour of a vertex is

  - gray if the vertex v represents the tuple identifier of e;
  - black if v represents a tuple identifier in a subexpression e' of e.

  For vertices $v \in V^A$, their colour is

  - *white* if v exists in the result of e, that is v represents a real attribute in the extended table that results from e;
  - *gray* if v represents a constant value contained in some predicate or scalar function within e; and
  - *black* otherwise.

Where convenient, we use V to represent the complete set of vertices in $G[V]$, where $V = V^A \cup V^C \cup V^R \cup V^J$, and we use E to represent the set of edges in $G[E]$, where $E = E^F \cup E^C \cup E^R \cup E^E \cup E^f \cup E^e \cup E^J$. We use the notation V^A_κ to represent the set of vertices in V^A that represent constants, which can stem from either $\kappa(C)$ for some predicate C or from $\kappa(\lambda)$, denoting a constant argument to a scalar function λ.

When referring to individual properties of $\tau(G)$ in the text, we will use the notation $\tau(G)[component]$ to represent a specific component of $\tau(G)$.

3.4 Modelling derived dependencies with FD-graphs

It may be helpful at this point to explain some of the assumptions we make with respect to the algorithms outlined below. First, we assume at the outset that the algebraic expression tree that represents the original SQL query is correctly built and is semantically equivalent to the original expression. Second, we do not consider the naming (or renaming) of attributes to be a problem; each attribute that appears in an algebraic expression tree is qualified by its range variable (in SQL terms the table name or correlation name). Derived columns—for example, `Avg(A + B)`—are given unique names. We assume the existence of a 1:1 mapping function χ that when given a vertex $v \in V^A$ will return that vertex's (unique) label, corresponding to the name of that attribute referenced in the query[22] and, vice-versa, when given a name will return that attribute's corresponding vertex $v \in V^A$. Third, to minimize the maintenance of redundant information, we maintain FD-graphs in simplified form. Fourth, in the algorithms below the reader will note that we are neither retaining the *closure* of an attribute with respect to $\mathcal{F}$, nor are we computing a minimal cover of $\mathcal{F}^+$. We consider either approach to be too expensive. Maier [192] and Ausiello, D'Atri and Saccà [20] both offer proof that finding a minimal cover for $\mathcal{F}$ with less than k edges is NP-complete (Maier references Bernstein's PhD. thesis). Maier also shows that finding a minimal cover with fewer than k vertices is also NP-complete.

Perhaps most importantly, we cannot guarantee that the algorithms below determine *all* possible dependencies in $\mathcal{F}^+$. Klug [162] showed that given an arbitrary relational algebra expression e it was impossible to determine all the dependencies that held in e. Particularly troublesome are the set operators difference and union, as is determining derived dependencies from a join of a projection of a relation with itself (Klug credits Codd with identifying the latter problem). We do claim, however, that the procedures below will derive a useful set of dependencies for a large class of queries.

---

22 See Definition 46 in Section 3.5.

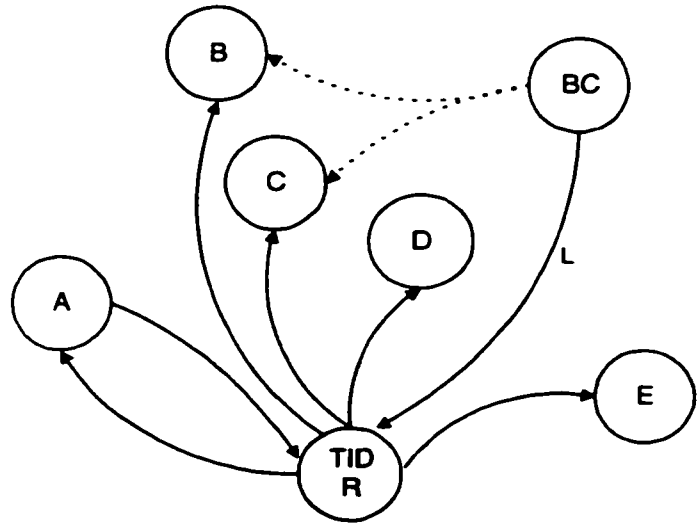


FIGURE 3.3: FD-graph for a base table $R(ABCDE)$ with primary key A and unique constraint BC. Attribute B is nullable.

3.4.1 Base tables

'Base tables' is the only procedure that creates an FD-graph from scratch; the other procedures analyze the FD-graphs of their inputs and generate a new FD-graph as output. Given an extended base table R, the basic idea is to construct an FD-graph G_R with one vertex, coloured white, in the set V^A for each of the attributes in $\alpha(R)$. For brevity we assume that the sets of vertices and edges are initialized to the empty set ($\emptyset$) before FD-graph construction begins.

Once attribute vertices are established, we add additional vertices and edges for any key dependencies in R. We first construct a vertex $v_r \in V^R$, coloured gray, to represent the tuple identifier of each tuple in $I(R)$, and add strict dependency edges from v_r to every attribute vertex in V^A. For the primary key of R, if one exists, or for any unique indexes defined on R, we add strict dependency edges from the vertex representing the primary key or the set of columns in the unique index to v_r. Note that the determinant could be from either of V^A (if the candidate key is a singleton attribute) or V^C (if compound). Similarly, we add dependency edges for **Unique** constraints, though if there is at least one nullable attribute reference in the **Unique** constraint then we make the dependency lax instead of strict. Figure 3.3 depicts an FD-graph for a base table that has both a primary key and a unique constraint.

```
1  Procedure: BASE-TABLE
2  Purpose: Construct an FD-graph for table R(A).
3  Inputs: schema for table R.
4  Output: FD-graph G_R.
5  begin
6    for each attribute a_i ∈ A_R do
7      Construct vertex v_i ∈ V^A corresponding to χ(a_i);
8      Colour[v_i] ← White;
9      if a_i ∈ A is defined as Not Null then
10       Nullability[v_i] ← Definite
11     else
12       Nullability[v_i] ← Nullable
13       fi
14     od ;
15   --Construct tuple identifier vertex v_r.
16   V^R ← v_R;
17   Colour[v_R] ← Gray;
18   for each v_i ∈ V^A do
19     E^F ← E^F ∪ (v_R, v_i)
20     od ;
21   for each primary key constraint or unique index of R do
22     --Let K denote the attributes specified in the constraint or index.
23     if K is compound then
24       Construct a vertex K to represent the composite key;
25       V^C ← V^C ∪ K;
26       for each v_i ∈ K do
27         --Add a dotted edge from K to each of its components.
28         E^C ← E^C ∪ (K, v_i)
29         od
30     else
31       Let K denote an existing vertex in V^A
32       fi ;
33     --Add a strict edge from K to the tuple identifier of R.
34     E^F ← E^F ∪ (K, v_R)
35     od ;
36   for each unique constraint defined for R do
37     --Let K denote the vertex representing the attributes in the unique constraint.
38     if K is compound then
39       Construct a vertex K to represent the composite candidate key;
40       V^C ← V^C ∪ K;
41       for each v_i ∈ K do
42         --Add a dotted edge from K to each of its components.
```

```
43          E^C ← E^C ∪ (K, v_i)
44          od
45        else
46          Let K denote an existing vertex in V^A
47          fi ;
48        if ∃ v_i ∈ K such that v_i is not Definite then
49          -- Add a lax edge from K to the tuple identifier of R.
50          E^f ← E^f ∪ (K, v_R)
51        else
52          -- Add a strict edge from K to the tuple identifier of R.
53          E^F ← E^F ∪ (K, v_R)
54          fi
55        od ;
56      return G_R
57      end
```

As mentioned previously, other arbitrary constraints on base tables can be handled as if they constitute a false- or true-interpreted restriction condition on R (see Section 3.2.1).

3.4.2 Handling derived attributes

In Section 3.1.4 we described the problem of dealing with derived attributes in an intermediate or final result. To model these dependencies in an FD-graph, we add vertices which represent each derived attribute to the FD-graph as necessary while analyzing each algebraic operator.

Darwen [70, pp. 145] extended his set of relational algebra operators to include *extension*, which *extends* a relation R with a derived attribute whose value for any row is the result of a particular function. Our approach, on the other hand, is slightly different. We do not assume that scalar functions or complex arithmetic conditions are added *a priori* to an intermediate result prior to dependency analysis. In contrast to Darwen we will add derived attributes to an FD-graph as we process each individual algebraic operator. A derived attribute can represent a scalar function in a query's `Select` list, `Group` by list, or within a query predicate, which could be part of a `Where` clause, `Having` clause, or `On` condition.

Consider an idempotent function $\lambda(X)$ that produces a derived attribute Y and hence implies that the strict functional dependency $X \longrightarrow Y$ holds in the operator's result R'. The following procedure can be used by any of the procedures of the other operators to add the dependencies for a derived attribute to an FD-graph. Note that we assume in all

cases that λ can return `Null`, regardless of the characteristics of its inputs. An optimization would be to make the setting of *nullability* dependent upon the precise characteristics of each function λ. The test for an exact match (line 63) ensures that two or more instances of any function λ are considered equivalent only if their parameters match exactly; otherwise they are considered unequal. Also note that we intentionally do not attach a attribute Y—doing so is the responsibility of the calling procedure, and can differ depending upon whether λ is a real attribute (in $\alpha(R')$ and coloured white), or a virtual attribute (in $\rho(R)$ and coloured black).

```
58  Procedure: EXTENSION
59  Purpose: Modify an FD-graph to consider X ⟶ λ(X).
60  Inputs: FD-graph G; set of attributes X; new attribute Y ≡ λ(X); tuple ID v_K.
61  Output: modified FD-graph G.
62  begin
63    if χ(λ(X)) ∈ V^A then
64      Let Y ← χ(λ(X))
65    else
66      Construct vertex Y ∈ V^A to represent λ(X);
67      V^A ← V^A ∪ Y;
68      E^F ← E^F ∪ (v_K, Y);
69      fi
70    --Assume the function λ(X) can return a null value.
71    Nullability[Y] ← Nullable;
72    for each constant value κ ∈ X do
73      Construct a vertex χ(κ) ∈ V^A_κ;
74      V^A ← V^A ∪ χ(κ);
75      Colour[χ(κ)] ← Gray;
76      if κ is the null value then
77        Nullability[χ(κ)] ← Nullable
78      else
79        Nullability[χ(κ)] ← Definite
80        fi
81      od;
82    if ||X|| > 1 then
83      --Construct the compound attribute P to represent the set of attributes X.
84      Construct vertex P ∈ V^C to represent the set {X};
85      V^C ← V^C ∪ P;
86      for each v ∈ X
87        --Add the dotted edges for the new compound vertex.
88        E^C ← E^C ∪ (P, χ(v))
89        od
90      E^F ← E^F ∪ (P, Y)
```

```
91      else
92        E^F ← E^F ∪ (X,Y)
93        fi;
94      return G
95      end
```

3.4.3 Projection

The projection operator can both add and remove functional dependencies to and from the set of dependencies that hold in its result. Figure 3.4 illustrates an FD-graph with strict functional dependencies $A \longrightarrow \iota(R)$, $\iota(R) \longrightarrow ABCDE$, and $BC \longrightarrow F$ and with attribute C projected out. The simple attribute vertex representing C is simply coloured black to denote its change from a real attribute to a virtual attribute, mirroring the semantics of both the projection and distinct projection operators.

If the projection includes the application of a scalar function λ, then the algorithm calls the EXTENSION procedure described above to construct the vertex representing its result, and also to construct a compound vertex of its parameters if there is more than one.

If the projection operator eliminates duplicates (i.e. the algebraic π_{Dist} or in SQL `Select Distinct`) then we create a new tuple identifier $v_P \in V^R$ to represent the distinct rows in the result. In this case, the entire projection list can be treated as a candidate superkey of e, generating a strict key dependency since in SQL duplicate elimination via projection treats null values as equivalent 'special values' in each domain. If the number of attributes in the result exceeds one, we need to construct a new compound attribute P, made up of those simple vertices V^A coloured white in G. Finally, we construct a strict dependency between P and v_P to represent the superkey of this derived table. Note that even with the construction of these additional vertices, information about existing candidate keys that survive the projection operation is not lost. An existing superkey in G will continue to transitively determine all the other attributes in G, and therefore will also functionally determine the new superkey of G' (see Figure 3.5).

```
96    Procedure: PROJECTION
97    Purpose: Modify an FD-graph to consider Q = π[A](e).
98    Inputs: FD-graph G for expression e; set of attributes A.
99    Output: FD-graph G_Q.
100   begin
101     copy G to G_Q;
102     v_I ← v ∈ V^R such that Colour[v] is Gray;
```

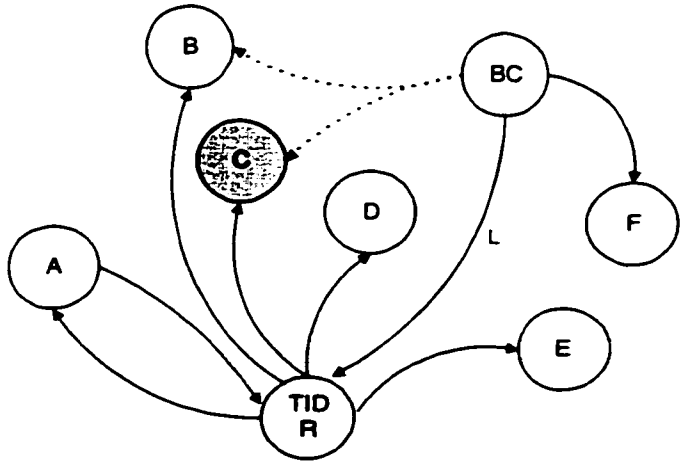


FIGURE 3.4: Marking attributes projected out of an FD-graph.

```
103   for each attribute a_i ∈ A in the projection do
104     if a_i is a function λ(X) then
105       call Extension(G_Q, X, a_i, v_I);
106       Colour[χ(a_i)] ← White
107       fi
108     od;
109   for each white vertex v_i ∈ V^A[G_Q] do
110     if χ(v_i) is not in A then
111       Colour[v_i ∈ V^A] ← Black
112       fi
113     od;
114   if duplicates are to be eliminated then
115     Colour[v_I] ← Black;
116     --Add the new tuple id vertex to represent a tuple in the derived result.
117     Construct vertex v_P ∈ V^R as a tuple identifier;
118     V^R ← V^R ∪ v_P;
119     Colour[v_P] ← Gray;
120     for each vertex v_i ∈ V^A[G_Q] do
121       if Colour[v_i] is White then
122         E^F ← E^F ∪ (v_P, v_i)
123         fi
124       od;
125     if ||A|| > 1 then
126       --Add the candidate key consisting of all projection attributes to V^C.
```

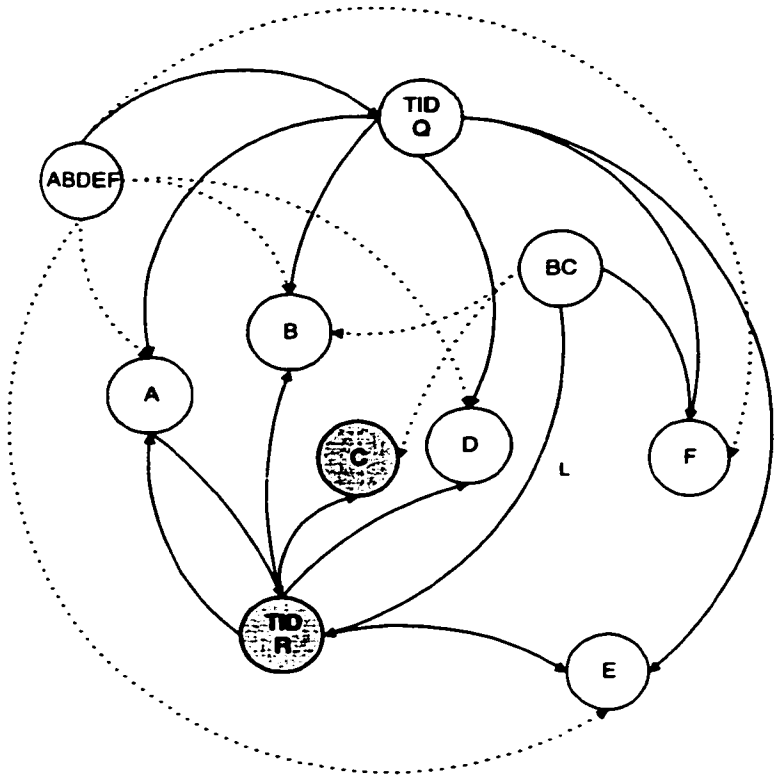


FIGURE 3.5: Development of an FD-graph for projection with duplicate elimination, using the example from Figure 3.4. Note that attribute A, by transitivity, still represents a superkey of the result by functionally determining the result's tuple identifier.

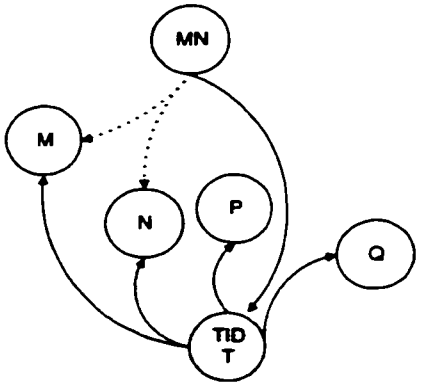
```
127       Construct vertex P ∈ V^C ← {A};
128       V^C ← V^C ∪ P;
129       --Add the dotted edges for the new compound vertex.
130       for each v_i ∈ P
131         E^C ← E^C ∪ (P, v_i);
132         od;
133       E^F ← E^F ∪ (P, v_P)
134     else
135       E^F ← E^F ∪ (A, v_P)
136       fi
137     fi;
138   return G_Q
139   end
```

3.4.4 Cartesian product

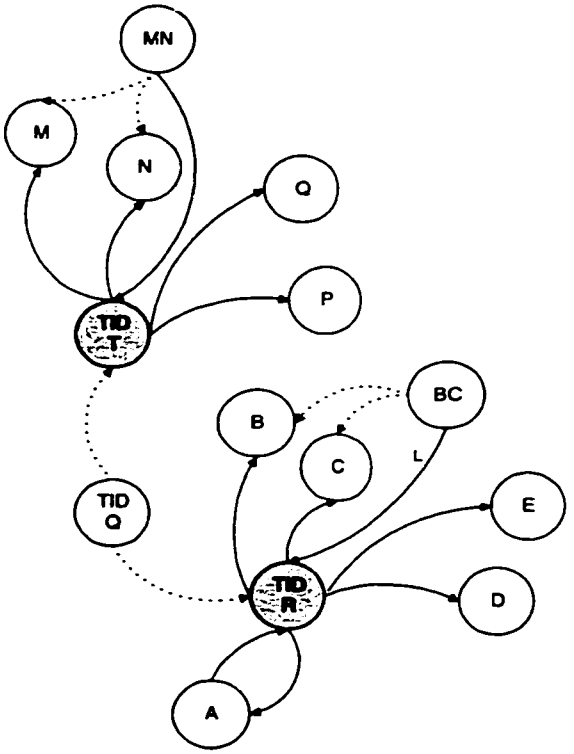
Figure 3.6 illustrates the FD-graph formed to represent the Cartesian product of extended table R with real attributes $ABCDE$, whose FD-graph appears in Figure 3.3, and extended table T with real attributes $MNPQ$ and primary key MN. Note from the resulting FD-graph that two superkeys, namely $\{MNA\}$ and $\{MNBC\}$, can be inferred from the FD-graph. The latter is a lax superkey because BC constitutes a unique constraint in R. However, subsequent discovery of a null-intolerant predicate in a **Where** or Having clause will trigger the conversion of lax dependencies to strict ones, and if so then $MNBC$ still has the possibility of forming a strict candidate key (see Section 3.4.5 below).

An optimization to this algorithm, though omitted in this thesis, is to recognize that if we can guarantee that either of the inputs can have at most one tuple then constructing a combined tuple identifier vertex is unnecessary.

```
140   Procedure: CARTESIAN-PRODUCT
141   Purpose: Construct an FD-graph for Q = R × T.
142   Inputs: FD-graphs G_R and G_T.
143   Output: FD-graph G_Q.
144   begin
145     Merge graphs G_R and G_T into G_Q;
146     --Add a new tuple id vertex to represent a tuple in the derived result.
147     Construct vertex v_K ∈ V^R as a tuple identifier;
148     V^R ← V^R ∪ v_K;
149     Colour[v_K] ← Gray;
```



(a) FD-graph for
$\pi_{All}[M, N, P, Q](T)$.



(b) FD-graph for
$Q = \pi_{All}[A, B, C, D, E, M, N, P, Q](R \times T)$.

FIGURE 3.6: Development of an FD-graph for the Cartesian product operator.

```
150     v_R ← v ∈ V^R[G_R] such that Colour[v] is Gray;
151     E^R ← E^R ∪ (v_K, v_R);
152     Colour[v_R] ← Black;
153     v_T ← v ∈ V^R[G_T] such that Colour[v] is Gray;
154     E^R ← E^R ∪ (v_K, v_T);
155     Colour[v_T] ← Black;
156     return G_Q
157     end
```

3.4.5 Restriction

The algebraic restriction operator $R' = \sigma[C](R)$ is used for both **Where** and **Having** clauses (see Section 2.3.1.4). Restriction is one operator that can only *add* strict functional dependencies to $\mathcal{F}$; it cannot remove any existing strict dependencies.

Extension. If we detect a scalar function $\lambda(X)$ during the analysis of a **Where** clause, then we call the EXTENSION procedure to construct a vertex to represent $\lambda(X)$ and the (strict) functional dependency $X \longrightarrow \lambda(X)$ in the FD-graph. Since $\lambda(X)$ is a virtual attribute in $\rho(R')$, we colour its vertex black.

Conversion of lax dependencies. In the algorithm below we assume that each conjunct of the restriction predicate is false-interpreted. In this case, any Type 1 or Type 2 equality condition concerning an attribute v will automatically eliminate any tuples from the result where v is the null value. Hence any algebraic operator higher in the expression tree can be guaranteed that v cannot be `Null`. Consequently we can mark v as definite in the FD-graph, and we can do so transitively for any other attribute equated to v. The following sub-procedure, SET DEFINITE, appropriately marks vertices in the set V^A as definite, and does so transitively using the temporary set S. The temporary vertex characteristic 'Visited' ensures that no attribute vertex is considered more than once.

```
158     Procedure: SET DEFINITE
159     Purpose: Mark each vertex in the closure of equality conditions as definite.
160     Inputs: FD-graph G.
161     Output: Modified FD-graph G.
162     begin
163       S ← ∅;
164       for each v_i ∈ V^A do
165         Visited[v_i] ← False;
166         if Nullability[v_i] is Definite then
167           S ← S ∪ v_i
```

```
168       fi
169     od;
170   while S ≠ ∅ do
171     select vertex v_i from S;
172     S ← S − v_i;
173     Visited[v_i] ← True;
174     D ← Mark Definite(v_i);
175     for each v_j ∈ D do
176       Nullability[v_j] ← Definite;
177       if Visited[v_j] is False then
178         S ← S ∪ v_j
179         fi
180       od
181     od;
182   return G
183   end
```

The MARK DEFINITE procedure below returns a set D of related attributes that can be treated as definite due to the existence of a strict equality constraint between each attribute in D and the input parameter, vertex v. The test for vertex colour ensures that we do not attempt to mark a vertex in V_{κ}^{A} representing a `Null` constant as definite.

```
184   Procedure: MARK DEFINITE
185   Purpose: Construct a set of vertices that can be considered definite.
186   Inputs: FD-graph G, vertex v.
187   Output: A set D consisting of those vertices related to v that can be made definite.
188   begin
189     D ← ∅;
190     for each v_i ∈ V^A such that (v, v_i) ∈ E^E do
191       if Nullability[v_i] is not Definite and Colour[v_i] is not Gray then
192         D ← D ∪ v_i
193         fi
194       od;
195     return D
196     end
```

We can convert any lax dependencies or equivalence constraints into strict ones once we can determine that both the left- and right-hand sides of the constraint cannot be Null. The following sub-procedure, CONVERT DEPENDENCIES, transforms lax dependencies or equivalence constraints between any two vertices that represent definite values into strict ones. In the case of composite determinants, we check that each of the individual component vertices that constitute the determinant are marked definite.

```
197  Procedure: CONVERT DEPENDENCIES
198  Purpose: Convert lax dependencies and equivalence constraints into strict ones.
199  Inputs: FD-graph G.
200  Output: Modified FD-graph G.
201  begin
202    for each edge (v_i, v_j) ∈ E^f do
203      --Note that vertex v_j can only exist in V^A ∪ V^R.
204      if v_i ∈ V^A and Nullability[v_i] is Definite then
205        if v_j ∈ V^R then
206          E^f ← E^f − (v_i, v_j);
207          E^F ← E^F ∪ (v_i, v_j)
208        else
209          if v_j ∈ V^A and Nullability[v_j] is Definite then
210            E^f ← E^f − (v_i, v_j);
211            E^F ← E^F ∪ (v_i, v_j)
212            if (v_i, v_j) ∈ E^e then
213              E^e ← E^e − (v_i, v_j);
214              E^E ← E^E ∪ (v_i, v_j)
215              fi
216            fi
217          fi
218      else
219        if v_i ∈ V^C then
220          if ∃ v_k such that (v_i, v_k) ∈ E^C and Nullability[v_k] is not Definite then
221            continue
222            fi
223          if v_j ∈ V^R then
224            E^f ← E^f − (v_i, v_j);
225            E^F ← E^F ∪ (v_i, v_j)
226          else
227            if v_j ∈ V^A and Nullability[v_j] is Definite then
228              E^f ← E^f − (v_i, v_j);
229              E^F ← E^F ∪ (v_i, v_j)
230              fi
```

```
231          fi
232        fi
233      fi
234    od;
235    return G
236    end
```

The RESTRICTION procedure assumes that the given restriction predicate is in conjunctive normal form, and immediately eliminates all disjunctive clauses. For each remaining Type 1 or Type 2 condition, the procedure adds the necessary vertices if they do not already exist, marks the vertices as definite, and adds the appropriate (strict) dependency and equivalence edges. In the last two steps of the procedure, the sub-procedures SET DEFINITE and CONVERT DEPENDENCIES are called to first mark those vertices in V^A that are guaranteed to be definite through transitive strict equivalence constraints, and second to convert lax dependencies and equivalence constraints into strict ones if both their determinants and dependents cannot be `Null`.

```
237  Procedure: RESTRICTION
238  Purpose: Construct an FD-graph for Q = σ[C](R).
239  Inputs: FD-graph G; restriction predicate C.
240  Output: FD-graph G_Q.
241  begin
242    copy G to G_Q;
243    separate C into conjuncts: C' ← P_1 ∧ P_2 ∧ ... ∧ P_n;
244    for each P_i ∈ C' do
245      if P_i contains an atomic condition not of Type 1 or Type 2 then
246        delete P_i from C'
247      else if P_i contains a disjunctive clause then
248          delete P_i from C'
249          fi
250        fi
251      od
252    if C' is simply True then return G_Q fi;
253    --C' now consists of entirely conjunctive components.
254    v_K ← v ∈ V^R such that Colour[v] is Gray;
255    for each conjunctive predicate P_i ∈ C' do
256      if P_i is a Type 1 condition (v = c) then
257        V^A ← V^A ∪ χ(c);
258        if v is a function λ(X) then
259          call Extension(G_Q, X, v, v_K);
```

```
260         Colour[χ(v)] ← Black
261         fi;
262       -- This condition will eliminate all null values of v.
263       Nullability[χ(c)] ← Definite;
264       Nullability[χ(v)] ← Definite;
265       Colour[χ(c)] ← Gray;
266       E^F ← E^F ∪ (χ(v), χ(c)) ∪ (χ(c), χ(v));
267       E^E ← E^E ∪ (χ(v), χ(c))
268     else
269       -- Component P_i is a Type 2 condition (v_1 = v_2).
270       if v_1 is a function λ(X) then
271         call Extension(G_Q, X, v_1, v_K);
272         Colour[χ(v_1)] ← Black
273         fi;
274       if v_2 is a function λ(X) then
275         call Extension(G_Q, X, v_2, v_K);
276         Colour[χ(v_2)] ← Black
277         fi
278       -- This condition will eliminate all null values of both v_1 and v_2.
279       Nullability[χ(v_1)] ← Definite;
280       Nullability[χ(v_2)] ← Definite;
281       E^F ← E^F ∪ (χ(v_1), χ(v_2)) ∪ (χ(v_2), χ(v_1));
282       E^E ← E^E ∪ (χ(v_1), χ(v_2))
283       fi
284     od;
285   Call SetDefinite(G_Q);
286   Call ConvertDependencies(G_Q);
287   return G_Q
288   end
```

Handling true-interpreted predicates. In the algorithm above, we presented only the pseudo-code for analyzing false-interpreted predicates. In cases where the restriction condition contains one or more conjunctive true-interpreted Type 1 or Type 2 conditions, the pseudo-code would be nearly identical but for a modification to generate lax dependencies and equivalence constraints instead of strict ones. Essentially the main loop from lines 255 through 284 would be repeated, with:

- the lines which set each vertex's nullability characteristic to 'definite' removed (lines 263, 264, 279, and 280); and

- lines which add strict functional dependencies (lines 266 and 281) and strict equivalence constraints (lines 267 and 282) changed to add lax functional dependencies and equivalence constraints, respectively.

The existence of a true-interpreted predicate will not alter the validity of any strict dependency or equivalence constraint previously discovered, nor does it affect the logic of the procedures SET DEFINITE or CONVERT DEPENDENCIES.

3.4.6 Intersection

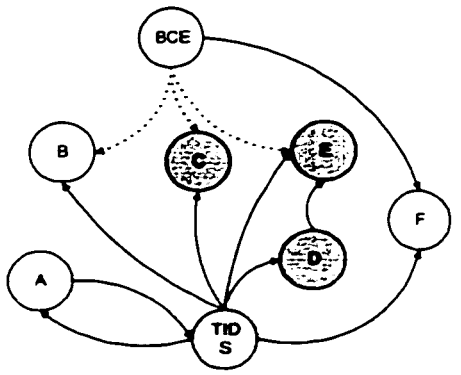
The FD-graph G_Q representing the intersection of extended tables S and T is constructed by performing a union of the FD-graphs for S and T and adding equivalence and strict dependency edges between the corresponding white vertices of their inputs, which correspond to the select list items of each query specification (see Figure 3.7). We assume that the merging of the two FD-graphs results in a graph where all attributes are uniquely named. We arbitrarily denote attributes from S to constitute the result of the intersection, and colour black all white attributes from T to model their inclusion as virtual attributes in Q.

To represent a tuple in the instance $I(Q)$, we arbitrarily choose the tuple identifier of S, and add strict dependency edges between the two tuple identifier vertices of the inputs. The tuple identifier of T is coloured black[23]. With this construction, if a complete simple or composite candidate superkey is present in either input, then that key becomes a superkey of the result (as per Corollary 1 on page 80).

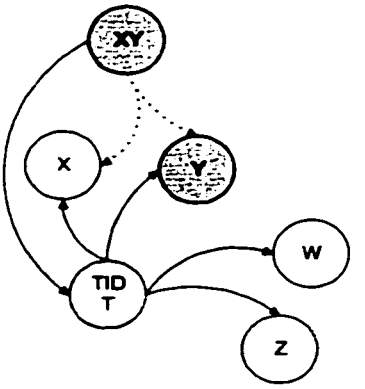
Finally, we mark as pseudo-definite any vertex representing a real attribute of S if it or its corresponding attribute in T is pseudo-definite, since there exists a strict equivalence constraint between each attribute pair and by the semantics of intersection the corresponding values of each attribute pair are identical. Moreover, the procedure SET DEFINITE may now be able to exploit these strict equivalence constraints to mark either of these corresponding vertices as definite. SET DEFINITE can also exploit the existence of other strict equivalence constraints *in either input* to mark some subset of the attributes in $sch(Q)$ as definite. Subsequently, we may be able to convert lax functional dependencies and lax equivalence constraints to strict ones in a manner identical to that for the restriction operator.

---

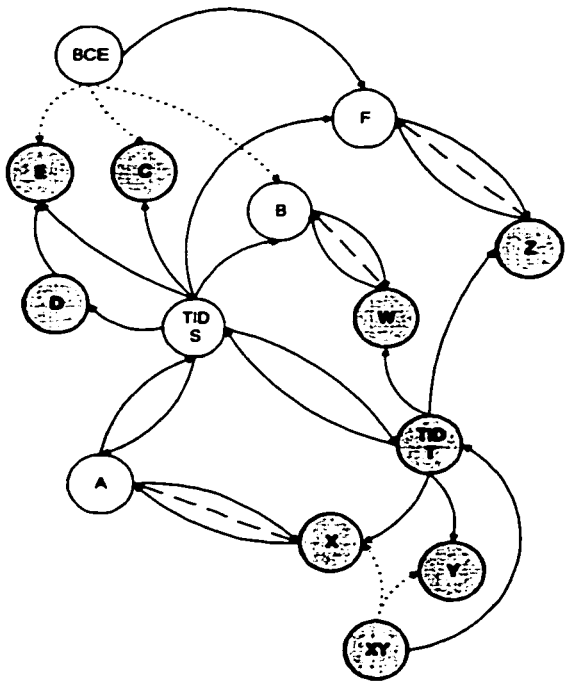
23 Recall the invariant that there must be one, and only one, tuple identifier vertex in V^R coloured gray in an FD-graph.



(a) FD-graph for $\pi_{All}[A, B, F](S)$.



(b) FD-graph for $\pi_{All}[X, W, Z](T)$.



(c) FD-graph for $Q =$
$\pi_{All}[A, B, F](S) \cap_{All} \pi_{All}[X, W, Z](T)$.

FIGURE 3.7: Development of an FD-graph for the **Intersection** operator. Note that the vertex representing A denotes a superkey of the result.

289 **Procedure:** INTERSECTION
290 **Purpose:** *Construct an FD-graph for* $Q = S \cap_{All} T$.
291 **Inputs:** FD-graphs G_S and G_T.
292 **Output:** FD-graph G_Q.
293 **begin**
294 Merge graphs G_S and G_T into G_Q;
295 *--Choose the tuple id vertex of* S *to represent a tuple in* $I(Q)$.
296 $v_S \leftarrow v \in V^R[G_S]$ such that Colour[v] is Gray;
297 $v_T \leftarrow v \in V^R[G_T]$ such that Colour[v] is Gray;
298 $E^F \leftarrow E^F \cup (v_S, v_T) \cup (v_T, v_S)$;
299 Colour[v_T] $\leftarrow$ Black;
300 *--Establish equivalences between the white vertices in* G_T *and* G_S.
301 **for each** white vertex $v_k \in V^A[G_T]$ **do**
302 *--Let* v_i *denote the corresponding white attribute from* G_T *in* G_Q.
303 *--Let* v_j *denote the union-compatible white attribute from* G_S *in* G_Q.
304 $E^F[G_Q] \leftarrow E^F[G_Q] \cup (v_i, v_j) \cup (v_j, v_i)$;
305 $E^E[G_Q] \leftarrow E^E[G_Q] \cup (v_i, v_j)$;
306 Colour[v_i] $\leftarrow$ Black;
307 **if** Nullability[v_i] is Pseudo-definite and Nullability[v_j] is Nullable **then**
308 Nullability[v_j] $\leftarrow$ Pseudo-definite
309 **fi**;
310 **if** Nullability[v_j] is Pseudo-definite and Nullability[v_i] is Nullable **then**
311 Nullability[v_i] $\leftarrow$ Pseudo-definite
312 **fi**
313 **od**;
314 Call SetDefinite(G_Q);
315 Call ConvertDependencies(G_Q);
316 **return** G_Q
317 **end**

3.4.7 Grouping and Aggregation

As described in Section 2.3 we model SQL's group-by operator with two algebraic operators. The *partition* operator, denoted $\mathcal{G}$, produces a *grouped table* as its result with one tuple per distinct set of group-by attributes. Each set of values required to compute any aggregate function is modelled as a set-valued attribute. The *grouped table projection* operator, denoted $\mathcal{P}$, projects a grouped table over a `Select` list composed of group-by attributes and aggregate function applications. Projection of a grouped table differs from an 'ordinary' projection in that it must deal not only with atomic attributes (those in

the Group by list) but also the set-valued attributes used to compute the aggregate functions.

3.4.7.1 Partition

The algorithm begins with creating a new tuple identifier v_K to represent each tuple in the partition; the tuple identifier of the original input v_I, now a virtual attribute in $\rho(Q)$, is coloured black. Subsequently, the EXTENSION procedure is called to create additional vertices for any scalar functions $\lambda(X)$ present in the group-by list. These new attributes are coloured white since they are real attributes in the extended table Q that results from the partition. Thereafter the algorithm colours black any vertex that does not represent a real attribute—that is, a group-by attribute or a set-valued attribute required for one or more aggregate functions—to denote its assignment to $\rho(Q)$. Finally, we create a superkey K consisting of the entire group-by list (if one exists) which forms the determinant of the set of strict functional dependencies for each of the set-valued attributes. If there already exists a superkey vertex X in the input—for example, the Group by clause contains the primary key(s) of the input base table(s)—then it follows that $X \subseteq K$ and consequently we trivially have $X \longrightarrow K$ and $K \longrightarrow X$. These other keys will then infer K, and consequently the FD-graph reflects all of the valid superkeys.

In Claim 22 (see page 83) we noted that a special case of partition is when the set A^G is empty, corresponding to a missing Group by clause in an aggregate ANSI SQL query. In this case, the result must consist of a single tuple containing one or more set-valued attributes containing the values required by each aggregation function $f \in F$. In this case, we model the key of the result by using a constant attribute. To ensure that we do not infer erroneous transitive dependencies or equivalence constraints from this constant, we assume that the value of the constant is definite and unique across the entire database instance (and hence unequal to any other constant discovered during dependency analysis). We assume the existence of a generating function $\aleph()$ that produces such a unique value when required.

318 **Procedure:** PARTITION
319 **Purpose:** *Construct an FD-graph for* $Q = \mathcal{G}[A^G, A^A](R)$.
320 **Inputs:** FD-graph G; n grouping attributes A^G;
321 m set-valued attributes A^A.
322 **Output:** FD-graph G_Q.
323 **begin**
324 copy G to G_Q;
325 *--Add a new tuple id vertex to represent a tuple in* $I(Q)$.
326 $v_I \leftarrow v \in V^R$ such that Colour[v] is Gray;

```
327   Colour[v_I] ← Black;
328   Construct vertex v_K ∈ V^R as a tuple identifier;
329   V^R ← V^R ∪ v_K;
330   Colour[v_K] ← Gray;
331   --Establish group-by attributes.
332   for each a_i^G ∈ A^G do
333     if a_i^G is a scalar function λ(X) then
334       --Construct the new vertex to represent the function's result.
335       call Extension(G_Q, X, a_i^G, v_I);
336       Colour[χ(a_i^G)] ← White;
337       fi
338     od;
339   --Establish a dependency with v_K for each group-by and set-valued attribute.
340   for each white vertex v_i ∈ V^A[G_Q] do
341     if χ(v_i) ∈ A^G then
342       E^F ← E^F ∪ (v_K, v_i)
343     else if χ(v_i) ∈ A^A then
344       --Construct set-valued vertex v_S to represent v_i.
345       V^A ← V^A ∪ v_S;
346       Colour[v_S] ← White;
347       Nullability[v_S] ← Nullability[v_i];
348       E^F ← E^F ∪ (v_K, v_S);
349       Colour[v_i] ← Black
350     else
351       Colour[v_i] ← Black
352       fi
353     od;
354   if A^G ≠ ∅ then
355     --Construct a superkey consisting of all grouping attributes.
356     if ||A^G|| > 1 then
357       Construct compound vertex K to represent the set {A^G};
358       V^C ← V^C ∪ K;
359       for each χ(v_i) ∈ A^G do
360         E^C ← E^C ∪ (K, v_i)
361         od
362     else
363       Let K ← χ(a_1^G)
364       fi;
365     E^F ← E^F ∪ (K, v_K)
366   else
367     --Construct a candidate key of the result using the constant ℵ().
368     Construct vertex K to represent the value ℵ();
```

```
369       V^A ← V^A ∪ K;
370       Colour[K] ← Gray;
371       Nullability[K] ← Definite;
372       E^F ← E^F ∪ (K, v_K)
373       fi;
374     return G_Q
375     end
```

3.4.7.2 Grouped table projection

The projection of a grouped table, denoted $\mathcal{P}$, projects a grouped table over the set of grouping columns and represents the computation of any aggregate functions over one or more set-valued attributes. The GROUPED TABLE PROJECTION algorithm below computes the derived dependencies that hold in the result of $Q = \mathcal{P}[A^G, F[A^A]](R)$. As described earlier, the number of tuples in Q is identical to the number of tuples in the extended grouped table R, hence no tuple identifier modifications are necessary. Set-valued attributes do not themselves appear in the projection; only grouping columns or the result of an aggregate function f_i can appear as real attributes in the result, and hence these input attributes are coloured black. Each aggregate function in the result is created by extending Q through a call to the EXTENSION procedure; the resulting vertex is coloured white to represent its membership in $\alpha(Q)$.

```
376   Procedure: GROUPED TABLE PROJECTION
377   Purpose: Construct an FD-graph for Q = P[A^G, F[A^A]](R).
378   Inputs: FD-graph G; n grouping attributes A^G;
379         m set-valued attributes A^A, k aggregate functions.
380   Output: FD-graph G_Q.
381   begin
382     copy G to G_Q;
383     v_K ← v ∈ V^R such that Colour[v] is Gray;
384     --Colour black all set-valued attributes that cannot exist in the projection.
385     for each white vertex v_i ∈ V^A do
386       if χ(v_i) ∉ A^G then
387         --v_i is a set-valued aggregation attribute, which cannot be in the result.
388         Colour[v_i] ← Black
389         fi
390       od;
391     for each f_i ∈ F | 1 ≤ i ≤ k do
392       --Construct a vertex Y to represent f_i(A_j^A) and
```

```
393       --make its set-valued input parameters its determinant.
394       call Extension(G_Q, A_j^A, Y, v_K);
395       Colour[Y] ← White;
396       if f_i = 'Count' or 'Count Distinct' then
397         Nullability[Y] ← Definite
398       else
399         Nullability[Y] ← Nullable
400         fi
401       od;
402     return G_Q
403     end
```

3.4.8 Left outer join

As mentioned previously in Section 3.3.1.7, for left outer joins we require a mechanism to group the attributes from the null-supplying side of an outer join so that once a null-intolerant `Where` clause predicate is discovered, all of the lax dependencies and lax equivalence constraints amongst the pseudo-definite attributes in the group—which form a null constraint—can be converted to strict dependencies and equivalence constraints. To group null-supplying attributes together, we utilize a hypervertex in V^J with mixed edges in E^J to each of its component vertices. In the case of nested outer joins, an edge in E^J may connect two vertices in V^J (see Figure 3.8). These edges form a tree of vertices in V^J, which represent each level of nesting in much the same way Bhargava, Goel, and Iyer [34] use *levels* of binding to determine the candidate keys of the results of outer joins[24].

EXAMPLE 22
Consider the FD-graph pictured in Figure 3.9. The FD-graph illustrates the functional dependencies that hold in the result of the expression

$$Q = \mathcal{R}_{\alpha}(\pi_{All}[W, X, A, B](T \xrightarrow{p} S)) \tag{3.4}$$

---

24 For full outer joins each side of the join will be represented by a separate instance of a vertex in V^J; see Section 3.4.9.

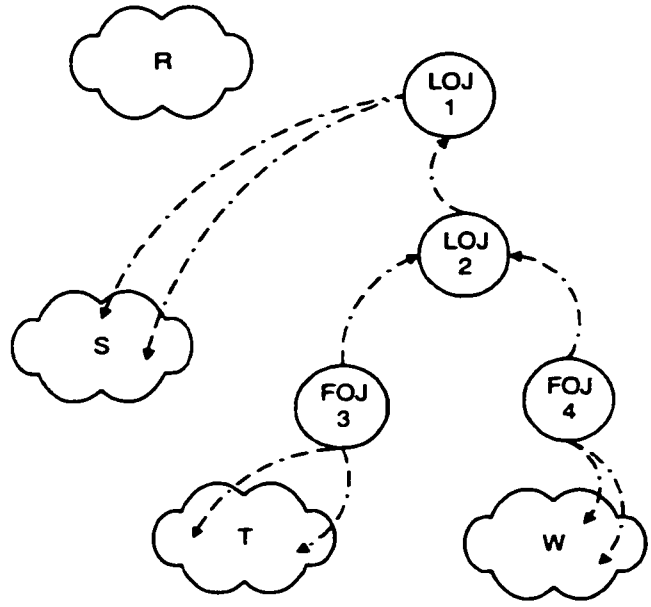


FIGURE 3.8: Summarized FD-graph for a nested outer join modelling the expression $Q = R \xrightarrow{p_1} (S \xrightarrow{p_2} (T \xleftarrow{p_3} W))$. Each vertex that represents a null-supplying side groups null-supplying attributes together. Furthermore, there exists a directed edge in E^J from each nested null-supplying vertex to a vertex representing its 'parent' table expression.

over extended tables T and S with real attributes $WXYZ$ and $ABCDE$ respectively and where predicate p consists of the conjunctive condition $T.X = S.B \wedge T.Z = S.A$, which corresponds to the SQL statement

```
Select W, X, A, B
From   Rₐ(T) Left Outer Join Rₐ(S) On (T.X = S.B and T.Z = S.A)
```

where XY is the primary key of T and A is the primary key of S. Note the existence of the lax functional dependency $A \longrightarrow Z$ and its corresponding lax equivalence constraint $A \simeq Z$, due to the fact that S is the null-supplying table. Note also that $A \longrightarrow BCDE$ remains reflected as four strict dependencies since A is a primary key and hence cannot be `Null`. The strict dependency $B \longrightarrow D$ has been transformed from a lax dependency in the input; the conjunctive predicate $T.X = S.B$ in the `On` condition ensures that the generation of an all-`Null` row cannot violate $B \longrightarrow D$ even though B is nullable in extended table S.

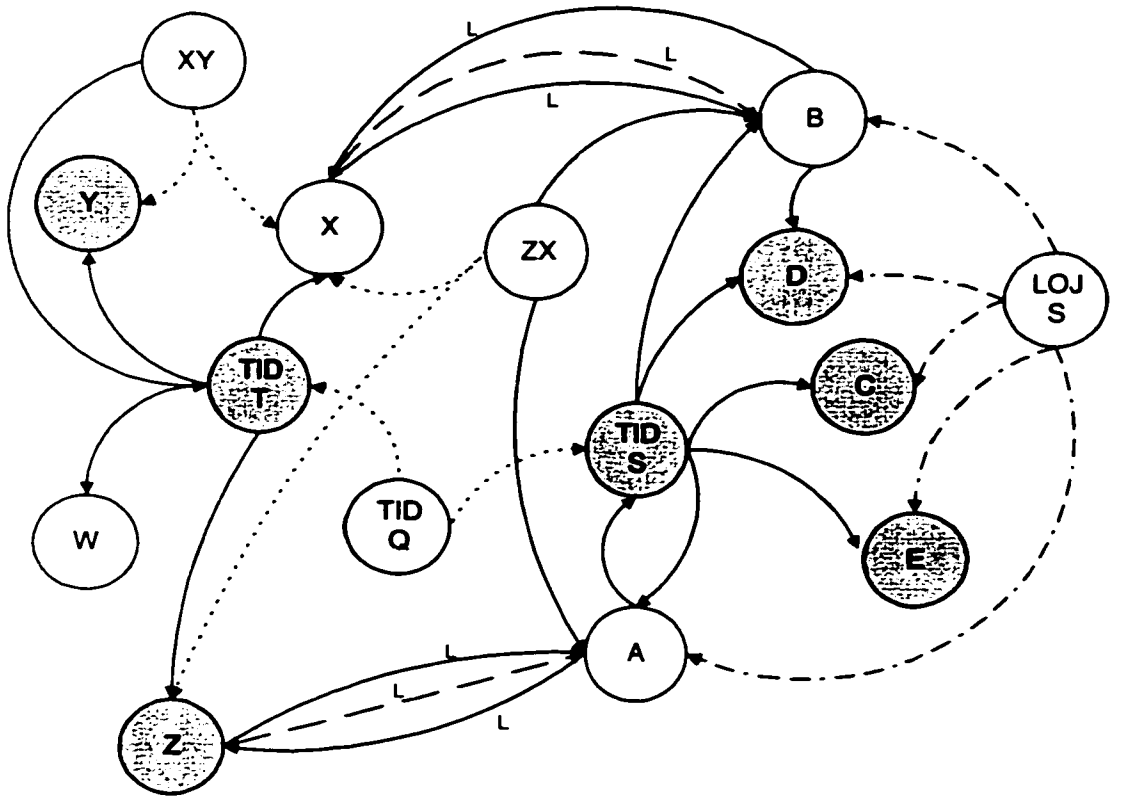


FIGURE 3.9: Resulting FD-graph for the left outer join in Example 22. Note the four different types of edges: solid edges denote functional dependencies, dotted edges link compound nodes to their components, dashed edges represent equivalence constraints, and mixed edges group attributes from a null-supplying side of an outer join. Note that the lax dependency $B \longmapsto D$, which stemmed from the nullable determinant B, has been converted to a strict dependency due to the null-intolerant predicate in the outer join's On condition.

3.4.8.1 Algorithm

The algorithm below accepts three inputs: the two FD-graphs corresponding to the inputs (preserved table S, null-supplying table T) of the left outer join, and the outer join's On condition p. The algorithm's logic can be divided into five distinct parts:

1. Graph merging and initialization (lines 410 to 430): the two input graphs are merged into a single graph G_Q that represents the dependencies and equivalences that hold in the result. A new tuple identifier vertex v_K is created to represent a derived tuple in the outer join's result, and a null-supplying vertex J is created to link the attributes on the null-supplying side of the left outer join. The algorithm also assumes that attributes are appropriately renamed to prevent logic errors due to duplicate names, and to simplify the exposition we assume that the On condition does not contain any scalar functions, although we could easily extend the algorithm to do so. The algorithm for right outer join is symmetric with the one for left, simply by interchanging the two inputs.

2. Dependency and constraint analysis for the null-supplying table (lines 431 to 472): strict dependencies that hold in T are analyzed to determine if they can hold in $I(Q)$, or if they must be 'downgraded' to a lax dependency because their determinant can be wholly Null, as per Lemma 7. As mentioned above, the algorithm relies on a function η, which is assumed to exist, to determine whether or not the left outer join's On condition can be satisfied if a given attribute can be Null. Lax dependencies that hold in T can also be converted to strict dependencies if the conditions specified in Lemma 8 hold. Moreover, lax equivalence constraints $e : X \simeq Y$ will continue to hold in Q. However, if the possibility of null values for X and Y are eliminated but for the all-Null row—that is, both $\eta(p, X)$ and $\eta(p, Y)$ return *true*—then e can be 'upgraded' to a strict equivalence constraint.

3. Generation of lax dependencies implied by the On condition (lines 473 to 516): following the analysis outlined in Section 3.2.9.2, the algorithm generates a valid subset of the lax and strict dependencies implied by null-intolerant conjunctive equality predicates in the On condition. The logic within this section is quite similar to that outlined in the RESTRICTION algorithm. A strict dependency will only result from a Type 2 condition involving two attributes from the null-supplying table; otherwise all of the generated dependencies constructed in this portion of the algorithm are lax dependencies.

4. Construction of strict dependencies implied by the On condition (lines 517 to 533): following the analysis outlined in Section 3.2.9.2, the algorithm generates the strict

dependencies $g : \alpha_S(p) \longrightarrow z$ between the set of preserved attributes referenced in p and each $z \in Z$, the set of null-supplying attributes included in null-intolerant conjunctive equality predicates in the `On` condition (see Theorem 3). The dependent set Z is constructed as a side-effect of the `On` condition analysis done in the previous step.

5. Marking attributes nullable (lines 534 to 541): finally, all null-supplying attributes must be marked as either nullable or pseudo-definite to correspond to the possible generation of an all-`Null` row in the result.

We have simplified the algorithm below to ignore the existence of (1) `is null` predicates, (2) any other null-tolerant condition, or (3) scalar functions $\lambda(X)$ in p. However, the algorithm can be easily extended to include such support, or to derive additional strict or lax dependencies depending upon the sophistication of the analysis on the `On` condition predicate p.

```
404  Procedure: LEFT OUTER JOIN
405  Purpose: Construct an FD-graph for Q = S --p--> T.
406  Inputs: FD-graphs for G_S, G_T, predicate p.
407  Output: FD-graph G_Q.
408
409  begin
410    Merge graphs G_S and G_T into G_Q;
411    --Add a new tuple id vertex to represent a tuple in the derived result.
412    Construct vertex v_K ∈ V^R as a tuple identifier;
413    V^R ← V^R ∪ v_K;
414    Colour[v_K] ← Gray;
415    v_S ← v ∈ V^R[G_S] such that Colour[v] is Gray;
416    E^R ← E^R ∪ (v_K, v_S)
417    Colour[v_S] ← Black;
418    v_T ← v ∈ V^R[G_T] such that Colour[v] is Gray;
419    E^R ← E^R ∪ (v_K, v_T)
420    Colour[v_T] ← Black;
421    --Construct an outer join vertex J to represent null-supplying attributes in G_T.
422    V^J[G_Q] ← V^J[G_Q] ∪ J;
423    for each v_i ∈ V^A[G_T] do
424      if ∄(v_j, v_i) ∈ E^J[G_T] for any v_j ∈ V^J[G_T] then
425        E^J ← E^J ∪ (J, v_i)
426        fi
427      od;
428    for each v_j ∈ V^J[G_T] such that ∄v_k ∈ V^J | (v_j, v_k) ∈ E^J do
429      E^J ← E^J ∪ (v_j, J)
```

```
430     od ;
431   --Determine those equivalences and dependencies from T that still hold in Q.
432   for each strict dependency edge (v_i, v_j) ∈ E^F do
433     if v_j ∉ V^A[G_T] ∪ V^R[G_T] then continue fi ;
434     if v_i ∈ V^R[G_T] then continue fi ;
435     if v_i ∈ V^A[G_T] then
436       if η(p, χ(v_i)) is False and (v_i, v_j) ∉ E^E then
437         E^F ← E^F − (v_i, v_j);
438         E^f ← E^f ∪ (v_i, v_j)
439         fi
440     else if v_i ∈ V^C[G_T] then
441       --For a compound determinant, ensure it has at least one definite attribute.
442       if ∄ v_k such that (v_i, v_k) ∈ E^C and η(p, χ(v_k)) is True then
443         E^F ← E^F − (v_i, v_j);
444         E^f ← E^f ∪ (v_i, v_j)
445         fi
446       fi
447     od ;
448   for each lax dependency edge (v_i, v_j) ∈ E^f do
449     if v_j ∉ V^A[G_T] ∪ V^R[G_T] then continue fi :
450     if v_i ∈ V^A[G_T] then
451       if η(p, χ(v_i)) is True and (v_j ∈ V^R or η(p, χ(v_j)) is True) then
452         E^F ← E^F ∪ (v_i, v_j);
453         E^f ← E^f − (v_i, v_j):
454         fi
455     else if v_i ∈ V^C[G_T] then
456       --For a compound determinant, it must have all definite attributes.
457       if ∃ v_k such that (v_i, v_k) ∈ E^C and η(p, χ(v_k)) is False then
458         continue
459         fi
460       if v_j ∈ V^R or η(p, χ(v_j)) is True then
461         E^F ← E^F ∪ (v_i, v_j);
462         E^f ← E^f − (v_i, v_j)
463         fi
464       fi
465     od ;
466   for each lax equivalence edge (v_i, v_j) ∈ E^e do
467     if {v_i ∪ v_j} ⊈ V^A[G_T] then continue fi ;
468     if η(p, χ(v_i)) is True and η(p, χ(v_j)) is True then
469       E^E ← E^E ∪ (v_i, v_j);
470       E^e ← E^e − (v_i, v_j)
471       fi
```

```
472     od ;
473   --Initialize the dependent set of the strict dependency g : α_S(p) ⟶ Z.
474   Z ← ∅;
475   --Handle the ON condition in a similar manner as restriction.
476   separate p into conjuncts: P' ← P_1 ∧ P_2 ∧ ... ∧ P_n;
477   for each P_i ∈ P' do
478     if P_i contains a disjunctive clause then
479       delete P_i from P'
480     else if P_i contains an atomic condition not of Type 1 or Type 2 then
481       delete P_i from P'
482       fi
483     od
484   if P' is not simply True then
485     --P' now consists of entirely null-intolerant conjunctive components.
486     for each conjunctive component P_i ∈ P' do
487       if P_i is a Type 1 condition (v = c) then
488         --Comparisons to preserved attributes do not imply a dependency.
489         if χ(v) ∉ V^A[G_T] then continue fi ;
490         Construct vertex χ(c) to represent c;
491         V^A[G_Q] ← V^A[G_Q] ∪ χ(c);
492         Nullability[χ(c)] ← Definite;
493         Colour[χ(c)] ← Gray;
494         E^f ← E^f ∪ (χ(v), χ(c)) ∪ (χ(c), χ(v));
495         E^e ← E^e ∪ (χ(v), χ(c));
496         Z ← Z ∪ χ(v)
497       else
498         --Component (v_1 = v_2) is a Type 2 condition; note that
499         --η(p, χ(v_1)) and η(p, χ(v_2)) will be true automatically.
500         if {χ(v_1), χ(v_2)} ⊆ V^A[G_S] then continue fi ;
501         if {χ(v_1), χ(v_2)} ⊆ V^A[G_T] then
502           E^F ← E^F ∪ (χ(v_1), χ(v_2)) ∪ (χ(v_2), χ(v_1));
503           E^E ← E^E ∪ (χ(v_1), χ(v_2));
504           Z ← Z ∪ {χ(v_1), χ(v_2)}
505         else
506           E^f ← E^f ∪ (χ(v_1), χ(v_2)) ∪ (χ(v_2), χ(v_1));
507           E^e ← E^e ∪ (χ(v_1), χ(v_2));
508           if χ(v_1) ∈ V^A[G_T] then
509             Z ← Z ∪ χ(v_1)
510           else
511             Z ← Z ∪ χ(v_2)
512             fi
513           fi
```

```
514         fi
515       od
516     fi
517   --Construct the strict dependency of preserved attributes to null-supplying ones.
518   if ||α_S(p)|| > 0 then
519     if ||α_S(p)|| > 1 then
520       --Create a compound determinant in V^C for the dependency g.
521       Construct vertex W ∈ V^C to represent α_S(p);
522       V^C ← V^C ∪ W;
523       --Add the dotted edges for the new compound vertex.
524       for each v_i ∈ W
525         E^C ← E^C ∪ (W, v_i);
526         od;
527     else
528       Let χ(W) denote the single preserved vertex in α_S(p);
529       fi
530     for each v_k ∈ Z do
531       E^F ← E^F ∪ (W, v_k)
532       od
533     fi;
534   --Finally, set the nullability characteristic of each attribute from the null-supplying side.
535   for each v_i ∈ V^A[G_T] do
536     if η(p, χ(v_i)) is True then
537       Nullability[v_i[G_Q]] ← Pseudo-definite
538       fi
539     od
540   return G_Q
541   end
```

3.4.9 Full outer join

Because full outer join only differs slightly in its semantics from left or right outer join, we omit the presentation of a detailed algorithm to construct an FD-graph representing the dependencies and equivalences that hold in the result of a full outer join. We claim that we can modify the LEFT OUTER JOIN algorithm in a straightforward manner, following Theorem 4, to model the correct semantics for full outer join.

3.4.10 Algorithm modifications to support outer joins

Galindo-Legaria and Rosenthal [97, 98] and Bhargava and Iyer [33] both describe query rewrite optimizations to convert outer joins to inner joins, or full outer joins to left (or right) outer joins, via the analysis and exploitation of *strong predicates* specified in the query. Such rewrite transformations are beyond the scope of this thesis. However, we do recognize that the presence of null-intolerant predicates in a **Where** or **Having** clause converts lax dependencies and equivalence constraints into strict ones. This result follows from the following algebraic identities [98, pp. 50]:

$$\sigma[C_1](R \xrightarrow{C_2} S) \equiv \sigma[C_1 \wedge C_2](R \times S) \quad \text{if } C_1 \text{ rejects null values on } \alpha(S) \tag{3.5}$$

$$\sigma[C_1](R \xleftrightarrow{C_2} S) \equiv \sigma[C_1](R \xleftarrow{C_2} S) \quad \text{if } C_1 \text{ rejects null values on } \alpha(S) \tag{3.6}$$

Consequently we modify the RESTRICTION algorithm to determine when lax dependencies can be transformed to strict ones by recognizing the existence of any null constraints. To do so simply involves modifying the MARK DEFINITE procedure to augment the set of vertices in an FD-graph that can be treated as definite to include those marked as pseudo-definite. If the input parameter to MARK DEFINITE() is part of the null-supplying side of a left- or full outer join, then all other pseudo-definite attributes—those that are guaranteed not to be **Null** except for the generated all-**Null** row—can also be marked as definite attributes by the calling SET DEFINITE procedure. Attributes that are marked as pseudo-definite as the result of a nested outer join can also be treated as definite, since the restriction condition will eliminate the all-**Null** row for the entire null-supplying table expression. Subsequently, the CONVERT DEPENDENCIES procedure can convert lax dependencies and equivalence edges based on these attributes to strict ones.

```
542  Procedure: MARK DEFINITE
543  Purpose: Construct a set of vertices that can be considered definite.
544  Note: Replaces previous version of pseudo-code with lines numbered 184 through 196.
545  Inputs: FD-graph G, vertex v.
546  Output: A set D consisting of those vertices related to v that can be made definite.
547  begin
548    D ← ∅;
549    for each v_i ∈ V^A such that (v, v_i) ∈ E^E do
550      if Nullability[v_i] is not Definite and Colour[v_i] is not Gray then
551        D ← D ∪ v_i
552        fi
553      od ;
```

```
554     --Add to D those null-supplying attributes that are pseudo-definite.
555     if ∃ J ∈ V^J such that (J, v) ∈ E^J then
556       --Construct the set S of attributes related to v via a null constraint.
557       S ← ∅;
558       while J ≠ ∅ do
559         --Add to S those null-supplying attributes related to outer join j_i.
560         S ← S ∪ v_i ∈ V^A such that (J, v_i) ∈ E^J and v_i ≠ v;
561         J ← v_j ∈ V^J such that (J, v_j) ∈ E^J
562         od;
563       for each v_i ∈ S do
564         if Nullability[v_i[G_Q]] is Pseudo-definite and Colour[v_i] is not Gray then
565           D ← D ∪ v_i
566           fi
567         od
568       fi;
569     return D
570     end
```

3.5 Proof of correctness

In Section 3.2 we outlined, from a relational theory standpoint, those strict and lax functional dependencies and equivalence constraints that hold for each algebraic operator in an expression e, and in Section 3.4 gave an algorithm to construct an FD-graph that represented those dependencies and constraints. In this section we give a formal proof that the algorithms to construct an FD-graph are correct. The proof is by induction on the height of the expression tree. We assume that the expression e correctly reflects the semantics of the original ANSI SQL query.

THEOREM 5 (FD-GRAPH CONSTRUCTION ALGORITHMS)
Consider an arbitrary algebraic expression e comprised of the operators defined in Section 2.3 over an arbitrary schema. The result of e is an extended table Q corresponding to the definitions in Section 2.2. Suppose an FD-graph G is constructed for e, as described in Section 3.3.1, using the algorithms for each algebraic operator as described in Section 3.4. Then every dependency and constraint inferred from G must hold in every instance $I(Q)$ that could result from expression e.

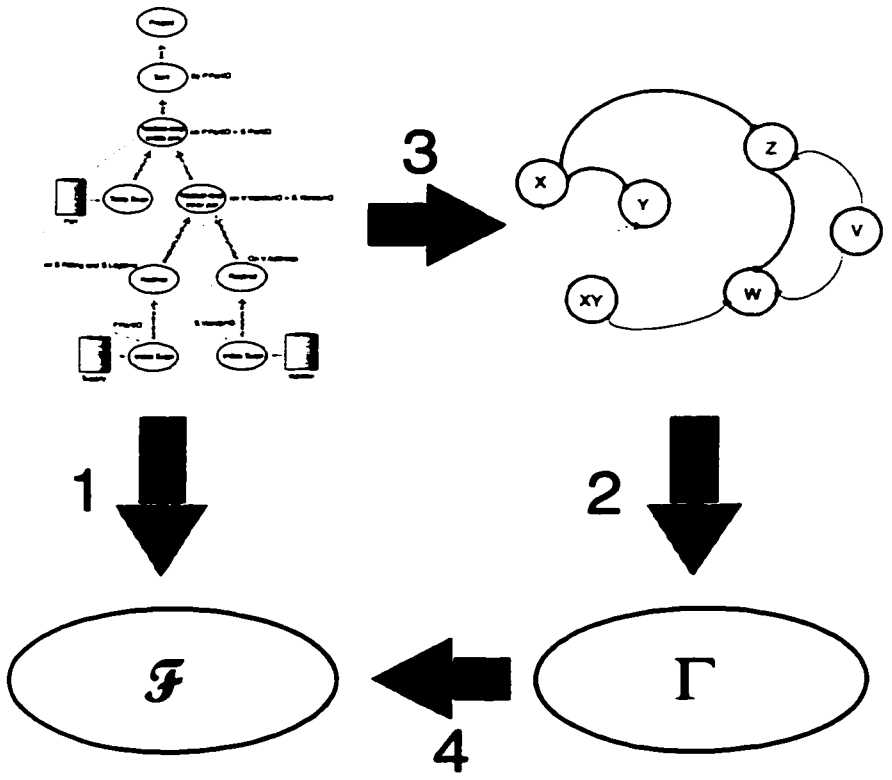


FIGURE 3.10: FD-graph proof overview for strict functional dependencies.

3.5.1 Proof overview

As discussed earlier, our interest in maintaining FD-graphs is to exploit strict functional dependencies, strict superkeys, and strict equivalences in optimizing algebraic expression trees. However, since we derive strict dependencies and equivalences from their lax equivalents when conditions warrant, we must also show that we correctly maintain FD-graph constructs pertaining to lax functional dependencies, lax equivalence constraints, and null constraints. Figure 3.10 illustrates an overview of the proof procedure as applied to strict functional dependencies only. Previously, in Section 3.2, we described a subset of the set of strict functional dependencies $\mathcal{F}$ that are implied by the semantics of each algebraic operator and hold in its result Q (denoted as transformation (1) in Figure 3.10). The set of strict functional dependencies Γ represented in an FD-graph—transformation (2) in Figure 3.10—is provided by an extended definition of the mapping function χ.

DEFINITION 46 (DEPENDENCIES AND EQUIVALENCE CONSTRAINTS IN AN FD-GRAPH) In Section 3.4 we introduced the function $\chi()$ that served to map attribute names to simple vertices and vice-versa in an FD-graph. We now redefine $\chi()$ to be a polymorphic function that provides the following correspondences:

- for each vertex $v \in V^A$ the function $\chi(v)$ returns the unique name of attribute v (and vice-versa).
- for each vertex $v \in V^R$ the function $\chi(v)$ returns the unique name given to the tuple identifier $\iota(R)$ of its corresponding base or derived table R (and vice-versa).
- $\chi(\{v_1, v_2, \ldots, v_n\}) = \{\chi(v_1), \chi(v_2), \ldots, \chi(v_n)\}$.

Furthermore, we extend χ to provide mappings from other features of an FD-graph to characteristics of its corresponding extended table as follows:

- $\chi(v \in V^C) = \{\chi(b) \mid (v, b) \in E^C\}$.
- $\chi(E^C) = \{\chi(v) \longrightarrow \chi(y) \mid v \in V^C \wedge (v, y) \in E^C\}$.
- $\chi(E^F) = \{\chi(x) \longrightarrow \chi(y) \mid (x, y) \in E^F\}$.
- $\chi(E^R) = \{\chi(x) \longrightarrow \chi(y) \mid (x, y) \in E^R\} \cup \{Y \longrightarrow \chi(x) \mid x \in E^R \wedge Y = \{\chi(y) \mid (x, y) \in E^R\}\}$.
- $\chi(E^f) = \{\chi(x) \longmapsto \chi(y) \mid (x, y) \in E^f\}$.
- $\chi(E^E) = \{\chi(x) = \chi(y) \mid (x, y) \in E^E\}$.
- $\chi(E^e) = \{\chi(x) \simeq \chi(y) \mid (x, y) \in E^e\}$.

These additional mappings define the set of functional dependencies and equivalence constraints represented by an FD-graph. Note that χ cannot be used to map functional dependencies or equivalence constraints that may exist in an extended table to components in its corresponding FD-graph since not all such dependencies and constraints may be captured by the FD-graph.

We denote the complete set of strict dependencies modelled by an FD-graph with Γ, which is equivalent to $\chi(E^C) \cup \chi(E^F) \cup \chi(E^R)$. The combined set of strict and lax dependencies γ modelled by an FD-graph is equivalent to $\Gamma \cup \chi(E^f)$, since by inference rule FD5 (weakening) any strict functional dependency is a lax dependency. Similarly, the complete set of strict equivalence constraints modelled by an FD-graph, which we denote as

Type	*Mapping*
strict functional dependencies	$\Gamma = \chi(E^C) \cup \chi(E^F) \cup \chi(E^R)$
lax functional dependencies	$\gamma = \Gamma \cup \chi(E^f)$
strict equivalence constraints	$\Xi = \chi(E^E)$
lax equivalence constraints	$\xi = \Xi \cup \chi(E^e)$

TABLE 3.2: Summary of constraint mappings in an FD-graph

Ξ, and the complete combined set of strict and lax equivalence constraints, which we denote as ξ, are equivalent to $\chi(E^E)$ and $\chi(E^e) \cup \chi(E^E)$ respectively. These definitions are summarized in Table 3.2.

What remains to be proved is that the transformation (3) is well-defined and results in the subset relationship $\Gamma \subseteq \mathcal{F}$ represented by (4) in Figure 3.10. Proof of (3) shows that each algorithm in Section 3.4 manipulates its input FD-graph(s) correctly, resulting in an FD-graph that correctly mirrors the schema of the extended table that constitutes the result Q of the algebraic expression e:

(1) $v \in V^A \wedge$ colour[v] is white $\iff \chi(v) \in \alpha(Q)$

(2) $v \in V^R \wedge$ colour[v] is gray $\iff \chi(v) \in \iota(Q)$

(3) $v \in V^A \wedge$ colour[v] is gray $\iff \chi(v) \in \kappa(Q)$

(4) $v \in V^A \cup V^R \wedge$ colour[v] is black $\iff \chi(v) \in \rho(Q)$

(5) $v \in V^A \wedge$ Nullability[v] is definite $\implies \chi(v)$ is a definite attribute in $I(Q)$.

Proof of (4) requires that we show for each strict dependency $f \in \Gamma$ that its counterpart in $\mathcal{F}$ holds; that is, $\Gamma \subseteq \mathcal{F}$. Similarly, we must also prove that any null constraints, strict equivalence constraints (the set Ξ), lax functional dependencies (the set γ), and lax equivalence constraints (the set ξ) derived from an FD-graph are guaranteed to hold in our extended relational model. More formally, for any sets of attributes X and Y and atomic attributes w and z such that $XYzw \subseteq sch(Q)$:

(1) $X \longrightarrow Y \in \Gamma \implies \chi(X) \longrightarrow \chi(Y) \in \mathcal{F}_Q$

(2) $X \longmapsto Y \in \gamma \implies \chi(X) \longmapsto \chi(Y) \in \mathcal{F}_Q$

(3) $w \stackrel{\omega}{=} z \in \Xi \implies \chi(w) \stackrel{\omega}{=} \chi(z) \in \mathcal{E}_Q$

(4) $w \simeq z \in \xi \implies \chi(w) \simeq \chi(z) \in \mathcal{E}_Q$

(5) ISNULLCONSTRAINT$(w, z) \implies w \dashv z$ holds in $I(Q)$.

where ISNULLCONSTRAINT is a procedure defined in Section 3.5.1.2 below that determines whether or not a null constraint exists between two attributes in $sch(Q)$.

3.5.1.1 Assumptions for complexity analysis

Along with the proof of correctness for each algorithm, we will also present a brief worst-case complexity analysis of each. For this analysis we assume that an extended FD-graph is implemented as follows. Each set of vertices in V is represented using a hash table. The hash key of attribute vertices is their unique name, which corresponds to our vertex mapping function χ. The hash tables for the other vertex sets utilize the polymorphic version of χ. We further assume that FD-graph edges are represented using adjacency lists. Directed edges are linked as usual from their source vertex; we assume that undirected edges between attribute vertices v_i and v_j in V^A which represent equivalence edges appear in the adjacency lists for both v_i and v_j, thereby doubling their maintenance cost. With this construction, we assume that we can perform:

- constant-time ($O(1)$) vertex lookup and insertion, and
- constant-time edge lookup, insertion, and deletion.

We believe these assumptions are defensible with the availability of sophisticated, low-cost hash-based data structures. For example, consider the dynamic hash tables recently proposed by Dietzfelbinger, Karlin, and Mehlhorn et al. [80], which use space proportional to the size of the input. Their construction, which utilizes a perfect hashing algorithm, provides $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions. Through the use of these hash table we can achieve $O(1)$ lookup and insertion of vertices in V, and moreover achieve constant time lookup, insertion, and deletion of edges in an FD-graph by using them to implement FD-graph adjacency lists.

Finally, as with restriction conditions, we assume that an outer join's `On` condition p is specified in conjunctive normal form, and that the nullability function $\eta(p, X)$ returns a result in time proportional to the size of p, which we write as $\|P\|$ (see Table 3.3).

Ordinarily, we will express the worst-case expected running time as a function of the size of the input FD-graph(s). However, we will require a different yardstick for procedure BASE-TABLE, which constructs an FD-graph from scratch. We will use the notation in Table 3.3 to denote the sizes of various inputs to particular algorithms. For procedures like CARTESIAN PRODUCT, which construct an FD-graph by combining the components of its two inputs, we will differentiate between individual FD-graph components by using subscripts, as in $\|V_S\|$. Note that since we are using maximal FD-graph sizes to state complexity bounds, it follows that $\|A\| < \|V\|$, and similarly $\|F\| < \|V\|$. Finally, we also note that $\|E\|$ is $O(\|V\|^2)$ since the number of edges between any two vertices is bounded.

$\|\|A\|\|$	Maximum cardinality of a set of attributes A
$\|\|P\|\|$	Maximum size of a CNF predicate P
$\|\|K\|\|$	Maximum number of primary key and unique indexes on a table R
$\|\|U\|\|$	Maximum number of uniqueness constraints on a table R
$\|\|F\|\|$	Maximum number of aggregate functions in a group-by projection
$\|\|V\|\|$	Maximum number of vertices in an FD-graph G
$\|\|E\|\|$	Maximum number of edges in an FD-graph G

TABLE 3.3: Notation for procedure analysis.

3.5.1.2 Null constraints

Procedure ISNULLCONSTRAINT determines whether or not a null constraint (see Definition 38 on page 95) exists between two attributes in $sch(Q)$.

```
571  Procedure: ISNULLCONSTRAINT
572  Purpose: Determine if a null constraint holds between attributes w and z.
573  Inputs: FD-graph G, attributes w and z.
574  Output: True if there is a null constraint path from χ(w) to χ(z), and false otherwise.
575  begin
576    if Nullability[χ(w)] is Pseudo-definite then
577      if Nullability[χ(z)] is Pseudo-definite then
578        if ∃ M, N ∈ V^J such that {(M, χ(w)), (N, χ(z))} ⊆ E^J then
579          while M ≠ ∅ do
580            if M = N return true fi;
581            M ← J such that J ∈ V^J and (M, J) ∈ E^J
582            od
583          fi
584        fi
585      fi
586    return false
587    end
```

Procedure ISNULLCONSTRAINT determines if there exists a null constraint path (Definition 45) between two attributes $\chi(w), \chi(z) \subseteq V^A$. The algorithm relies on the characteristics of vertices in $V^J[G]$, in particular that the edges in E^J are acyclic, and that at most one outer join vertex can directly reference any vertex in V^A. As an example, consider an expression e consisting of several left-deep nested outer joins. The FD-graph representing e will then contain one outer join vertex for each of the k outer joins in e, with $k-1$ directed edges in E^J forming a path through the k vertices in V^J. In the case of nested outer joins consisting of table expressions containing multiple (unnested) outer joins or full outer joins, a tree of edges in E^J will result.

LEMMA 11 (ANALYSIS)
Given as input two attributes w and z and an FD-graph G, procedure ISNULLCONSTRAINT executes time proportional to $O(\|V\|)$.

PROOF. If attributes w and z each participate in an outer join, then the main loop beginning on line 579 and ending on line 582 must terminate if the invariants in both $\tau(G)[V^J]$ and $\tau(G)[E^J]$ hold. If either does not participate in an outer join, there is no loop. Since no vertex in V^J is considered more than once, we conclude that ISNULLCONSTRAINT executes in time proportional to $O(\|V\|)$. □

LEMMA 12 (CORRECTNESS OF ISNULLCONSTRAINT)
Given as input two attributes w and z and an FD-graph G, procedure ISNULLCONSTRAINT returns *true* if and only if there exists a null constraint path in G from $\chi(w)$ to $\chi(z)$.

PROOF. Straightforward from Definition 45 and from the invariants for outer join vertices V^J and outer join edges E^J that must hold in the result of e. □

3.5.2 Basis

Our proof that the procedures specified in Section 3.4 are correct is by induction on the height of the expression tree e representing the query. The basis of the induction is to show that we correctly construct an FD-graph for a base table R, since only base tables are permitted as the leaves of an expression tree.

CLAIM 28 (ANALYSIS)
Procedure BASE-TABLE executes in time proportional to $O(\|V\|)$.

PROOF (SKETCH). It is immediate that each main loop in Procedure BASE-TABLE executes over a finite set:

- Lines 6 through 14 are executed once for each attribute a_i in $\mathcal{R}_\alpha(R)$, and hence in time proportional to $O(\|A\|)$;
- Lines 21 through 35 execute once for each primary key or unique index on $\mathcal{R}_\alpha(R)$, and hence in time proportional to $O(\|K\|)$; and
- Lines 36 through 55 execute once for each unique constraint defined on $\mathcal{R}_\alpha(R)$, and hence in time proportional to $O(\|U\|)$.

Clearly Procedure BASE-TABLE terminates; hence our claim of overall execution time proportional to $O(\|A\| + \|K\| + \|U\|)$ follows. Since uniqueness constraints cannot be trivially redundant, for a given extended table R BASE TABLE must construct at least $\|A_R\| + \|K_R\| + \|U_R\|$ vertices; hence the overall running time is bounded by $O(\|V\|)$. □

CLAIM 29 (SCHEMA OF A BASE TABLE)
Procedure BASE-TABLE constructs an FD-graph whose vertices correctly represent the schema of an extended base table R.

PROOF (SKETCH).

$\alpha(R) = \mathcal{R}_\alpha(R)$.

By a straightforward analysis of the BASE-TABLE procedure, it is easy to see that each attribute in $\alpha(R)$ is represented by a white vertex in V^A (line 7). Each of R's attribute vertices in V^A are coloured white (line 8).

$\iota(R) =$ a new tuple identifier attribute.

Only one tuple identifier vertex in V^R is constructed in G (line 16) and its colour is gray (line 17). Hence this vertex correctly represents $\iota(R)$. Note also that R's tuple identifier vertex is the source of only strict full arcs (line 19).

$\rho(R) = \kappa(R) = \emptyset$.

Obvious.

Definite attributes. Each attribute in $\alpha(R)$ has a vertex in V^A and the 'Nullability' attribute of each vertex is appropriately marked as 'Definite' or 'Nullable' depending on the schema definition of $\mathcal{R}_\alpha(R)$ (lines 10 and 12 respectively). □

CLAIM 30 (CONSTRAINTS AND DEPENDENCIES IMPLIED BY A BASE TABLE)
Procedure BASE-TABLE constructs an FD-graph whose vertices and edges correctly represent the functional dependencies that hold in an extended base table R.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By a straightforward analysis of the BASE-TABLE procedure, it is easy to see that edges in E^F are constructed only for (1) the tuple identifier vertex v_K as the determinant of each attribute in R (line 19), (2) for primary keys or unique indexes whose target is v_K (line 34), and (3) for unique constraints whose attributes are defined as definite (line 53). Moreover, compound vertices in V^C are only constructed for composite keys, either strict (line 25) or lax (line 40). There are no edges constructed in E^R since G_R will contain only a single tuple identifier vertex. Edges in E^C are constructed only for composite strict candidate keys that stem from a primary key constraint or unique index (line 25) and strict or lax candidate keys that stem from a unique constraint (line 40); and no compound vertex is the target of any edge in $E[G]$. Hence for any strict functional dependency $f : X \longrightarrow Y \in \Gamma$ the strict functional dependency $\chi(X) \longrightarrow \chi(Y)$ holds in $I(R)$. □

PROOF (LAX FUNCTIONAL DEPENDENCIES). BASE-TABLE constructs edges in E^J only for unique constraints (line 50) containing at least one nullable attribute. Hence it is easy to see that any lax functional dependency $f : X \longmapsto Y \in \gamma$ implies that the lax functional dependency $\chi(X) \longmapsto \chi(Y)$ holds in $I(R)$. □

PROOF (EQUIVALENCE AND NULL CONSTRAINTS). As there are no other edges constructed by BASE-TABLE, $E^E = E^e = E^J = \emptyset$, correctly mirroring the lack of any equivalence constraint or null constraint implied by the definition of table $\mathcal{R}_\alpha(R)$. □

THEOREM 6 (FD-GRAPH CONSTRUCTION FOR BASE TABLES)
The interpretation $\tau(G)$ of the FD-graph G constructed by the procedure BASE TABLE for an arbitrary base table R correctly reflects the attributes in R, their characteristics, and both the strict and lax functional dependencies inherent in R.

PROOF. Follows from Claims 28 through 30. □

3.5.3 Induction

The basis of the induction, proved above, shows that the procedure BASE TABLE correctly constructs an FD-graph for base tables, which constitute the leaves of an expression tree. We now prove that the procedure used to construct an FD-graph for each algebraic operator is correct. More formally, we intend to show that given an operator $\odot$ at the root

of an expression tree e of height n, the procedure to construct an FD-graph that represents the characteristics of the dependencies that hold for $\odot$ produces a correct FD-graph assuming that its input FD-graph(s) for each subtree of height $k \mid 0 \leq k < n$ are correct.

3.5.3.1 Projection

Given an arbitrary expression tree e' of height $n > 0$ rooted with a unary projection operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure PROJECTION based on the input FD-graph G_R for expression e of height $n - 1$ correctly reflects the characteristics (attributes, equivalences, and dependencies) of the derived extended table Q resulting from the projection of expression e which produced the input extended table R. The PROJECTION procedure modifies an FD-graph to:

1. model the projection (the algebraic operator π) of a set of tuples over a set of attributes A;
2. model duplicate elimination (the algebraic operator π_{Dist}) by constructing a new candidate key for the FD-graph of e' by creating an FD-path from $\chi(A)$ to the new tuple identifier representing a tuple in Q; and
3. model the dependencies implied by the use of scalar functions in the projection list.

To model these changes, the PROJECTION procedure:

1. colours black all vertices in $v \in V^A$ such that $\chi(v) \notin A$; (line 111);
2. constructs a new tuple identifier vertex v_P (line 118) to uniquely identify a tuple in Q when duplicate elimination is necessary, and:
   (a) adds a strict dependency edge from v_P to each attribute that exists in the `Select` list and hence survives projection (line 122);
   (b) if $\|A\| > 1$, PROJECTION adds a compound vertex P to G_Q (line 128) and constructs the appropriate dotted edges to its components (line 131) and a full edge to v_P (line 133);
   (c) otherwise, if $\|A\| = 1$ then PROJECTION simply adds a full edge between $\chi(A)$ and v_P (line 135).
3. for each scalar function λ in the `Select` list, PROJECTION calls the EXTENSION procedure to add the strict dependencies from λ's inputs to the vertex representing the function's result (line 105). The EXTENSION procedure constructs a single strict dependency between the function's inputs and its result. The vertex corresponding to

the result is constructed first (line 67), followed by the construction of the strict edge to λ from the given tuple identifier (line 68), followed by the construction of the strict edge denoting the dependency (line 90 or 92), possibly including the construction of a compound node if λ has more than one input parameter (line 85).

By the induction hypothesis, $\tau(G_R)$, which represents the characteristics of the FD-graph G_R that models the expression tree e of height $n-1$, is correct. We proceed with the proof of PROJECTION by first stating that the procedure terminates.

CLAIM 31 (ANALYSIS)
Procedure PROJECTION completes in time proportional to $O(\|V\|^2)$.

PROOF. The first step of the PROJECTION procedure is to copy the input FD-graph (line 101), which will take time proportional to $O(\|V\| + \|E\|)$. The second step is to create vertices in V^A that correspond to scalar functions in the **Select** list (lines 103 through 108). By observation, it is clear that for a scalar function $\lambda(X)$ the EXTENSION procedure executes in time proportional to $\|X\|$. Since the set of arguments to any scalar function may include each attribute in the FD-graph, this second phase executes in time proportional to $O(\|A\| \times \|V\|)$. The third step (lines 109 through 113) colours black each attribute that does not survive projection, and hence executes in $O(\|V\|)$ time. The final two steps are necessary only if the projection involves duplicate elimination. First, lines 120 through 124 create strict dependency edges from the new tuple identifier v_P to each white vertex in V^A, and hence executes in $O(\|A\|)$ time. Finally, lines 130 through 132 create dotted edges for the compound vertex representing the superkey of the result, again in $O(\|A\|)$ time. Since $\|A\| < \|V\|$ and $\|E\|$ is $O(\|V\|^2)$, the total execution time is proportional to $O(\|V\|^2)$. □

LEMMA 13 (SCHEMA OF THE RESULT OF PROJECTION)
Procedure PROJECTION constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = \pi_{All}[A](e)$ for an arbitrary expression e whose result is the extended table R.

PROOF.

$\alpha(Q) = A$.

We need to show that the white vertices in $V^A[G_Q]$ correctly reflect that $\alpha(Q)$ is equal to the set A. We first note that PROJECTION does not colour any preexisting vertex white. Thus a white attribute vertex $w \in V^A$ can arise from one of two possible sources. Either w denotes the result of a scalar function λ, added by procedure EXTENSION (line 67) and

coloured white (line 106), or w was already a white vertex in G_R. In both cases $\chi(w) \in A$, since any white attribute vertex $v \in V^A[G_R]$ is coloured black (line 111) if $\chi(v) \notin A$.

$\iota(Q) = \iota(R)$.

Obvious.

$\kappa(Q) = \kappa(R) \cup \kappa(\Lambda)$.

Obvious.

$\rho(Q) = \rho(R) \cup \{\alpha(R) \setminus A\}$.

PROJECTION retains all vertices in the FD-graph G_R and colours black those vertices representing real attributes of R that are not in A (line 111).

Definite attributes. By observation, PROJECTION does not alter the nullability characteristic of any attribute vertex in G_R. Hence if $w \in V^A[G_R]$ is marked definite, then by the induction hypothesis $\chi(w)$ is a definite attribute in R and consequently $\chi(w)$ is also definite in Q. If $w \notin V^A[G_R]$ then w must represent a scalar function λ. However, attribute vertices constructed by EXTENSION for scalar functions are nullable, since we assume that the result of $\lambda() \in A$ is possibly `Null` (line 71). □

LEMMA 14 (DEPENDENCIES AND CONSTRAINTS IN THE RESULT OF PROJECTION) Procedure PROJECTION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = \pi_{All}[A](e)$ for an arbitrary expression e whose result is the extended table R.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the corresponding strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

Case (1). If $f : X \longrightarrow Y \in \Gamma$ in G_R, then by the induction hypothesis the dependency $\chi(X) \longrightarrow \chi(Y)$ held in R. If so, then $\chi(X) \longrightarrow \chi(Y)$ must also hold in $I(Q)$, a contradiction. This is because $\chi(XY) \subseteq sch(R)$, procedure PROJECTION does not alter any existing edges in G_R, and the projection operator π_{All} does not alter or remove any existing strict functional dependency from R.

Case (2). Otherwise, $f \notin \Gamma$ in G_R. In the case of π_{All} the PROJECTION procedure does not alter existing edges nor add additional edges to E^F except in the case of scalar functions. However, the EXTENSION procedure is only called by PROJECTION to extend

Q with a scalar function λ, and edges to E^F are added in only two cases: to add the edge $(v_I, \chi(\lambda))$ (line 68) and the edge $(P, \chi(\lambda))$ to denote the strict dependency between λ and its input arguments (lines 90 or 92). Hence f can only exist in G_Q if $\chi(Y)$ represents the result of a scalar function, and by the definition of the projection operator f' must hold in $I(Q)$, a contradiction.

Hence we conclude that if $f \in \Gamma$ then $\chi(X) \longrightarrow \chi(Y) \in \mathcal{F}_Q$. □

PROOF (LAX DEPENDENCIES AND EQUIVALENCE CONSTRAINTS). Since PROJECTION does not modify or create any edges in E^E or E^e, and the semantics of π_{All} does not affect equivalence constraints that hold in the input, then if $e : w \overset{\omega}{=} z \in \Xi$ then $e \in \Xi$ in G_R and by the induction hypothesis the strict equivalence constraint $\chi(w) \overset{\omega}{=} \chi(z)$ holds in $I(Q)$. An identical situation occurs with lax equivalence constraints. □

PROOF (NULL CONSTRAINTS). Procedure PROJECTION does not mark any attributes as pseudo-definite, nor does it construct any vertices in V^J or edges in E^J. Hence if ISNULLCONSTRAINT(w, z) returns *true* in G_R then by the induction hypothesis the null constraint $w \dashv z$ holds in e. ISNULLCONSTRAINT(w, z) must also return *true* in G_Q since $w \dashv z$ still holds in Q. □

LEMMA 15 (SCHEMA OF THE RESULT OF DISTINCT PROJECTION)
Procedure PROJECTION constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = \pi_{Dist}[A](e)$ for an arbitrary expression e whose result is the extended table R.

PROOF.

$\alpha(Q) = A$.

The proof for this component of $sch(Q)$ is identical to that for projection (Lemma 13 above).

$\iota(Q) =$ a new tuple identifier attribute.

In the case of distinct projection, the PROJECTION procedure add a new tuple identifier vertex v_P to G_Q (line 118), which is coloured gray to denote the new tuple identifier attribute $\iota(Q)$. The existing tuple identifier v_I is coloured black (line 115) to denote its addition to $\rho(Q)$.

$\kappa(Q) = \kappa(R) \cup \kappa(\Lambda)$.

Obvious.

$\rho(Q) = \rho(R) \cup \iota(R) \cup \{\alpha(R) \setminus A\}$.

By observation, PROJECTION colours real attributes $\alpha(R) \notin A$ black, denoting their addition to $\rho(Q)$, and colours the tuple identifier vertex v_I black to denote its move to $\rho(Q)$ as well.

Definite attributes. The proof of the correct modelling of definite attributes for distinct projection is identical to that for projection. □

LEMMA 16 (CONSTRAINTS IN THE RESULT OF DISTINCT PROJECTION)
Procedure PROJECTION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = \pi_{Dist}[A](e)$ for an arbitrary expression e whose result is the extended table R.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the corresponding strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

There are four possible scenarios:

- *Case (1).* If $Y \equiv v_P$, representing $\iota(Q)$, then $\chi(X) \equiv A$, as by observation the only edge added to E^F with v_P as a dependent is to represent the dependency $A \longrightarrow \iota(Q)$ (line 133 or 135).

- *Case (2).* If $X \equiv v_P$ then $\chi(Y) \in A$, because the only edge added to E^F with v_P as the determinant is to represent the strict dependency between the new tuple identifier and an attribute in A (line 122).

- *Case (3).* If $\chi(XY) \subseteq sch(R)$ and $f : X \longrightarrow Y \in \Gamma$ in G_R, then by the induction hypothesis the dependency $\chi(X) \longrightarrow \chi(Y)$ held in R. If so, then $\chi(X) \longrightarrow \chi(Y)$ must also hold in $I(Q)$, a contradiction. This is because $\chi(XY) \subseteq sch(R)$, procedure PROJECTION does not alter any edges that existed in G_R, and the projection operator π_{Dist} does not alter or remove any existing strict functional dependency from its input.

- *Case (4).* Otherwise, $f \notin \Gamma$ in G_R and neither $\chi(X)$ nor $\chi(Y)$ represents the new tuple identifier $\iota(Q)$. There are two remaining possibilities: either (1) $X \equiv \{A\}$, so that $(X, Y) \in E^C$ (line 131) representing the reflexive dependency $\chi(X) \longrightarrow \chi(Y)$, or (2) Y is a scalar function and X is its input parameters, denoting the strict functional dependency $\chi(X) \longrightarrow \chi(Y)$. Both these dependencies hold in $I(Q)$.

Since no other edges are added to G_Q by the PROJECTION procedure, we conclude that if $f : X \longrightarrow Y \in \Gamma$ then $\chi(X) \longrightarrow \chi(Y) \in \mathcal{F}_Q$. □

PROOF (LAX DEPENDENCIES AND CONSTRAINTS). As with the projection operator, distinct projection does not invalidate any functional dependency, equivalence constraint, or null constraint that held in its input. PROJECTION does not introduce any new edges into E^f or E^e, and hence any such constraint that appears in $\chi(E^f)$ or $\chi(E^e)$ represents one that held in R and continues to hold in Q. □

THEOREM 7 (PROJECTION)
Procedure PROJECTION correctly constructs an FD-graph G_Q corresponding to the projection $\pi[A](e)$ for any algebraic expression e.

PROOF. Follows from Claim 31 and Lemmas 13 through 16. □

3.5.3.2 Cartesian product

Given an arbitrary expression tree e' of height $n > 0$ rooted with a binary Cartesian product operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure CARTESIAN PRODUCT based on two input FD-graphs G_R and G_T for expressions e_R and e_T, one or both having a maximum height $n - 1$, correctly reflects the characteristics (attributes, equivalences, and dependencies) of the derived table Q resulting from the Cartesian product of expressions e_R and e_T.

CLAIM 32 (ANALYSIS)
Procedure CARTESIAN-PRODUCT completes in time proportional to $O(\|V\| + \|E\|)$.

PROOF. Obvious, since the two input FD-graphs must be copied into the FD-graph of the result. □

LEMMA 17 (SCHEMA OF THE RESULT OF CARTESIAN PRODUCT)
Procedure CARTESIAN PRODUCT constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = e_R \times e_T$ for arbitrary expressions e_R and e_T whose results are the extended table R and T respectively.

PROOF.

$\alpha(Q) = \alpha(S) \cup \alpha(T)$.

Obvious.

$\iota(Q) =$ a new tuple identifier attribute.

Procedure CARTESIAN-PRODUCT constructs a new tuple identifier vertex v_K (line 148) and colours it gray (line 149) to represent $\iota(Q)$. The existing tuple identifiers for R and T are coloured black to denote their addition to $\rho(Q)$ (lines 152 and 155).

$\kappa(Q) = \kappa(R) \cup \kappa(T)$.

Obvious.

$\rho(Q) = \rho(R) \cup \rho(T) \cup \iota(R) \cup \iota(T)$.

Obvious.

Definite attributes. By observation, CARTESIAN-PRODUCT does not alter the nullability characteristic of any attribute vertex in G_Q. Hence any attribute vertex $w \in V^A$ marked definite in either input will remain marked definite in G_Q, mirroring the semantics of the Cartesian product operator. □

LEMMA 18 (CONSTRAINTS IN THE RESULT OF CARTESIAN PRODUCT)
Procedure CARTESIAN-PRODUCT constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = e_R \times e_T$ for arbitrary expressions e_R and e_T whose results are the extended tables R and T respectively.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

Case (1). First, consider the case where $\chi(XY) \subseteq sch(R)$ and $f \in \Gamma$ in G_R. By the induction hypothesis f' held in $I(R)$. If so, then f' must also hold in $I(Q)$, a contradiction. This is because $\chi(XY) \subseteq sch(Q)$, procedure CARTESIAN-PRODUCT does not alter any existing edges in G_R, and the Cartesian product operator does not alter or eliminate any existing strict functional dependencies from either of its inputs. An identical argument can be used if $\chi(XY) \subseteq sch(T)$.

Case (2). If $f \notin \Gamma$ in G_R or G_T, then there is only one possible instance of such an edge existing in Γ in G_Q: that edge must include the newly-constructed row identifier vertex v_K added to G_Q (line 148), and that is the edge $(X, Y) \in E^R$ which represents the strict functional dependency between the new tuple identifier and the tuple identifiers from either R or T (and vice-versa). □

PROOF (LAX DEPENDENCIES AND CONSTRAINTS). As with the projection operator, Cartesian product does not invalidate any functional dependency, equivalence constraint, or null constraint that held in its input. CARTESIAN PRODUCT does not introduce any new edges into E^f or E^e, and hence any such constraint that appears in $\chi(E^f)$ or $\chi(E^e)$ represents one that held in either R or T and continues to hold in Q. □

THEOREM 8 (CARTESIAN PRODUCT)
Procedure CARTESIAN PRODUCT correctly constructs an FD-graph G_Q corresponding to $e_R \times e_T$ for two arbitrary algebraic expressions e_R and e_T.

PROOF. Follows from Claim 32 and Lemmas 17 and 18. □

3.5.3.3 Restriction

Given an arbitrary expression tree e' of height $n > 0$ rooted with a unary restriction operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure RESTRICTION based on the input FD-graph G_R for expression e of height $n-1$ correctly reflects the characteristics (attributes, equivalences, and dependencies) of the derived table Q resulting from the restriction of expression e (with result R) by a predicate C.

By the induction hypothesis, $\tau(G_R)$, which represents the characteristics of the FD-graph G_R that models the expression tree e of height $n-1$, is correct. We proceed with the proof of RESTRICTION by first proving that the procedure terminates.

CLAIM 33 (ANALYSIS)
Procedure RESTRICTION executes in time proportional to $O(\|E\|^2 + \|V\|^2 + (\|V\| \times \|P\|))$.

PROOF. By observation, note that the CNF preprocessing loop beginning on line 244 and the RESTRICTION procedure's main loop beginning on line 255 iterate once per conjunct in the combined predicate C'. As each atomic condition $P_i \in C'$ may contain one or two scalar functions $\lambda(X)$, the main loop will execute in time proportional to $O(\|V\| \times \|P\|)$, following from our previous analysis of the EXTENSION procedure in Claim 31 on page 153.

Subsequent to these two loops, the SETDEFINITE procedure first loops over all attribute vertices in G_Q (lines 164 through 169), and then processes each vertex added to the set S at most once (lines 170 through 181). The inner loop (lines 175 through 180) processes those vertices returned by MARK DEFINITE, which in the worst case is $O(\|V\|)$. Procedure MARK DEFINITE, as modified in Section 3.4.10, consists of two sections. The first, from lines 549 through 553, executes in time proportional to $O(\|E\|)$, *for each call* from procedure SET DEFINITE. The second section consists of two loops that consider null constraints that stem from the existence of outer joins. The first loop (lines 558 through 562) constructs the set S by ranging over all attribute vertices for each outer

join in e. Since no attribute vertex can be directly connected to more than one outer join vertex in V^J, its execution time is proportional to $O(\|V\|)$. The second loop (lines 563 through 567) ranges over the vertices in S, which can be at most $O(\|V\|)$. Hence the complete running time of SET DEFINITE is $O(\|V\| \times (\|V\| + \|E\|)$.

Finally, we observe that the main loop in procedure CONVERTDEPENDENCIES (lines 202 through 234) iterates once for each edge in E^f. Since no edge is ever added to E^f in the entire execution of RESTRICTION, it is immediate that CONVERTDEPENDENCIES must terminate. The execution time of CONVERTDEPENDENCIES is, therefore, $O(\|E\|^2)$ due to the inner loop over all equivalence edges, implicit on line 220. Since $\|V\| \times \|E\|$ is $O(\|V\|^2 + \|E\|^2)$, we therefore claim that procedure RESTRICTION terminates in time proportional to $O(\|E\|^2 + \|V\|^2 + (\|V\| \times \|P\|))$. □

LEMMA 19 (SCHEMA OF THE RESULT OF RESTRICTION)
Procedure RESTRICTION constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = \pi_{All}[A](e)$ for an arbitrary expression e.

PROOF. The proof of the correct construction of each vertex in G_Q that mirrors $sch(Q)$ is straightforward. We devote our attention to the correct modelling of definite attributes. By contradiction, assume that $w \in V^A[G_Q]$ is marked definite but $\chi(w)$ is not guaranteed to be definite in $I(Q)$.

Case (1). If $w \in V^A$ is coloured white in G_Q, then w cannot be a vertex added by RESTRICTION, since all such vertices (representing constants or the result of a scalar function λ) are coloured gray or black, respectively (lines 260, 265, 272, and 276). If $w \in V^A[G_R]$ is marked definite, then by the induction hypothesis $\chi(w)$ represents a definite attribute in e. Since RESTRICTION does not mark any vertices as pseudo-definite or nullable, w must remain a definite vertex in G_Q; and if already definite, setting w to be definite through its inclusion in a Type 1 or Type 2 condition in C has no effect. However, if $\chi(w)$ was guaranteed to be definite in $I(R)$, then by the semantics of the restriction operator it must remain definite in $I(Q)$; a contradiction.

Case (2). If $w \in V^A$ is coloured gray, then $\chi(w)$ is a constant in $\kappa(Q)$. If $w \in V^A[G_R]$ then by the induction hypothesis $\chi(w) \in \kappa(R)$. Otherwise, w was added to G_Q a part of the processing of either (1) a Type 1 condition (line 257) and coloured gray (line 265), or (2) a constant argument to a scalar function added by the EXTENSION procedure (line 74) and coloured gray (line 75). Once the nullability of these constants is established, they are never changed; procedure MARK-DEFINITE ensures that constants in the FD-graph remain unaltered (lines 550 and 564). Hence we conclude that if $w \in V^A[G_Q]$ and w is coloured gray and marked definite then $\chi(w)$ represents a definite constant in $\kappa(Q)$.

Case (3). If $w \in V^A[G_R]$ is coloured black and marked definite, then by the induction hypothesis $\chi(w) \in \rho(R)$ and is definite in $I(R)$. If so, w cannot be a vertex modified by RESTRICTION, since the main loop of the RESTRICTION algorithm deals only with Type 1 or Type 2 conditions which must refer only to attributes in $\alpha(R)$, and RESTRICTION does not mark any vertices in G_Q as pseudo-definite or nullable. Hence w must remain a definite vertex in G_Q. However, if w was guaranteed to be definite in $I(R)$, then by the semantics of the restriction operator it must remain definite in $I(Q)$; a contradiction.

Case (4). If $w \notin V^A[G_R]$ and w is coloured black, then w is a vertex that corresponds to the result of a scalar function λ since these are the only vertices created by RESTRICTION (lines 259, 271, and 275) that are coloured black (lines 260, 272, and 276 respectively). In each case, these vertices are marked definite (lines 264, 279, and 280) since they are created only when they participate in a false-interpreted, null-intolerant condition of Type 1 or Type 2, and once marked definite remain definite. Hence if w is marked definite, the semantics of the restriction operator guarantees that $\chi(w)$ will be definite in $I(Q)$.

Case (5). Otherwise, $w \in V^A$ represents a pseudo-definite or nullable attribute $\chi(w) \in \alpha(R) \cup \rho(R)$. There are three possible ways in which RESTRICTION can mark an existing vertex as definite:

- *Case (A).* The vertex w represents an attribute $\chi(w) \in \alpha(R)$ that is equated to a constant in a conjunctive, null-intolerant, false-interpreted predicate $P_i \in C$ (line 264). If this is the case, then clearly $\chi(w)$ cannot be `Null` in $I(Q)$ by our definition of restriction; a contradiction.

- *Case (B).* The vertex w represents an attribute $\chi(w) \in \alpha(R)$ that is equated to another attribute $\chi(z)$ in a conjunctive, null-intolerant, false-interpreted predicate $P_i \in C$ (line 279 or 280). Again, it is obvious that both $\chi(z)$ and $\chi(w)$ cannot be null in $I(Q)$; a contradiction.

- *Case (C).* The vertex w is marked definite by procedure SET DEFINITE due to a transitive strict equivalence constraint to some other definite attribute vertex $z \in V^A$.

  Consider the point in procedure SET DEFINITE at which such a attribute vertex w is altered (line 176). If w is so marked, then procedure MARK DEFINITE must have returned w as one of a set of attribute vertices that are either (1) directly equated to a definite vertex v_i through a strict equivalence edge in E^E or (2) related to a definite vertex through a null constraint path (line 174). The vertex v_i must be a definite vertex since only definite vertices are added to S (lines 167 and 178).

Now consider the correctness of procedure MARK DEFINITE. There are two possibilities for a vertex v_i to be added to the set D:

- *Case i.* There exists a strict equivalence edge $e : (v, v_i) \in E^E$. If $e \in E^E[G_R]$ then by the induction hypothesis e represented a valid strict equivalence constraint $\chi(v) \stackrel{\omega}{=} \chi(v_i)$ in R. By the definition of the restriction operator, this equivalence constraint continues to hold in $I(Q)$. Otherwise, if $e \notin E^E[G_R]$ then e must have been constructed by the RESTRICTION procedure (note that we need not consider the outcomes of procedure CONVERT DEPENDENCIES since it does not alter the nullability of an attribute vertex, and is executed only after procedure SET DEFINITE has completed). There are only two sources of such edges: lines 267 and 282. However, these correspond to the existence of conjunctive false-interpreted equivalence predicates of Type 1 or 2 in C. Therefore if v is definite (by one of Cases 1–4 above) then $\chi(v_i)$ must be definite in $I(Q)$.
- *Case ii.* Otherwise, v_i is added to the set D as the result of the existence of a null constraint path between v and v_i. Since RESTRICTION does not alter any vertex in V^J nor any edge in E^J, then the only way for a null constraint path $\langle v, v_i \rangle$ to exist in G_Q is if it existed in G_R, and if so then the null constraint $\chi(v) \dashv \chi(v_i)$ held in R.

  Consider the pseudo-code in procedure MARK DEFINITE that considers null constraint paths (lines 554 through 568). An attribute vertex v_i is added to the set S (line 560) only if (1) v_i is related to the identical outer join vertex J as v, meaning that $\chi(v)$ and $\chi(v_i)$ are both null-supplying attributes of the same outer join, or (2) v_i is a null-supplying attribute in an outer join J' that includes J as part of its null-supplying side (see Figure 3.8). Hence if $\chi(v_i) \in S$ then the null constraint path $\langle v, v_i \rangle$ exists in G_Q. However, v_i is not added to D unless v_i is still marked as pseudo-definite (line 564). This is because if v_i has already been marked as definite due to some other processing by RESTRICTION then the null constraint cannot hold, since neither $\chi(v)$ nor $\chi(v_i)$ can be `Null`.

Hence we conclude from the contradictions shown in Cases 1-5 above that if $w \in V^A[G_Q]$ is marked definite then $\chi(w)$ is guaranteed to be definite in $I(Q)$. □

LEMMA 20 (CONSTRAINTS IN THE RESULT OF RESTRICTION)
Procedure RESTRICTION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the

extended table Q that results from the algebraic expression $e' = \sigma[C](e)$ for an arbitrary expressions e whose results is the extended table R.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

Case (1). First, consider the case where $\chi(XY) \subseteq sch(R)$ and $f \in \Gamma$ in G_R. By the induction hypothesis f' held in $I(R)$. If so, then f' must also hold in $I(Q)$, a contradiction. This is because $\chi(XY) \subseteq sch(Q)$, procedure RESTRICTION does not alter any existing strict dependencies that result from the sets of edges $E^F \cup E^C \cup E^R$, and by definition the restriction operator σ only adds strict functional dependencies to Q.

Case (2). If $f \notin \Gamma$ in G_R, then f can be formed by the RESTRICTION procedure under the following circumstances:

- X and Y denote the vertices representing the input parameters to a scalar function $\lambda(X)$, added by EXTENSION on lines 90 or 92; or
- $\chi(X) \subset \chi(Y)$ where $Y \in V^C$ and represents the set of inputs to a scalar function $\lambda(\chi(Y))$ (line 88); or
- X denotes the gray tuple identifier vertex v_K and Y represents the scalar function $\lambda()$ (line 68); or
- X and Y are vertices that represent two attributes or function results $\chi(X)$ and $\chi(Y)$ respectively that are operands in a Type 1 or Type 2 equality condition (lines 266 or 281); or
- f is a converted lax functional dependency transformed by the procedure CONVERT DEPENDENCIES once each definite vertex has been so marked by the SET DEFINITE procedure. The four situations where this can occur involve (1) a lax key dependency with a single key attribute (line 207), (2) a lax dependency between two attributes (line 211), (3) a lax key dependency involving a wholly non-`Null` composite key (line 225), and (4) a lax dependency with a wholly non-`Null` composite determinant (line 229).

In each case, these modifications to G_Q correctly imply that the strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ must hold in $I(Q)$; a contradiction. □

PROOF (LAX FUNCTIONAL DEPENDENCIES). By observation, procedure RESTRICTION does not add edges to E^f to denote any additional lax functional dependencies implied by σ, nor are any edges in E^f deleted outright; they are only converted to strict dependency edges in E^F when valid to do so: when both the dependent and determinant vertices are marked as definite, which has already been proven in Lemma 19 above. Since $\gamma = \chi(E^F) \cup \chi(E^F)$, and since inference axiom FD5 (weakening) implies that every strict dependency is also a lax dependency, then $f : X \longmapsto Y \in \gamma$ must imply that $f' : \chi(X) \longmapsto \chi(Y)$ holds in $I(Q)$. □

PROOF (STRICT AND LAX EQUIVALENCE CONSTRAINTS). The proof of these constraints mirrors the above proofs for strict and lax functional dependencies. □

PROOF (NULL CONSTRAINTS). The sufficient conditions for proving that procedure ISNULLCONSTRAINT(X, Y) returns *true* only if there exists a valid null constraint $X \dashv Y$ between attributes X and Y in e' have already been stated in the proof for definite attributes. □

3.5.3.4 Intersection

Given an arbitrary expression tree e' of height $n > 0$ rooted with a binary intersection operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure INTERSECTION based on two input FD-graphs G_S and G_T for expressions e_S and e_T, one or both having a maximum height $n - 1$, correctly reflects the characteristics (attributes, equivalences, and dependencies) of the derived table Q resulting from the intersection of expressions e_S and e_T.

CLAIM 34 (ANALYSIS)
Procedure INTERSECTION terminates in time proportional to $O(\|V\|^2 + \|E\|^2)$.

PROOF. It is easy to see that the main loop in the INTERSECTION procedure (lines 301 through 313) completes in time proportional to $O(\|V_T\|)$ since it is over a (finite) set of white attributes in V^A that corresponds to the set of union-compatible real attributes in the result of e_T. This loop adds $O(\|V_T\|)$ edges to the combined FD-graph produced by the INTERSECTION procedure. Lemma 33 already showed that procedures SET DEFINITE and CONVERT DEPENDENCIES execute in time proportional to $O(\|V\| \times (\|V\| + \|E\|)$ and $O(\|E\|^2)$ respectively. Since $\|V\| \times \|E\|$ is $O(\|V\|^2 + \|E\|^2)$, we can simplify the analysis in a manner similar to that for the RESTRICTION procedure above. Hence we claim that procedure INTERSECTION executes in time proportional to $O(\|V\|^2 + \|E\|^2)$. □

LEMMA 21 (SCHEMA OF THE RESULT OF INTERSECTION)
Procedure INTERSECTION constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = e_S \cap_{All} e_T$ for arbitrary expressions e_S and e_T whose results are extended tables S and T respectively.

PROOF.

$\alpha(Q) = \alpha(S)$

Each white vertex v that represents a union-compatible attribute $\chi(v) \in \alpha(T)$ is coloured black (line 306) denoting its move from $\alpha(T)$ to $\rho(Q)$. The only remaining white vertices will be all the existing white attributes vertices in G_S.

$\iota(Q) = \iota(S)$

Obvious.

$\kappa(Q) = \kappa(S) \cup \kappa(T)$

Obvious.

$\rho(Q) = \rho(S) \cup \rho(T) \cup \iota(T)$

Obvious.

Definite attributes. By contradiction, assume that a vertex $v \in V^A$ is marked definite in G_Q but the attribute $\chi(v)$ is not guaranteed to be definite in $I(Q)$.

If v is marked definite in either G_S or G_T then by the induction hypothesis $\chi(v)$ was definite in $I(S)$ or $I(T)$, respectively. Since the semantics of intersection does not change any values in either input, $\chi(v)$ must be definite in the result; a contradiction.

Otherwise, v is either nullable or pseudo-definite. Without loss of generality, assume that $v \in V^A[G_T]$. The nullability of v is altered by procedure INTERSECTION in two possible ways:

1. If its corresponding union-compatible attribute vertex v_S is pseudo-definite in G_S, then v is marked pseudo-definite in G_Q (line 311). In this case $\chi(v)$ must be pseudo-definite, since the semantics of the intersection operator requires that the real attributes of a tuple $s_0[\alpha(S)]$ in S must match exactly (using the null equivalence operator $\overset{\omega}{=}$) with the corresponding real attributes of at least one tuple $t_0[\alpha(T)]$ in order to construct a corresponding tuple in the result. Hence the corresponding attributes must either match in value, or both be `Null`. Therefore if either input attribute is pseudo-definite, so must be the other in the result.

2. Otherwise, v is marked definite due to either (1) the existence of a strict equivalence edge between v and some other constant or attribute vertex $v_Q \in V^A[G_Q]$, or (2) the existence of a null constraint between v and another attribute vertex v_Q. The proof for the correct behaviour of the procedures SET DEFINITE and MARK DEFINITE were already presented in Lemma 19 above.

Hence we conclude that if v is marked definite in G_Q then $\chi(v)$ is guaranteed to be definite in the result. □

LEMMA 22 (CONSTRAINTS IN THE RESULT OF INTERSECTION)
Procedure INTERSECTION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = S \cap_{All} T$ for arbitrary expressions e_S and e_T whose results are the extended tables S and T respectively.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

Case (1). Suppose f held in Γ in G_S; by the induction hypothesis f' held in $I(S)$. Observe that INTERSECTION does not remove nor alter any existing strict dependency that stemmed from S; hence f remains in G_Q unchanged. By Claim 21 all strict dependencies in S now hold in Q; a contradiction.

Case (2). The identical situation exists if f held in Γ in G_T.

Case (3). Otherwise, f is produced by the INTERSECTION procedure. There are three possible cases where f may be derived:

1. If X and Y refer to the two tuple identifier vertices $\iota(S)$ and $\iota(T)$, then this dependency was produced on line 298. By Claim 21 this dependency holds in Q; contradiction.

2. If X and Y refer to two paired, union-compatible attributes (one from each input) then f must have been constructed by INTERSECTION on line 304. By the semantics of intersection, these two vertices represent attributes that must have identical values in the result Q; hence both $X \longrightarrow Y$ and $Y \longrightarrow X$ hold in Q, a contradiction.

3. Otherwise, f must have been a lax dependency in either G_S or G_T that has now been converted to a strict dependency through the modification of one or more vertices to reflect that, by the semantics of the intersection operator their values are definite

in the result. The proof of the operation of procedures SET DEFINITE, MARK DEFINITE, and CONVERT DEPENDENCIES has already been described in Lemmas 19 and 20 above.

Hence we conclude that if $f \in \Gamma$ in G_Q then the strict functional dependency $\chi(X) \longrightarrow \chi(Y)$ holds in Q. □

PROOF (LAX DEPENDENCIES AND CONSTRAINTS). Proofs of lax functional dependencies, equivalence constraints, and null constraints are similarly proved. □

3.5.3.5 Partition

Given an arbitrary expression tree e' of height $n > 0$ rooted with a unary partition operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure PARTITION based on the input FD-graph G_R for expression e of height $n-1$ correctly reflects the characteristics (attributes, equivalences, and dependencies) of the grouped extended table Q resulting from the partition of expression e by grouping columns A^G.

CLAIM 35 (ANALYSIS)
Procedure PARTITION executes in time proportional to $O(\|V\|^2)$.

PROOF. The proof of this claim is straightforward; from observation it is clear that the algorithm terminates. Copying the input FD-graph takes time proportional to $O(\|V\|+\|E\|)$. Moreover, other than the loop over each Group-by attribute (lines 332 through 338), which executes in time proportional to $O(\|V\|^2)$ due to the possible existence of scalar functions, the remaining loops in the procedure execute in time linear to the number of vertices in the graph. □

LEMMA 23 (SCHEMA OF THE RESULT OF PARTITION)
Procedure PARTITION constructs an FD-graph G_Q whose vertices correctly represent the schema of the grouped extended table Q that results from the algebraic expression $e' = \mathcal{G}[A^G, A^A](R)$ for an arbitrary expression e whose result is the extended table R.

PROOF. This proof is similar to the proof for distinct projection (Lemma 15). □

LEMMA 24 (CONSTRAINTS IN THE RESULT OF PARTITION)
Procedure PARTITION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the grouped extended table Q that results from the algebraic expression $e' = \mathcal{G}[A^G, A^A](R)$ for an arbitrary expressions e whose result is the extended table R.

PROOF. This proof is similar to that for distinct projection (Lemma 16). □

3.5.3.6 Grouped table projection

Given an arbitrary expression tree e' of height $n > 0$ rooted with a unary grouped table projection operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure GROUPED TABLE PROJECTION based on the input FD-graph G_R for expression e of height $n-1$, which by definition must be rooted with a unary partition operator, correctly reflects the characteristics (attributes, equivalences, and dependencies) of the extended table Q resulting from the grouped table projection of expression e.

CLAIM 36 (ANALYSIS)
Procedure GROUPED TABLE PROJECTION executes in time proportional to $O(\|V\|^2)$.

PROOF. Straightforward. The main loop that constructs vertices corresponding to aggregate functions (lines 391 through 401) executes in time proportional to $O(\|V\| \times \|F\|)$, since the EXTENSION procedure executes in $O(\|V\|)$ time. Since copying the input FD-graph takes $O(\|V\| + \|E\|)$ time, $\|F\| < \|V\|$, and $\|E\|$ is $O(\|V\|^2)$, procedure GROUPED TABLE PROJECTION executes in time proportional to $O(\|V\|^2)$. □

LEMMA 25 (SCHEMA OF THE RESULT OF PARTITION)
Procedure GROUPED TABLE PROJECTION constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = \mathcal{P}[A^G, F[A^A]](R)$ for an arbitrary expression e rooted with a partition operator whose result is the grouped extended table R.

PROOF. This proof is similar to that for projection (Lemma 13). □

LEMMA 26 (CONSTRAINTS IN THE RESULT OF GROUPED TABLE PROJECTION)
Procedure GROUPED TABLE PROJECTION constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = \mathcal{P}[A^G, F[A^A]](R)$ for an arbitrary expressions e rooted with a partition operator whose result is the grouped extended table R.

PROOF. This proof is similar to that for projection (Lemma 14). □

3.5.3.7 Left outer join

Given an arbitrary expression tree e' of height $n > 0$ rooted with a binary left outer join operator, we must show that $\tau(G_Q)$ of the FD-graph G_Q constructed by the procedure LEFT OUTER JOIN based on two input FD-graphs G_S and G_T for expressions e_S and

e_T, one or both having a maximum height $n-1$, correctly reflects the characteristics (attributes, equivalences, and dependencies) of the derived table Q resulting from the left outer join $e' = e_S \xrightarrow{p} e_T$ of expressions e_S and e_T with On condition p.

CLAIM 37 (ANALYSIS)
Procedure LEFT OUTER JOIN executes in time proportional to $O(\|V\|^2 + \|E\| \times \|P\| \times \|V\|)$.

PROOF. We proceed by code section, following our explanation of the algorithm in Section 3.4.8.1 beginning on page 137.

1. Graph merging and initialization (lines 410 to 430). As with Cartesian product, graph merging consists of creating a combined FD-graph of the two inputs in $O(\|V\| + \|E\|)$. The first loop (lines 423 through 427), which establishes an edge between each null-supplying attribute and the new outer join vertex as a prerequisite for the testing of null constraints, executes in time proportional to $O(\|V\|^2)$. The second loop (lines 428 through 430) links this new outer join vertex with unnested outer join vertices in G_T in $O(\|V\|)$.

2. Dependency and constraint analysis for the null-supplying table (lines 431 to 472). There are three loops in this section, over strict dependency edges, lax dependency edges, and lax equivalence edges respectively. It is immediate that their execution time is proportional to $O(\|V\| \times \|E\| \times \|P\|)$, $O(\|V\| \times \|E\| \times \|P\|)$, and $O(\|E\| \times \|P\|)$ respectively.

3. Generation of lax dependencies implied by the On condition (lines 473 to 516). This section analyzes the On condition predicate p several times, first to break up p into conjuncts, next to eliminate any conjunctive term containing disjuncts, and finally to infer additional dependencies and equivalences from each conjunctive atomic condition that remains. However, since we are assuming constant time updates of the FD-graph, this code section executes in time proportional to $O(\|P\|)$.

4. Construction of strict dependencies implied by the On condition (lines 517 to 533). It is immediate that in the worst case, this section of code executes in time proportional to $O(\|V\|)$.

5. Marking attributes nullable (lines 534 to 541). In this final section, each definite attribute from the null-supplying side is marked pseudo-definite. Since we assume that the nullability function η is $O(\|P\|)$, this section of pseudocode executes in $O(\|V\| \times \|P\|)$ time.

We therefore claim that procedure LEFT OUTER JOIN executes in time proportional to $O(\|V\|^2 + \|E\| \times \|P\| + \|V\|)$. □

LEMMA 27 (SCHEMA OF THE RESULT OF LEFT OUTER JOIN)
Procedure LEFT OUTER JOIN constructs an FD-graph G_Q whose vertices correctly represent the schema of the extended table Q that results from the algebraic expression $e' = e_S \xrightarrow{p} e_T$ for arbitrary expressions e_S and e_T whose results are extended tables S and T respectively.

PROOF. The proof of this Lemma is virtually identical to the corresponding proof for Cartesian product. □

LEMMA 28 (CONSTRAINTS IN THE RESULT OF LEFT OUTER JOIN)
Procedure LEFT OUTER JOIN constructs an FD-graph G_Q whose vertices and edges correctly represent the functional dependencies, equivalence constraints, and null constraints of the extended table Q that results from the algebraic expression $e' = e_S \xrightarrow{p} e_T$ for On condition p and arbitrary expressions e_S and e_T whose results are the extended tables S and T respectively.

PROOF (STRICT FUNCTIONAL DEPENDENCIES). By contradiction, assume that the strict functional dependency $f : X \longrightarrow Y \in \Gamma$ but the strict functional dependency $f' : \chi(X) \longrightarrow \chi(Y)$ does not hold in $I(Q)$.

Case (1). Suppose f held in Γ in G_S (the FD-graph for the preserved table S). If so, then by the induction hypothesis f' held in S. By Theorem 3 f' holds in $I(Q)$; a contradiction.

Case (2). Suppose f held in Γ in G_T (the FD-graph for the null-supplying table T). If so, then by the induction hypothesis f' held in T. There are two situations in which f could be altered by LEFT OUTER JOIN:

1. f denotes a dependency with a singleton attribute vertex X as its determinant, and $\chi(X)$ cannot be guaranteed to be definite in the result except for the all-`Null` row (line 436); or

2. f denotes a dependency with a compound vertex X as its determinant, and no single attribute $\chi(x) \in \chi(X)$ can be guaranteed to be definite in the result but for the all-`Null` row (line 442).

Thus, if f is unaltered, either X represents a tuple identifier vertex in the set V^R, $XY \subseteq V^A$ and the strict equivalence constraint $e : X \overset{\omega}{=} Y$ held in Ξ in G_T, or $\eta(p, \chi(X))$ is true. Under these conditions, by Theorem 3 f' must hold in $I(Q)$, a contradiction.

Case (3). Otherwise, f must be developed through the analysis of the `On` condition p. We consider each possible modification to the edges in $E^F \cup E^R \cup E^C$ in G_Q that could result in the new dependency f:

1. If $f \in \Gamma$ is represented by an edge $(X, Y) \in E^R$ then f corresponds to one of the strict functional dependencies between the tuple identifier of the result Q and the tuple identifiers of both inputs, or vice-versa (lines 416 and 419) which clearly hold in the result Q.

2. If $f \in \Gamma$ is represented by an edge $(X, Y) \in E^C$ then f must represent the strict reflexive dependency formed by the construction of the composite determinant W, all of whose attributes are from the preserved table S; clearly this dependency also holds in Q.

3. Otherwise, f must stem from an edge $(X, Y) \in E^F$. There are four ways that LEFT OUTER JOIN constructs such an edge:

   (a) Line 452: $X \longmapsto Y$ was a lax dependency that held in T, $\eta(p, \chi(X))$ is *true*, meaning that the singleton attribute X is either definite in T or p is such that it cannot evaluate to *true* if $\chi(X)$ is `Null` (line 436), and either Y denotes a tuple identifier in G_T or $\eta(p, \chi(Y))$ is *true*. These conditions match the corresponding case in Theorem 3, and hence f' must hold in $I(Q)$, a contradiction.

   (b) Line 461: similarly, if $X \longmapsto Y$ is a lax dependency that held in T and X is a compound determinant such that $\chi(X) \subseteq sch(T)$ and $\eta(p, \chi(X))$ is *true* (line 442) then f' must hold in $I(Q)$.

   (c) Line 502: f stems from an Type 2 equality condition between null-supplying attributes $\chi(X)$ and $\chi(Y)$. Since we falsely-interpret Type 1 and Type 2 conditions in p, the nullability function η will evaluate to *true* for both $\eta(p, \chi(X))$ and $\eta(p, \chi(Y))$. This case is also explicit in Theorem 3 and hence it must follow that f' holds in $I(Q)$, a contradiction.

   (d) Line 531: in this case we add a set of strict dependency edges $\alpha_S(p) \longrightarrow z$ for all $z \in Z$ between all of the preserved attributes referenced in the `On` condition p and each null-supplying vertex z referenced in each false-interpreted Type 1 or Type 2 condition in p (lines 496, 504, 509, and 511). By their inclusion in such a condition $\eta(p, \chi(z))$ for each $\chi(z) \in \chi(Z)$ is automatically true. The tests on line 518 verifies that $\alpha_S(p)$ is not empty. This combined set of conditions mirrors those conditions specified in Theorem 3 (Case 4), and therefore f' must hold in Q, a contradiction.

As we have shown that in each case an edge in Γ correctly represents a strict functional dependency in $\mathcal{F}$, we conclude that procedure LEFT OUTER JOIN correctly constructs an FD-graph representing strict dependencies that hold in Q. □

PROOF (LAX FUNCTIONAL DEPENDENCIES). By contradiction, assume that the lax functional dependency $f : X \longmapsto Y \in \gamma$ but the lax functional dependency $f' : \chi(X) \longmapsto \chi(Y)$ does not hold in $I(Q)$.

Case (1). Suppose f held in Γ in G_S (the FD-graph for the preserved table S). If so, then by the induction hypothesis f' held in S. By Theorem 3 f' holds in $I(Q)$; a contradiction.

Case (2). Suppose f held in γ in G_T (the FD-graph for the null-supplying table T). If so, then by the induction hypothesis f' held in T. There are four situations in which f could be altered or removed by LEFT OUTER JOIN:

1. Line 438: $X \longrightarrow Y$ denotes a strict dependency in E^F with a singleton attribute vertex X as its determinant, and X cannot be guaranteed to be definite in the result except for the all-`Null` row (line 436). By Theorem 3 this dependency laxly holds in the result since the generation of an all-`Null` row may produce a strict dependency violation.

2. Line 444: similarly, $X \longrightarrow Y$ denotes a strict dependency with a compound determinant vertex X, and no single attribute $\chi(x) \in \chi(X)$ can be guaranteed to be definite but for an all-`Null` row in the result (line 442).

3. Line 453: f represents the lax dependency f' that held in T, but both its determinant and dependent attributes cannot be `Null` except for the all-`Null` row. In this case, as argued above for strict dependencies in Q, the dependency can be made strict. However, by inference axiom FD5 (weakening) f' still laxly holds in $I(Q)$, as per Theorem 3.

4. Line 462: similarly, f denotes a lax dependency that held in T, where X is a compound determinant and at least one of the attributes $\chi(x) \in \chi(X)$ is guaranteed definite in Q but for the all-`Null` row. Once again, by Theorem 3 f can be made strict, hence f' still laxly holds in Q.

Hence, if none of the conditions in the cases above are met, f is retained unaltered in G_Q. By Theorem 3, any lax dependency that held in $I(T)$ must hold in $I(Q)$; a contradiction.

Case (3). Otherwise, f must be a new dependency produced via the analysis of the `On` condition p. f represents a lax dependency formed by an equality condition between an attribute from $\alpha(T)$ and either a constant (line 494) or an attribute from $\alpha(S)$ (line 506).

This situation is also explicitly mentioned in Theorem 3; hence f' holds in Q, again a contradiction. □

PROOF (STRICT EQUIVALENCE CONSTRAINTS). By contradiction, assume that the strict equivalence constraint $e : X \stackrel{\omega}{=} Y \in \Xi$ but the corresponding strict equivalence constraint $e' : \chi(X) \stackrel{\omega}{=} \chi(Y)$ does not hold in $I(Q)$.

- *Case (1).* Suppose e held in Ξ in G_S; by the induction hypothesis, $e' \in \mathcal{E}$ in S. By Theorem 3, therefore, e' must hold in Q; a contradiction.
- *Case (2).* Suppose e held in Ξ in G_T; by the induction hypothesis, $e' \in \mathcal{E}$ in T. By Theorem 3, therefore, e' must hold in Q; a contradiction.
- *Case (3).* Suppose $e : X \simeq Y$ held as a lax equivalence constraint in ξ in G_T; by the induction hypothesis, $\chi(X) \simeq \chi(Y)$ in T. There is only one circumstance where e is made into a strict equivalence constraint (line 469), and that is when both $\eta(p, \chi(X))$ and $\eta(p, \chi(Y))$ evaluate to *true*. By Corollary 2, under these conditions e' holds as a strict equivalence constraint in Q; a contradiction.
- *Case (4).* Otherwise, the only remaining possibility is that e was generated through the analysis of a Type 2 equality condition in p (line 503). In this case $\chi(XY) \subseteq sch(T)$, and by Theorem 3 e' must hold in Q; a contradiction.

As we have shown that e holds in each case, we have proved that if G_Q contains a strict equivalence constraint $e : X \stackrel{\omega}{=} Y$ then $\chi(X) \stackrel{\omega}{=} \chi(Y)$ holds in $\mathcal{E}$. □

PROOF (LAX EQUIVALENCE AND NULL CONSTRAINTS). The proof for lax equivalence constraints is similar to the proof for strict equivalence constraints above; the proof for null constraints is straightforward, by construction. □

By similarly showing that the procedures for union, difference, and full outer join correctly modify an FD-graph such that the dependencies and constraints modelled by the graph are guaranteed to hold in its result, we will have proven the theorem. Moreover, we have also shown that the complete algorithm executes in time polynomial in the size of its inputs. Q.E.D.

3.6 Closure

The mapping function χ, as defined in Definition 46, straightforwardly converts edges in an FD-graph G into strict or lax functional dependencies and equivalence constraints that are guaranteed to hold in the result of the algebraic expression e modelled by G. By the

soundness of the inference axioms for strict and lax dependencies (Theorem 1) and strict and lax equivalence constraints (Theorem 2), any dependency or constraint in the closure of these dependencies and constraints must hold in $I(e)$ as well.

One method to compute the closure of the strict functional dependencies modelled in G would be to:

1. use the mapping function χ to create the set of strict functional dependencies Γ;
2. develop the closure of the these dependencies in the standard manner, that is to apply the inference rules augmentation, union, and strict transitivity defined in Lemma 1 to the set of dependencies in Γ; and, if desired,
3. eliminate from the closure any dependency whose determinant or dependent contained an attribute in $\rho(e)$, retaining only those dependencies that involve real attributes, constants, and the tuple identifier of the result of e.

Instead, we shall use the data structures comprising the FD-graph G to compute the closure of Γ directly. In this section, we present two algorithms, DEPENDENCY-CLOSURE and EQUIVALENCE-CLOSURE, that computes the closures of Γ, γ, Ξ, and ξ. Observe that the closures of these sets of dependencies and constraints correspond to the definitions of FD-paths and equivalence-paths described earlier (Definitions 41 through 44):

DEFINITION 47 (STRICT DEPENDENCY CLOSURE)
The *strict dependency closure* of a set of vertices $X \subseteq V$ with respect to the strict FD-paths in an FD-graph G, denoted X_Γ^+, is defined as follows:

$$X_\Gamma^+ = X \cup V_\kappa^A \cup \{Y\} \tag{3.7}$$

such that for each vertex $y \in Y$ the strict FD-path $\langle X \cup V_\kappa^A, y\rangle$ exists in G.

DEFINITION 48 (LAX DEPENDENCY CLOSURE)
The *lax dependency closure* of a set of vertices $X \subseteq V$ with respect to the lax FD-paths in an FD-graph G, denoted X_γ^+, is defined as follows:

$$X_\gamma^+ = X \cup V_\kappa^A \cup \{Y\} \tag{3.8}$$

such that for each vertex $y \in Y$ the lax FD-path $\langle X \cup V_\kappa^A, y\rangle$ exists in G.

DEFINITION 49 (STRICT EQUIVALENCE CLOSURE)
The *strict equivalence closure* of a single vertex $x \in V^A$ with respect to the strict equivalence-paths in an FD-graph G, denoted x_Ξ^+, is defined as follows:

$$x_\Xi^+ = x \cup \{Y\} \tag{3.9}$$

such that for each vertex $y \in Y \subset V^A$ the strict equivalence-path $\langle x, y \rangle$ exists in G.

DEFINITION 50 (LAX EQUIVALENCE CLOSURE)
The *lax equivalence closure* of a single vertex $x \in V^A$ with respect to the lax equivalence-paths in an FD-graph G, denoted x_ξ^+, is defined as follows:

$$x_\xi^+ = x \cup \{Y\} \tag{3.10}$$

such that for each vertex $y \in Y \subset V^A$ the lax equivalence-path $\langle x, y \rangle$ exists in G.

3.6.1 Chase procedure for strict and lax dependencies

Procedure DEPENDENCY-CLOSURE, which implements a chase procedure for dependencies represented in an FD-graph, is a modified version of Ausiello, D'Atri, and Saccà's algorithm NODE-CLOSURE. DEPENDENCY-CLOSURE computes the set of all strict or lax FD-paths in G with head $\chi(X) \cup V_\kappa^A$ using the temporary set variables S^+ and S. As each FD-path in G represents a functional dependency for an expression e, the former represents that portion of the closure X_Γ^+ or X_γ^+ derived from functional dependencies, and the latter represents vertices on FD-paths, used to determine transitive dependencies. Lines 603 through 607 add constants to S so that in turn all attributes functionally determined by constants are added to S^+. In addition, our version requires logic to ensure that only attribute vertices coloured white, and tuple identifier vertices coloured gray, appear in the closure S^+. This eliminates from S^+ any vertex representing an attribute in $\rho(e)$, and at the same time enables a calling procedure to easily determine if any FD-path rooted with $\chi(X) \cup V_\kappa^A$ represents a key dependency. Note, however, that attributes of any colour can be placed in S since transitive dependencies through projected-out attributes in $\rho(e)$ continue to hold. Lines 643 through 676 compute the closure of any lax dependencies, enforcing the condition that a lax FD-path is transitive only over definite attributes. DEPENDENCY-CLOSURE also contains an invariant: a compound determinant is never added to the set S until each of its component vertices have been added to S and considered for inclusion in S^+.

588 **Procedure:** DEPENDENCY-CLOSURE
589 **Purpose:** *Determine the closure of* $\{X\}$ *with respect to* Γ or γ.
590 **Inputs:** FD-graph G, attributes $X_1, X_2, \cdots, X_n$, closure type
591 **Output:** the set of vertices representing the closure of X, denoted S^+.
592 **begin**
593 $S^+ \leftarrow S \leftarrow \emptyset$;
594 *-- Create a temporary structure 'Visited' for vertices in G.*
595 **for each** $v_j \in \{V^C \cup V^R \cup V^A\}$ **do**

```
596     Visited[v_j] ← false
597     od ;
598   --Initialize S and S+ with vertices representing those attributes in X.
599   for each X_i ∈ {X} do
600     S ← S ∪ χ(X_i)
601     od ;
602   --Add to S any constant values in G.
603   for each v_i ∈ V^A do
604     if Colour[v_i] is gray then
605       S ← S ∪ v_i
606       fi
607     od ;
608   --Construct the closure X_Γ+.
609   while S ≠ ∅ do
610     select v_i from S;
611     S ← S − v_i;
612     Visited[v_i] ← true;
613     if v_i ∈ V^A then
614       --v_i is a simple node; determine if a compound node including v_i
615       --is now transitively dependent on S.
616       for each v_j ∈ V^C | (v_j, v_i) ∈ E^C do
617         if Visited[v_j] is false and ∀ v_k | (v_j, v_k) ∈ E^C: Visited[v_k] is true then
618           S ← S ∪ v_j;
619           fi
620         od ;
621       if Colour[v_i] is white then
622         S+ ← S+ ∪ v_i
623         fi
624     else if v_i ∈ V^R then
625       --v_i is a tuple identifier; determine if we have found a key.
626       if Colour[v_i] is Gray then
627         S+ ← S+ ∪ v_i
628         fi ;
629       --Determine if a compound tuple identifier including v_i
630       --is now transitively dependent on S.
631       for each v_j ∈ V^R | (v_j, v_i) ∈ E^R do
632         if Visited[v_j] is false and ∀ v_k | (v_j, v_k) ∈ E^R: Visited[v_k] is true then
633           S ← S ∪ v_j;
634           fi
635         od
636       fi ;
637     for each v_k | (v_i, v_k) ∈ E^F do
```

```
638         if Visited[v_k] is false then
639           S ← S ∪ v_k;
640           fi
641         od
642       --Include lax dependencies if desired by the calling procedure.
643       if closure type = 'γ' then
644         if v_i ∈ V^A then
645           if Nullability[v_i] is Definite then
646             for each v_k | (v_i, v_k) ∈ E^f do
647               if v_k ∈ V^A then
648                 if Colour[v_k] is white then
649                   S^+ ← S^+ ∪ v_k;
650                   fi;
651                 if Nullability[v_k] is Definite and Visited[v_k] is false then
652                   S ← S ∪ v_k
653                   fi
654               else if v_k ∈ V^R and Visited[v_k] is false then
655                 S ← S ∪ v_k
656                 fi
657               od
658             fi
659         else if v_i ∈ V^C then
660           if ∄ v_j ∈ V^A | (v_i, v_j) ∈ E^C and Nullability[v_j] is not Definite then
661             for each v_k | (v_i, v_k) ∈ E^f do
662               if v_k ∈ V^A then
663                 if Colour[v_k] is white then
664                   S^+ ← S^+ ∪ v_k;
665                   fi;
666                 if Nullability[v_k] is Definite and Visited[v_k] is false then
667                   S ← S ∪ v_k
668                   fi
669               else if v_k ∈ V^R and Visited[v_k] is false then
670                 S ← S ∪ v_k
671                 fi
672               od
673             fi
674           fi
675         fi
676       fi
677     od;
678   return S^+
679   end
```

Observe that if S is implemented as a queue then DEPENDENCY-CLOSURE corresponds to a breadth-first traversal of G. We could, if desired, optimize the algorithm to simply return all white vertices in G once a gray tuple identifier vertex $v_K \in V^R$ has been found.

LEMMA 29 (ANALYSIS)
Given as input an arbitrary set of valid attributes X and an FD-graph $G < V, E >$, procedure DEPENDENCY-CLOSURE executes in time proportional to $O(\|V\|^2)$.

PROOF. We can make the following straightforward observations:

1. Clearly the initialization loops execute in $O(\|V\|)$ time since they are over finite sets (lines 593 through 607).

2. Consider the main closure loop beginning on line 609. The loop terminates when S is empty; the size of S can never exceed $\|V\|$ since no vertex is visited more than once. After initialization, when S contains the vertices of $\chi(X)$, there are only the following ways in which a vertex may be added to S:

   (a) a compound node may be added to S once all of its components have been visited, executing in time proportional to $O(\|V\|^2)$ (line 618);

   (b) a node $v_i \in V^R$ may be added to S upon discovery of all of its component tuple identifiers that together form v_i, again in time proportional to $O(\|V\|^2)$ (line 633);

   (c) a node in V^A or V^R may be added to S upon discovery of a strict edge in E^F, taking time $O(\|E\|)$ (line 639);

   (d) a node in V^A may be added to S upon discovery of a lax edge in E^f (lines 652 and 667, executing in time proportional to $O(\|E\|)$ and $O(\|V\|^2)$ respectively); or

   (e) a node in V^R may be added to S upon discovery of a lax edge in E^f (lines 655 and 670, executing in time proportional to $O(\|E\|)$ and $O(\|V\|^2)$ respectively).

   In no case can the node be added to S if already visited (line 612); hence even if cycles exist in E^F or E^f each vertex in $V[G]$ will be considered at most once. Moreover, it is impossible for the algorithm to traverse any single edge in E more than once. Hence, we claim that the main loop beginning on line 609 executes in time proportional to $O(\|V\|^2 + \|E\|)$.

Since G contains a finite number of vertices and edges, and $\|E\|$ is $O(\|V\|^2)$, we conclude that DEPENDENCY-CLOSURE must terminate, and in the worst case executes in time proportional to $O(\|V\|^2)$. □

LEMMA 30 (STRICT CLOSURE)
Given inputs of an arbitrary set of valid attributes X, an FD-graph G representing the dependencies in e, and closure type of Γ, procedure DEPENDENCY-CLOSURE returns a set containing a vertex $\chi(Y)$ if and only if $Y \in \iota(e) \cup \alpha(e)$ and $\chi(Y) \in X_\Gamma^+$.

PROOF (SUFFICIENCY). For strict closures there are only two ways in which a vertex v_i can be added to S^+. The first, on line 622, requires that $v_i \in V^A$ was previously part of the set S. If $\chi(Y) \in (\chi(X) \cup V_\kappa^A)$ then S^+ will automatically contain $\chi(Y)$, since lines 600 and 605 will add $\chi(Y)$ to S, and if $\chi(Y)$ is a white attribute vertex then it will be added to S^+ on line 622. The second, on line 627, adds the vertex v_i to S^+ if $v_i \in V^R$ is coloured gray, indicating that $\chi(v_i) \equiv \iota(e)$.

Otherwise, since the elements of S correspond to vertices on strict FD-paths, it is clear that the traversal of each strict FD-path component will result in a vertex added to S to represent a vertex on that FD-path, and either (1) a white vertex added to S^+ if it appears on the FD-path, or (2) a gray tuple-identifier vertex added to S^+ if it appears on the FD-path. Hence we claim that if the strict FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y) \rangle$ exists in G then $\chi(Y)$ will be returned in the result of DEPENDENCY-CLOSURE. Therefore, the result of DEPENDENCY-CLOSURE will contain $\chi(Y)$ if $Y \in \iota(e) \cup \alpha(e)$ and $\chi(Y) \in X_\Gamma^+$. □

PROOF (NECESSITY). To prove necessity, by contradiction assume that DEPENDENCY-CLOSURE returns a set which contains a vertex $\chi(Y)$, but either $\chi(Y)$ is not a white vertex in the set V^A and $\chi(Y)$ is not a gray tuple identifier vertex in V^R, or G does not contain the strict FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y) \rangle$. Then at some point during the traversal of G DEPENDENCY-CLOSURE must add $\chi(Y)$ to S^+. There are only two possible ways this may occur: at line 622 or line 627. $\chi(Y)$ must also be either a white vertex in V^A, due to the test on line 621, or a gray vertex in V^R, due to the tests on lines 624 and 626. Furthermore, $\chi(Y)$ was previously an element of the set S. For $\chi(Y)$ to exist in S one of the following must have occurred:

1. $\chi(Y) \in \chi(X)$, added to S during initialization at line 600, contradicting our initial assumption.

2. $\chi(Y)$ is a constant and is coloured gray (line 604), and is added to S on line 605. In this case the trivial FD-path $\langle V_\kappa^A, \chi(Y) \rangle$ exists in G, again contradicting our initial assumption.

3. In our last case we carry on the proof by induction on the number of strict edges traversed in G. If $\chi(Y) \subseteq S^+$ then there must exist a directed edge with target $\chi(Y)$. $\chi(Y)$ can be added to S only at line 639, as a result of an edge in E^F, or at line 633,

as a result of a set of edges in E^R. These are the sole remaining possibilities since we are not considering lax dependency edges at this point (lines 643 through 676).

Basis. The base case of the induction is that there exists a directed edge $(v_j, \chi(Y)) \in E^F \mid v_j \in (\chi(X) \cup V_\kappa^A)$, which represents the strict FD-path $\langle v_j, \chi(Y)\rangle$; hence the strict FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y)\rangle$ is also in G.

Induction. Otherwise, in our FD-graph implementation there are four possible sources of an edge with target $\chi(Y)$:

- *Case (1).* The source vertex v_i is a tuple identifier vertex $v_i \in V^R$. If so, then v_i was also an element of S.
- *Case (2).* $\chi(Y)$ is a tuple identifier vertex in the set V^R with edges in E^R to each of its component tuple identifiers, all of which must already be in S.
- *Case (3).* The source is a compound vertex $v_i \in V^C$. If so, then v_i must also have been added to S, and in addition all of its component vertices must have already been visited during the traversal of G due to the test on line 617.
- *Case (4).* The source vertex is an ordinary vertex $v_i \in V^A$.

In each case, the vertex v_i was added to S only through the direct or indirect traversal of strict edges in G, indicating the existence of a direct or transitive strict FD-path from $\chi(X) \cup V_\kappa^A$ to v_i. Since there exists a strict FD-path $\langle v_i, \chi(Y)\rangle$ in G, we then have a combined FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y)\rangle$, a contradiction.

Hence we conclude that DEPENDENCY-CLOSURE will return $\chi(Y)$ as part of the strict closure of an attribute set X only if $Y \in \iota(e) \cup \alpha(e)$ and $\chi(Y) \in X_\Gamma^+$. □

LEMMA 31 (LAX CLOSURE)
Given inputs of an arbitrary set of valid attributes X, an FD-graph G, and closure type of γ, procedure DEPENDENCY-CLOSURE returns a set containing an element $\chi(Y)$ if and only if $Y \in \iota(e) \cup \alpha(e)$ and $\chi(Y) \in X_\gamma^+$.

PROOF (SUFFICIENCY). Observe that if $Y \subseteq X$ then S^+ will automatically contain Y, since this situation corresponds to a strict FD-path from $\chi(X)$ to $\chi(Y)$. Otherwise, for lax closures, the only way in which a vertex v_i can be added to S^+ is on lines 622, 627, 649, or 664, each of which requires that v_i is a target of some strict or lax edge in G. If v_i represents a definite attribute or a tuple identifier then v_i will also be added to S, corresponding to the definition of a lax FD-path. Hence it is clear that every lax FD-path traversed by DEPENDENCY-CLOSURE will result in (1) that path's target vertex added to S^+ if representing a real attribute in $\alpha(e)$ or a tuple identifier in $\iota(e)$ and (2) added to

the set S if a definite attribute or tuple identifier for use as a determinant. Therefore we claim that if there exists a lax FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y) \rangle$ and $Y \in \iota(e) \cup \alpha(e)$ then $\chi(Y)$ will be returned in the result of DEPENDENCY-CLOSURE. □

PROOF (NECESSITY). Clearly $X_\Gamma^+ \subseteq X_\gamma^+$ since the strict closure of X is computed in both cases. Following an approach similar to that in Lemma 30, assume that DEPENDENCY-CLOSURE returned $\chi(Y)$ in the result but either $Y \notin \iota(e) \cup \alpha(e)$ or $\chi(Y) \notin X_\gamma^+$. We must have $Y \not\subseteq (X_\Gamma^+ \cup V_\kappa^A)$ since we have already shown in Lemma 30 that DEPENDENCY-CLOSURE correctly computes the strict closure of X_Γ^+. Therefore $\chi(Y)$ must have been added to S^+ only due to the existence of:

- *Case (1).* a strict dependency edge in E^F whose target is $\chi(Y)$ and whose source is already in S (line 622), or
- *Case (2).* a set of edges in E^R whose source is a gray vertex in V^R representing $\iota(e)$ and each of the targets of such edges are tuple identifier vertices in V^R that are already in S, or
- *Case (3).* a lax dependency edge in E^f whose source is a simple vertex in S and whose target is $\chi(Y)$ (line 649), or
- *Case (4).* a lax dependency edge in E^f whose source is a compound vertex in S and whose target is $\chi(Y)$ (line 664).

Cases (1) and (2) were proven correct in Lemma 30; we now consider cases (3) and (4). In both cases the addition of a vertex to S^+ is valid since they both represent instances of a valid lax FD-path. We now argue inductively that the existence of the source vertex $v_i \in S$ is correct. If $v_i \in S$ then it must have been added to S either:

- during initialization, implying that either $v_i \in \chi(X)$ or v_i is a constant (basis);
- $v_i \in V^C$ (line 618) and each component of v_i has been transitively inferred by the traversal of other edges in G, and furthermore each component of v_i is definite (line 660);
- $v_j \in V^R$ (line 633), v_j is a compound tuple identifier vertex, and v_j has been transitively inferred by its component tuple identifier vertices that are already in S;
- $v_i \in V^A \cup V^R$ (line 639), v_i has been transitively inferred by the traversal of a strict dependency edge in G, and either v_i represents a definite attribute (lines 645 or 651) or $v_i \in V^R$;

- $v_i \in V^A$ (line 652), v_i has been laxly inferred by a definite simple vertex (line 645), and v_i itself represents a definite attribute (line 651);
- $v_i \in V^R$ (line 655), transitively inferred by a definite simple vertex in V^A (line 645);
- $v_i \in V^A$ (line 667), transitively inferred by a definite compound vertex in V^C (line 660), and v_i itself represents a definite vertex (line 666);
- $v_i \in V^R$ (line 670), transitively inferred by a definite compound vertex in V^C (line 660).

We have shown that the only way in which a vertex $\chi(Y)$ can be added to the result contained in S^+ is either for $Y \subseteq X$ or for $\chi(Y)$ to be directly or transitively connected to one or more vertices in $\chi(X)$ through the existence of a lax FD-path. Hence we conclude that G must contain a lax FD-path $\langle \chi(X) \cup V_\kappa^A, \chi(Y) \rangle$. □

THEOREM 9 (DEPENDENCY CLOSURE)
Procedure DEPENDENCY-CLOSURE is correct.

PROOF. Follows from Lemmas 29, 30, and 31. □

3.6.2 Chase procedure for strict and lax equivalence constraints

For a given attribute X as input, the procedure EQUIVALENCE-CLOSURE given below computes that attribute's equivalence class for real attributes in X_Ξ^+ or X_ξ^+. Included in the closure are gray vertices representing constants; therefore if a vertex $v \in V_\kappa^A$ then the calling procedure can conclude that X is strictly or laxly equivalent to a constant. Note that the complete set of strictly equivalent attributes are also returned for a lax equivalence closure. In a similar fashion to DEPENDENCY-CLOSURE, the input parameter 'closure type' is either 'Ξ' or 'ξ' to represent strict and lax equivalence closures respectively. The basic algorithm is a straightforward implementation of determining the connected components of an undirected graph; handling black or gray vertices and computing the set of laxly connected components are the two specializations of the basic algorithm.

680 **Procedure:** EQUIVALENCE-CLOSURE
681 **Purpose:** *Determine the equivalence class of an attribute X in an* FD*-graph G.*
682 **Inputs:** FD-graph G, attribute X, closure type
683 **Output:** the set $S^+ = \chi(X) \cup \{y_i\}$ for each path $\langle X, y_i \rangle \mid \chi(y_i) \in \alpha(e) \cup \kappa(e)$.
684 **begin**
685 $S^+ \leftarrow \emptyset$;
686 *- - Establish a 'visited' indicator for each vertex in V^A.*

```
687  for each v_i ∈ V^A do
688    Visited[v_i] ← false
689    od ;
690  S ← X;
691  --Construct the equivalence class of the attributes in S.
692  while S ≠ ∅ do
693    select v_i from S;
694    S ← S − v_i;
695    Visited[v_i] ← true;
696    if Colour[v_i] is white or gray then
697      S^+ ← S^+ ∪ v_i
698      fi ;
699    --Add to the closure those attributes transitively equivalent to v_i.
700    for each v_k | (v_i, v_k) ∈ E^E do
701      if Visited[v_k] is false then
702        S ← S ∪ v_k;
703        fi
704      od
705    --Include lax equivalence constraints if desired.
706    if closure type = 'ξ' then
707      for each v_k | (v_i, v_k) ∈ E^e do
708        if Visited[v_k] is false then
709          if Nullability[v_i] is Definite or ∃ v_j ∈ V^A | (v_k, v_j) ∈ E^E then
710            S ← S ∪ v_k;
711            fi
712          fi
713        od
714      fi
715    od ;
716  return S^+
717  end
```

LEMMA 32 (ANALYSIS)
Given as input an arbitrary attribute X and an FD-graph G, procedure EQUIVALENCE-CLOSURE executes in time proportional to $O(\|V\| + \|E\|)$.

PROOF. We proceed with our analysis of procedure EQUIVALENCE-CLOSURE by making the following observations:

1. Clearly the initialization loop executes in time $O(\|V\|)$ since it is over a finite set (lines 685 through 689).

2. Consider the main closure loop beginning on line 692. The loop terminates when S is empty; again, $\|S\|$ can never exceed $\|V\|$. After initialization, when S contains the vertex $\chi(X)$ (line 690), there are only the following ways in which a vertex may be added to S:

   (a) a node in V^A may be added to S upon discovery of a strict edge in E^E (line 702); or,

   (b) a node in V^A may be added to S upon discovery of a lax edge in E^e (line 710).

   In neither case can the node be added to S if already visited; hence even if cycles exist in E^E or E^e each vertex in $V[G]$ will be considered at most once. Moreover, no edge in $E^F \cup E^f$ will be considered more than once, hence bounding the overall execution time to $O(\|V\| + \|E\|)$.

Since G contains a finite number of vertices and edges, we conclude that EQUIVALENCE-CLOSURE must terminate, and executes in time proportional to $O(\|V\| + \|E\|)$. □

We now show that for strict equivalence closures (that is, closures over edges in E^E) EQUIVALENCE-CLOSURE returns a set containing Y if and only if $Y \in X_{\Xi}^{+}$.

LEMMA 33 (STRICT CLOSURE)
Given inputs of an arbitrary attribute X, an FD-graph G representing the constraints that hold in the algebraic expression e, and closure type of Γ, procedure EQUIVALENCE-CLOSURE returns a set containing an element $\chi(Y)$ if and only if $Y \in \alpha(e) \cup \kappa(e)$ and $\chi(Y) \in X_{\Xi}^{+}$.

PROOF (SUFFICIENCY). For strict equivalence closures, the only way in which a vertex v_i can be added to S^+ is on line 697, which requires that v_i was previously part of the set S. Since the elements of S correspond to target attributes of strict equivalence-paths, it is clear that every strict equivalence edge traversed by EQUIVALENCE-CLOSURE will

result in (1) that path's target attribute vertex added to S^+ if either representing a real attribute or a constant, and (2) it will be added to S to represent the head of another equivalence-path. Hence we claim that if the strict equivalence-path $\langle \chi(X), \chi(Y) \rangle$ exists in G and $Y \in \alpha(e) \cup \kappa(e)$ then $\chi(Y)$ will be returned in the result of EQUIVALENCE-CLOSURE. □

PROOF (NECESSITY). To prove necessity, by contradiction assume that EQUIVALENCE-CLOSURE returns the equivalence class of $\chi(X)$ which contains $\chi(Y)$, but $\chi(Y) \neq \chi(X)$ and either $Y \notin \alpha(e) \cup \kappa(e)$ or the strict equivalence-path $\langle \chi(X), \chi(Y) \rangle$ is not in G. Then at some point during the traversal of G EQUIVALENCE-CLOSURE must add $\chi(Y)$ to S^+ at line 697. $\chi(Y)$ must also be either representing a real attribute in $\alpha(e)$ or a constant in $\kappa(e)$, due to the test on line 696. Furthermore, $\chi(Y)$ was previously an element of the set S. For $\chi(Y)$ to exist in S one of the following must have occurred:

1. Y is the input parameter X, so $\chi(Y)$ is added to S during initialization (line 690), which contradicts our initial assumption that X and Y are not in the same equivalence class.

2. Otherwise, if X and Y are different attributes then we prove the remainder of the cases by induction on the number of strict equivalence edges traversed in G. If $\chi(Y) \subseteq S^+$ then there must exist a strict undirected edge with target $\chi(Y)$, since $\chi(Y)$ can be added to S only at line 702; this is the sole remaining possibility since we are not considering lax equivalence edges at this point (lines 707 through 713). Therefore $\chi(Y) \in S$ only as the result of the existence of a strict equivalence edge in E^E with $\chi(Y)$ as the target vertex.

   Basis. The base case of the induction is that there exists an edge $(\chi(X), \chi(Y)) \in E^E$ which represents the (direct) equivalence-path $\langle \chi(X), \chi(Y) \rangle$, contradicting our initial assumption.

   Induction. Otherwise, in our FD-graph implementation there is only one other possible source of a strict undirected edge with target $\chi(Y)$, and that is another single vertex $v_i \in V^A$. The vertex v_i was added to S only through the direct or indirect traversal of strict equivalence edges in G, indicating the existence of a transitive strict equivalence-path from $\chi(X)$ to v_i. Such an equivalence-path, however, implies that there exists the strict equivalence path $\langle \chi(X), \chi(Y) \rangle$ in G, again contradicting our initial assumption.

Hence we conclude that EQUIVALENCE-CLOSURE will return $\chi(Y)$ as part of the strict closure of X only if $Y \in \alpha(e) \cup \kappa(e)$ and $\chi(Y) \in X^+_{\equiv}$. □

LEMMA 34 (LAX CLOSURE)
Given inputs of an arbitrary attribute X, an FD-graph G representing the constraints that hold in the algebraic expression e, and closure type of ξ, procedure EQUIVALENCE-CLOSURE returns a set containing an element $\chi(Y)$ if and only if $Y \in \alpha(e) \cup \kappa(e)$ and $\chi(Y) \in X_\xi^+$.

PROOF (SUFFICIENCY). As was the case for strict equivalence closures, for lax equivalence closures the only way in which a vertex v_i can be added to S^+ is on line 697, which requires that v_i was previously part of the set S. The elements of S correspond to either target attributes of strict equivalence-paths or definite target attributes of lax equivalence-paths. By observation, it is clear that every lax equivalence-path traversed by EQUIVALENCE-CLOSURE will result in (1) that path's target attribute added to S^+ if either representing a real attribute or a constant, and (2) it will be added to S, if guaranteed to be definite, to represent the head of another lax equivalence-path. Hence we claim that if the lax equivalence-path $\langle \chi(X), \chi(Y) \rangle$ exists in G and $Y \in \alpha(e) \cup \kappa(e)$ then $\chi(Y)$ will be returned in the result of EQUIVALENCE-CLOSURE. □

PROOF (NECESSITY). To prove that EQUIVALENCE-CLOSURE returns the correct lax equivalence closure of X, by contradiction assume that EQUIVALENCE-CLOSURE returns the equivalence class of X which contains $\chi(Y)$, but $Y \neq X$ and either Y does not represent a real attribute in e or a constant, or the lax equivalence-path $\langle \chi(X), \chi(Y) \rangle$ is not in G. Then at some point during the traversal of G EQUIVALENCE-CLOSURE must add $\chi(Y)$ to S^+ at line 697. $\chi(Y)$ must represent a real attribute or a constant, due to the test on line 696. Furthermore, $\chi(Y)$ was previously an element of the set S. For $\chi(Y)$ to exist in S one of the following must have occurred:

1. Y is the input parameter X, so $\chi(Y)$ is added to S during initialization (line 690), which contradicts our initial assumption that X and Y are not in the same equivalence class.

2. Otherwise, if X and Y are different then we again prove the remainder of the cases by induction on number of strict and lax equivalence edges traversed in G. If $\chi(Y) \subseteq S^+$ then there must exist a strict or lax undirected edge with target $\chi(Y)$, since $\chi(Y)$ must first be added to S at either lines 702 or 710. Therefore $\chi(Y) \in S$ only as the result of the existence of a strict or lax edge in E^E or E^e with $\chi(Y)$ as the target vertex.

   Basis. The base case of the induction is that there exists an edge $(\chi(X), \chi(Y)) \in E^E \cup E^e$ which represents a direct strict or lax equivalence-path $\langle \chi(X), \chi(Y) \rangle$, contradicting our initial assumption.

Induction. Otherwise, in our FD-graph implementation there are two additional sources of a strict undirected edge with target $\chi(Y)$, and that is an edge linking another single vertex $v_i \in V^A$ such that $(v_i, \chi(Y)) \in E^E$ or $(v_i, \chi(Y)) \in E^e$. In either case the vertex v_i was added to S only through the direct or indirect traversal of strict (or lax) equivalence edges in G. By Lemma 33 we know that $\chi(Y) \subseteq S^+$ if the strict equivalence path $\langle v_i, \chi(Y)\rangle$ is in G. For lax equivalence, $\chi(Y)$ is added to S only on line 710, and the prior tests (line 709) ensure that the lax equivalence path is transitive only over a definite attribute. Hence the addition of v_i to S implies the existence of a transitive lax equivalence-path from X to v_i. Such an equivalence-path, however, implies that the lax-equivalence path $\langle \chi(X), \chi(Y)\rangle$ exists in G, again contradicting our initial assumption.

Hence we conclude that EQUIVALENCE-CLOSURE will return $\chi(Y)$ as part of the lax closure of X only if $Y \in \alpha(e) \cup \kappa(e)$ and $\chi(Y) \in X_\xi^+$. □

THEOREM 10 (EQUIVALENCE CLOSURE)
Procedure EQUIVALENCE-CLOSURE is correct.

PROOF. Follows from Lemmas 32, 33, and 34. □

3.7 Related work

Early work on functional dependencies concentrated on *schema* decomposition. In other words, at that time the goal of functional dependency analysis was to determine the 'best' database design given a set of attributes and a set of functional (including key) dependencies. References [14, 22, 24, 28, 79, 87, 185, 191, 193, 201] are representative of this early research on dependency theory. In addition, several researchers have studied the interaction of functional dependencies with other forms of constraints; noteworthy examples include the work of Beeri et al. [23], Fagin [84], and Nicolas [218] who studied the interaction of functional and multivalued dependencies, and Mitchell [206], Casanova, Fagin, and Papadimitriou [45, 46], and Johnson and Klug [145] who studied the interaction of functional dependencies with inclusion dependencies.

Klug [162] developed a high-level procedure for determining those functional dependencies that hold in an arbitrary relational algebra expression consisting of the projection, restriction, selection (a restriction predicate that references a literal), cross product, and union operators. Klug's procedure relies on computing the transitive closure of a set of dependencies, but he gave no details as to how this was to be done. Klug's motivation was to determine the validity of functional dependencies that hold in a view, in

the context of an exported schema in an ANSI-SPARC multidatabase system. Klug later [164] studied derived dependencies in the context of tableaux, but again the application was schema design. Darwen [70], on the other hand, considered the application of dependency analysis for query optimization. Darwen reiterated much of Klug's earlier work, but also considered additional algebraic operators such as intersection, natural join, grouping, and aggregation. Both authors considered a 'classical' relational model where null values and duplicate tuples are not permitted (as opposed to ANSI SQL semantics). Darwen's early results were also incorporated into the draft SQL3 standard [137], but the failure to take into account duplicates and null values produced a series of flaws that subsequent work has tried to correct [303]. Nevertheless we are unaware of any study of derived functional dependencies that takes into account conjunctive atomic conditions in an outer join's `On` condition.

Darwen's [70] main contribution is a much more detailed explanation of how to determine the closure of a set of dependencies, though his algorithm is exponential as it computes a minimal cover of Γ. His paper, and its corresponding rules in the draft SQL3 standard, provide the motivation for the present work. In addition to Darwen, Abiteboul, Hull, and Vianu [4, pp. 177–80] also looked at exploiting functional dependencies in the optimization of tableaux queries using the *chase* algorithm.

Determining the closure of a set of functional dependencies is a problem that has been extensively studied over the past two decades, though once again this research was conducted in the context of database schema normalization. Bernstein [28] and, in a follow-up paper, Beeri and Bernstein [22] use *derivation trees* [28] to represent functional dependencies and provide several algorithms for their analysis, including transitive closure. Maier, Mendelzon, and Sagiv [194] describe a chase algorithm for determining transitive dependencies in tableau queries; a forerunner of the chase can be found in reference [7]. Mannila and Räihä [195] and Ausiello et al. [19] both describe algorithms to compute the closure of a set of dependencies using a hypergraph representation. This latter work, which also provided an algorithm to produce a minimal covering so as to synthesize a relational schema into 3NF, provides the basis for the representation of functional dependencies in this thesis. Diederich and Milton [78, 79] use a slightly modified definition of attribute closure, which they term *r-closures*, to enable more efficient elimination of extraneous attributes and redundant dependencies, which in turn permit the determination of a minimal covering for $\mathcal{F}$. Such a minimal covering leads to a decomposition of the fewest possible number of relations. As mentioned earlier in Section 3.2.4, Yan [294, pp. 75–78] presents an algorithm which, given an input set of dependencies and an arbitrary predicate P, determines the closure of a set of attributes S for simple SPJ queries. The goal of the algorithm is to determine if a query can be rewritten to commute `Group-by` and join, one of sev-

eral semantic query optimization techniques Yan introduces for grouped queries. The algorithm is similar to the RESTRICTION procedure presented above, but Yan's algorithm considers a smaller class of queries and exploits a smaller set of constraints.

Three independent, though quite similar problems, are closely related to determining the set of functional dependencies that hold in a derived relation. The first related problem is that of finding the candidate key(s) of a base or derived relation, which (obviously) relies on the determination of transitive functional dependencies. Lucchesi and Osborn [191] were one of the first to study this problem. Their approach utilized Beeri and Bernstein's transitive closure algorithms (using derivation trees) for determining candidate keys of a set of relations. A recent paper by Saiedian and Spencer [246] offers yet another technique, using another form of directed graph called *attribute graphs*. By categorizing attributes into three subsets—those which are determinants only, dependents only, or both—the authors claim to reduce the algorithm's running time for a large subset of dependency families. Saiedian and Spencer also contrast other key-finding algorithms that have been proposed: references [82, 167, 273], [16, pp. 115], and [81, pp. 431] are five such algorithms. All these papers rely on duplicate-free relations: finding a determinant that determines all the attributes in a relation R does not necessarily imply a key when duplicate tuples are permitted. Consequently, Pirahesh et al. [230], Bhargava et al. [33, 34], Paulley and Larson [228], and Yan and Larson [295, 296] use similar but more straightforward approaches to determine the key of a derived (multiset) table in the context of semantic query optimization. This is done through the (simple) exploitation of equivalence comparisons in a query's `Where` clause, and finds only keys; other dependencies that are discovered in the process are ignored.

The second related problem is *query containment* [139, 245], the fundamental component of common subexpression analysis [89] that play a large role in query optimization (e.g. reference [254]), utilization of materialized views [39, 174, 239, 274], and multiple query optimization [10, 144, 227, 251]. Determining query containment relies on the analysis of relational expressions, the same type of analysis required to determine which new functional dependencies are introduced in the derived relation as the result of an arbitrary `Where` clause.

The third related problem is *view updatability* [75, 91, 93, 147–149, 169, 185, 198]. The problem of translating an update operation on a view into one or more update operations on base tables requires knowledge of which underlying tuples make up the view, ordinarily determined through analysis of key dependencies. A typical requirement of updating through a view is that the underlying functional dependencies in the base tables must continue to hold [185] [169, pp. 55]; hence key attributes cannot be projected out of a view [147, 185]. Medeiros and Tompa [197–199] describe a validation algorithm that

takes into account the existence of functional dependencies when deciding how to map an update operation on a view into one or more base tables.

3.8 Concluding remarks

Our complexity analysis of the FD-graph construction algorithm given in Section 3.4 demonstrated that its running time was proportional to the square of its input, though we assumed $O(1)$ vertex and edge lookup, insertion, and deletion. Similarly, the algorithms for computing an FD-graph's dependency and equivalence closure were also polynomial in the size of the FD-graph. Clearly, our stated bounds are not tight; there are a variety of minor improvements we could make to reduce the running time of the more complex procedures. For example, we could quite easily reduce the running time of the DEPENDENCY CLOSURE algorithm from $O(\|V\|^2)$ to $O(\|V\| + \|E\|)$, using Beeri and Bernstein's [22, pp. 44–5] 'counter' technique for computing the closure of dependencies represented with derivation trees[25].

However, it is not clear that the use of the hash tables described by Dietzfelbinger, Karlin, and Mehlhorn et al. [80] is indeed 'optimal' for the construction and maintenance of FD-graphs in a typical relational database system. One set of tradeoffs is in terms of both the space required for the data structures themselves, and the additional software required to maintain them. In addition, our approach centered on deferring the computation of any closure; but naïvely recomputing the strict or lax closure of a set of attributes on demand may, in the end, prove more expensive, depending on the number of times attribute closure is required during optimization. Hence it may be worthwhile to consider other techniques for representing an FD-graph.

Several authors have developed algorithms for *on-line* computation of the transitive closure of a directed graph, where the closure is automatically maintained in the face of edge insertions and deletions [130, 141, 168]. In a recent paper, Ausiello, Nanni, and Italiano [21] modified their representation of FD-graphs so that they could be maintained in a dynamic fashion—that is, so that the transitive closure was maintained along with the graph during both vertex and edge insertion and deletion. They introduced several additional data structures to do this, in addition to the 'base' representation of an FD-graph, which is done using adjacency lists. The first is an $n \times n$ array A of pointers that represents the closure of simple attributes. If the dependency $X \longrightarrow Y$ exists in G then the array value $A[X, Y]$ points to the last simple (or compound) vertex in G that is on an

25 Beeri and Bernstein's technique is also utilized by Ausiello, D'Atri, and Saccà [19] for computing the closure of dependencies represented with their FD-graphs.

FD-path from X to Y (but excluding Y itself). Secondly, they use n reachability vectors to quickly determine if a simple vertex Y is on an FD-path originating at each vertex X. Thirdly, they use an AVL-tree to maintain the compound vertices in sorted order. This permits a faster search to determine whether or not a compound determinant to be introduced corresponds to a vertex already in the graph. With this construction, their approach requires $O(n^2)$ elements for the closure array, and a balanced tree implementation for compound nodes. They also do not address the issue of *deleting* a dependency from the graph, which will likely be more complex with the extra structures.

Another set of tradeoffs lies in the complexity of the query being analyzed. Suppose we have a schema composed of tables that have large numbers of attributes but with simple (non-compound) keys. Then the $n \times n$ closure array may be quite large, even for very simple queries, but will be quite sparse. Maintaining the array will have little if any benefit, since the length of any FD-path is likely to be limited to at most, say, 2 or 3. This 'sparseness' of directed edges is also a weakness of the hash-table based approach we assumed in Section 3.5. Additional research is needed to determine the 'best' technique for maintaining FD-graphs given a representative set of queries. Diederich and Milton [79] have done a similar analysis on closure algorithms, and their approach may be useful in this context.

In the remainder of the thesis, we will utilize our extended relational model, and exploit the knowledge of implied functional dependencies and constraints developed in this chapter, to improve the optimization of large classes of queries. We assume that the reader will be able to make the necessary transformations between extended tables, and algebraic expressions over them defined by our extended relational model, to ANSI SQL base tables and SQL expressions over them. We also will use the more conventional notation RowID(R) to denote the tuple identifier of an extended table, instead of $\iota(R)$.

4 Rewrite optimization with functional dependencies

4.1 Introduction

SQL[26] queries that contain `Distinct` are common enough to warrant special consideration by commercial query optimizers because duplicate elimination often requires an expensive sort of the query result. It is worthwhile, then, for an optimizer to identify redundant `Distinct` clauses to avoid the sort operation altogether. Example 23 illustrates a situation where a `Distinct` clause is unnecessary.

EXAMPLE 23
Consider the query

```
Select  Distinct S.VendorID, P.PartID, P.Description
From    Supply S, Part P
Where   S.PartID = P.PartID AND P.Cost > 100
```

which lists all parts with cost greater than $100 and the identifiers of vendors that supply them. The `Distinct` in the query's `Select` clause is unnecessary because each tuple in the result is uniquely identified by the combination of `VendorID` and `PartID`, the primary key of SUPPLY. Conversely, Example 24 presents a case where duplicate elimination *must* be performed.

EXAMPLE 24
Consider a query that lists expensive parts along with each distinct supply code:

```
Select  Distinct S.SupplyCode, P.PartID, P.Description
From    Supply S, Part P
Where   S.PartID = P.PartID and P.Cost > 100.
```

In this case, duplicate elimination is required because two parts, supplied by different vendors, can have the same supply code. These two examples raise the following questions:

- Under what conditions is duplicate elimination unnecessary?

- Are there other types of queries where duplicate analysis enables alternate execution strategies?
- If so, when are these other execution strategies beneficial, in terms of query performance?

In this chapter we explore the first two questions. Our main theorem provides a necessary and sufficient condition for deciding when duplicate elimination is unnecessary. Testing the condition utilizes FD-graphs developed in the previous chapter, but in addition requires minimality analysis of (super)keys, which cannot always be done efficiently. Instead, we offer a practical algorithm that handles a large class of possible queries yet tests a simpler, sufficient condition. The rest of the chapter is organized as follows. Section 4.2 formally defines the main result in terms of functional dependencies. Section 4.3 presents our algorithm for detecting when duplicate elimination is redundant for a large subset of possible queries. Section 4.4 illustrates some applications of duplicate analysis; we consider transformations of SQL queries using schema information such as constraints and candidate keys. Section 4.5 summarizes related research, and Section 4.6 presents a summary and lists some directions for future work.

4.2 Formal analysis of duplicate elimination

Section 2.4 detailed the SQL2 mechanisms for declaring primary and candidate keys of base tables. A key declaration implies that all attributes of the table are functionally dependent on the key. For duplicate elimination, we are interested in which functional dependencies hold in a *derived* table—a table defined by a query or view. We call such dependencies *derived functional dependencies.* Similarly, a key dependency that holds in a derived table is a *derived key dependency.* The following example illustrates derived functional dependencies.

EXAMPLE 25
Consider the derived table defined by the query

```
Select All  S.VendorID, S.SupplyCode, P.PartID, P.Description
From   Supply S, Part P
Where  P.PartID = S.PartID and S.VendorID = :Supplier-No
```

which lists the supplier ID and supply code, and part name and number, for all parts supplied by vendor `:Supplier-No`. We claim that `PartID` is a key of the derived table. `PartID` is certainly a key of the derived table D where $D = \sigma$[`VendorID` = `:Supplier-No`](`Supply`). In this case, `:Supplier-No` is a host variable in an application program, assumed to have the same domain as `S.VendorID`. Each tuple of D joins

with at most one tuple from PART since **PartID** is PART's primary key. Therefore, **PartID** remains the key of the derived table obtained after projection. Since the key dependency **VendorID** $\longrightarrow$ **SupplyCode** holds in the SUPPLY table, it should also hold in the derived table. In this case, a key dependency in a source table became a non-key functional dependency in the derived table.

4.2.1 Main theorem

Example 25 illustrates the usefulness of derived functional dependencies in determining if duplicate elimination is required, because the existence of a primary key in each output tuple—**PartID** in the example—means that duplicates cannot exist. In this section, we formally define the conditions necessary to determine if a key exists in a derived table, taking into account SQL's three-valued logic and multiset semantics.

Consider a simple SQL query specification that involves only projection, restriction, and Cartesian product; for simplicity, we do not permit queries to contain arithmetic expressions, outer joins, **Group by** clauses, or **Having** clauses. A query's **Where** clause may contain host variables—constants whose values are known only at query execution. We assume that each restriction predicate expression containing host variables compares them to other union-compatible arguments, for example, domains of particular columns. We define a host variable's domain as the intersection of the column domains with which the host variable is compared.

We would like to determine if the result of the query

```
Select A
From   R, S
Where  C_R ∧ C_S ∧ C_R,S
```

may contain duplicate rows. Intuitively, the uniqueness condition will be met if:

- both R and S have primary keys, so that the key of $(R \times S)$ is the concatenation of Key(R) with Key(S), denoted Key(R) $\circ$ Key(S);
- if either R or S lack a key, then we can utilize the respective tuple identifiers of each tuple to act as a surrogate key;
- either all the columns of Key($R \times S$) are in the projection list, or
- a subset of the key columns is present in the projection list, and the values of the other key columns are equated to constants or can be inferred through the restriction predicate or table constraints.

This notion corresponds to the following theorem.

THEOREM 11 (UNIQUENESS CONDITION)
Consider a query involving only projection, restriction, and Cartesian product over two tables R and S where R and S each have *at least* one candidate key. The restriction predicate $C_R \wedge C_S \wedge C_{R,S}$ may contain expressions that include host variables; we denote this set of input parameters by h. Thus we identify the test of a restriction predicate, which includes host variables, on tuple r of R with the notation $C_R(r, h)$. Then the two expressions

$$Q = \pi_{All}[A](\sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S))$$

and

$$V = \pi_{Dist}[A](\sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S))$$

are equivalent if and only if the following condition holds:

$$\begin{aligned}
&\forall\, r, r' \in \text{Domain}(R \times S); \forall\, h \in \text{Domain}(H) : \qquad (4.1)\\
&\quad \{\, \lceil T_R(r) \rceil \wedge \lceil T_R(r') \rceil \wedge \lceil T_S(r) \rceil \wedge \lceil T_S(r') \rceil \wedge \\
&\quad (\textit{for each } K_i(R) : (r[K_i(R)] \stackrel{\omega}{=} r'[K_i(R)]) \Longrightarrow r[\alpha(R)] \stackrel{\omega}{=} r'[\alpha(R)]) \wedge \\
&\quad (\textit{for each } U_i(R) : (\lfloor r[U_i(R)] = r'[U_i(R)] \rfloor) \Longrightarrow r[\alpha(R)] \stackrel{\omega}{=} r'[\alpha(R)]) \wedge \\
&\quad (\textit{for each } K_i(S) : (r[K_i(S)] \stackrel{\omega}{=} r'[K_i(S)]) \Longrightarrow r[\alpha(S)] \stackrel{\omega}{=} r'[\alpha(S)]) \wedge \\
&\quad (\textit{for each } U_i(S) : (\lfloor r[U_i(S)] = r'[U_i(S)] \rfloor) \Longrightarrow r[\alpha(S)] \stackrel{\omega}{=} r'[\alpha(S)]) \wedge \\
&\quad \lfloor C_R(r, h) \rfloor \wedge \lfloor C_R(r', h) \rfloor \wedge \lfloor C_S(r, h) \rfloor \wedge \\
&\quad \lfloor C_S(r', h) \rfloor \wedge \lfloor C_{R,S}(r, h) \rfloor \wedge \lfloor C_{R,S}(r', h) \rfloor \Longrightarrow \\
&\qquad [\, (r[A] \stackrel{\omega}{=} r'[A]) \Longrightarrow \\
&\qquad\quad (r[\text{Key}(R \times S)] \stackrel{\omega}{=} r'[\text{Key}(R \times S)]) \,] \,\}
\end{aligned}$$

PROOF (SUFFICIENCY). We assert that if the theorem's condition is true then the query result contains no duplicates. In contradiction, assume the condition stated in Theorem 11 holds but $Q \neq V$; i.e. Q contains duplicate rows. If $Q \neq V$, then there exists a valid instance $I(R)$ and a valid instance $I(S)$ giving different results for Q and V. Then there exist (at least) two different tuples $r_0, r_0' \in (I(R) \times I(S))$ such that $r_0[A] \stackrel{\omega}{=} r_0'[A]$. Projecting r_0 and r_0' onto base tables $I(R)$ and $I(S)$, r_0 and r_0' are derived from the tuples $r_0[\alpha(S)]$, $r_0'[\alpha(S)]$, $r_0[\alpha(R)]$, and $r_0'[\alpha(R)]$. Furthermore, $r_0[\alpha(R)], r_0'[\alpha(R)] \in \sigma[C_R](R)$ and $r_0[\alpha(S)], r_0'[\alpha(S)] \in \sigma[C_S](S)$. If $Q \neq V$, then the extended Cartesian product of these tuples, which satisfies the condition $C_{R,S}$, yields at least two tuples in Q's result. This means that either the tuples in $I(S)$ are different ($r_0[\alpha(S)] \stackrel{\omega}{\neq} r_0'[\alpha(S)]$), the tuples in $I(R)$ are different, or both. It follows that the consequent $r_0[\text{Key}(R \times S)] \stackrel{\omega}{=} r_0'[\text{Key}(R \times S)]$

must be false, since if either $r_0[\alpha(S)] \not\stackrel{\omega}{=} r'_0[\alpha(S)]$, $r_0[\alpha(R)] \not\stackrel{\omega}{=} r'_0[\alpha(R)]$, or both, then the keys of the respective tuples must be different; a contradiction. Therefore, we conclude that no duplicate rows can appear in the query result if the condition of Theorem 11 holds. □

PROOF (NECESSITY). Assume that for every valid instance of the database, Q cannot generate any duplicate rows, but the condition stated in Theorem 11 does not hold. To prove necessity, we must show that we can construct valid instances of R and S for which Q results in duplicate rows.

If Theorem 11's condition does not hold, then there must exist two tuples $r_0, r'_0 \in \text{Domain}(R \times S)$ so that the consequent $(r_0[A] \stackrel{\omega}{=} r'_0[A]) \implies (r_0[\text{Key}(R \times S)] \stackrel{\omega}{=} r'_0[\text{Key}(R \times S)])$ is false, but its antecedents (table constraints, key dependencies, and query predicates) are true. If r_0 and r'_0 disagree on their key, then there must exist at least one column $\epsilon \in \text{Key}(R) \circ \text{Key}(S)$ where $r[\epsilon] \not\stackrel{\omega}{=} r'[\epsilon]$. Projecting r_0 and r'_0 onto base tables R and S, we get the database instance consisting solely of the tuples $r_0[\alpha(S)]$, $r'_0[\alpha(S)]$, $r_0[\alpha(R)]$, and $r'_0[\alpha(R)]$. This instance is valid since the tuples satisfy the table and uniqueness constraints for R and S. Furthermore $r_0[\alpha(S)], r'_0[\alpha(S)] \in \sigma[C_S](S)$ and $r_0[\alpha(R)], r'_0[\alpha(R)] \in \sigma[C_R](R)$. Because all constraints are satisfied and $r_0[A] \stackrel{\omega}{=} r'_0[A]$, V contains a single tuple. Suppose $\epsilon \in \text{Key}(S)$. Then $r_0[\alpha(S)] \not\stackrel{\omega}{=} r'_0[\alpha(S)]$, and the extended Cartesian product with $r_0[\alpha(R)]$ and $r'_0[\alpha(R)]$ satisfying $C_{R,S}$ yields at least two tuples. A similar result occurs if $\epsilon \in \text{Key}(R)$. In either case, Q contains at least two tuples, so $Q \neq V$. Therefore, we conclude that the condition in Theorem 11 is both necessary and sufficient. □

Note that we can extend this result to involve more than two tables in the Cartesian product.

EXAMPLE 26

Consider the query from Example 25, modified to eliminate duplicate rows:

```
Select Distinct  S.VendorID, S.SupplyCode, P.PartID, P.Description
From   Supply S, Part P
Where  P.PartID = S.PartID and S.VendorID = :Supplier-No.
```

We can safely ignore the **Distinct** specification in the above query if the condition of Theorem 11 holds:

$\forall\, r, r' \in \text{Domain}(\mathbf{S} \times \mathbf{P})$;

$\forall$:Supplier-No $\in$ Domain(S.VendorID) :

Tuple constraints (**Check** *conditions*)

$\{ \lceil r[\mathbf{P.Price}] \geq r[\mathbf{P.Cost}] \rceil \wedge$

$(\lceil r[\texttt{P.Qty}] \neq 0\rceil \vee \lceil r[\texttt{P.Status}] = \text{Inactive}\rceil)\wedge$

$(\lceil r[\texttt{S.Rating}] = \text{'A'}\rceil \vee \lceil r[\texttt{S.Rating}] = \text{'B'}\rceil \vee \lceil r[\texttt{S.Rating}] = \text{'C'}\rceil)\wedge$

$\lceil r'[\texttt{P.Price}] \geq r'[\texttt{P.Cost}]\rceil\wedge$

$(\lceil r'[\texttt{P.Qty}] \neq 0\rceil \vee \lceil r'[\texttt{P.Status}] = \text{Inactive}\rceil)\wedge$

$(\lceil r'[\texttt{S.Rating}] = \text{'A'}\rceil \vee \lceil r'[\texttt{S.Rating}] = \text{'B'}\rceil \vee \lceil r'[\texttt{S.Rating}] = \text{'C'}\rceil)\wedge$

Primary key dependency for **Supply**

$(r[\texttt{S.VendorID}] \stackrel{\omega}{=} r'[\texttt{S.VendorID}]\wedge$

$r[\texttt{S.PartID}] \stackrel{\omega}{=} r'[\texttt{S.PartID}]) \Longrightarrow$

$(r[\texttt{S.Rating}] \stackrel{\omega}{=} r'[\texttt{S.Rating}] \wedge r[\texttt{S.SupplyCode}] \stackrel{\omega}{=}$

$r'[\texttt{S.SupplyCode}] \wedge r[\texttt{S.Lagtime}] \stackrel{\omega}{=} r'[\texttt{S.Lagtime}])\wedge$

Primary key dependency for **Part**

$(r[\texttt{P.PartID}] \stackrel{\omega}{=} r'[\texttt{P.PartID}]) \Longrightarrow$

$(r[\texttt{P.Description}] \stackrel{\omega}{=} r'[\texttt{P.Description}] \wedge r[\texttt{P.Status}] \stackrel{\omega}{=} r'[\texttt{P.Status}]\wedge$

$r[\texttt{P.Qty}] \stackrel{\omega}{=} r'[\texttt{P.Qty}] \wedge r[\texttt{P.Price}] \stackrel{\omega}{=} r'[\texttt{P.Price}])\wedge$

$r[\texttt{P.Cost}] \stackrel{\omega}{=} r'[\texttt{P.Cost}] \wedge r[\texttt{P.Support}] \stackrel{\omega}{=} r'[\texttt{P.Support}])\wedge$

$r[\texttt{P.ClassCode}] \stackrel{\omega}{=} r'[\texttt{P.ClassCode}])\wedge$

Query predicate conditions

$\lfloor r[\texttt{S.VendorID}] = \texttt{:Supplier-No}\rfloor \wedge \lfloor r'[\texttt{S.VendorID}] =$

$\texttt{:Supplier-No}\rfloor \wedge \lfloor r[\texttt{P.PartID}] = r[\texttt{S.PartID}]\rfloor\wedge$

$\lfloor r'[\texttt{P.PartID}] = r'[\texttt{S.PartID}]\rfloor\ ] \Longrightarrow$

Projection attributes

$[(r[\texttt{S.VendorID}] \stackrel{\omega}{=} r'[\texttt{S.VendorID}] \wedge r[\texttt{S.SupplyCode}] \stackrel{\omega}{=}$

$r'[\texttt{S.SupplyCode}] \wedge r[\texttt{P.PartID}] \stackrel{\omega}{=} r'[\texttt{P.PartID}]\wedge$

$r[\texttt{P.Description}] \stackrel{\omega}{=} r'[\texttt{P.Description}])\ ] \Longrightarrow$

Key of P ∘ Key of S

$(r[\texttt{P.PartID}] \stackrel{\omega}{=} r'[\texttt{P.PartID}] \wedge r[\texttt{S.PartID}] \stackrel{\omega}{=} r'[\texttt{S.PartID}]\wedge$

$r'[\texttt{S.VendorID}] \stackrel{\omega}{=} r[\texttt{S.VendorID}])]\}$

Although complex, this expression is satisfiable: ignoring the table constraints and key dependencies, we can see from the consequent

$(r[\texttt{S.VendorID}] \stackrel{\omega}{=} r'[\texttt{S.VendorID}] \wedge r[\texttt{S.SupplyCode}] \stackrel{\omega}{=}$

$r'[\texttt{S.SupplyCode}] \wedge r[\texttt{P.PartID}] \stackrel{\omega}{=} r'[\texttt{P.PartID}]\wedge$

$r[\texttt{P.Description}] \overset{\omega}{=} r'[\texttt{P.Description}])\,] \Longrightarrow$

Key of P $\circ$ *Key of S*

$(r[\texttt{P.PartID}] \overset{\omega}{=} r'[\texttt{P.PartID}] \wedge r[\texttt{S.PartID}] \overset{\omega}{=} r'[\texttt{S.PartID}] \wedge$

$r'[\texttt{S.VendorID}] \overset{\omega}{=} r[\texttt{S.VendorID}])$

that the conjuncts containing P.PartID and S.PartID in the final consequent are trivially true. The conjunct containing S.VendorID is also true, since the antecedent $[r[\texttt{S.VendorID}] = \texttt{:Supplier-No}] \wedge [r'[\texttt{S.VendorID}] = \texttt{:Supplier-No}]$ implies that S.VendorID is constant. Therefore, the entire condition is true, and duplicate elimination is not necessary.

In the next section, we propose a straightforward algorithm for determining if a uniqueness condition, like the one above, holds for a given query and database instance.

4.3 Algorithm

We need to test whether a particular query, for any instance of a database, satisfies the conditions of Theorem 11 so that we can decide if duplicate elimination is unnecessary. Since the conditions are quantified Boolean expressions, the test is equivalent to deciding if the expression is satisfiable—a PSPACE-complete problem [102, pp. 171–2]. However, we can determine satisfiability of a simplified set of conditions through exploiting the strict functional dependency relationships known to hold in the result, computed by the various algorithms described in Section 3.4. Our algorithm to determine if duplicate elimination is unnecessary, described below, utilizes the FD-graph built for a query Q and checks if the transitive closure of strict dependencies (denoted Γ) whose determinants are in the query's Select list contains a key of each table in the From clause[27].

720 **Procedure:** DUPLICATE-ELIMINATION
730 **Purpose:** *Determine if duplicate elimination is unnecessary.*
740 **Inputs:** A query Q.
750 **Output:** Yes or No.
760 **begin**
770 $Q' \leftarrow Q$ with **Distinct** removed from the outermost query specification;
780 Construct the FD-graph G for query Q';
790 *--Compute the closure of the query's projection list, denoted A.*
800 $A_\Gamma^+ \leftarrow$ DEPENDENCY-CLOSURE(G, A, Γ);

27 For the moment we still presume that the class of queries under consideration comprises those containing only projection, restriction, and extended Cartesian product.

```
810  if any element of A_Γ^+ is a tuple identifier vertex then
820    return Yes
830  else
840    return No
850    fi
860  end
```

4.3.1 Simplified algorithm

While an FD-graph can be used to determine whether or not duplicate elimination is unnecessary, it is possible to optimize a slightly smaller class of queries by using a simplified form of FD-graph than that presented earlier. Our proposed algorithm exploits information about primary keys, candidate keys, and equality conditions in a **Where** clause. As with the analysis of restriction predicates in Section 3.2.4, we classify equality conditions into two types: Type 1 of the form $(v = c)$ and Type 2 of the form $(v_1 = v_2)$ where v, v_1, v_2 are columns and c is a constant.

Our simplified algorithm, SIMPLIFIED-DUPLICATE-ELIMINATION, uses a simplified form of FD-graph that does not require tuple identifiers, lax dependency edges, outer join vertices and edges, or equivalence edges. The algorithm requires the infrastructure to support only base tables, extended Cartesian product, and restriction. We first construct a hypergraph consisting of all of the base tables in the query's **From** clause, using a simplified version of the BASE TABLE algorithm, which builds a hypergraph representing the functional dependencies that hold for a base table's primary and definite candidate keys.

```
870  Procedure: BASE-TABLE (SIMPLIFIED)
880  Purpose: Construct a simplified FD-graph for table R.
890  Inputs: scheme of table R; query Q.
900  Output: FD-graph G.
910  begin
920    for each attribute a_i ∈ A_R do
930      Construct vertex v_i ∈ V^A;
940      Colour[v_i] ← Black;
950      if a_i is nullable in R then
960        Nullability[v_i] ← Nullable
970      else
980        Nullability[v_i] ← Definite
990        fi
1000     od ;
1010   for each primary or candidate key of R do
```

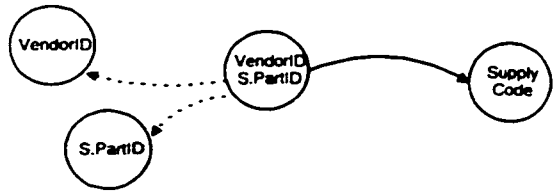
```
1020    if ∃ any key column that is nullable then continue fi;
1030    if Key(R) is composite then
1040      Construct the composite vertex K;
1050      V^C ← V^C ∪ K;
1060      for each v ∈ K do
1070        E^C ← E^C ∪ (K, v)
1080        od
1090    else
1100      Let K denote the singleton key vertex K ∈ V^A
1110      fi
1120    for each v ∈ V^A such that v ∉ K do
1130      E^F ← E^F ∪ (K, v)
1140      od
1150    od;
1160  for each attribute a_i ∈ A_R referred to in the Select list of Q do
1170    Colour[v_i] ← White
1180    od
1190  return G
1200  end
```

Construction of the simplified FD-graph to determine if a uniqueness condition holds consists of three steps. First, we union the base-table FD-graphs together (as usual, we assume that all attributes can be uniquely identified by their correlation names). Second, we add strict functional dependency edges to G for each Type 1 and Type 2 equality condition in the query's **Where** clause[28]. The algorithm supports only a simplified set of comparison conditions: conjunctive, null-intolerant equality conditions of Type 1 or 2, with no support for scalar functions. Third, we utilize the DEPENDENCY-CLOSURE algorithm to compute the transitive closure of the white attributes in G (the attributes in the projection) and then determine if the closure covers a primary or candidate key from each table.

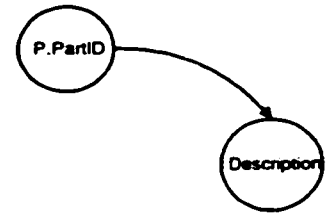
```
1210  Algorithm: SIMPLIFIED-DUPLICATE-ELIMINATION
1220  Purpose: Determine if duplicate elimination is unnecessary.
1230  Inputs: predicates C_R, C_S, C_{R,S}; key constraints U(R), U(S), K(R), K(S); projection list A.
1240  Output: Yes or No.
1250  begin
1260    --Construct an FD-graph for the query's extended Cartesian product.
```

28 As with the algorithms in Chapter 3, we assume *a priori* that each **Where** clause, if necessary, has been converted to conjunctive normal form.

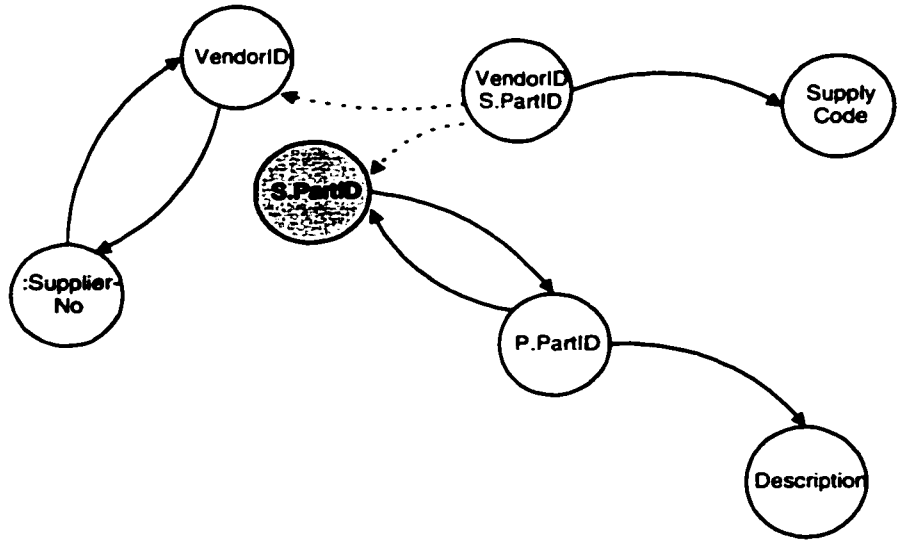
```
1270  G ← ∅;
1280  for each table T_i in the From clause (the set {R, S}) do
1290    G ← G ∪ SIMPLIFIED-BASE-TABLE(T_i);
1300    od
1310  --Construct strict edges for search conditions in the Where clause.
1320  Separate C_R ∧ C_S ∧ C_{R,S} ∧ T into conjuncts: C' = P_1 ∧ P_2 ∧ ... ∧ P_n;
1330  for each P_i ∈ C' do
1340    if P_i contains an atomic condition not of Type 1 or Type 2 then delete P_i from C'
1350    else if P_i contains a disjunctive clause then delete P_i from C' fi fi
1360    od
1370  for each conjunctive predicate P_i ∈ C' do
1380    --Consider Type 1 conditions that equate an attribute to a constant.
1390    if P_i is a Type 1 condition (v = c) then
1400      Construct vertex χ(c) to represent the constant c;
1410      V[G] ← V[G] ∪ χ(c);
1420      Colour[χ(c)] ← Gray;
1430      Nullability[χ(c)] ← Definite;
1440      E^F ← E^F ∪ (χ(v), χ(c));
1450      E^F ← E^F ∪ (χ(c), χ(v))
1460    else
1470      --Consider Type 2 conditions in P_i.
1480      if P_i is a Type 2 condition (v_1 = v_2) then
1490        E^F ← E^F ∪ (χ(v_1), χ(v_2));
1500        E^F ← E^F ∪ (χ(v_2), χ(v_1))
1510        fi
1520      fi
1530    od
1540  A_Γ^+ ← DEPENDENCY-CLOSURE(G, A, Γ);
1550  for each table T_i ∈ Q do
1560    if any candidate Key((T_i)) ∈ A_Γ^+ then continue
1570    else
1580      return No
1590      fi
1600    od
1610  return Yes
1620  end
```



(a) Simplified FD-graph for SUPPLY.



(b) Simplified FD-graph for PART.



(c) Simplified FD-graph for

```
Select  Distinct  S.VendorID, S.SupplyCode, P.PartID, P.Description
From    Supply S, Part P
Where   P.PartID = S.PartID and S.VendorID = :Supplier-No.
```

FIGURE 4.1: Development of a simplified FD-graph for the query in Example 26.

EXAMPLE 27
Suppose we are given the query of Example 26:

```
Select Distinct  S.VendorID, S.SupplyCode, P.PartID, P.Description
From   Supply S, Part P
Where  P.PartID = S.PartID and S.VendorID = :Supplier-No.
```

Applying SIMPLIFIED-DUPLICATE-ELIMINATION to this query we can trace the following steps:

Line 1290: The simplified FD-graphs for the base tables PART and SUPPLY are shown in Figure 4.1. The strict functional dependencies in each graph correspond to the primary keys of each table. The simplified FD-graph that represents the Cartesian product of these two tables is the union of the vertices and edges of these two graphs.

Line 1320: $C' \Longleftrightarrow$ `S.VendorID = :Supplier-No` $\wedge$ `S.PartID = P.PartID` $\wedge T$.

Lines 1330–1360: C' is unchanged.

Line 1410: Here a vertex representing the unknown constant in the host variable `:Supplier-No` is added to the graph, as are two strict edges between it and the vertex representing `S.VendorID`.

Line 1480: At this point we add two other strict edges to the graph between the two existing vertices that represent `S.PartID` and `P.PartID`. At this point no further edges are to be added, and the resulting FD-graph is shown in Figure 4.1(c).

Line 1540: Executing the DEPENDENCY-CLOSURE algorithm on the simplified FD-graph G, which embodies the set of strict functional dependencies Γ, yields $A_\Gamma^+ = \{$ `S.VendorID, S.SupplyCode, P.PartID, P.Description, S.PartID` $\}$.

Line 1560: A_Γ^+ contains both the primary key of PART (`P.PartID`) and the primary key of SUPPLY ({ `S.VendorID, S.PartID` }); we proceed.

Line 1610: Return **Yes** and stop.

Since the algorithm returns **Yes**, we know that the `Distinct` clause in the query is unnecessary.

4.3.2 Proof of correctness

Algorithm SIMPLIFIED-DUPLICATE-ELIMINATION tests a simpler, sufficient condition than that stated in Theorem 11; it ignores table constraints T_R and T_S and considers only conjunctive, atomic equality conditions in the query's **Where** clause. Any P_i deleted on line 1340 weakens condition C, but C remains a sufficient condition for testing if duplicate elimination is unnecessary. Similarly, any P_i deleted on line 1350 removes conditions like '`X = 5 or X = 10`'. Therefore, we need to show that the simplified condition

$$
\begin{aligned}
&\forall\, r, r' \in \text{Domain}(R \times S); \forall\, h \in \text{Domain}(H) : \qquad (4.2)\\
&\quad (\textit{for each } K_i(R) : (r[K_i(R)] \stackrel{\omega}{=} r'[K_i(R)]) \Longrightarrow r[\alpha(R)] \stackrel{\omega}{=} r'[\alpha(R)]) \wedge \\
&\quad (\textit{for each } K_i(S) : (r[K_i(S)] \stackrel{\omega}{=} r'[K_i(S)]) \Longrightarrow r[\alpha(S)] \stackrel{\omega}{=} r'[\alpha(S)]) \wedge \\
&\quad \lfloor C_R(r, h) \rfloor \wedge \lfloor C_R(r', h) \rfloor \wedge \lfloor C_S(r, h) \rfloor \wedge \\
&\quad \lfloor C_S(r', h) \rfloor \wedge \lfloor C_{R,S}(r, h) \rfloor \wedge \lfloor C_{R,S}(r', h) \rfloor \Longrightarrow \\
&\qquad [\, (r[A] \stackrel{\omega}{=} r'[A]) \Longrightarrow \\
&\qquad\quad (r[\text{Key}(R \times S)] \stackrel{\omega}{=} r'[\text{Key}(R \times S)])\,]\, \}
\end{aligned}
$$

where C'_R, C'_S, and $C'_{R,S}$ contain only atomic conditions using '=', is true when the algorithm returns **Yes**. Assuming SIMPLIFIED-DUPLICATE-ELIMINATION returns **Yes**, consider one iteration of the main loop starting on line 1370. Since line 1560 yields **True** (the primary keys of both R and S occur in Γ^+), then we know that the $\text{Key}(R) \circ \text{Key}(S)$ is functionally determined from the result attributes; a derived functional dependency. This means that the consequent $(r[A] \stackrel{\omega}{=} r'[A]) \Longrightarrow (r[\text{Key}(R \times S)] \stackrel{\omega}{=} r'[\text{Key}(R \times S)])$ must be true. Since we assume that all key dependencies hold, and we considered only conjunctive components $P_i \in C'$ then the simplified condition must hold for C since $P_1 \wedge P_2 \wedge \ldots \wedge P_m \Longleftrightarrow C'_R \wedge C'_S \wedge C'_{R,S}$.

4.4 Applications

Our goal is to show how relational query optimizers can employ Theorem 11 to expand the space of possible execution strategies for a variety of queries. Once the optimizer identifies possible transformations, it can then choose the most appropriate strategy on the basis of its cost model. In this section, we identify four important query transformations: detection of unnecessary duplicate elimination, conversion of a subquery to a join, conversion of set intersection to a subquery, and conversion of set difference to a subquery. Other researchers have described these query transformations elsewhere [74, 157, 212, 230] but with relatively little formalism. Later, in Section 6.2, we show the applicability of these transformations in nonrelational environments.

4.4.1 Unnecessary duplicate elimination

We believe that many queries contain unnecessary **Distinct** clauses, for two reasons. First, CASE tools often generate queries using 'generic' query templates. These templates specify **Distinct** as a conservative approach to handling duplicate rows. Second, some practitioners [71] encourage users to always specify **Distinct**, again as a conservative approach to simplify query semantics. We feel that recognizing redundant **Distinct** clauses is an important optimization, since it can avoid a costly sort.

EXAMPLE 28
Consider the following query which lists the vendor ID and part data for every part supplied by a vendor with the name **:VendorName**:

```
Select  Distinct V.VendorID, P.PartID, P.Description, P.Qty
From    Supply S, Vendor V, Part P
Where   V.Name = :VendorName and V.VendorID = S.VendorID
        and S.PartID = P.PartID.
```

This query satisfies the conditions in Theorem 11, and, consequently, **Distinct** in the Select clause is unnecessary.

4.4.2 Subquery to join

A number of researchers over the years, including Kim [157], Ganski and Wong [101], Muralikrishna [211, 212], Dayal [74], Pirahesh, Hellerstein, and Hasan [230], and Steenhagen, Apers, and Blanken [265] have studied the rewriting of correlated, positive existential subqueries as joins. Their rationale is to avoid processing the query with a naive nested-loop strategy. Instead, they convert the query to a join so that the optimizer can consider alternate join methods.

The class of queries we consider corresponds to Type J nested queries in Kim's paper; however, we explicitly consider three-valued logic and duplicate rows. Pirahesh et al. consider merging existential subquery blocks in Rule 7 of their suite of rewrite rules in the STARBURST query optimizer. We believe that it is worthwhile to analyze several subquery-to-join transformations, particularly when duplicate rows are permitted.

EXAMPLE 29
Consider the correlated query

```
Select  All Q.QuoteID, Q.Date, Q.QtyPrice
From    Quote Q
Where   Q.MinOrder > :Minimum-Qty and
        Exists (Select *
                From   Part P
                Where  Q.PartID = P.PartID and
                       P.PartID = :Part-No)
```

which lists all part quotations for a given part (from all the part's suppliers) as long as the minimum order quantity in the quote is greater than the parameter **'Minimum-Qty'**, which is a host variable. We claim that this query may be rewritten as

```
Select  All Q.QuoteID, Q.Date, Q.QtyPrice
From    Quote Q, Part P
Where   Q.MinOrder > :Minimum-Qty and
        Q.PartID = P.PartID and
        P.PartID = :Part-No
```

since the conditions in the subquery block can, at most, identify a single tuple in the PART table for each candidate tuple in QUOTE.

THEOREM 12 (SUBQUERY TO JOIN)
Consider a nested query on tables R and S that contains a positive existential subquery block. Assume that R and S have at least one candidate key and the same preconditions for host variables, as described in Theorem 11, hold. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R \wedge \exists(\sigma[C_S \wedge C_{R,S}](S))](R))$$

and

$$V = \pi_{All}[A_R](\sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S))$$

are equivalent if and only if the following condition holds:

$$\begin{aligned}
&\forall\, r \in \text{Domain}(R);\ \forall\, h \in \text{Domain}(H) : \\
&\quad \{\, [\, \lceil T_R(r) \rceil \wedge \lfloor C_R(r,h) \rfloor\,] \Longrightarrow \\
&\qquad [\, [\forall\, s, s' \in \text{Domain}(S) : \\
&\qquad \lceil T_S(s) \rceil \wedge \lceil T_S(s') \rceil \wedge \\
&\qquad (\textit{for each } K_i(S) : (s[K_i(S)] \stackrel{\omega}{=} s'[K_i(S)]) \Longrightarrow s[\alpha(S)] \stackrel{\omega}{=} s'[\alpha(S)]) \wedge \\
&\qquad (\textit{for each } U_i(S) : (\lfloor\, s[U_i(S)] = s'[U_i(S)]\, \rfloor) \Longrightarrow s[\alpha(S)] \stackrel{\omega}{=} s'[\alpha(S)]) \wedge \\
&\qquad \lfloor C_S(s,r,h) \rfloor \wedge \lfloor C_S(s',r,h) \rfloor \wedge \lfloor C_{R,S}(s,r,h) \rfloor \wedge \lfloor C_{R,S}(s',r,h) \rfloor\,] \Longrightarrow \\
&\qquad\quad (s[\text{RowID}(S)] \stackrel{\omega}{=} s'[\text{RowID}(S)])\,]\, \}
\end{aligned} \tag{4.3}$$

PROOF (SUFFICIENCY). We assert that at most one tuple from S can match the restriction predicate $C_S \wedge C_{R,S}$ if the condition in Theorem 12 holds. We prove this claim by contradiction; assume the condition in Theorem 12 holds, but the expressions Q and V are not equivalent. Then there must exist instances $I(R)$ and $I(S)$, a tuple $r_0 \in I(R)$, and (at least) two different tuples $s_0, s_0' \in I(S)$ such that $C_S(s_0, h)$, $C_S(s_0', h)$, $C_{R,S}(r_0, s_0, h)$, and $C_{R,S}(r_0, s_0', h)$ are satisfied. Since all the antecedents in the condition hold, and the table and key constraints hold for every tuple in Domain$(R \times S)$, then s_0 and s_0' must agree on their key. However, if the two tuples s_0 and s_0' agree on their key, then they violate the candidate key constraint for S, a contradiction.

We now argue that the semantics of Q and V are equivalent if at most one tuple from S matches each tuple from R. If the predicate $C_S \wedge C_{R,S}$ in Q is false or unknown, then the existential predicate $\exists(\sigma[C_S \wedge C_{R,S}](S))$ must return *false*, and the tuple represented by r_0 cannot be part of the result. Otherwise, if $C_S \wedge C_{R,S}$ is true then r_0 appears in the result. Similarly, for query V, any tuple r_0 that satisfies C_R will join with *at most* one tuple s_0 of S if the condition in Theorem 12 holds. If $C_S \wedge C_{R,S}$ is false or unknown for the two tuples r_0 and s_0 the restriction predicate is false; hence r_0 will not appear in the result. If $C_S \wedge C_{R,S}$ is true then at most one tuple of S qualifies, and the extended Cartesian product produces only a single tuple from R. Therefore, if at most one tuple from S matches each tuple of R, then $Q = V$. □

PROOF (NECESSITY). Assume that for every valid instance of the database, the subquery block on S can match at most one tuple r of R but the condition in Theorem 12 does not hold. To prove necessity, we must show we can construct valid instances $I(R)$ and $I(S)$ so that evaluating Q and V on those instances yields a different result.

If the condition in Theorem 12 is false there must exist two different tuples $s_0, s_0' \in$ Domain(S) and a tuple $r_0 \in$ Domain(R) such that the consequent $(s_0[\text{Key}(S)] \stackrel{=}{=} s_0'[\text{Key}(S)])$ is false, but its antecedents are true. The instance of S formed by tuples s_0 and s_0' is certainly valid, since it satisfies the table and uniqueness constraints for $I(S)$. In turn, r_0 is a valid instance of R because it satisfies the constraints on R. Since r_0 satisfies the condition C_R and since both s_0 and s_0' satisfy the restriction predicate $C_S \wedge C_{R,S}$, then Q yields one instance of r_0 in the result, but V yields two, a contradiction. We conclude that the condition in Theorem 12 is both necessary and sufficient. □

At this point, we can make several observations. Trivially, if the subquery in Q includes more than one table so that the subquery involves an extended Cartesian product of, say, tables S and W, we can extend Theorem 12 to include the corresponding conditions of W (similar to Theorem 11). Moreover, we observe that the two expressions

$$Q = \pi_{Dist}[A_R](\sigma[C_R \wedge \exists(\sigma[C_S \wedge C_{R,S}](S))](R))$$

and

$$V = \pi_{Dist}[A_R](\sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S))$$

are *always* equivalent, since duplicate elimination in the projection automatically excludes duplicate tuples obtained from the Cartesian product if more than one tuple in S matches the restriction predicate. This means that if we can alter the projection $\pi_{All}[A_R]$ to $\pi_{Dist}[A_R]$ without changing the query's semantics, then we can always convert a nested query to a join, as illustrated by the following example.

EXAMPLE 30
Consider the correlated query

```
Select All V.VendorID, V.Name, V.Address
From   Vendor V
Where  Exists (Select *
               From   Part P, Supply S, Quote Q
               Where  P.PartID = S.PartID and V.VendorID = S.VendorID
                      and Q.PartID = P.PartID and Q.VendorID = S.VendorID
                      and Q.MinOrder < 500 and P.Qty > 1000 )
```

which lists all suppliers who supply at least one part that is significantly overstocked, but whose minimum order quantity has been less than 500. Note that the uniqueness condition does not hold on the subquery block since many quotes can exist for the same part sold by the same vendor. However, this query may be rewritten as

```
Select Distinct V.VendorID, V.Name, V.Address
From   Vendor V, Part P, Supply S, Quote Q
Where  P.PartID = S.PartID and V.VendorID = S.VendorID
       and Q.PartID = P.PartID and Q.VendorID = S.VendorID
       and Q.MinOrder < 500 and P.Qty > 1000
```

since the uniqueness condition is satisfied for the outer query block (**VendorID** is the key of VENDOR). The optimizer converts the query to a join, disregards any columns from the other tables in the **From** clause, and then applies duplicate elimination that outputs only one VENDOR tuple for each unique **VendorID** in the Cartesian product. This observation leads to the following corollary:

COROLLARY 3 (SUBQUERY TO DISTINCT JOIN)
Consider a nested query on tables R and S that contains a positive existential subquery block. Assume that R and S have at least one candidate key and the same preconditions for host variables, as described in Theorem 12, hold. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R \wedge \exists(\sigma[C_S \wedge C_{R,S}](S))](R))$$

and

$$V = \pi_{Dist}[A_R](\sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S))$$

are equivalent if $\pi_{All}[A_R](\sigma[C_R](R))$ contains no duplicate rows. Duplicate elimination in the projection can be implemented through the use of tuple identifier(s) if suitable primary keys are either missing or are absent from the projection list $[A_R]$.

Two additional, though perhaps straightforward, observations are noteworthy here. First, note that we can safely transform a **Select Distinct** to a **Select All** in any subquery that is involved in a set-oriented search condition (e.g. **In, Any, All, Some**). This is because the select list of any **Exists** subquery is of no consequence to its result, and can be replaced by a simple literal[29]. We cannot, however, safely ignore **Distinct** in a subquery that is part of a scalar theta-comparison, since the RDBMS must generate a run-time execution error if the subquery returns more than one row.

Second, note that once all set-oriented predicates involving nested subqueries (e.g. **In, Any, All, Some**) have been transformed to simple (correlated) **Exists** predicates we can *always* flatten nested SPJ queries into joins without the need for duplicate elimination as long as the query's outermost block is not involved—that is, the subquery being 'flattened' is itself contained within another subquery. This is (again) because the select list of any **Exists** subquery is of no consequence to its result, so retaining duplicate tuples will not affect the result of any **Exists** predicate.

Thus far we have proved the equivalence of nested queries and joins in a variety of situations. Commercial relational database systems exploit this equivalence to transform nested queries to joins whenever possible [230] so that their optimizers' join enumeration algorithms can try to construct a less expensive access plan. To our knowledge, no commercial RDBMS performs the reverse transformation: rewriting a join as a subquery as a semantic transformation. Later, in Section 6.2, we consider this opposite case and show its potential as a semantic optimization in different database environments, including hierarchical and object-oriented database systems. For now, we present examples where converting an SQL query expression—specifically those involving **Except** or **Intersect**—into a nested query specification could lead to a cheaper access plan.

---

29 In fact, the ANSI standard defines **Exists** subqueries in precisely this manner.

4.4.3 Distinct intersection to subquery

Typically, most relational query optimizers execute the **Intersect** operation by evaluating each operand, sorting each intermediate result if necessary, and merging the inputs. Recall that the semantics of **Intersect** requires ignoring duplicates and, more troublesome, equating two tuples if:

- all non-**Null** columns are equal and
- for each **Null** column, its counterpart in the other (derived) table is also **Null**.

A subtle difficulty with the transformation of query expressions to nested query specifications arises because the equivalence of tuples, normally handled by a set operator that treats null values as equivalent, is now moved into a **Where** clause. Pirahesh et al. [230] do not handle this situation adequately in their paper (Rule 8); they transform a query without considering possibly **Null** keys.

THEOREM 13 (DISTINCT INTERSECTION TO EXISTS)
Consider a query expression that contains the set intersection operator on two tables R and S where R and S each have at least one candidate key. Either restriction predicate $C_R(R)$ or $C_S(S)$ may contain host variables. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R](R)) \cap_{Dist} \pi_{All}[A_S](\sigma[C_S](S))$$

and

$$V = \pi_{All}[A_R](\sigma[C_R \wedge \exists(\sigma[C_S \wedge C_{R,S}](S))](R)),$$

where $C_{R,S} = \bigwedge_{i=1}^{m} R[A_i] \stackrel{\omega}{=} S[A_i]$ are equivalent if the derived table $\pi_{All}[A_R](\sigma[C_R](R))$ does not contain duplicate rows.

PROOF. Omitted. □

Recall that the semantics of $R \cap_{Dist} S$ are to include a tuple from R iff it exists in S, and eliminate any duplicates in the result. If each result tuple from R is unique, then a tuple from $\pi_{All}[A_R](\sigma[C_R](R))$ may appear in the final result if at least one matching tuple is found in $\pi_{All}[A_S](\sigma[C_S](S))$.

Observe that the predicate $C_{R,S} = \bigwedge_{i=1}^{m} R[A_i] \stackrel{\omega}{=} S[A_i]$ can be expressed in SQL as `(R.X Is Null and S.X Is Null) or R.X = S.X` for each attribute **X** in the projection list (though a plain equijoin predicate will suffice for primary key columns, since a primary key is guaranteed not to contain any **Null** values). Using a primary key makes the transformation in Example 5 of reference [230] correct.

EXAMPLE 31
As an example of Theorem 13, consider the SQL query expression

```
Select All P.PartID
From   Part P
Where  P.ClassCode = 'BX'
Intersect
Select All Q.PartID
From   Quote Q
Where  Q.MinOrder < 500 and Q.UnitPrice < 0.75
```

which lists part numbers for those parts in part class 'BX' who have at least one quotation from any vendor where the unit price is less than $0.75 and the minimum order quantity is less than 500. Since **PartID** is the key of PART, the derived table from PART cannot contain duplicate rows, and we may rewrite the query as

```
Select All P.PartID
From   Part P
Where  P.ClassCode = 'BX' and
       Exists( Select *
               From   Quote Q
               Where  Q.MinOrder < 500 and Q.UnitPrice < 0.75 and
                      ( Q.PartID = P.PartID or ( Q.PartID Is Null
                      and P.PartID Is Null ) )
```

Obviously we can perform this transformation if either of the derived tables from PART or QUOTE have unique rows. Subsequent conversion of the **Exists** subquery to a join is possible [230] if the tests for **Nulls** are maintained[30]. We can make two additional observations:

- We now have a means of converting a nested query specification to a query expression involving intersection, another possible execution strategy.
- The semantics of **Intersect** and **Intersect All** are equivalent if *at least one of the derived tables cannot produce duplicate rows.* This leads to the following corollary:

---

30 Because the derived table from PART in Example 31 is a primary key column, and thus can never be Null, the test for null values in the transformed nested query is actually unnecessary.

COROLLARY 4 (ALL INTERSECTION TO EXISTS)
Consider a query expression that contains the set operator **Intersect All**. Assume that R and S have at least one candidate key, and the same preconditions for host variables, as described in Theorem 11, hold. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R](R)) \cap_{All} \pi_{All}[A_S](\sigma[C_S](S))$$

and

$$V = \pi_{All}[A_R](\sigma[C_R \wedge \exists(\sigma[C_S \wedge C_{R,S}](S))](R)),$$

where $C_{R,S}$ is defined as in Theorem 13 are equivalent if the expression $\pi_{All}[A_R](\sigma[C_R](R))$ does not contain duplicate rows. Similarly, Q and V (modified by interchanging R and S) are equivalent if the query specification on S does not contain duplicate rows.

4.4.4 Set difference to subquery

In a manner similar to the typical computation of set intersection, which sorts the two inputs and performs a merge, SQL's two set difference operators are typically processed by sorting the two operands and subsequently computing the difference of the two tuple streams. However, the semantics of set difference offers a natural transformation to a nested query form containing a **Not Exists** predicate, which can offer additional optimization opportunities. Once again, we have to be careful about the possible existence of null values in order to ensure a correct result.

EXAMPLE 32
Consider the SQL query expression

```
Select Distinct  P.PartID
From   Part P
Where  P.ClassCode = 'BX'
Except All
Select Distinct S.PartID
From   Supply S
```

which lists part numbers for those parts in part class 'BX' that are not supplied by any supplier. Since **PartID** is the key of PART, the derived table from PART cannot contain duplicate rows, and we may rewrite the query as

```
Select All P.PartID
From   Part P
Where  P.ClassCode = 'BX' and
       Not Exists( Select *
                   From   Supply S
                   Where  ( S.PartID = P.PartID or ( S.PartID Is Null
                          and S.PartID Is Null ) ).
```

Note that because **PartID** cannot be **Null** in either table, the disjunction that tests for Null can be omitted. Note as well that we may hereafter utilize Dayal's [74] techniques to convert this negated **Exists** predicate into an outer join—more precisely, to a grouped query over an outer join with a **Having** clause containing the aggregate function Count(*)—which offers yet another syntactic form that could lead to a more efficient access plan.

Example 32 illustrates three separate transformations. The first is to rewrite the query expression involving **Except All** as a nested query. This is possible because the **Distinct** in the first query specification means that there cannot be any duplicate rows in the result; hence in this case **Except** and **Except all** computes the identical result. To compute the result utilizing the nested query, we need only verify that each qualifying **PartID** from PART does not exist in any row of SUPPLY. Second, note that the **Distinct** in the second query specification is unnecessary since we are implementing the semantics of Except—duplicate tuples in the subquery do not affect the output. Third, we can eliminate the **Distinct** from the outer block in the nested query since **PartID** is the key of PART and hence there cannot be duplicate parts in the output.

More formally, we state the equivalence of these two queries as follows:

LEMMA 35 (EXCEPT ALL TO EXCEPT)
Consider a query expression that contains the set difference operator on two tables R and S where R and S each have at least one candidate key (an actual key or tuple identifier). Either restriction predicate $C_R(R)$ or $C_S(S)$ may contain host variables. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R](R)) -_{All} \pi_{All}[A_S](\sigma[C_S](S))$$

and

$$V = \pi_{All}[A_R](\sigma[C_R](R)) -_{Dist} \pi_{All}[A_S](\sigma[C_S](S))$$

are equivalent iff the derived table $\pi_{All}[A_R](\sigma[C_R](R))$ does not contain duplicate rows.

PROOF. Straightforward from the definition of **Except all**. □

THEOREM 14 (DISTINCT DIFFERENCE TO NOT EXISTS)
Consider a query expression that contains the set difference operator on two tables R and S where R and S each have at least one candidate key. Either restriction predicate $C_R(R)$ or $C_S(S)$ may contain host variables. Then the two expressions

$$Q = \pi_{All}[A_R](\sigma[C_R](R)) -_{All} \pi_{Dist}[A_S](\sigma[C_S](S))$$

and

$$V = \pi_{All}[A_R](\sigma[C_R \wedge \nexists(\sigma[C_S \wedge C_{R,S}](S))](R)),$$

where $C_{R,S} = \bigwedge_{i=1}^{m} R[A_i] \stackrel{\omega}{=} S[A_i]$ are equivalent if the derived table $\pi_{All}[A_R](\sigma[C_R](R))$ does not contain duplicate rows.

PROOF. Omitted. □

4.5 Related work

Semantic transformation of SQL queries using our uniqueness condition is a form of *semantic query optimization* [161]. Kim [157] originally suggested rewriting correlated, nested queries as joins to avoid nested-loop execution strategies. Subsequently, several researchers corrected and extended Kim's work, particularly in the aspects of grouping and aggregation [47, 74, 101, 155, 211, 212]. Much of the earlier work in semantic transformations ignored SQL's three-valued logic and the presence of `Null` values. To help better understand these problems, Negri, Pelagatti, and Sbattella [216] and von Bültzingsloewen [288] defined formal semantics for SQL using an extended relational calculus, although neither paper tackled the problems of duplicates. A significant contribution of Negri et al. is their notion of *query equivalence classes* for syntactically different, yet semantically equivalent, SQL queries.

Several authors discuss the properties of derived functional dependencies in two-valued logic. Klug [162] studied the problem of derived strict dependencies in two-valued relational algebra expressions with the operators projection, selection, restriction, cross-product, union, and difference. His paper's main contributions were (1) the problem of determining the equivalence of two arbitrary relational expressions is undecidable, (2) the definition and proof of a transitive closure operator for strict functional dependencies, and (3) an algorithm to derive all strict functional dependencies for an arbitrary expression, without set difference, and with a restricted order of algebraic operators. Maier [193] describes query modification techniques with respect to minimizing the number of rows in tableaux, which is equivalent to minimizing the number of joins in relational algebra. Maier's *chase computation* uses functional and join dependencies to transform

tableaux. Darwen [70] reiterates Klug's work, and gives an exponential algorithm for generating derived strict functional dependencies. Darwen concentrates on deriving candidate keys for arbitrary algebraic expressions and their applications, notably view updatability and join optimization. Ceri and Widom [48] discuss derived key dependencies with respect to updating materialized views. They define these dependencies in terms of an algorithm for deducing *bound columns*, quite similar in purpose to our SIMPLIFIED-DUPLICATE-ELIMINATION algorithm. In our approach, however, our formal proofs take into account other static constraints and explicitly handle the existence of **Null** values; our algorithm is simply a sufficient condition for determining candidate keys.

Pirahesh, Hellerstein, and Hasan [230] draw parallels between optimization of SQL subqueries in relational systems and the optimization of path queries in object-oriented systems. Their work in STARBURST focuses on rewriting complex **Select** statements as select-project-join queries. One of the query rewrite rules identifies when duplicate elimination is not required, through isolation of two conditions: uniqueness, termed the 'one-tuple-condition', and existence of a primary key in a projection list, termed the 'quantifier-nodup-condition'. However, we feel that optimization opportunities may be lost upon their insistence that the STARBURST rewrite engine convert *all* queries, whenever possible, to joins. In contrast, we believe that converting joins *to* subqueries offers possibilities for optimization in nonrelational systems. We explore that possibility in Chapter 6.

Bhargava, Goel and Iyer [32–34] extended the work described in this chapter and applied it to the optimization of (a) outer joins and (b) the set operators **Union**, **Intersect**, and **Except** and their interaction with projections of query specifications that eliminate duplicates. Their research into outer join optimization covered (1) join elimination of an outer join in the presence of distinct projection, (2) simplifications of outer joins to inner joins, (3) discovery of a uniqueness condition for a query block containing (possibly nested) outer joins. Their approach to uniqueness conditions with respect to outer joins was based on *key sets*, and influenced our approach to the maintenance of lax dependencies in FD-graphs. Some of their later work on set operations mirrors our own research, which due to space constraints was omitted from publication [228].

4.6 Concluding remarks

We have formally proved the validity of a number of semantic query rewrite optimizations for a restricted set of SQL queries, and shown that these transformations can potentially improve query performance in both relational and nonrelational database systems. Although testing the conditions for transformation is PSPACE-complete, our algorithm detects a large subclass of queries for which the transformations are valid. Our approach

takes into account static constraints, as defined by the SQL2 standard, and explicitly handles the 'semantic reefs' [155] referred to by Kiessling—duplicate rows and three-valued logic—which continue to complicate optimization strategies.

5 Tuple sequences and functional dependencies

An obvious benefit of using an ordered data structure like a B$^+$-tree in the implementation of a relational database system is that tuples may be retrieved in ascending or descending secondary key sequence, which quite often matches the ordering of the result tuples desired by an application program. Indexes present one of the few opportunities for exploiting ordering since tuples in base tables are not typically maintained in key sequence (clustered indexes are one exception). In this chapter, we illustrate how we can exploit *tuple sequences* [2, 3] in optimizing queries over ANSI SQL relational databases. In addition to project-select-join queries we formally defined in Section 2.3, we will also look at the possibility of exploiting tuple sequences to compute the result of SQL query specifications containing **Group by**, and query expressions containing **Union** or **Union all**, **Intersect** or **Intersect all**, and **Except** or **Except all**. Throughout this section, for simplicity we assume that the domains of attributes involved in any query can be totally ordered [103].

5.1 Possibilities for optimization

EXAMPLE 33
Consider the SQL query

```
Select D.Name, E.Surname, E.GivenName, E.Phone
From   Division D, Employee E
Where  D.Name = E.DivName
Order by E.Surname
```

that lists information about each employee in the firm in ascending sequence by surname, using the manufacturing schema described in Appendix A.

There are several possible ways to process the above query (see Figure 5.1). One possible access strategy, shown in Figure 5.1(a), is to perform a sort-merge join of the two tables over **Name** and **Divname**. Given the lack of any other restriction predicate, it may be necessary to first sort both the DIVISION and EMPLOYEE tables in their entirety, perform the merge join, and then sort the join's output to satisfy the **Order by** clause. Sorts

of the input relations could be avoided if there exist the appropriate indexes on each table, though retrieving each tuple randomly through the index is likely to significantly increase the cost of retrieval.

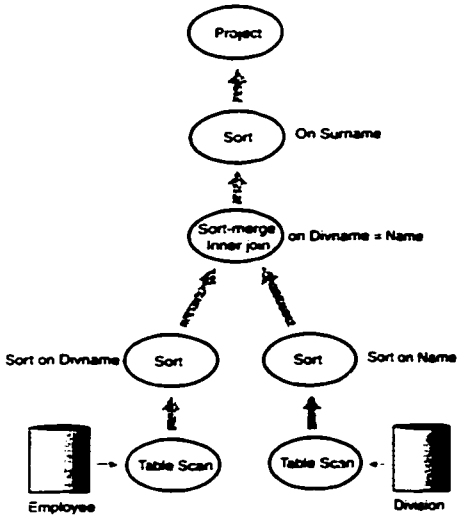
A second possible strategy (b) is to perform an indexed nested-loop join with the DIVISION as the 'outer' table, and EMPLOYEE as the indexed 'inner', assuming an index on the `Divname` attribute of EMPLOYEE. Conversely, strategy (c) reverses the join order and scans EMPLOYEE as the 'outer' table. In either case, however, we will need to sort the entire result to satisfy the query's **`Order by`** clause. However, suppose there exists an ascending index on **Surname**. Then a fourth possible strategy (d) is to scan the outer EMPLOYEE table by the index on **Surname**, and join each EMPLOYEE tuple with at most one from DIVISION as before. In this case a final sort of the result would be unnecessary, assuming that the nested-loop join implementation is order-preserving.

The process of query optimization is responsible for analyzing these tradeoffs to determine the cheapest access plan. Because every EMPLOYEE tuple will be in the result, the most efficient way to retrieve them is to perform a sequential scan. However, this access strategy requires a final sort, so it may not be the cheapest overall. Furthermore, it is not possible to return any result tuples to the application until all the EMPLOYEE tuples have been retrieved.

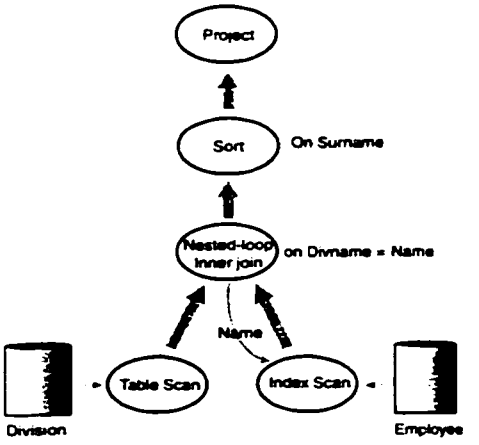
Using strategy (d), though possibly more costly, avoids both of these problems. This strategy may be particularly appropriate if only a subset of the tuples in the result will actually be retrieved by the application [42–44]. But this strategy will not always be considered in a commercial database system. For example, the query optimizer in ORACLE Version 7 rejects outright such a strategy as too expensive [69]. ORACLE will exploit an index to satisfy an **`Order by`** clause only if there exists at least one restriction predicate on the index's secondary key (in this case, **Surname**).

In this chapter, we are interested in how we can exploit 'interesting orders' [247] of tuple sequences in a multiset relational model. Some opportunities are:

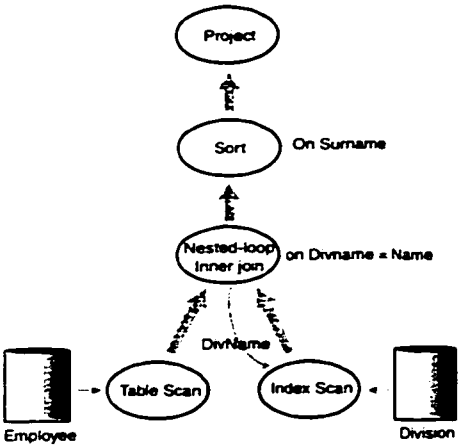
Sort avoidance. Avoiding an unnecessary sort can dramatically improve query execution time. The SQL language offers several possibilities where the analysis of a lexicographic tuple sequence can avoid an unnecessary sort. First, it may be possible to avoid a redundant sort to satisfy a query's **`Order by`** clause. Second, a redundant sort can be avoided for one or both of the inputs to a merge join, which can provide a significant reduction in query execution time and buffer pool utilization [100, 261]. A third example is to eliminate the need to materialize intermediate results during query processing. For instance, we may be able to exploit the ordered nature of sequences for processing queries containing `Distinct` or **`Group by`** [92]. The elimination of a materialization step is important not only because it may take fewer resources to compute the query's result; it



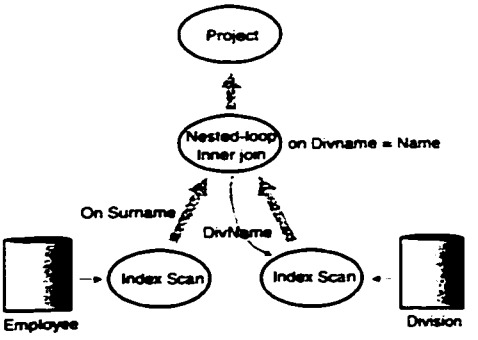
(a) Sort-merge join strategy.



(b) Nested-loop join strategy.



(c) Nested-loop strategy with Employee as 'outer' table.



(d) Nested-loop join strategy requiring no explicit sorting.

FIGURE 5.1: Some possible physical access plans for Example 33.

also means that the database system can begin returning result tuples to the application program at once, rather than after the computation of the entire (intermediate) result. This is a critical determination when a relational query optimizer attempts to optimize a query for response time, as opposed to most commercial query optimizers' goal of minimizing resource consumption.

A recent paper by Simmen, Shekita, and Malkemus [261] provides a framework for the analysis of tuple sequences to avoid redundant sorts. However their framework, which utilizes Darwen's [70] analysis of derived functional dependencies, lacks a solid theoretical foundation and one piece of analysis is missing: how to determine the lexicographic order of two or more relations involved in a nested-loop (or sort-merge) join.

Scan factor reduction. Consider a nested-loop join with an indexed inner relation. With one or more suitable conditions in the query's **Where** clause, we can exploit the fact that the inner tuples are retrieved in sequence so that we can 'cut' the search as soon as a tuple is retrieved whose indexed attribute(s) is greater than the one desired. A similar technique can be used to compute the aggregate functions **Max()** and **Min()**. If a query specifies `Min(X)` and attribute **X** is indexed in ascending sequence, then under certain conditions the database need only retrieve the first non-null value in the index to compute `Min(X)`. The situation with **Max(X)** is analogous.

More accurate cost estimation. In any given strategy, it may be useful to know for purposes other than joins if an intermediate result is ordered. For example, consider IBM's DB2/MVS that memoizes [202] the previously computed results of subqueries[31]. If it can be determined that the correlation variables for the subquery are sorted (that is, they correspond to the order in which the tuples of the outer query block are retrieved), then memoizing the subquery's result will not require more than a single row buffer: once a range of correlation values has been processed, the subquery will never again be executed with those values. As another example, Yu and Meng [299, pp. 142–3] give an algorithm for converting left-deep join strategies to bushy strategies in a multidatabase global query optimizer. Preserving the sortedness of each join means not only that the introduction of additional sort nodes can be avoided, but estimating the cost of the bushy access plan can be more efficient as the optimizer must re-estimate only a subset of the nodes in the transformed subtree.

Sort introduction. Consider an indexed nested-loop join strategy to compute the join of two tables R and S. If lookups on the indexed inner table (say S) are done using a sorted

---

31 Guy M. Lohman, IBM Almaden Laboratory, personal communication, 30 June 1996.

list of values, then it is likely that the join's cost will be decreased since each index leaf page for table S will be referenced only once. If the index is a clustering index, then it is likely that each base table page of S will also be referenced only once.

This examples illustrates the possibility of *sort introduction* to decrease the overall cost of an access plan. Moreover, it illustrates that there are more ways to exploit 'interesting orders' than simply for the optimization of joins, `Distinct`, and `Group by`—a query optimizer can exploit the ordering of intermediate results in a myriad of ways. For example, an optimizer can:

- push an interesting order down the algebraic expression tree to cheapen the execution cost of operators higher in the tree (also termed *sort-ahead* [261]);
- in addition, modify the ordering requirement to include additional attributes to eliminate the need for an additional sort operation higher in the tree;
- push down cardinality restrictions on the result (i.e. in the case of `Select Top` n queries) through order-preserving operations to restrict the size of intermediate results [44].

Such analysis, however, comes at a cost to the process of optimization [261]. Utilizing the sort order of a tuple stream means that an optimizer implemented with a classic dynamic programming algorithm [247] can no longer produce optimal access plans, since the choice of join strategy for a sub-plan may differ depending on the sort order of the strategy for an outer query block [100]. Hence dynamic programming optimizers, such as DB2's, use heuristics to guide the pruning of access strategies when exploiting sort order [261].

The topics covered in this chapter are as follows. After introducing some formalisms to describe order properties and interesting orders, we describe the infrastructure necessary to exploit order properties in a query optimizer, with some ideas as to their implementation. Thirdly, we look at how various implementations of relational algebra operators affect the ordering of tuples, and how to exploit that order in query processing. We conclude the chapter with a summary of related work and some thoughts on future research.

5.2 Formalisms for order properties

We begin by formally defining what we mean by an ordering of tuples in a multiset relational model with duplicates, null values, and three-valued logic. We then restate some of the concepts developed by Simmen, Shekita, and Malkemus [261] and show how they correspond to facets of *lexicographic indexes* previously described by Abiteboul and Ginsburg [2, 3].

DEFINITION 51 (TUPLE SEQUENCE)
A *sequence* of m tuples $r^* = r_1, r_2, \cdots r_m$ defines an explicit ordering to the m tuples of the instance $I(R)$, which can denote an instance of either a base or derived table. While it is possible to define a factorial number of sequences for any (finite) instance, the notion of a tuple sequence is concerned with physical access, i.e. tuple r_1 is 'retrieved' prior to tuple r_2.

To consider ordering tuple sequences that can contain nullable attributes, we need to consider how to treat null values in terms of their lexicographic order. We do so by following SQL2's treatment of null values, that is as a special value that we arbitrarily define as having a value less than *any other value in its domain*[32].

DEFINITION 52 (ORDERING COMPARISON OF NULL VALUES)
We define the binary comparison operator $\overset{\omega}{<}$ as follows. For the expression $a \overset{\omega}{<} b$:

- if neither a nor b are `Null` then the operator returns the same (two-valued) truth value as $a < b$;
- if a is `Null` and b is not, the result is *true;*
- otherwise the result is *false.*

DEFINITION 53 (ORDER PROPERTY)
An *order property,* which we abbreviate OP, is an ordered list of n attributes, written $\text{OP}(a_1, a_2, \cdots, a_n)$, taken from the set of attributes $A \subseteq \alpha(I(R))$, where $I(R)$ can denote an instance of a base or derived relation. A sequence of m tuples $r^* = r_1, r_2, \cdots, r_m$ where $r^* \equiv I(R)$ satisfies $\text{OP}(a_1, \cdots, a_n)$ if for all $r_i, r_j \in r^*$ such that $i < j$, either

1. $r_i[a_k] \overset{\omega}{=} r_j[a_k]$ for each $k \mid 1 \leq k \leq n$, or
2. there exists some k, where $0 \leq k < n$, such that $r_i[a_1 \cdots a_k] \overset{\omega}{=} r_j[a_1 \cdots a_k]$ and $r_i[a_{k+1}] \overset{\omega}{<} r_j[a_{k+1}]$.

---

32 In ANSI SQL the collating sequence of null values is implementation-defined. Sybase SQL Anywhere and Adaptive Server Enterprise follow the above convention (less than). IBM s DB2 2.1 and ORACLE Versions 6 and 7 implement the opposite: null values are defined as *greater than* every other value in each data type's domain.

Ordinarily $n \geq 1$; if $n = 0$ then any sequence of tuples trivially satisfies the order property. Hence an order property[33] is simply a dependency defined over a tuple sequence, precisely the meaning of a 'lexicographic index' described by Abiteboul and Ginsburg [3].

It is clear from this definition that if a tuple sequence satisfies $\text{OP}(X, Y, Z)$ then it also satisfies $\text{OP}(X, Y)$ and $\text{OP}(X)$, thus forming a partial order [233]. The *coverage* of this partial order is precisely what Simmen et al. mean by 'covering' two or more order properties—that is, when a particular order property 'covers' two or more interesting orders [261].

Our definition of order property is a generalization of Abiteboul and Ginsburg's *lexicographic indexes* [3]. Their formalism only considered total orders; each index was a unique index, and, consequently, also defined a candidate (possibly composite) key for its base relation. They also only considered key dependencies in their development of axioms for order properties. While key dependencies are important, we also consider derived functional dependencies to develop similar axioms for our definition of order properties.

LEMMA 36 (AUGMENTATION OF ORDER PROPERTIES)
Suppose a sequence of tuples r^* satisfies $\text{OP}(x_1, x_2, \cdots, x_k, x_{k+1}, \cdots, x_n)$. If the functional dependency $\{x_1, \ldots, x_k\} \longrightarrow A$ holds for some subset of attributes $A \subseteq \alpha(R)$ then r^* also satisfies $\text{OP}(x_1, x_2, \cdots, x_k, A, x_{k+1}, \cdots, x_n)$. We say that the order property is *augmented* by attribute A.

PROOF. We break x_1 through x_n into two subsets $X \equiv \{x_1, x_2, \cdots, x_k\}$ and $Y \equiv \{x_{k+1}, x_{k+2}, \cdots, x_n\}$. Therefore r^* satisfies $\text{OP}(X, Y)$. Consider any two tuples $r_i, r_j \in r^* \mid i < j$. If $r_i[X] \overset{\omega}{<} r_j[X]$ then $r_i[XA] \overset{\omega}{<} r_j[XA]$, so the two tuples are still in the correct sequence. Otherwise, if r^* satisfies $\text{OP}(X, Y)$ then $r_i[X] \overset{\omega}{=} r_j[X]$. However, if this is true then $r_i[XA] \overset{\omega}{=} r_j[XA]$ if the functional dependency $X \longrightarrow A$ holds, so r^* satisfies both $\text{OP}(X, Y)$ and $\text{OP}(X, A, Y)$. □

COROLLARY 5 (AUGMENTATION POSITION)
Suppose the tuple sequence r^* satisfies the order property $\text{OP}(X)$ where X is the ordered list $x_1, x_2, \cdots, x_k, x_{k+1}, \cdots, x_n$ and the functional dependency $X' \longrightarrow A$ holds in R, where X' denotes some subset of X with highest subscript k. Then r^* also satisfies any order property consisting of the concatenation of

1. the subsequence $X'' = x_1, x_2, \cdots, x_k$, and

---

33 For simplicity and without loss of generality we have only considered the case of ascending order properties.

2. any subsequence consisting of the set $Y = \{x_{k+1}, x_{k+2}, \cdots, x_n\} \cup A$ such that the relative position of each $x_i \in Y$ is preserved—that is, A can appear anywhere within the subsequence $x_{k+1}, x_{k+2}, \cdots, x_n$.

PROOF. If the dependency $X' \longrightarrow A$ holds, by Armstrong's axioms we can trivially add any attribute to its determinant; hence $\{X' \cup x_{k+1}\} \longrightarrow A$ also holds. Therefore we can add attribute A at any position in the order property after its determinant. Similarly, if r^* satisfies OP(X, Y) then r^* also satisfies OP(A, X, Y) if and only if the 'empty-headed' [70] functional dependency $\{\} \longrightarrow A$ holds. In this case A can be added to the order property at any position in the list. □

COROLLARY 6 (REDUCED ORDER PROPERTIES)
Let r^* denote a tuple sequence on a single table R. Then if r^* satisfies OP(X, A, Y) then r^* also satisfies OP(X, Y) if the functional dependency $X \longrightarrow A$ holds in R.

COROLLARY 7 (DUPLICATE ATTRIBUTES IN AN ORDER PROPERTY)
Let r^* denote a tuple sequence on a single table R. Then if r^* satisfies OP(X, A, Y, Z) then r^* also satisfies OP(X, A, Y, A, Z), since R satisfies the trivial dependency $A \longrightarrow A$.

Finally, if it can be inferred that, for each tuple in the result, two attribute values are always equivalent—that is, $X = Y$ so that we have $X \longrightarrow Y \wedge X \longrightarrow Y$ (see Section 3.2.4)—then we can perform attribute substitution within an order property.

LEMMA 37 (ATTRIBUTE SUBSTITUTION)
Let r^* represent a tuple sequence over a single table R. If r^* satisfies OP(W, X, Z) for every valid instance of R, then r^* also satisfies OP(W, Y, Z) if and only if for any two tuples $r, r' \in r^*$ we have $(r[X] \stackrel{\omega}{=} r[Y]) \wedge (r'[Y] \stackrel{\omega}{=} r'[X])$.

PROOF (SUFFICIENCY). If X and Y are $\stackrel{\omega}{=}$-equivalent for each tuple in r^* then we can substitute each X-value with each Y-value, and vice versa. Therefore r^* trivially satisfies OP(W, Y, Z) by substituting each X-value with its matching Y-value from the same tuple. □

PROOF (NECESSITY). Assume that for every valid instance of the database r^* satisfies both OP(W, X, Z) and OP(W, Y, Z) but the attributes X and Y are not equivalent. We must show that we can construct a valid instance of R so that r^* satisfies OP(W, X, Z) but not OP(W, Y, Z).

Consider an instance $I(R)$ of R consisting of two valid tuples r_0, r_0' such that $r_0[X] \stackrel{\omega}{=} r_0[Y]$, $r_0[X] \stackrel{\omega}{=} r_0'[X]$ but $r_0'[X] \stackrel{\omega}{\neq} r_0'[Y]$. Let each attribute value W in each tuple be a constant; thus the tuple sequence $r^* \equiv (r_0, r_0')$ satisfies OP(W, X). We can, however, select any value of Y for r_0' as long as $r_0'[X] \stackrel{\omega}{\neq} r_0'[Y]$. Let $r_0'[Y] \stackrel{\omega}{<} r_0'[X]$. Then r_0 and r_0'

constitute a valid instance of R, but r^* does not satisfy OP(W, Y), which it must satisfy to satisfy OP(W, Y, Z). Hence we conclude that the equivalence of X and Y is both necessary and sufficient. □

5.2.1 Axioms

In summarizing the proofs above, we have the following axioms to use in reasoning about the interaction of order properties and functional dependencies:

$$\text{OP}(X, Y) \Longrightarrow \text{OP}(X, A, Y) \quad \text{if } X \longrightarrow A \text{ (augmentation)}, \tag{5.1}$$

$$\text{OP}(X, A, Y) \Longrightarrow \text{OP}(X, Y) \quad \text{if } X \longrightarrow A \text{ (reduction)}, \tag{5.2}$$

$$\text{OP}(W, X, Z) \Longrightarrow \text{OP}(W, Y, Z) \quad \text{if } X = Y \text{ (substitution)}. \tag{5.3}$$

Abiteboul and Ginsburg [3] define yet another axiom that shows how a combination of order properties can be satisfied by the same tuple sequence. This axiom, along with axioms 5.1 and 5.2, form a sound and complete basis for inferencing with (unique) lexicographic indexes. While interesting from a theoretical standpoint, such a result does not assist in the problem of order optimization, since we cannot guarantee the satisfaction of two arbitrary order properties (other than satisfying prefixes of an order property) without a formal specification of the order dependencies that exist in the database [103]. Of much greater interest is how order properties hold in the context of derived relations.

5.3 Implementing order optimization

As described in Chapter 3, an execution strategy for an SQL query is made up of various relational algebra operators that form a tree. Each 'node' in the tree represents an algebraic operator that takes as input one or two tuple sequences and outputs another sequence, which either constitutes the query's final result or an intermediate result used as an input sequence for another operator placed higher in the tree. So it is important to determine how order properties are propagated through algebraic operator (sub)trees so as to determine the order property of each intermediate result in the strategy. Once the order property of a tuple sequence generated by an operator subtree is known, a separate problem is to determine in what ways can this ordering be exploited during query processing.

DEFINITION 54 (INTERESTING ORDER)
Selinger et al. [247] coined the phrase *interesting order* to denote a desired order property that could lead to a more efficient access plan. Hence an interesting order is merely

a *specification* for a desired order property, and can be defined in the same way. An interesting order, abbreviated IO, over an instance $I(R)$ of relation R is an ordered list of n attributes, written $\text{IO}(a_1, a_2, \cdots, a_n)$, taken from the set of attributes $A \subseteq \alpha(R)$. Ordinarily $n \geq 1$; if $n = 0$ then there is no sort requirement to be satisfied.

Lemmas 36 and 37 provide the basic axioms to manipulate order properties so that one can determine if the 'interesting order' desired is satisfied by a tuple sequence.

A critical aspect of order property analysis is the reduction of an order property into its *canonical form* [261]. Reducing order properties and interesting orders serves two purposes: it expands the space of the number of other order properties that can cover it, and if a sort is required the reduced version of an interesting order gives the minimal set of sorting columns. Lemma 36 formally proves the basis for the algorithms 'reduce order', 'test order', and 'cover order' in reference [261].

In Chapter 3 we described a data structure (an FD-graph) and an algorithm to keep track of derived functional dependencies and attribute equivalences that propagate through the query's algebraic expression tree. With this information, we can apply the axioms previously described to manipulate order properties to determine the coverage of any specific order property. To exploit the various possibilities of order optimization, a query optimizer must keep track of the following for each tuple sequence:

1. *the order property satisfied by the sequence in its canonical (reduced) form.* An order property's canonical form is an order property stripped of any redundancies, either duplicate attributes or attributes whose order is implied by (derived) functional dependencies. This is the major distinction between Abiteboul and Ginsburg's work [3] and that of Simmen et al. from IBM Almaden [261]: Abiteboul and Ginsberg consider only functional dependencies that hold for every database instance, whereas Simmen et al. consider not only functional dependencies implied by the database schema but derived dependencies as well.

2. *a set of (derived) functional dependencies.* By generating and maintaining a set of functional dependencies that hold for a given relational operator, it is possible to augment a canonical order property using these dependencies in order to determine if an interesting order is satisfied.

3. *a set of attribute equivalences.* As with functional dependencies, the optimizer can use the equivalences of attributes to augment an order property or substitute one attribute for another within an order property.

5.4 Order properties and relational algebra

In Section 5.1 we briefly described the situations in which we can exploit the order nature of sequences to speed query processing. Sort avoidance is an obvious application of order properties; if a query's `Order by` clause (an 'interesting order') coincides with the order property satisfied by the tuple sequence constituting the result, then a sort of the final result is unnecessary. In query optimization we are interested in how properties of the database, and even properties of the given database instance, can be exploited to speed query execution. In particular, we can use query predicates to determine what functional dependencies hold in any intermediate result, and then attempt to match the order property of the result with the interesting order required by the query itself, or subsequent physical algebra operators higher in the expression tree.

A note of caution: while we describe the affects of relational algebra operators on order properties, it should be obvious that not all *implementations* of these operators propagate order properties in the same way. For example, most implementations of both hash-join [41] and block nested-loop join [156] do not preserve the sequence of their inputs. Consequently, we assume in what follows that the implementation of each relational algebra operator is 'order preserving'. Where applicable, we give examples of order-preserving implementations of these algorithms to illustrate that orderings can indeed be preserved. We note, however, that there are substantially more sophisticated implementations of these algebraic operators in commercial database systems that are beyond the scope of this thesis. We refer the interested reader to two surveys on the subject [107, 203].

5.4.1 Projection

From the definition of an order property (Definition 53) we know that if a given tuple sequence satisfies a composite order property, say $\text{OP}(X, Y, Z)$ then it trivially satisfies any prefix of that property, say $\text{OP}(X, Y)$.

THEOREM 15 (PROJECTION)
Suppose the order property $\text{OP}(x_1, \cdots, x_n)$, which has been reduced to its canonical form, is satisfied by an arbitrary tuple sequence. Then after a projection (with or without duplicate elimination) that preserves $x_1, x_2, \cdots, x_{j-1}$ but eliminates x_j the prefix $\text{OP}(x_1, \cdots, x_{j-1})$ holds and cannot be extended to include any of $x_j, x_{j+1}, \cdots, x_n$. If j is 1 then the order property becomes empty $\text{OP}(\emptyset)$, that is, the tuple sequence cannot be guaranteed to satisfy any order property.

PROOF. Obvious. □

Projection is an excellent example of the need to first reduce an order property to its canonical form. For example, projecting away a given column from the result may in fact have no effect on the sequence's order property if that column is functionally determined by higher-order attributes. Projection may also permit the augmentation of the reduced order property so that a larger (longer) order property can cover it [261].

As an aside, some relational systems such as Sybase SQL Anywhere extend the SQL2 standard and permit an `Order by` clause to reference columns or derived columns that do not appear in the query's `Select` list. The way in which this can be handled (with respect to projection) is to interpret the query's `Select` list as the union of those attributes that actually appear in the `Select` list with those attributes in the `Order by` clause. The above lemma can still be used to determine the order property of the result of a projection in this case.

See Section 5.4.6 for a discussion on order properties and duplicate elimination.

5.4.2 Restriction

The algorithms used by Simmen, Shekita, and Malkemus [261] to determine the order properties of derived relations use the presence of equivalence conditions in the query's `Where` clause to infer functional dependencies for that database instance that can be used to reduce an OP to its 'canonical' (reduced) form. While equivalence operators do provide opportunities for reduction, we can also utilize table or column constraints, candidate or primary keys, and any other query predicates to infer which OP holds.

THEOREM 16 (RESTRICTION PREDICATES AND ORDER PROPERTIES)
Consider a query involving only restriction on a single table R consisting of at least 3 attributes X, A, Y. The restriction predicate C_R may contain expressions that include host variables; we denote this set of input parameters by h. Thus we identify the test of a restriction predicate, which includes host variables, on tuple r of R with the notation $C_R(r, h)$. Then if the expression

$$Q = \sigma[C_R](R)$$

represents a sequence of tuples that satisfies $\mathrm{OP}(X, Y)$ then Q also satisfies $\mathrm{OP}(X, A, Y)$ if the following condition holds:

$$\begin{aligned}
&\forall\, r, r' \in \mathrm{Domain}(R); \forall\, h \in \mathrm{Domain}(H) : \qquad (5.4)\\
&\quad \{ [\, \lceil T_R(r) \rceil \wedge \lceil T_R(r') \rceil \wedge \\
&\quad (\textit{for each } K_i(R) : (r[K_i(R)] \stackrel{\omega}{=} r'[K_i(R)]) \Longrightarrow r[\alpha(R)] \stackrel{\omega}{=} r'[\alpha(R)]) \wedge \\
&\quad (\textit{for each } U_i(R) : (\lfloor\, r[U_i(R)] = r'[U_i(R)] \,\rfloor) \Longrightarrow r[\alpha(R)] \stackrel{\omega}{=} r'[\alpha(R)]) \wedge
\end{aligned}$$

$$\lfloor C_R(r,h)\rfloor \wedge \lfloor C_R(r',h)\rfloor \wedge$$
$$(r[X] \stackrel{\omega}{=} r'[X])\,] \Longrightarrow (r[A] \stackrel{\omega}{=} r'[A])\}$$

PROOF. Follows directly from Lemma 36. □

The satisfaction of the condition in Theorem 16 permits an optimizer some 'degrees of freedom' in trying to determine if an order property satisfies an interesting order. An independent but related problem is the analysis of the predicates in a query's **Where** clause to determine if indexed retrieval of a base or derived table is beneficial. Two such techniques are index union and intersection [207] that merge sets of row identifiers (RIDs) from two or more indexes. The result of the merge is a sorted list of RIDs used to probe the underlying table; hence the tuple sequence fails to satisfy any order property that corresponds to any syntactic component of the original SQL statement. However, RIDs can act as surrogate data values (in fact, primary keys) in a variety of situations and many commercial system employ physical algebra operators that utilize sorted RIDs for efficient processing.

5.4.3 Inner join

Consider an inner join between two tables R and S. If we restrict the physical inner join operator to order-preserving implementations of either sort-merge or nested-loop join. then it is easy to see that if the tuple sequence r^* of R satisfies OP(X,Y,Z) then the derived table consisting of the inner join of R and S also satisfies OP(X,Y,Z) regardless of the characteristics of the tuple sequence s^* of S [261]. Of more interest is when we can augment the order property satisfied by R with that of S when joining R and S.

5.4.3.1 Nested-loop inner join

EXAMPLE 34
Consider the following SQL query:

```
Select D.Location, D.Name, E.EmpID, E.Surname, E.GivenName
From   Division D, Employee E
Where  D.Name = E.DivName
Order by D.Location, E.EmpID
```

that lists information about each employee in the firm in ascending sequence by employee ID within division location. The nested-loop strategy illustrated in Figure 5.2 will *not* return the correct order of results for every instance of the database. This is because the `Location` attribute is not unique. It is possible that a database instance will have two divisions with identical locations and with different (but overlapping) sets of employee identifiers—so the result of the join may not be in the desired ascending sequence. This

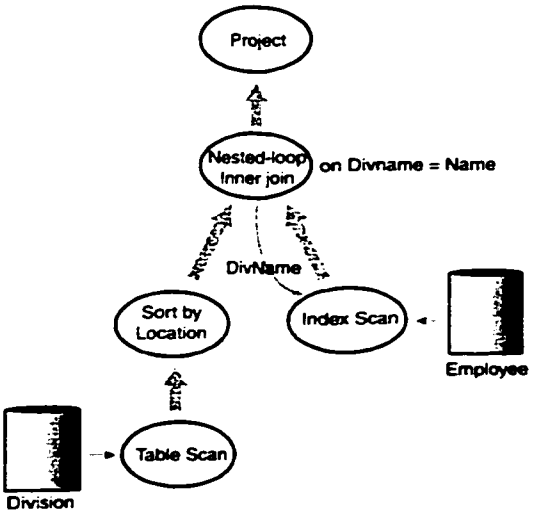


FIGURE 5.2: Erroneous nested-loop strategy for Example 34.

situation raises the following question: under what conditions can we guarantee that the result is properly ordered without sorting?

Algorithm 1 describes a straightforward implementation of nested-loop join that is order-preserving. It is easy to see that with this algorithm the output of the join satisfies the order property of the 'outer' table R, since each tuple of R is accessed in order only once. We wish to show that we can augment the order property of the result with attributes of S if certain conditions hold. Without loss of generality, assume that R represents the 'outer' table, and S the 'inner'.

THEOREM 17 (NESTED LOOP INNER JOIN)
Consider a query involving the nested-loop inner join of two tables R and S. Assume the nested loop join is implemented using simple tuple-iteration semantics (e.g. the join does not perform bulk operations on groups of tuples) and is order preserving on the tuple sequences r^* of R and s^* of S (see Algorithm 1). Each of the restriction predicates in the query C_R, C_S and $C_{R,S}$ may contain expressions that include host variables. If the tuple sequence r^* of R satisfies $\mathrm{OP}(X)$ and the tuple sequence s^* of S satisfies $\mathrm{OP}(Y)$, then the expression

$$Q = \sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S),$$

```
1630 Algorithm: Order-preserving nested-loop inner join.
1631 Inputs: join predicate C_{R,S}; tuple sequences r* and s*.
1632 Output: the ordered result σ[C_{R,S}](R × S).
1633 begin
1634   r ← r*_1;
1635   while r ≠ EOF do
1636     s ← first tuple of s* such that ⌊ C_{R,S} ⌋ is true;
1637     while s ≠ EOF and ⌊ C_{R,S} ⌋ is true do
1638       output the tuple ⟨r, s⟩ to the result;
1639       s ← next tuple after s of s*
1640       od;
1641     r ← next tuple after r of r*
1642     od
1643   end
```

ALGORITHM 1: Basic order-preserving nested-loop join algorithm.

which represents the sequence of tuples q^* returned by the nested-loop join of R and S, satisfies $\text{OP}(X, Y)$ if the following condition holds:

$$
\begin{aligned}
&\forall\, q, q' \in \text{Domain}(R \times S); \forall\, h \in \text{Domain}(H): \qquad (5.5)\\
&\quad \{ [\, \lceil T_R(q) \rceil \wedge \lceil T_R(q') \rceil \wedge \lceil T_S(q) \rceil \wedge \lceil T_S(q') \rceil \wedge\\
&\quad (\textit{for each } K_i(R) : (q[K_i(R)] \stackrel{\omega}{=} q'[K_i(R)]) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\quad (\textit{for each } U_i(R) : (\lfloor\, q[U_i(R)] = q'[U_i(R)] \,\rfloor) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\quad (\textit{for each } K_j(S) : (q[K_j(S)] \stackrel{\omega}{=} q'[K_j(S)]) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\quad (\textit{for each } U_j(S) : (\lfloor\, q[U_j(S)] = q'[K_j(S)] \,\rfloor) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\quad \lfloor C_R(q, h) \rfloor \wedge \lfloor C_R(q', h) \rfloor \wedge \lfloor C_S(q, h) \rfloor \wedge\\
&\quad \lfloor C_S(q', h) \rfloor \wedge \lfloor C_{R,S}(q, h) \rfloor \wedge \lfloor C_{R,S}(q', h) \rfloor\,] \Longrightarrow\\
&\qquad [\, (q[X] \stackrel{\omega}{=} q'[X]) \Longrightarrow\\
&\qquad\quad [\, (q[Y] \stackrel{\omega}{=} q'[Y]) \vee\\
&\qquad\quad (\forall\, r, r' \in \text{Domain}(R):\\
&\qquad\qquad q[X] \stackrel{\omega}{=} r[X] \wedge q'[X] \stackrel{\omega}{=} r'[X] \wedge r[X] \stackrel{\omega}{=} r'[X] \Longrightarrow\\
&\qquad\qquad\quad r[\text{RowID}(R)] = r'[\text{RowID}(R)]\,)\,]\,]\}
\end{aligned}
$$

PROOF. If the condition holds then for any two tuples r, r' in the result either the functional dependency $X \longrightarrow Y$ holds, or the (possibly composite) attribute values of $r[X]$ and $r'[X]$ are unique in R. If the former case, then we have a simple case of augmentation,

so the theorem holds by Lemma 36. We now consider the latter case: where X uniquely identifies the two tuples of R. R is specified as the outer relation, hence we are guaranteed that q^* satisfies OP(X). Since s^* satisfies OP(Y), then any tuples from S that join with a single tuple from R will satisfy OP(Y). Since the X attributes are unique, then for tuples $r_i, r_j \in r^* \mid i < j$ we have $r_i[X] \overset{\omega}{<} r_j[X]$. Therefore we conclude that q^* satisfies OP(X, Y). □

The condition in Theorem 17 must hold for each pair of tuples in the sequence r^*, which can be difficult to test on a tuple-by-tuple basis for any instance of the database. Since the conditions are quantified Boolean expressions, the test is equivalent to deciding if the expression is satisfiable, which in general is PSPACE-complete [102, pp. 171–2]. Instead, an easier test that constitutes a sufficient condition is to determine whether or not the set of attributes X form a candidate key of R, or if the functional dependency $X \longrightarrow Y$ holds in the result for any instance of the database. As with the algorithms in Chapter 4, assuming that we have an FD-graph G representing the constraints that hold in Q, testing the result of DEPENDENCY-CLOSURE(G, X, Γ) will provide the desired answer. Also note that the theorem holds for not only nested-loop equijoins, but for any arbitrary join condition on R and S.

LEMMA 38 (NESTED-LOOP INNER JOIN WITH COVERED JOIN ATTRIBUTES)
The condition in Theorem 17 is necessary if the attributes from R involved in the join condition are a subset of those attributes in X, that is $\alpha_R(C_{R,S}) \subseteq X$.

PROOF. Assume that q^* satisfies OP(X, Y) but the condition in Theorem 17 does not hold—that is, the attribute set X is neither a candidate key of R nor does the functional dependency $X \longrightarrow Y$ hold for any two tuples in Q. If so, then there must exist at least two tuples r_i, r_j in R such that $r_i[X] \overset{\omega}{=} r_j[X]$. Let the subset of the attributes of X that are referenced in the join condition be denoted as J; hence $J \subseteq X$. Then $r_i[J] \overset{\omega}{=} r_j[J]$. It follows that if tuple r_i from R is present in Q then tuple r_j will join with the same tuples of S and will also be in the result. Furthermore, since the condition in Theorem 17 does not hold, we know that for those tuples in Q that stem from tuples r_i and r_j in R the Y values are unequal; hence these two tuples of R each joined with at least two tuples of S, and there exists at least four tuples in the result that stem from r_i and r_j.

Consider now the (at least) four tuples in Q that stem from r_i and r_j. There must be an even number of tuples since r_i and r_j joined with the identical tuples of S. Assume that in r^* tuple r_i appears before r_j. Then regardless as to which tuples of S are joined, there is no way for the order property OP(X, Y) to be satisfied since the X values are equal, but the Y values are not; a contradiction. Hence we conclude that the condition in Theorem 17 is both necessary and sufficient if $\alpha_R(C_{R,S}) \subseteq X$. □

Algorithm: *Order-preserving sort-merge inner equijoin.*
Inputs: equijoin predicate $C_{R,S}$; tuple sequences r^* and s^*.
Output: the ordered result $\sigma[C_{R,S}](R \times S)$.
begin
  $r \leftarrow r_1^*$;
  $s \leftarrow s_1^*$;
  $g \leftarrow s$;
  while $r \neq$ EOF and $s \neq$ EOF **do**
    while $r \neq$ EOF and $r[\alpha_R(C_{R,S})] \overset{\omega}{<} g[\alpha_S(C_{R,S})]$ **do**
      $r \leftarrow$ next tuple after r of r^*
      od;
    while $g \neq$ EOF and $g[\alpha_S(C_{R,S})] \overset{\omega}{<} r[\alpha_R(C_{R,S})]$ **do**
      $g \leftarrow$ next tuple after g of s^*
      od;
    while $r \neq$ EOF and $\lfloor r[\alpha_R(C_{R,S})] = g[\alpha_S(C_{R,S})] \rfloor$ **do**
      $s \leftarrow g$;
      while $s \neq$ EOF and $s[\alpha_S(C_{R,S})] = r[\alpha_R(C_{R,S})]$ **do**
        output the tuple $\langle r, s \rangle$ to the result:
        $s \leftarrow$ next tuple after s of s^*
        od
      $r \leftarrow$ next tuple after r of r^*
      od
    $g \leftarrow s$
    od
  end

ALGORITHM 2: A straightforward implementation of sort-merge inner equijoin that preserves the ordering of both input sequences, taken from Ramakrishnan [232, pp. 292].

5.4.3.2 Sort-merge inner join

Sort-merge join relies on both inputs being sorted on the attribute(s) being joined (see Algorithm 2). Hence the attributes involved in the equijoin condition must constitute the prefix of the order properties of both inputs. However, each input sequence may satisfy a longer order property that covers the one necessary to perform the join.

THEOREM 18 (SORT-MERGE INNER EQUIJOIN)
Consider a query involving the sort-merge inner equijoin of two tables R and S. Assume the sort-merge inner join implementation is order preserving on both tuple sequences r^* of R and s^* of S (see Algorithm 2). The join predicate $C_{R,S}$ may contain only equality conditions; each of the other restriction predicates in the query (C_R and C_S) may contain

expressions that include host variables. If the tuple sequence r^* of R satisfies OP(J_R, X) and the tuple sequence s^* of S satisfies OP(J_S, Y) such that $C_{R,S}$ constitutes equality conditions between corresponding attributes in J_R and J_S, then the expression

$$Q = \sigma[C_R \wedge C_S \wedge C_{R,S}](R \times S)$$

which represents the sequence of tuples q^* returned by the sort-merge inner join of R and S, satisfies OP(J_R, X, Y) and, via axiom 5.3, also OP(J_S, X, Y), if and only if the following condition holds:

$$\begin{aligned}
&\forall\, q, q' \in \text{Domain}(R \times S); \forall\, h \in \text{Domain}(H) : \qquad (5.6)\\
&\quad \{ [\lceil T_R(q) \rceil \wedge \lceil T_R(q') \rceil \wedge \lceil T_S(q) \rceil \wedge \lceil T_S(q') \rceil \wedge\\
&\quad (\textit{for each } K_i(R) : (q[K_i(R)] \stackrel{\omega}{=} q'[K_i(R)]) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\quad (\textit{for each } U_i(R) : (\lfloor q[U_i(R)] = q'[U_i(R)] \rfloor) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\quad (\textit{for each } K_j(S) : (q[K_j(S)] \stackrel{\omega}{=} q'[K_j(S)]) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\quad (\textit{for each } U_j(S) : (\lfloor q[U_j(S)] = q'[K_j(S)] \rfloor) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\quad \lfloor C_R(q, h) \rfloor \wedge \lfloor C_R(q', h) \rfloor \wedge \lfloor C_S(q, h) \rfloor \wedge\\
&\quad \lfloor C_S(q', h) \rfloor \wedge \lfloor C_{R,S}(q, h) \rfloor \wedge \lfloor C_{R,S}(q', h) \rfloor] \Longrightarrow\\
&\qquad [(q[J_R, X] \stackrel{\omega}{=} q'[J_R, X]) \Longrightarrow\\
&\qquad\quad [(q[Y] \stackrel{\omega}{=} q'[Y]) \vee\\
&\qquad\quad (\forall\, r, r' \in \text{Domain}(R) :\\
&\qquad\qquad q[J_R] \stackrel{\omega}{=} r[J_R] \wedge q'[J_R] \stackrel{\omega}{=} r'[J_R] \wedge r[J_R] \stackrel{\omega}{=} r'[J_R] \Longrightarrow\\
&\qquad\qquad\quad r[\text{RowID}(R)] = r'[\text{RowID}(R)]\,)\,]\,]\,\}
\end{aligned}$$

PROOF. The proof for sort-merge join is similar to the proofs of Theorem 17 and Corollary 38. If the condition does not hold, then it is possible that the existence of two or more tuples in R with identical values for the set of join attributes J_R will cause the algorithm to re-process the same tuples of S, and hence the ordered attributes of Y will repeat, breaking the sequence. □

5.4.3.3 Applications

If the conditions in Theorems 17 and 18 hold then an optimizer can use this information to choose an access plan for a class of SPJ queries that does not require an additional sort of the final result.

EXAMPLE 35 (SORT AVOIDANCE AND INNER JOINS)
The SQL query:

```
Select Q.PartID, Q.UnitPrice, Q.MinOrder, S.VendorID, S.SupplyCode
From   Quote Q, Supply S
Where  Q.PartID = S.PartID and Q.VendorID = S.VendorID
       and S.Rating = 'A'
Order by Q.PartID, S.VendorID, S.SupplyCode
```

that lists part quotations for vendors with an 'A' supply rating for that part. The query's predicates enable a query optimizer to substitute S.VendorID with Q.VendorID and S.PartID with Q.PartID. That, combined with the fact that the functional dependency {S.VendorID, S.PartID} $\longrightarrow$ S.SupplyCode holds means that the optimizer can choose an access plan that does not require an explicit sort of the entire result. A possible strategy is to perform an index scan on the QUOTE table by VendorID within PartID, and subsequently perform a nested-loop index join or sort-merge join with the SUPPLY table on those two attributes. It is unnecessary to consider the SupplyCode attribute since it is functionally determined by the key of SUPPLY.

Even though such a strategy may involve an index scan of the outer relation, it may still be cheaper than alternative strategies if the number of tuples fetched by the application is small. It also has the advantage of accessing the SUPPLY table in primary key sequence, which should reduce the amount of I/O to retrieve tuples from that table.

Another application of this analysis is in the optimization of disjunctive predicates in queries which, in fact, do not necessarily contain a join at all.

EXAMPLE 36 (ORDERED INDEX SCAN)
Consider the following SQL query:

```
Select *
From   Employee E
Where  E.EmpID in ( 100, 115, ..., 1230 )
Order by E.EmpID
```

that lists employee information for each employee in the list. This type of In predicate is ubiquitous in production applications. Moreover, in our experience it is not uncommon to see SQL statements containing In lists of hundreds, even thousands, of values, the result of the widespread use of *ad-hoc* query tools that generate SQL statements.

One possible way of rewriting this query is a join of EMPLOYEE with the single-column relation made up of the distinct elements of the list. If we denote this relation as TEMP with column EmpID then the rewritten query is

```
Select E.*
From   Employee E, Temp T
Where  E.EmpID = T.EmpID
Order by E.EmpID.
```

Consider a nested-loop join strategy for this query. If EMPLOYEE is the outer table, then we can only satisfy the query's **Order by** clause by either (1) sorting the entire result or (2) performing an index scan on EMPLOYEE by **EmpID**. However, if the **EmpID** attribute in the EMPLOYEE table is indexed, then it may be preferable to do n probes into EMPLOYEE to compute the result set, where n is the number of elements in the list. Note however that duplicate elements in the **In** list must be removed to preserve the query's semantics. If we chose to eliminate those duplicate elements through sorting, then a nested-loop join with TEMP as the outer table may be a cost-effective strategy, and furthermore satisfies the condition in Theorem 17, avoiding a sort of the final result.

5.4.4 Left outer join

In terms of order properties, order-preserving implementations of outer joins work in much the same way as inner joins. We first consider nested-loop implementations of left-outer joins. Note that the cases for right-outer joins are equivalent since the two operations are symmetric.

5.4.4.1 Nested-loop left outer join

Consider the left outer join of table R and S, i.e. $R \xrightarrow{p} S$ on some **On** condition p, computed by the nested-loop left outer join algorithm in Algorithm 3. If the tuple sequence r^* of R satisfies some order property $\text{OP}(X, Y, Z)$ then it is easy to see that the derived table consisting of the left outer join of R and S also satisfies $\text{OP}(X, Y, Z)$ since each tuple of R (the preserved side of the outer join) is retrieved only once in sequence. As with inner join, we can augment the order property satisfied by r^* with the order property satisfied by s^* if certain conditions hold.

THEOREM 19 (NESTED LOOP LEFT OUTER JOIN)
Consider a query involving the nested-loop outer join of two tables R and S. Assume the nested loop outer join is implemented using simple tuple-iteration semantics (e.g. the join does not perform bulk operations on groups of tuples) and is order preserving on both tuple sequences r^* of R and s^* of S. Each of the restriction predicates in the query C_R, C_S

and $C_{R,S}$ may contain expressions that include host variables. If the tuple sequence r^* of R satisfies OP(X) and the tuple sequence s^* of S satisfies OP(Y), then the expression[34]

$Q = \sigma[C_R](R \xrightarrow{p} \sigma[C_S](S))$ where $p = C_{R,S}$,

which represents the sequence of tuples q^* returned by the nested-loop outer join of R and S, satisfies OP(X, Y) if the following condition holds:

$$
\begin{aligned}
&\forall\, h \in \text{Domain}(H) : \qquad (5.8)\\
&\quad \forall\, q, q' \in \text{Domain}(R \xrightarrow{p} S) :\\
&\qquad \{ [\, \lceil T_R(q) \rceil \wedge \lceil T_R(q') \rceil \wedge \lceil T_S(q) \rceil \wedge \lceil T_S(q') \rceil \wedge\\
&\qquad (\textit{for each } K_i(R) : (q[K_i(R)] \stackrel{\omega}{=} q'[K_i(R)]) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\qquad (\textit{for each } U_i(R) : (\lfloor\, q[U_i(R)] = q'[U_i(R)] \,\rfloor) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\qquad (\textit{for each } K_j(S) : (q[K_j(S)] \stackrel{\omega}{=} q'[K_j(S)]) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\qquad (\textit{for each } U_j(S) : (\lfloor\, q[U_j(S)] = q'[K_j(S)] \,\rfloor) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\qquad \lfloor C_R(q,h) \rfloor \wedge \lfloor C_R(q',h) \rfloor \wedge\\
&\qquad (\lceil q[\alpha(S)] \rceil \vee \lfloor C_S(q,h) \rfloor) \wedge (\lceil q'[\alpha(S)] \rceil \vee \lfloor C_S(q',h) \rfloor) \wedge\\
&\qquad (\lceil q[\alpha(S)] \rceil \vee \lfloor C_{R,S}(q,h) \rfloor) \wedge (\lceil q'[\alpha(S)] \rceil \vee \lfloor C_{R,S}(q',h) \rfloor)\,] \Longrightarrow\\
&\qquad\quad [\, (q[X] \stackrel{\omega}{=} q'[X]) \Longrightarrow\\
&\qquad\qquad [\, (q[Y] \stackrel{\omega}{=} q'[Y]) \vee\\
&\qquad\qquad (\forall\, r, r' \in \text{Domain}(R) :\\
&\qquad\qquad\quad q[X] \stackrel{\omega}{=} r[X] \wedge q'[X] \stackrel{\omega}{=} r'[X] \wedge r[X] \stackrel{\omega}{=} r'[X]\\
&\qquad\qquad\qquad \Longrightarrow r[\text{RowID}(R)] = r'[\text{RowID}(R)]\,)\,]\,]\,\}
\end{aligned}
$$

PROOF. If the condition holds then for any two tuples r, r' in the result either the functional dependency $X \longrightarrow Y$ holds, or the (possibly composite) attribute values of $r[X]$ and $r'[X]$ are unique in R. If the former case, then we have a simple case of augmentation, so the theorem holds by Lemma 36. We now consider the latter case: where X uniquely identifies the two tuples of R. R is specified as the outer relation, hence we are guaranteed that q^* satisfies OP(X). Since s^* satisfies OP(Y), then any tuples from S that join

---

34 To simplify the theorem, we assume that any portion of the **On** condition that solely refers to $\alpha(S)$ is pushed down the physical algebra expression tree by the identity [98, pp. 50]:

$$R_1 \xrightarrow{p_1 \wedge p_2} R_2 \equiv R_1 \xrightarrow{p_1} \sigma[p_2](R_2) \qquad \text{if } \alpha(p_2) \subseteq \alpha(S). \qquad (5.7)$$

This ensures that any two tuples from the preserved side of the join with the same values of the join attribute(s) will join (or not) with the same set of tuples of S.

```
1669 Algorithm: Order-preserving nested-loop left-outer join.
1670 Inputs: outer join predicate C_{R,S}; tuple sequences r* and s*.
1671 Output: the ordered result R --p--> S where p = C_{R,S}.
1672 begin
1673   r ← r*_1;
1674   while r ≠ EOF do
1675     s ← first tuple of s* such that ⌊ C_{R,S} ⌋ is true;
1676     if s = EOF then
1677       output the tuple ⟨r, NULL⟩ to the result
1678     else
1679       while s ≠ EOF and ⌊ C_{R,S} ⌋ is true do
1680         output the tuple ⟨r, s⟩ to the result;
1681         s ← next tuple after s of s*
1682         od
1683       fi;
1684     r ← next tuple after r of r*
1685     od
1686   end
```

ALGORITHM 3: Basic order-preserving nested-loop left outer join.

with a single tuple from R will satisfy OP(Y). Since the X attributes are unique, then for tuples $r_i, r_j \in r^* \mid i < j$ we have $r_i[X] \overset{\omega}{<} r_j[X]$. Therefore we conclude that q^* satisfies OP(X, Y). □

In practice, the dependency $X \longrightarrow Y$ will only hold when the left outer join's `On` condition contains the single equality condition $X = Y$, because the presence of any other condition can affect whether or not a specific tuple from R 'matches' a tuple from S, and if not, the output contains that tuple of R combined with an all-`Null` row, violating the dependency (see Section 3.2.9). Hence for conjunctive `On` conditions the condition in Theorem 19 will hold only if the attributes in X constitute a candidate key of R.

5.4.4.2 Sort-merge left outer join

THEOREM 20 (SORT-MERGE LEFT OUTER JOIN)
Consider a query involving the sort-merge outer join of two tables R and S. Assume the sort-merge outer join implementation is order preserving on both tuple sequences r^* of R and s^* of S. Each of the restriction predicates in the query C_R, C_S and $C_{R,S}$ may contain expressions that include host variables. If the tuple sequence r^* of R satisfies OP(J_R, X)

and the tuple sequence s^* of S satisfies $\text{OP}(J_S, Y)$ such that $C_{R,S}$ constitutes equality conditions between corresponding attributes in J_R and J_S, then the expression

$$Q = \sigma[C_R](R \xrightarrow{p} \sigma[C_S](S)) \text{ where } p = C_{R,S},$$

which represents the sequence of tuples q^* returned by the sort-merge outer join of R and S, satisfies $\text{OP}(J_R, X, Y)$ if and only if the following condition holds:

$$\begin{aligned}
&\forall\, h \in \text{Domain}(H) : \qquad (5.9)\\
&\quad \forall\, q, q' \in \text{Domain}(R \xrightarrow{p} S) :\\
&\qquad \{ [\, \lceil T_R(q) \rceil \wedge \lceil T_R(q') \rceil \wedge \lceil T_S(q) \rceil \wedge \lceil T_S(q') \rceil \wedge\\
&\qquad (\textit{for each } K_i(R) : (q[K_i(R)] \stackrel{\omega}{=} q'[K_i(R)]) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\qquad (\textit{for each } U_i(R) : (\lfloor q[U_i(R)] = q'[U_i(R)] \rfloor) \Longrightarrow q[\alpha(R)] \stackrel{\omega}{=} q'[\alpha(R)]) \wedge\\
&\qquad (\textit{for each } K_j(S) : (q[K_j(S)] \stackrel{\omega}{=} q'[K_j(S)]) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\qquad (\textit{for each } U_j(S) : (\lfloor q[U_j(S)] = q'[K_j(S)] \rfloor) \Longrightarrow q[\alpha(S)] \stackrel{\omega}{=} q'[\alpha(S)]) \wedge\\
&\qquad \lfloor C_R(q, h) \rfloor \wedge \lfloor C_R(q', h) \rfloor \wedge\\
&\qquad (\lceil q[\alpha(S)] \rceil \vee \lfloor C_S(q, h) \rfloor) \wedge (\lceil q'[\alpha(S)] \rceil \vee \lfloor C_S(q', h) \rfloor) \wedge\\
&\qquad (\lceil q[\alpha(S)] \rceil \vee \lfloor C_{R,S}(q, h) \rfloor) \wedge (\lceil q'[\alpha(S)] \rceil \vee \lfloor C_{R,S}(q', h) \rfloor) \,] \Longrightarrow\\
&\qquad\quad [\, (q[J_R, X] \stackrel{\omega}{=} q'[J_R, X]) \Longrightarrow\\
&\qquad\qquad [\, (q[Y] \stackrel{\omega}{=} q'[Y]) \vee\\
&\qquad\qquad (\forall\, r, r' \in \text{Domain}(R) :\\
&\qquad\qquad\quad q[J_R] \stackrel{\omega}{=} r[J_R] \wedge q'[J_R] \stackrel{\omega}{=} r'[J_R] \wedge r[J_R] \stackrel{\omega}{=} r'[J_R] \Longrightarrow\\
&\qquad\qquad\qquad r[\text{RowID}(R)] = r'[\text{RowID}(R)] \,) \,] \,] \,\}
\end{aligned}$$

PROOF. Omitted. □

5.4.5 Full outer join

Straightforward implementations of nested loop join or its variants, such as block nested loop join, cannot realistically be used for full outer joins, for example $R \xleftrightarrow{p} S$. This is because of the need to scan both inputs twice: once to produce the left outer join result $R \xrightarrow{p} S$ and the other to produce those tuples in S that do not join with any tuple in R. Consequently hash or sort-merge joins are used to compute full outer joins. As mentioned previously, the former algorithm, by design, is not order-preserving on either input. Sort-merge implementations of full outer joins are not usually order-preserving either, however, because **Null** values can be introduced at any point during the merge of the two sorted input sequences.

```
1687  Algorithm: Order-preserving sort-merge left-outer equijoin.
1688  Inputs: predicate C_{R.S}; tuple sequences r* and s*.
1689  Output: the ordered result (R --p--> S) where p = C_{R.S}.
1690  begin
1691    r ← r*_1;
1692    s ← s*_1;
1693    g ← s;
1694    while r ≠ EOF do
1695      if g = EOF then
1696        output the tuple ⟨r, NULL⟩ to the result
1697      else
1698        while r ≠ EOF and r[α_R(C_{R.S})] ≤̃ g[α_S(C_{R.S})] do
1699          output the tuple ⟨r, NULL⟩ to the result;
1700          r ← next tuple after r of r*
1701          od;
1702        while g ≠ EOF and g[α_S(C_{R.S})] ≤̃ r[α_R(C_{R.S})] do
1703          g ← next tuple after g of s*
1704          od;
1705        while r ≠ EOF and ⌊r[α_R(C_{R.S})] = g[α_S(C_{R.S})]⌋ do
1706          s ← g;
1707          while s ≠ EOF and ⌊s[α_S(C_{R.S})] = r[α_R(C_{R.S})]⌋ do
1708            output the tuple ⟨r, s⟩ to the result;
1709            s ← next tuple after s of s*
1710            od
1711          r ← next tuple after r of r*
1712          od
1713        g ← s
1714        fi
1715      od
1716  end
```

ALGORITHM 4: An implementation of sort-merge left-outer equijoin that preserves the ordering of both input sequences.

5.4.6 Partition and distinct projection

It has long been realized that `Group by` processing can be simplified by processing an ordered input stream [92, 107]. An implementation of the partition operator does not need to materialize its result if its input is a sequence of tuples ordered on the grouping attributes; if the query does not contain any `Distinct` aggregate functions, any aggregation can be done trivially in memory. Consequently the database can return the first row

in the result set almost immediately to the application program.

As pointed out by both Simmen et al. [261] and Furtado and Kerschberg [92] the 'interesting order' for both **Group by** and **Select Distinct** queries is any permutation of the grouping columns (for grouped queries) and any permutation of the columns in the select list (for distinct queries). After the input is partitioned into groups, the candidate key of the derived relation becomes the columns in the group-by list (see Section 3.2.8.1). For example, the query

```
Select D.Name, Avg(E.Salary)
From   Division D, Employee E
Where  D.Name = E.DivName
Group by D.Name
```

can be processed without materialization if there exists an index on **Division.Name**. With multiple grouping attributes, any permutation of the grouping columns in an index will satisfy the query's interesting order. In addition, we can exploit both ascending and descending sequences of attributes since only their clustering, not rank order, is important [261]. After simplification of an interesting order through the analysis of functional dependencies, for n grouping columns there are $n! \times 2^n$ possible interesting orders that are applicable to computing the derived table—$n!$ permutations of the attributes, with 2^n combinations of ascending or descending order for each attribute.

Suppose we modify the above query slightly to eliminate duplicate salaries from the computation:

```
Select D.Name, Avg(Distinct E.Salary)
From   Division D, Employee E
Where  D.Name = E.DivName
Group by D.Name.
```

With this query we can still compute the result without a temporary table if the tuple sequence satisfies the order property (**D.Name, E.Salary**), which is equivalent to the OP (**E.DivName, E.Salary**). In general, if there are multiple grouping columns with such a query the interesting order we need to satisfy consists of

- the $n! \times 2^n$ possible interesting orders for the grouping columns, augmented (suffixed) by
- the aggregation attribute (again, in ascending or descending lexicographical sequence). Note that with sort-based aggregation we can compute at most one distinct aggregate function in this way.

With grouped queries containing joins, the aggregation can be pipelined with the join if the **Group by** columns constitute the primary key columns of each relation [74]. This

condition, however, can be relaxed: *any* primary or candidate key of the base relations will do [294] and we can also utilize table constraints and restriction conditions to infer the values of key attributes [228].

If there is no **Group by** clause then we have the situation mentioned earlier on page 222: if the aggregation function is **Min()** or **Max()** we can simply retrieve the first (last) non-null value in the index [219, pp. 566–7]—we must ignore null values in the sequence since both **Min()** and **Max()** cannot return **Null** if the query does not contain a **Group by** clause. With **Avg (Distinct)** or **Sum (Distinct)** we retrieve all of the tuples satisfying the query's **Where** clause but eliminate duplicate values from the input.

5.4.6.1 Pipelining join with duplicate elimination

Duplicate elimination also benefits from sorted input, as **Select Distinct** is semantically equivalent to grouping over all of the attributes in the query's **Select** list. However, if we consider duplicate elimination of SPJ queries, there are several additional ways to exploit ordered tuple sequences. We illustrate some possible optimization techniques with the following example.

EXAMPLE 37
Consider the query

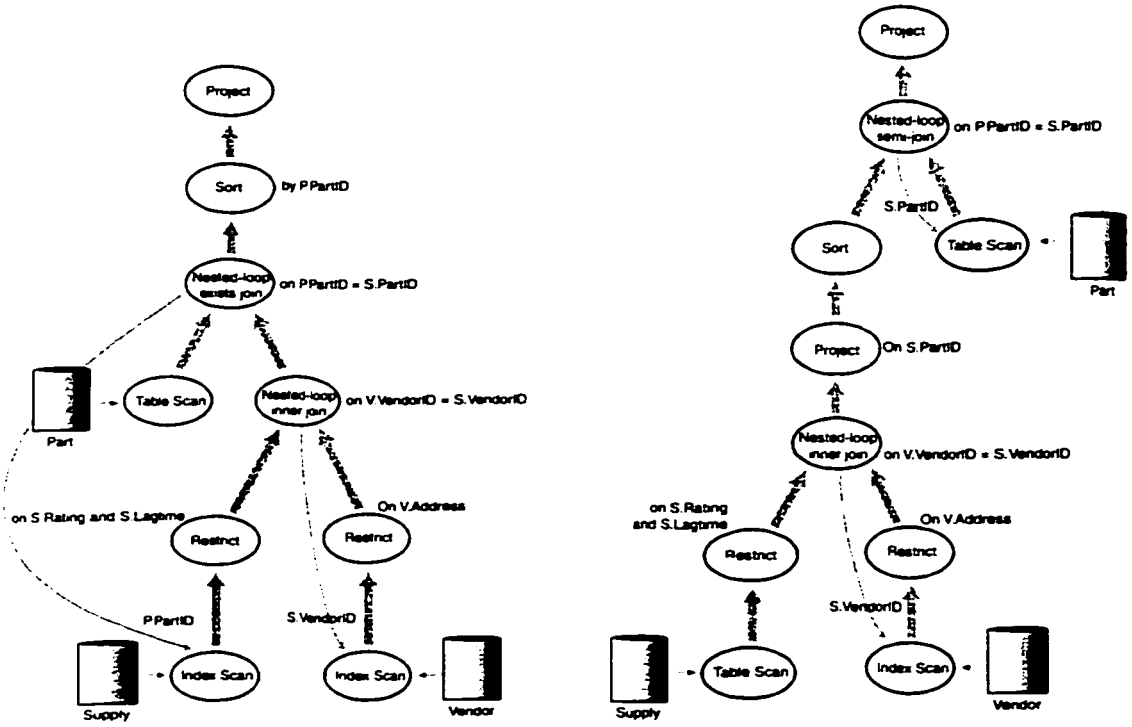
```
Select Distinct P.PartID, P.Description, P.ClassCode
From   Part P, Supply S, Vendor V
Where  P.PartID = S.PartID and V.VendorID = S.VendorID and
       S.Rating in ( 'A', 'B' ) and S.Lagtime < 3 and
       V.Address like '%Quebec%'
Order by D.Name
```

which lists the parts supplied by vendors located in Quebec that can be reliably delivered in less than three business days.

Several potential execution strategies are possible[35]; two potential nested-loop strategies are shown in Figure 5.3. From Corollary 3 in Section 4.4.2 we know that we can rewrite this query as the nested query

---

35 These nested-loop strategies serve only to illustrate possible access plans. Which plan offers the best performance will greatly depend on the relative sizes of the tables and the selectivity of each predicate.



(a) Nested-loop strategy with Part as the outer table, utilizing an exists join to the rest of access plan.

(b) Nested-loop strategy with Part as the inner table. The semijoin eliminates duplicate part identifiers to retain the correct semantics.

FIGURE 5.3: Two potential nested-loop physical access plans for Example 37.

```
Select P.PartID, P.Description, P.ClassCode
From   Part P
Where  Exists( Select *
               From   Vendor V, Supply S
               Where  V.VendorID = S.VendorID and
                      S.PartID = P.PartID and
                      S.Rating in ( 'A', 'B' ) and
                      S.Lagtime < 3 and
                      V.Address like '%Quebec%')
Order by D.Name
```

which corresponds to a semi-join between the two tables [74]. The first potential strategy (a) embodies this approach; it sequentially scans the PART table, and for each part

determines whether or not matching tuples exist in the SUPPLY and VENDOR tables.

Figure 5.3(b) illustrates an alternative join strategy with the PART table as the innermost table in the plan. Strategy (b) involves determining the set of part identifiers required from the join of SUPPLY and VENDOR, which are then sorted. The semijoin operator then accesses each PART tuple but only does so once; duplicate part identifiers are ignored (an alternative way of constructing the physical expression tree would be to eliminate duplicate part identifiers before the join).

Strategy (b) is an example of both sort-ahead and *duplicate elimination pushdown*, identical to interchanging the order of **Group by** and join [54, 57, 294–296]. Such a strategy takes advantage of the additional restriction predicates on both SUPPLY and VENDOR tables, which may pay off depending on the selectivities of those predicates. Note as well that the sort of part identifiers could be avoided by performing an index scan on SUPPLY by `S.PartID`, rather than a table scan. However, this could add considerable cost to the retrieval of tuples from the SUPPLY table.

5.4.7 Union and distinct union

Another possible exploitation of tuple sequences is for the optimization of query expressions: `Union`, `Intersect`, and the like [136]. For example, consider a **Union** query expression (which we denote as $T \cup_{Dist} V$) that, by definition, eliminates duplicate rows. It is possible to perform a simple merge of the two query specifications, thus eliminating the need for a temporary table and a duplicate elimination step, if the tuple sequence of both T and V satisfies the same order property and the order properties include each attribute in the `Select` list. We can pipeline duplicate elimination with merging in the case where we have a **Union all** query expression (which we denote as $T \cup_{All} V$) and both query specifications satisfy the same requirement as to ordering.

Albeit advantageous, neither of the two scenarios above are likely to occur often in practice. However, two other optimizations are possible that can exploit the order properties of the underlying query specifications. The first optimization uses the idea of *quotient relations* from reference [92] to exploit order properties for **Union** queries, which by definition require duplicate elimination. The insight is to realize that if both query specifications satisfy the same OP then that OP forms a *partition* of the tuples in the derived tables defined by each query specification. Consequently, for duplicate elimination *only those tuples in that partition need be examined to eliminate duplicate rows.* Consequently the size of the temporary table (or other data structure) used to eliminate duplicates can be greatly reduced in size.

The second optimization assists with the computation of **Union all** query expressions. Since duplicate tuples are not eliminated, the result can be computed simply as the concatenation of the two inputs. However, consider a **Union all** query expression that contains an **Order by** clause. We can eliminate the subsequent sort step if the order property of both of its input query specifications can satisfy the interesting order specified in the query—that is, we *push down* the **Order by** clause into the underlying query specifications, and then merely merge the inputs. Simmen et al. [261] consider the possibility of sort-ahead only for SPJ queries, particularly for sort-merge joins.

5.4.8 Intersection and difference

Similar to **Union** expressions, we can exploit order properties in the computation of the set operations **Intersect** and **Intersect all**. As with **Union**, for **Intersect** query expressions we can exploit the partitioning of the rows if each of the inputs to the distinct intersection operator satisfy the same OP. For **Intersect all**, however, a simple merge of the inputs is not sufficient to yield the correct result unless both query specifications satisfy the same order property that contains each item in their respective **Select** lists. However, partitioning the tuples by their order property (in each input) can still yield a strategy that may be cheaper than sorting both inputs in their entirety. Similar processing can be performed for query expressions involving **Except** and **Except all** (the operators distinct difference and difference, respectively).

5.5 Related work in order optimization

Commercial relational database systems, such as IBM S DB2 *Universal Database*, have exploited sort order in indexed access methods for some time. To our knowledge, however, Simmen, Shekita, and Malkemus [261, 262] appear to be the first researchers to discuss the theory of order optimization in the literature. They also describe several aspects of the implementation of this theory in DB2. Much of the work presented in this chapter was developed independently of Simmen et al. One of our main contributions is to prove the sufficient and necessary conditions for propagating order properties through a join, a problem which Simmen et al. mention only in passing.

The basic theory of tuple sequences and several ideas about the relationship between functional dependencies and unique indexes is given by Abiteboul and Ginsburg [3]. Ginsberg and Hull [104] mention applications of sort set analysis (a class of order dependencies) to physical storage implementations in relational database systems, and allude to

the possibility of reducing the time spent on sorting during query processing by taking advantage of the order of the tuples as they are stored on disk. They do not, however, investigate the application of the theory to query processing and optimization. Similarly, some recent results by Ng [217] parallel both the work herein, and that of Simmen et al. Ng defines a sound and complete axiom system for *ordered functional dependencies* over an *ordered* relational model, using two similar but subtly different extensions for domains: *pointwise orderings* and *lexicographical orderings*. Ng's model, however, is not based on ANSI SQL and does not address the existence of null values, duplicate rows, or three-valued logic. Interestingly, the main application of his work appears to be normalized schema design for an ordered relational database; there is no mention of the usefulness of order dependencies in query optimization. Dayal and Goodman [76] study tree query optimization in the context of the MULTIBASE system but do not consider exploiting tuple sequences. In their TIMBER system, Stonebraker and Kalash [268] consider sequences of tuples as a possible storage mechanism to support text retrieval applications but do not elaborate on optimization techniques that exploit TIMBER's storage model.

Another possible technique for combining order properties with the semantics of sequences of tuples is to extend the notion of quotient relations [92], which offer a well-defined relational algebra that operates over partitions of relations that are equivalent for some set of attributes X. In effect, these partitions are the 'groups' that result when performing a `Group by` over X on any arbitrary query. The extension required is to add an ordering relationship between the groups of tuples.

5.6 Conclusions

In this chapter we formally described the necessary and sufficient conditions for augmenting and reducing order properties, which included the specification of table and column constraints and handled the three-valued logic of ANSI SQL. We also presented sufficient conditions for determining if the order properties satisfied by two tuple sequences could be concatenated when performing inner or outer nested-loop and sort-merge joins. Part of this research brought together formalisms on tuple sequences [3] and prior work in query processing [92] that was absent in reference [261]. We also expanded the number of types of 'interesting orders' that we can exploit in query processing: in addition to sort avoidance with joins, `Distinct`, and `Group by`, order properties are useful to:

- optimize query expressions with or without duplicate elimination,
- optimize simple queries containing `Min()` or `Max()` aggregate functions,
- optimize aggregate functions with duplicate elimination,

- reduce the size of a cache for subquery memoization,
- *cut* the processing of an indexed search condition, and
- lower the cost of indexed retrieval by probing with a sorted set of secondary keys.

We believe there are several additional opportunities for exploiting order properties in query optimization. One very promising area mentioned previously is in the optimization of queries where the result set size is bounded, for example with the specification of `Top` n or `Bottom` n queries [42–44]. Exploiting this information is an active research area for vendors whose databases are used in OLAP applications. Sort order analysis is also useful in environments where the user desires an access plan optimized for response time instead of resource consumption.

6 Conclusions

Knowledge of functional dependencies, particularly key dependencies, is vital to the operation of a sophisticated query optimizer. All commercial database systems exploit functional dependencies in a variety of ways. However, we believe that most, if not all, of the query optimizers in these implementations could be improved by expanding the set of maintained constraints to include lax functional dependencies, equivalence constraints, and null constraints. The fact that ORACLE8i [26] permits a database administrator to explictly declare functional dependencies that hold in a given schema using ORACLE's `Dimension` objects is evidence that not only are dependencies useful for optimization—in the case of reference [26], to permit the query optimizer to exploit the existence of materialized views—but that the existing tools and techniques for their analysis are insufficient.

Our primary goal in this thesis was to develop an algorithm to capture this array of dependency and constraint information for an arbitrary relational algebra expression. Our main contributions can be summarized as follows:

1. a formal definition of an extended relational model that includes real and virtual attributes, three-valued logic, and multiset semantics, and a set of algebraic operators over that model that correspond to the major algebraic operators supported by ANSI SQL, particularly outer joins;

2. definitions of lax functional dependencies and equivalence constraints that effectively model ANSI SQL's unique constraints, true-interpreted predicates, and outer joins;

3. a sound axiom system for a combined set of strict and lax functional dependencies and equivalence constraints;

4. a formal characterization of the dependencies and constraints that hold in the result of each algebraic operator, particularly the problematic operators left-, right-, and full-outer join;

5. an extended FD-graph representation and an algorithm that correctly represents the set of valid dependencies and constraints that hold in the result of each operator and that can be simply augmented to capture additional constraints;

6. a suite of rewrite optimizations, formally proved, that exploit derived functional dependencies to transform queries into semantically equivalent variants that may offer the opportunity to discover better access plans;

7. a set of theorems that describe the interaction of order properties and functional dependencies, and examples of how exploiting these dependencies can lead to improved access plans, particularly by avoiding unnecessary sorts.

Our theoretical results provide a metaphorical channel through the 'semantic reef' of the optimization of outer joins. By characterizing the dependencies and equivalence constraints that hold with outer joins, we permit a wider class of optimization techniques to be applied to queries, views, or materialized views containing them. However, we believe that additional work on outer join optimization is still necessary to 'widen' and 'deepen' this channel. If we have learned anything through the development of this thesis, it is that the optimization of outer join queries remains a considerable challenge, both theoretically and in practice.

Some of the work contained herein has already been adopted into commercial database products, providing their optimizers with an expanded set of tools to optimize complex queries. Two variants of the simplified FD-graph algorithms described in Chapter 4 have been implemented in Sybase SQL Anywhere, where they are used to determine the correctness of subquery-to-join transformations, including those which require subsequent duplicate elimination, and `Distinct` elimination on SPJ queries, nested queries, and SPJ views. These algorithms have been extended to now support queries containing left outer joins, grouping, and aggregation, and now also utilize equivalence constraints derived from conditions in a `Where` clause. A significant result from their implementation is that a larger class of join elimination optimizations are now possible, which usually has a direct affect on a query's execution time. We believe that other commercial systems have utilized the results in Chapter 4 (actually the results published in our earlier paper [228]) to improve their query rewrite transformations. Moreover, Bhargava, Goel, and Iyer [34] based their work on outer join optimization on the formalisms developed in that paper. In addition, some of the work in Chapter 5, notably on the optimization of `In`-list predicates (see Example 36), has also been implemented in Sybase SQL Anywhere.

6.1 Developing additional derived dependencies

While the dependency and constraint inference algorithms presented in Chapter 3 develop and maintain a comprehensive set of constraints, there are many ways in which the algorithms can be improved to exploit additional information. For example:

1. The current analysis ignores the possible existence of other forms of complex 3-valued logic predicates in ANSI SQL. For example, to simplify the algorithms and proofs, we intentionally ignored predicates of the form '$P(x)$ `is unknown`', which occur rarely in practice.

2. Note that only a limited set of lax dependencies are developed for the `On` condition in a full outer join. Hence a subsequent null-intolerant restriction predicate that could convert the full outer join to a left- or right-outer join will be unable to exploit the missing dependencies.

3. Corollary 2, which provides an alternative means to maintain an existing strict functional dependency $f : X \longrightarrow Y$ from the null-supplying side of an outer join, is also merely a sufficient condition. One could utilize an existing null constraint $X \dashv Y$ in the outer join's null-supplying side to show that f continues to hold in the outer join's result as a strict functional dependency.

4. Consider the left outer join $Q = S \xrightarrow{p} T$ where p contains the conjunctive, null-intolerant condition `S.X = T.Y`. If X is a key of S then the strict functional dependency $f : X \longrightarrow Y$ will *always* hold in the result, even though $X \not\Rightarrow Y$ (due to the possible generation of an all-`Null` row) and, for the same reason, $Y \not\longrightarrow X$.

5. Similarly, consider the left outer query Q as above, but where p consists of the single atomic condition `S.X = 5`. While this condition does not produce a lax or strict dependency, it does produce another form of constraint: for each tuple in the result where `S.X` is not equal to 5, the value of each attribute in $sch(T)$ is `Null`. In fact, this is a generalization of a null constraint. Rather than a constraint between two attributes X and Y, as per Definition 38 on page 95, we instead could write $P(X) \dashv Y$ to reflect that if the predicate P on attribute(s) X evaluates to *true* then attribute Y must be `Null`. This more generalized form of null constraint could be exploited during optimization in several ways. In the example above, such constraints can used to determine the distribution of values for each result attribute that stems from $sch(T)$, which could lead to a more accurate cost estimate. A similar situation exists with full outer joins. If $Q = S \xleftrightarrow{p} T$ and the `On` condition p contains the conjunct `S.X = 5`, then any tuple q_0 in the result containing an `S.X`-value that is neither 5 nor `Null` contains the all-`Null` row of T.

6. As mentioned in Section 3.4.2, we could develop *classes* of scalar functions so that if we have $\lambda(X)$ in a `Select` list or predicate, and we can guarantee that the value returned from λ cannot be `Null`, then we can 'push' that restriction to make the inference that X also cannot be `Null`, forming the existence constraint $x \vdash \lambda(X)$

for each $x \in X$. There are likely other possible ways in which we can manufacture and exploit existence constraints with respect to inferring functional dependencies.

Moreover, improvements to the RESTRICTION algorithm that would recognize strict or lax functional dependencies from a more varied Boolean expression could yield additional opportunities for optimization improvements, as this predicate analysis could be utilized not only for restriction but for the outer join operators as well[36].

The changes required to extend the set of dependencies and/or constraints captured in an FD-graph largely depend on whether or not the new information simply adds another instance of an existing constraint, or forms an altogether new class of constraint. For example, suppose that we wished to extend Cartesian product (Section 3.4.4) to exploit the knowledge that one or both of its inputs consists of at most a single row. Consider the expression $Q = S \times T$. Such a case would occur, for example, if either of the input expressions, say e_T, represented a grouped query that did not contain any grouping attributes ($A^G = \emptyset$). In this case, there is no need to construct a new tuple identifier vertex to represent the result of Q. Instead, we need only add strict functional dependencies from the vertex $v_k \in V^R[G_S]$ that represents $\iota(S)$ to each of the vertices in $V^A[G_T]$.

On the other hand, capturing a new form of constraint will likely require additional classes of vertices and edges. Such an enhancement would involve:

1. proving that the new dependency or constraint would hold in the result of that operator over any instance of the database;
2. analyzing the other operators to determine if the new dependency would remain valid in the result of each;
3. extending the data structures in the FD-graph to represent the constraint;
4. modifying the appropriate FD-graph algorithm(s) to capture the new constraint;
5. possibly modifying other algorithms in order to retain or remove this constraint as necessary;
6. proving the correctness of each modified algorithm.

---

36 For example, David Toman has suggested modelling scalar functions and other complex predicates of n parameters by constructing a table with $n + 1$ columns of infinite domains, and rewriting the original query to include these 'virtual' tables by deriving the necessary join predicates from the set(s) of function parameters. In this manner the analysis of strict and lax dependencies due to functions can be reduced to the more straightforward analysis of conjunctive equality conditions.

6.2 Exploiting uniqueness in nonrelational systems

Our original motivation for determining how derived functional dependencies could be used in semantic query optimization was to find ways to expand the strategy space for optimizing ANSI SQL queries—particularly nested queries and joins—against relational views of IMS databases [131]. We believe these transformations are useful for *any* database model that uses pointers between objects. Pointer-based systems differ from traditional relational systems in that the cost of processing a particular algebraic operator in a pointer-based database system can vary significantly from the cost of processing the same operator in a 'pure' relational system.

As mentioned previously, several researchers [74, 101, 155, 157, 212, 230, 291] have studied ways to rewrite nested queries as joins to avoid a nested-loops execution plan. When the query is converted to a join, the optimizer is free to choose the most efficient join strategy while maintaining the semantics of the original query; the assumption is that a nested-loops strategy is inefficient and seldom worth considering.

On the other hand, non-relational systems such as IMS and various object-oriented database systems are essentially *navigational* and queries against these data models inherently use a nested-loops approach. In this section, we propose converting joins to subqueries as a possible execution strategy in these systems. Our examples below illustrate that nested-loop processing remains an attractive execution strategy, under certain conditions, with a variety of database architectures.

6.2.1 IMS

Part of the CORDS multidatabase project at the University of Waterloo [15] aimed to find ways to support ANSI-standard SQL queries against relational views of IMS databases [170, 171]. Essentially, the SQL gateway optimizer attempted to translate an SQL query into an iterative DL/I program consisting of nested loops of IMS calls [133]. Queries that cannot be directly translated by the *data access layer*—which executes the iterative program—require facilities of the *post-processing layer* that can perform more complex operations not directly supported by DL/I, such as sorting or aggregation, but at increased cost [170]. Therefore, nested-loop strategies, which require only the gateway's data access layer, may often be cheaper to execute.

EXAMPLE 38
Consider the select-project-parent/child join query

```
Select All V.*
From   Vendor V, Supply S
Where  S.VendorID = V.VendorID and S.PartID = :PARTNO
```

which lists all suppliers who have supplied a particular part, denoted by the host variable :PARTNO. This query can be handled exclusively by the data access layer by utilizing the application view in Figure A.4(a). A straightforward nested-loop join strategy is:

```
1717  GU VENDOR;
1718  while status = ' ' do
1719    GNP VSUPPLY (PartID = :PARTNO);
1720    while status = ' ' do
1721      output VENDOR tuple;
1722      GNP VSUPPLY (PartID = :PARTNO)
1723     od ;
1724    GN SUPPLIER
1725   od
```

The subquery block satisfies conditions similar to those in Theorem 12, which in turn depends on being able to infer derived key dependencies, and therefore can exploit the mechanisms detailed in Chapter 3. For this example, a necessary condition is that at most a single instance (segment) of VSUPPLY can join with each VENDOR. Therefore, we can rewrite this query as

```
Select All  S.*
From   Vendor V
Where  Exists (Select  *
               From    Supply S
               Where   V.VendorID = S.VendorID
                       and S.PartID = :PARTNO ).
```

This transformation simplifies the iterative method above, since the inner nested loop can stop as soon as one qualifying VSUPPLY segment is found:

```
1726  GU VENDOR;
1727  while status = ' ' do
1728    GNP VSUPPLY (PartID = :PARTNO);
1729    if status = ' ' then
1730      output VENDOR tuple
1731     fi ;
1732    GN VENDOR
1733   od
```

This version reduces the number of DL/I calls against the VSUPPLY segment by half, since the second GNP call in the join strategy (line 1722) will always fail with a 'GE' (not found) status code. A greater cost reduction may occur if the optimizer can convert a join that specifies non-key attributes in the join predicate to a nested query. For example, suppose the Supply table contained the attribute OEM-PartID, which in the IMS implementation would likely be represented as a unique SRCH field in the VSUPPLY segment. In the join strategy above, DL/I would have to scan all VSUPPLY segments with the given OEM part number, instead of halting the search when the next segment's key was greater than :PARTNO. The nested version halts the search immediately once DL/I finds a match.

6.2.2 Object-oriented systems

In some object-oriented database systems, physical *object identifiers* (OIDs) take the place of foreign keys; both EXODUS and O_2 take this approach [256]. However, OIDs differ from pointers in IMS because each child object points to its parent (see Figure 6.1.) This pointer scheme does not effectively support select-project-join queries in which the selection predicate on the parent class (for example, VENDOR is more restrictive than the predicate on a subordinate class, because the most efficient way to process this type of join would require pointers in the opposite direction [256, pp. 46].

EXAMPLE 39
Consider the following join between VENDOR and SUPPLY in an object-oriented database system (assume that the object-oriented system supports the use of *path expressions*, as in reference [301], in the SQL variant used herein):

```
Select All V.*
From   Supply S, Vendor V
Where  V.VendorID Between '000AA000' and '000AAB000' and
       V.S.SupplyCode = :SC
```

which lists all part vendors whose identifiers lie in the range '000AA000' to '000AAB000' and whose supply code is equivalent to the input host variable :SC. A straightforward nested-loop join strategy is:

```
1734  retrieve SUPPLY;
1735  while suppliers remaining do
1736   if SUPPLY.SUPPLYCODE = :SC then
1737    retrieve SUPPLY.VENDOR;
1738    if SUPPLY.VENDOR.VendorID is between '000AA000' and '000AAB000' then
1739     output VENDOR object
1740     fi
1741    fi;
```

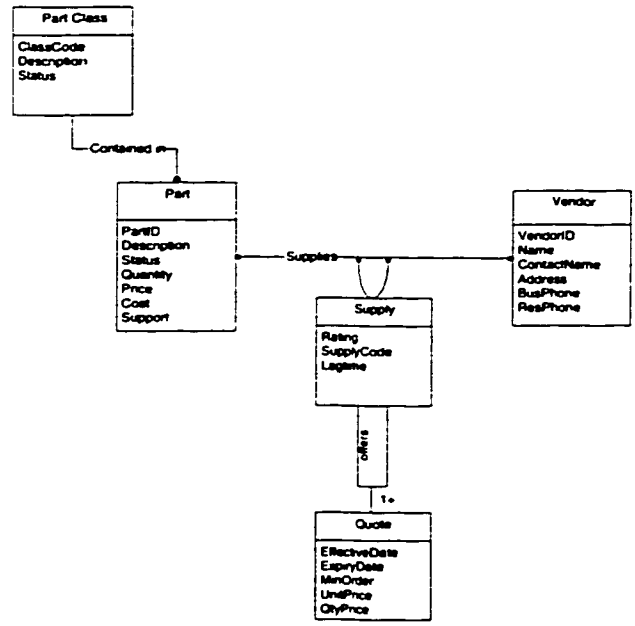


FIGURE 6.1: Rumbaugh OMT object-oriented data model for the parts-related classes. We assume object identifiers (OIDs), implemented as physical pointers, replace foreign keys as the relationship mechanism between objects. Each class has a surrogate key attribute to aid in object identification.

```
1742    retrieve next SUPPLY
1743    od
```

This strategy may be inefficient because many vendor objects may be referenced, only to find that their identifier is not in the specified range—this would be the case, for instance, if many vendors supplied parts with a supply code of :SC. From Theorem 12, however, we can rewrite the query in Example 39 as the nested query

```
Select All V.*
From   Vendor V
Where  V.VendorID between'000AA000' and '000AAB000' and
       Exists (Select  *
               From    Supply S
               Where   S.SupplyCode = :SC ).
```

Assuming we have an index on SUPPLY by supply code, and an index on VENDOR by vendor identifier, then a more efficient strategy may be as follows:

```
1744  retrieve VENDOR (VendorID between'000AA000' and '000AAB000');
```

```
1745  while vendors remaining do
1746    retrieve SUPPLY (SupplyCode = :SC and SUPPLY.VENDOR.OID = VENDOR.OID);
1747    if found then
1748      output VENDOR object
1749      fi;
1750    retrieve next VENDOR (VendorID between'000AA000' and '000AAB000');
1751    od
```

depending on the objects' selectivity. The idea is to restrict the search of the SUPPLY class to only those instances that correspond to a VENDOR instance whose vendor identifier matches the range predicate.

6.3 Other applications and open problems

The formalisms we developed in Chapter 2, in particular our definitions of strict and lax functional dependencies, equivalence constraints, and null constraints, and the formalisms for order properties and their interaction with these constraints, allow for the systematic study of other, closely related optimization problems. For example, the set of algebraic operators described in this thesis are merely the 'basic' ones needed to mirror a reasonably large class of query expressions in ANSI SQL. However, database systems implement a wider variety of *executable* algebraic operators: inner semijoin, left-outer semijoin, full-outer semijoin, exists join [29, 74, 146], variant forms of 'generalized outer join' [30, 31, 34, 94, 95, 98], and so on; Graefe's recent survey [107] provides a useful overview. Each of these operators requires careful analysis to determine the sets of dependencies and constraints that hold in their result.

The theoretical results contained herein can be extended in a variety of ways. We limited our modelling of constraints to those ANSI SQL column and table constraints that could be specified in a **Check** clause; we did not attempt to exploit other forms of constraints, such as multivalued dependencies, referential integrity constraints (inclusion dependencies), and the more general form of SQL table constraints termed *assertions*. We made no attempt to prove the completeness of the inference axioms for strict and lax dependencies, defined in Chapter 2, and strict and lax equivalence constraints, defined in Chapter 3. We have also not analyzed the algorithms' efficiency, nor have we constructed a prototype to experiment with various implementations to find an efficient one (see Section 3.8 for some additional remarks).

We believe these results have wide applicability to query processing and optimization. As a first example, consider the optimization of universally-quantified subqueries (see Section 2.3.1.2). Most query optimizers do not optimize these types of queries in a

sophisticated fashion [109]; part of the reason is the existence of true-interpreted correlation predicates, which are difficult to exploit for index processing. However, modelling a query's equivalence constraints with an FD-graph may permit such a predicate to be transformed into a semantically equivalent, false-interpreted one, permitting the correlation predicate to be used as a 'standard' matching predicate for indexed retrieval.

Other applications of our work on derived dependencies include:

1. exploiting derived dependencies during the optimization of queries over materialized views [26, 52, 53, 57, 174, 297];

2. using dependencies, equivalence constraints, and null constraints for materialized view maintenance, particularly for those views containing one or more outer joins [118, 119];

3. extending the work of Medeiros and Tompa [198] on *view update policies* to support the update of views over ANSI SQL tables, not simply relations, and to verify that the constraints (unique indexes, unique constraints, **Check** constraints, assertions, and so on) defined on them cannot be violated.

A Example schema

Our example database scheme contains employee, part, and supplier information for a hypothetical mechanical parts distribution firm, with divisions located in Chicago, New York, and Toronto. The firm's inventory consists of a wide variety of parts, from fasteners to widgets, manufactured by a variety of suppliers throughout North America. Figure A.1 contains an entity-relationship diagram that models the schema.

A.1 Relational schema

Parts. The firm's parts inventory is represented by several base tables which contain information about each part, its supplier(s), its status, and its cost. Parts are organized into classes, which serves to classify parts into groups for easier management and tracking.

```
Create Table Class (
ClassCode    char(2) not null,
Description  char(20) not null,
Status       char(1)
  Primary Key (ClassCode));


Create Table Part (
PartID       char(8) not null,
Description  char(30) not null,
Status       char(8),
Qty          numeric(7),
Price        numeric(7,2) not null,
Cost         numeric(7,2) not null,
Support      char(2) not null,
ClassCode    char(2) not null
  Primary Key (PartID)
  Foreign Key (ClassCode) references Class
  Check (Qty ≠ 0 Or Status = 'InStock')
  Check (Price ≥ Cost));
```

Vendors. Each part may be supplied by more than one supplier (termed a vendor), and the SUPPLY table contains a row for each part-vendor relationship. Vendors are 'ranked' for each part they supply.

```
Create Table Supply (
VendorID            char(8) not null,
PartID              char(8) not null,
Rating              char(1),
SupplyCode          char(4),
Lagtime             numeric(7) not null
  Primary Key (PartID, VendorID)
  Foreign Key (PartID) references Part
  Foreign Key (VendorID) references Vendor
  Check (Rating in( 'A', 'B', 'C') ));
```

It is assumed that periodically a part vendor will respond to a quotation request and offer a specific part for a certain price. The QUOTE table represents this intersection data detailing the quote of a part's price by that particular vendor for a certain date range.

```
Create Table Quote (
QuoteID             char(7) not null,
EffectiveDate       date not null,
ExpiryDate          date not null,
MinOrder            numeric(5) not null,
UnitPrice           numeric(7,2) not null,
QtyPrice            numeric(7,2) not null,
PartID              char(8) not null,
VendorID            char(8) not null
  Primary Key (PartID, VendorID, QuoteID)
  Foreign Key (PartID, VendorID) references Supply);
```

Finally, part vendors and their contacts are defined by the VENDOR table. The data model assumes that vendor names are unique.

```
Create Table Vendor (
VendorID      char(8) not null,
Name          char(40),
ContactName   char(30),
Address       char(40),
BusPhone      char(10),
ResPhone      char(10)
  Primary Key (VendorID)
  Unique (Name));
```

Employees. The EMPLOYEE table contains information regarding the firm's employees, organized by the corporate divisions within the firm. The DIVISION table simply identifies a division within the firm, including a foreign key to that division's manager.

```
Create Table Division (
Name          char(20) not null,
Location      char(40),
ManagerID     char(5)
  Primary Key(Name)
  Foreign Key(ManagerID) references Employee
  Check (Location in ('Chicago', 'New York', 'Toronto')));
```

Each division in the firm may have several employees who are assigned to one, and only one, division. Employees are uniquely identified by their Employee ID, which is unique across all company divisions. An employee is either salaried, or earns an hourly wage. A candidate key for an employee is the employee's name.

```
Create Table Employee (
EmpID         char(5) not null,
Surname       char(20) not null,
GivenName     char(15) not null,
Title         char(20),
Salary        numeric(6,2) not null,
Wage          numeric(6,2) not null,
Phone         char(10),
DivName       char(20)
  Primary Key (EmpID)
  Unique (Surname, GivenName)
  Foreign Key (DivName) references Division
  Check (EmpID Between 1 and 30000)
  Check (Salary ≠ 0 Or Wage ≠ 0) );
```

The MANAGES table embodies the manager-division relationship; a division can have only one manager, but an employee may manage several divisions.

```
Create Table Manages (
ManagerOf     char(20) not null,
EmpID         char(5) not null
  Primary Key (EmpID, ManagerOf)
  Foreign Key (EmpID) references Employee
  Foreign Key (ManagerOf) references Division);
```

Finally, each employee is responsible for the inventory of one or more parts, which are identified by a unique part identifier. Each part may be managed by more than one employee.

```
Create Table ResponsibleFor (
PartID      char(8) not null,
EmpID       char(5) not null
  Primary Key (PartID, EmpID)
  Foreign Key (EmpID) references Employee
  Foreign Key (PartID) references Part);
```

A.2 IMS schema

IMS, jointly developed by IBM Corporation and Rockwell International for the Apollo space program in the 1960s, permits application programs to *navigate* through a set of *database records* stored as a *hierarchy*. The hierarchy defines one-to-many relationships between *segments* (or, more properly, segment types), with a *root segment* at the top of each 'tree'. Each database record consists of a root segment occurrence and all occurrences of its *dependent segments*. Each root segment type in a HIDAM, HDAM, or HISAM database must have a *sequence field* that may either be unique or non-unique. The sequence field is used by IMS to locate a specific root segment occurrence: with HDAM a hashing technique is used, while with HIDAM and HISAM databases an index is used to retrieve root segment occurrences by their sequence field. With dependent segments the sequence field is optional. If one is defined, IMS stores the dependent segment occurrences in ascending order of the sequence field. If a dependent segment type lacks a sequence field, then IMS will insert new segment occurrences at an arbitrary point under that segment type's ancestor in the hierarchy, the precise position determined by the application program at execution.

If the sequence field of a particular segment type is unique, and each of its physical ancestors in the database record also have unique sequence fields, then each segment occurrence can be uniquely identified by the concatenation its sequence field and the sequence fields of each segment occurrence in its hierarchic path. In IMS terminology this is termed the segment's *fully concatenated key*.

A physical IMS database may have up to 15 *levels* (the PARTS database illustrated in Figure A.3 has four) and up to 255 segment types. The database administrator defines a database with a *database description*, commonly referred to as a DBD.

Application programs navigate through the database hierarchy, retrieving one segment at a time, using the IMS application program interface *Data Language/One* (DL/I).

The *application view* of a physical or logical IMS database is described in a *Program Control Block*, or PCB (see Figure A.4). The segment hierarchy defined in a PCB may be composed of a physical hierarchy, meaning that all the segments in the view are from the same physical IMS database, or they may be composed of a *logical hierarchy*, which utilizes logical child/logical parent relationships to form a hierarchical view of segments from different physical databases. Note that a database level cannot be 'missing' from an application view described by an IMS PCB.

Since DL/I is such a low-level API (see Table A.1) the application programmer is responsible for optimizing how the application retrieves its required information, including the use of any indexes that may exist; hence index usage in IMS is not transparent to the application. Furthermore, the programmer must be aware of the effects of different IMS access methods. For example, an HDAM and a HIDAM database with identical schemas can return different results to an application program because of HDAM's hashed access to root segments.

DL/I *Call*	*Purpose*
GU	Get Unique: retrieve the first segment occurrence in the database that satisfies the SSAs
GN	Get Next: retrieve the next segment from the current position in hierarchical order
GNP	Get Next Within Parent: retrieve the next dependent segment occurrence under the present ancestor

TABLE A.1: DL/I calls. Each retrieval call has a 'hold' option (GHU, GHN, and GHNP, respectively) that positions a program on a particular segment occurrence and locks it, prior to its replacement or deletion.

A complete description of IMS databases and application programming are beyond the scope of this thesis. More details regarding the DL/I interface can be found in references [36, 133, 276].

A.2.1 IMS physical databases

Figures A.2 and A.3 depict the three example databases used to implement the IMS version of the data model described in Figure A.1. In the EMPLOYEE physical database, employees are modelled as dependent segments under the DIVISION that employs them. RE-

SPBFOR and MANAGES are logical child segments that respectively implement the many-to-many relationship employees and the parts they are responsible for, and the one-to-many relationship between divisions and managers. With many-to-many relationships, logical child segments are often *paired*, meaning that while two different segment types are used to model the many-to-many relationship (one in each physical hierarchy) only one segment type is actually stored in the database. In this way, IMS ensures a consistent database when an application program inserts, updates, deletes a relationship segment. Table A.2 cross-references each logical child segment in the example schema with its pair. In contrast, MANAGES is a one-to-many relationship, embodying the constraint that each division can have only one manager. Hence MANAGES is not logically paired with any other segment.

A.2.2 Mapping segments to a relational view

Background information regarding the mapping of an IMS schema to a relational view can be found in references [170] and [172], but we briefly state the essentials here; a comprehensive discussion is beyond the scope of this thesis. Each IMS segment type is mapped to a virtual table that contains corresponding attributes. Parent-child relationships are modeled as foreign keys; the fully concatenated key of each dependent segment type is used as the key of its virtual table in the relational view. In this thesis we assume that each segment type has a unique sequence field, which can serve as a key. In practice, most segment types in IMS databases have unique sequence fields, enabling faster retrieval and avoiding the situation where the position of a segment in the database has some intrinsic meaning to application programs. Key attributes inherited from parent segment types are termed *virtual* attributes. For example, `DivName` is a virtual column, derived from the DIVISION segment, in the relational view of the EMPLOYEE segment.

Logical relationships provide a particular challenge for query optimization. For example, the paired bi-directional logical child segments PSUPPLY and VSUPPLY embody the many-to-many relationship between parts and vendors. There are two logical child segment types to permit navigation in both directions, though the contents of a paired PSUPPLY and VSUPPLY are identical but for their logical parent pointers. In the relational view, one table is created (SUPPLY) to represent this data; however, the gateway's optimizer must choose the direction, and therefore the logical child segment, in which to traverse the relationship to retrieve a query's result using the fewest resources.

As an aside, we note that there do exist several commercial products, notably Ingres' gateway products [135] (now offered by Computer Associates), IBM's DataJoiner, and Oracle's SQL Connect to IMS [222] that all provide relational access, via SQL queries, to IMS databases. All of these commercial products fail to offer a comprehensive solution

Segment	*Paired with*	*Pairing method*	*Logical parent*
RESPBFOR	EMPRESPB	virtual	PART
VSUPPLY	PSUPPLY	physical	PART
EMPRESPB	RESPBFOR	virtual	EMPLOYEE
MANAGES	N/A	unidirectional	DIVISION
PSUPPLY	VSUPPLY	physical	VENDOR

TABLE A.2: Logical relationships in the IMS schema

to both the problem of mapping an IMS database to a relational view, and to the problem of transforming update operations done on the view to physical database operations.

In the PARTS physical IMS database, the PART segment is a *child* of the CLASS segment, which is the root segment (see Figure A.3). In turn, EMPRESPB and PSUPPLY are both *dependent segments* of PART, and are thus termed *siblings*. Using analogous terminology, *twin* segments are multiple occurrences of a segment type under the same parent segment occurrence.

Figure A.4 diagrams two application views of the Vendor database. Figure (a) illustrates a straightforward view of the physical Vendor database consisting of the two physical segments VENDOR and VSUPPLY. The second PCB in Figure (b) illustrates the affect logical relationships has on the structure of the hierarchical view seen by the application program. In this example, the concatenated logical parent consisting of VSUPPLY and PART permits the application to view a hierarchy combining the Vendor database with components of the Part database. Moreover, note how the CLASS segment, the root of the physical Parts database, is now described in the hierarchy as a child of the concatenated logical parent segment VSUPPLY/PART.

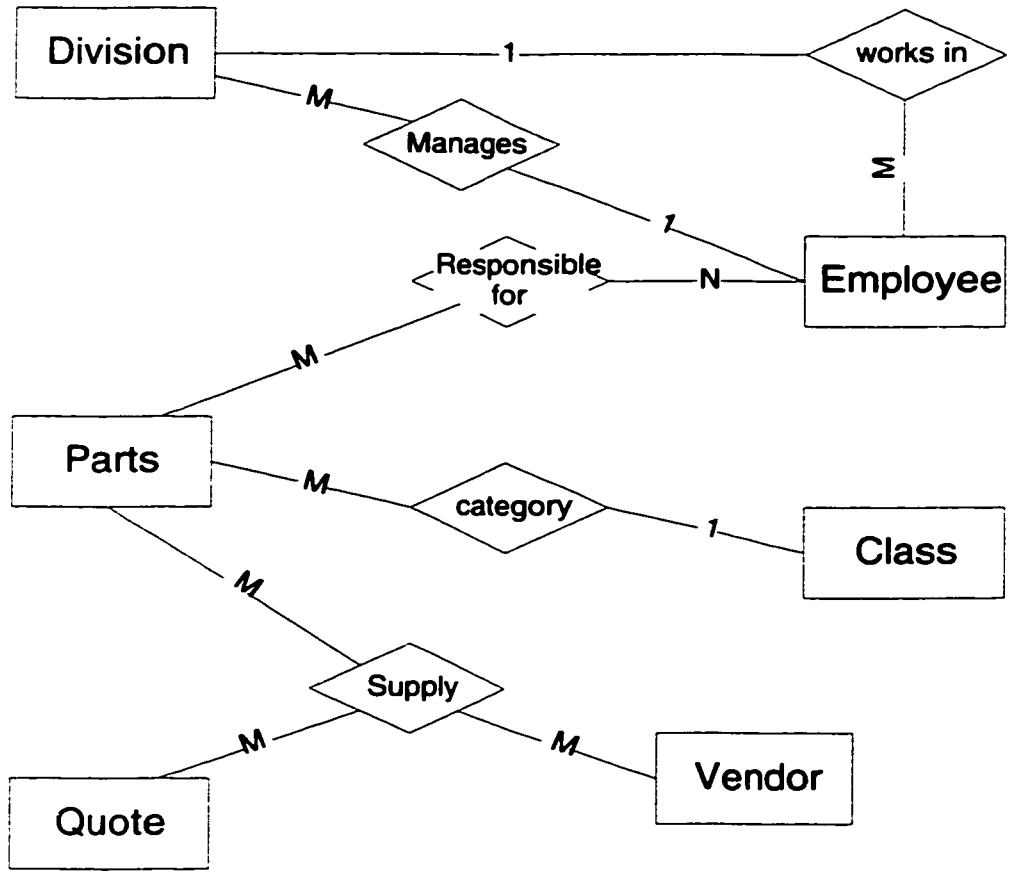


FIGURE A.1: E/R diagram of the manufacturing schema. Entities and relationships that begin with capital letters are represented by base tables.

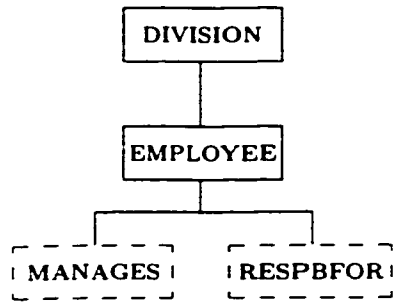
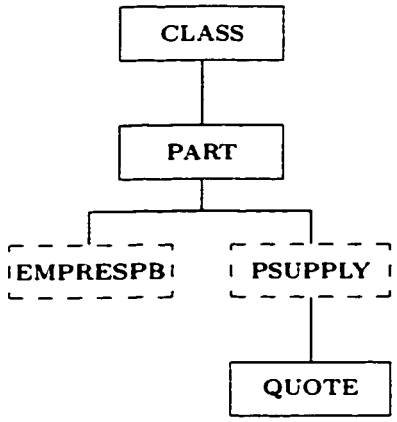
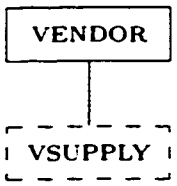


FIGURE A.2: Employee IMS database. Solid boxes denote *physical* segments; dashed boxes denote *logical*, or *pointer*, segments. The database organization is HIDAM [132] with parent-child/twin pointers; root segments are key-sequenced through the database's primary index.

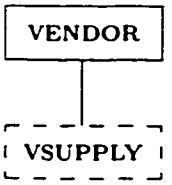


(a) Parts IMS database. EMPRESPB is a paired logical child of the EMPLOYEE segment in the Employee database. QUOTE is intersection data for each supplied part from a particular vendor.

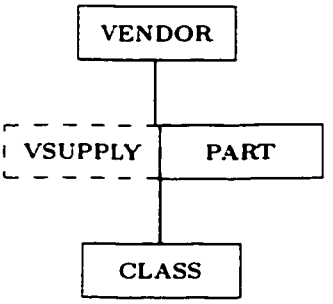


(b) Vendor IMS database. VSUPPLY is a *logical child segment*, physically paired with PSUPPLY in the Parts database, to implement the many-to-many relationship between parts and vendors.

FIGURE A.3: Parts and Vendor IMS databases.



(a) Application view (PCB) of the physical Vendor database.



(b) Application view (PCB) of a logical Vendor database using the concatenated logical child segment consisting of VSUPPLY and PART.

FIGURE A.4: Two application views of the Vendor IMS database.

B Trademarks

The following acronyms and abbreviations used in this thesis are trademarks or servicemarks in Canada, the United States and/or other countries:

- IBM, DB2, DB2/MVS, IMS/ESA, IMS/TM, DL/1, STARBURST, DataJoiner, System R, VSAM, and MVS/ESA are trademarks of International Business Machines Corporation;
- INGRES and QUEL are trademarks of Computer Associates;
- ORACLE, ORACLE8i, and SQL*CONNECT are trademarks of Oracle Corporation;
- SYBASE, SYBASE IQ, SQL Anywhere Studio, Adaptive Server Anywhere, and Adaptive Server Enterprise are trademarks of Sybase, Inc.

Other product names contained herein may be trademarks or servicemarks of their respective companies.

Bibliography

[1] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 254–263, Seattle, Washington, June 1998. Association for Computing Machinery.

[2] Serge Abiteboul and Seymour Ginsburg. Tuple sequences and indexes. In *Proceedings, 11th Colloquium on Automata, Languages, and Programming* (ICALP), pages 41–50, Antwerp, Belgium, July 1984. Springer-Verlag.

[3] Serge Abiteboul and Seymour Ginsburg. Tuple sequences and lexicographic indexes. *Journal of the* ACM, 33(3):409–422, July 1986.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Massachusetts, 1995.

[5] M[ichel] Adiba. Derived relations: A unified mechanism for views, snapshots and distributed data. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 293–305, Cannes, France, September 1981. IEEE Computer Society Press.

[6] Michel E. Adiba and Bruce G. Lindsay. Database snapshots. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 86–91, Montréal, Québec, October 1980. IEEE Computer Society Press.

[7] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. ACM *Transactions on Database Systems*, 4(3):297–314, September 1979.

[8] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. SIAM *Journal on Computing*, 8(2):218–246, May 1979.

[9] J[amal] R. Alsabbagh and V[ijay] V. Raghavan. A framework for multiple-query optimization. In *Proceedings, Second International Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, pages 157–162, Tempe, Arizona, February 1992. IEEE Computer Society Press.

[10] Jamal R. Alsabbagh and Vijay V. Raghavan. Analysis of common subexpression exploitation models in multiple-query processing. In *Proceedings, Tenth* IEEE *International Conference on Data Engineering*, pages 488–497, Houston, Texas, February 1994. IEEE Computer Society Press.

[11] Gennady Antoshenkov. Dynamic query optimization in RDB/VMS. In *Proceedings, Ninth* IEEE *International Conference on Data Engineering*, pages 538–547. IEEE Computer Society Press, April 1993.

[12] Gennady Antoshenkov. Dynamic optimization of index scans restricted by booleans. In *Proceedings, Twelfth* IEEE *International Conference on Data Engineering*, pages 430–440, New Orleans, Louisiana, February 1996. IEEE Computer Society Press.

[13] W. W. Armstrong. Dependency structures of database relationships. In *Proceedings of the* IFIP *Congress*, pages 580–583, Stockholm, Sweden, August 1974. North-Holland.

[14] W. W. Armstrong and C[laude] Delobel. Decompositions and functional dependencies in relations. ACM *Transactions on Database Systems*, 5(4):404–430, December 1980.

[15] G[opi] K. Attaluri, D[exter] P. Bradshaw, N[eil] Coburn, P[er-Åke] Larson, P[atrick] Martin, A[vi] Silberschatz, J[acob] Slonim, and Q[iang] Zhu. The CORDS multidatabase project. IBM *Systems Journal*, 34(1):39–62, 1995.

[16] Paolo Atzeni and Valeria De Antonellis. *Relational Database Theory.* Benjamin/Cummings, Redwood City, California, 1993.

[17] Paolo Atzeni and Nicola M. Morfuni. Functional dependencies in relations with null values. *Information Processing Letters*, 18:233–238, 1984.

[18] Paolo Atzeni and Nicola M. Morfuni. Functional dependencies and constraints on null values in database relations. *Information and Control*, 70(1):1–31, 1986.

[19] Giorgio Ausiello, Alessandro D'Atri, and Domenico Saccà. Graph algorithms for functional dependency manipulation. *Journal of the* ACM, 30(4):752–766, October 1983.

[20] G[iorgio] Ausiello, A[lessandro] D'Atri, and D[omenico] Saccà. Minimal representation of directed hypergraphs. SIAM *Journal on Computing*, 15(2):418–431, May 1986.

[21] Giorgio Ausiello, Umberto Nanni, and Giuseppe F. Italiano. Dynamic maintenance of directed hypergraphs. *Theoretical Computer Science*, 72(2-3):97–117, 1990.

[22] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relation schemas. ACM *Transactions on Database Systems*, 4(1):30–59, March 1979.

[23] Catriel Beeri, Ronald Fagin, and John H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In ACM SIGMOD *International Conference on Management of Data*, pages 47–61, Toronto, Ontario, August 1977.

[24] Catriel Beeri and P[eter] Honeyman. Preserving functional dependencies. SIAM *Journal on Computing*, 10(3):647–656, August 1981.

[25] D. A. Bell, D. H. O. Ling, and S. I. McClean. Pragmatic estimation of join sizes and attribute correlations. In *Proceedings, Fifth* IEEE *International Conference on Data Engineering*, pages 76–84, Los Angeles, California, February 1989. IEEE Computer Society Press.

[26] Randall G. Bello, Karl Dias, Alan Downing, James Feenan, et al. Materialized views in ORACLE. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 659–664, New York, New York, August 1998. Morgan-Kaufmann.

[27] Kristin Bennet, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 400–407, San Diego, California, 1991. Morgan-Kaufmann.

[28] Philip A. Bernstein. Synthesizing third normal form relations from functional dependencies. ACM *Transactions on Database Systems*, 1(4):277–298, December 1976.

[29] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *Journal of the* ACM, 28(1):25–40, January 1981.

[30] Gautam Bhargava, Piyush Goel, and Bala[krishna] Iyer. Reordering of complex queries involving joins and outer joins. Research Report TR-03.567, IBM Corporation, Santa Teresa Laboratory, San Jose, California, July 1994.

[31] Gautam Bhargava, Piyush Goel, and Bala[krishna] Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In ACM SIGMOD *International Conference on Management of Data*, pages 304–315, San Jose, California, May 1995.

[32] Gautam Bhargava, Piyush Goel, and Bala[krishna] Iyer. No regression algorithm for the enumeration of projections in SQL queries with joins and outer joins. In *Proceedings of the 1995* CAS *Conference*, pages 87–99, Toronto, Ontario, November 1995. IBM Canada Laboratory Centre for Advanced Studies.

[33] Gautam Bhargava, Piyush Goel, and Bala[krishna] Iyer. Simplification of outer joins. In *Proceedings of the 1995* CAS *Conference*, pages 63–75, Toronto, Ontario, November 1995. IBM Canada Laboratory Centre for Advanced Studies.

[34] Gautam Bhargava, Piyush Goel, and Bala[krishna] Iyer. Efficient processing of outer joins and aggregate functions. In *Proceedings, Twelfth* IEEE *International Conference on Data Engineering*, pages 441–449, New Orleans, Louisiana, February 1996. IEEE Computer Society Press.

[35] Joachim Biskup. A formal approach to null values in database relations. In Hervé Gallaire, Jack Minker, and Jean Nicolas, editors, *Advances in Database Theory*, volume 1, pages 299–341. Plenum Press, New York, New York, 1981.

[36] Dines Bjørner and Hans Henrik Løvengreen. Formalization of database systems—and a formal definition of IMS. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 334–347, Mexico City, Mexico, September 1982. VLDB Endowment.

[37] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. ACM *Transactions on Database Systems*, 14(3):369–400, September 1989.

[38] José A. Blakeley and Héctor Hernández. Multiple-query optimization for materialized view maintenance. Technical Report 267, Indiana University, Bloomington, Indiana, January 1989.

[39] José A. Blakeley, Per-Åke Larson, and F. W. Tompa. Efficiently updating materialized views. In ACM SIGMOD *International Conference on Management of Data*, pages 61–71, Washington, D.C., May 1986.

[40] José A. Blakeley and Nancy L. Martin. Join index, materialized view, and hybrid-hash join: A performance analysis. In *Proceedings, Sixth* IEEE *International Conference on Data Engineering*, pages 256–263, Los Angeles, California, February 1990.

[41] Kjell Bratbergsengen. Hashing functions and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, Singapore, August 1984. VLDB Endowment.

[42] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. In ACM SIGMOD *International Conference on Management of Data*, pages 219–230, Tucson, Arizona, May 1997. Association for Computing Machinery.

[43] Michael J. Carey and Donald Kossmann. Processing top n and bottom n queries. IEEE *Data Engineering Bulletin*, 20(3):12–19, September 1997.

[44] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 158–169, New York, New York, August 1998. Morgan-Kaufmann.

[45] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 29–59, Los Angeles, California, March 1982. Association for Computing Machinery.

[46] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

[47] Stefano Ceri and Georg Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. IEEE *Transactions on Software Engineering*, 11(4):324–345, April 1985.

[48] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991. Morgan Kaufmann.

[49] U[pen] S. Chakravarthy, John Grant, and Jack Minker. Foundations of semantic query optimization for deductive databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann, Los Altos, California, 1987.

[50] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. ACM *Transactions on Database Systems*, 15(2):162–207, June 1990.

[51] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 34–43, Seattle, Washington, June 1998. Association for Computing Machinery.

[52] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. Technical Report HPL-DTD-94-16, HP Research Laboratories, Palo Alto, California, February 1994. 25 pages.

[53] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings, Eleventh* IEEE *International Conference on Data Engineering*, pages 190–200, Taipei, Taiwan, March 1995. IEEE Computer Society Press.

[54] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, Santiago, Chile, September 1994. Morgan Kaufmann.

[55] Surajit Chaudhuri and Kyuseok Shim. An overview of cost-based optimization of queries with aggregates. IEEE *Data Engineering Bulletin*, 18(3):3–9, September 1995.

[56] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 87–98, Bombay, India, September 1996. Morgan Kaufmann.

[57] Surajit Chaudhuri and Kyuseok Shim. Optimizing queries with aggregate views. In P. Apers, M. Bouzeghoub, and G[eorges] Gardarin, editors, *Advances in Database Technology*—EDBT'96 *(Proceedings of the 5th International Conference on Extending Database Technology)*, pages 167–182, Avignon, France, March 1996. Springer-Verlag.

[58] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings,* ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 59–70, Washington, D. C., May 1993. Association for Computing Machinery.

[59] Mitch Cherniack and Stan Zdonik. Inferring function semantics to optimize queries. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 239–250, New York, New York, August 1998. Morgan-Kaufmann.

[60] Stavros Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[61] Stavros Christodoulakis. Estimating block selectivities. *Information Systems*, 9(1):69–79, 1984.

[62] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. ACM *Transactions on Database Systems*, 9(2):163–186, 1984.

[63] Stavros Christodoulakis. On the estimation and use of selectivities in database performance evaluation. Technical Report CS–89–24, University of Waterloo, Waterloo, Ontario, Canada, June 1989.

[64] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proceedings of the Fifth International Conference on Database Theory*—ICDT *'95*, pages 54–67, Prague, Czech Republic, January 1995. Springer-Verlag.

[65] E. F. Codd. A relational model of data for large shared data banks. *Communications of the* ACM, 13(6):377–387, June 1970.

[66] E. F. Codd. Extending the database relational model to capture more meaning. ACM *Transactions on Database Systems*, 4(4):397–434, December 1979.

[67] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In ACM SIGMOD *International Conference on Management of Data*, pages 469–480, Montréal, Québec, June 1996. Association for Computing Machinery.

[68] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting multiple view maintenance policies. In ACM SIGMOD *International Conference on Management of Data*, pages 405–416, Tucson, Arizona, May 1997. Association for Computing Machinery.

[69] Peter Corrigan and Mark Gurry. ORACLE *Performance Tuning*. O'Reilly & Associates, Sebastopol, California, 1993.

[70] Hugh Darwen. The role of functional dependence in query decomposition. In *Relational Database Writings 1989–1991*, chapter 10, pages 133–154. Addison-Wesley, Reading, Massachusetts, 1992.

[71] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, Massachusetts, fifth edition, 1990.

[72] C. J. Date and Hugh Darwen. *A Guide to the* SQL *Standard*. Addison-Wesley, Reading, Massachusetts, fourth edition, 1997.

[73] Umeshwar Dayal. Query processing in a multidatabase system. In Kim et al. [159], pages 81–108.

[74] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 197–208, Brighton, England, August 1987. Morgan Kaufmann.

[75] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. ACM *Transactions on Database Systems*, 8(3):381–416, September 1982.

[76] Umeshwar Dayal and Nathan Goodman. Query optimization for CODASYL database systems. In ACM SIGMOD *International Conference on Management of Data*, pages 138–150, Orlando, Florida, June 1982.

[77] János Demetrovics, Leonid Libkin, and Ilya B. Muchnik. Functional dependencies in relational databases: A lattice point of view. *Discrete Applied Mathematices*, 40(2):155–185, December 1992.

[78] Jim Diederich. Minimal covers revisited: Correct and efficient algorithms. ACM SIGMOD *Record*, 20(1):12–13, March 1991.

[79] Jim Diederich and Jack Milton. New methods and algorithms for database normalization. ACM *Transactions on Database Systems*, 13(3):339–365, September 1988.

[80] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. SIAM *Journal on Computing*, 23(4):738–761, August 1994.

[81] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, California, 2nd edition, 1995.

[82] Raymond Fadous and John Forsyth. Finding candidate keys for relational data bases. In ACM SIGMOD *International Conference on Management of Data*, pages 203–210, San Jose, California, May 1975.

[83] R[onald] Fagin. Functional dependencies in a relational database and propositional logic. IBM *Journal of Research and Development*, 21(6):534–544, November 1977.

[84] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. ACM *Transactions on Database Systems*, 2(3):262–278, September 1977.

[85] Ronald Fagin. Normal forms and relational database operators. In ACM SIGMOD *International Conference on Management of Data*, pages 153–160, Boston, Massachusetts, May 1979.

[86] Ronald Fagin. Horn clauses and database dependencies (extended abstract). In *Proceedings, Twelfth Annual* ACM *Symposium on the Theory of Computing*, pages 123–134, Los Angeles, California, April 1980. Association for Computing Machinery.

[87] Ronald Fagin. A normal form for relational databases that is based on domains and keys. ACM *Transactions on Database Systems*, 6(3):387–415, September 1981.

[88] Ronald Fagin. Horn clauses and database dependencies. *Journal of the* ACM, 29(4):952–985, October 1982.

[89] Sheldon Finkelstein. Common expression analysis in database applications. In ACM SIGMOD *International Conference on Management of Data*, pages 235–245, Orlando, Florida, June 1982.

[90] Johann Christoph Freytag. A rule-based view of query optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 173–180, San Francisco, California, May 1987.

[91] Antonio L. Furtado and Marco A. Casanova. Updating relational views. In Kim et al. [159], pages 127–142.

[92] Antonio L. Furtado and Larry Kerschberg. An algebra of quotient relations. In ACM SIGMOD *International Conference on Management of Data*, pages 1–8, Toronto, Ontario, August 1977.

[93] A[ntonio] L. Furtado, K. C. Sevcik, and C. S. dos Santos. Permitting updates through views of data bases. *Information Systems*, 4(4):269–283, December 1979.

[94] César Galindo-Legaria. *Algebraic Optimization of Outer Join Queries.* PhD thesis, University of Wisconsin, Madison, Wisconsin, June 1992.

[95] César Galindo-Legaria. Outerjoins as disjunctions. In ACM SIGMOD *International Conference on Management of Data*, pages 348–358, Minneapolis, Minnesota, May 1994. Association for Computing Machinery.

[96] César Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection: Why use transformations? In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994. Morgan-Kaufmann.

[97] César Galindo-Legaria and Arnon Rosenthal. How to extend a conventional query optimizer to handle one- and two-sided outerjoin. In *Proceedings, Eighth* IEEE *International Conference on Data Engineering*, pages 402–409, Tempe, Arizona, February 1992. IEEE Computer Society Press.

[98] César Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. ACM *Transactions on Database Systems*, 22(1):43–74, March 1997.

[99] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Uniformly-distributed random generation of join orders. In *Proceedings of the Fifth International Conference on Database Theory*—ICDT *'95*, pages 280–293, Prague, Czech Republic, January 1995. Springer-Verlag.

[100] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In ACM SIGMOD *International Conference on Management of Data*, pages 9–18, San Diego, California, June 1992. Association for Computing Machinery.

[101] Richard A. Ganski and Harry K. T. Wong. Optimization of nested queries revisited. In ACM SIGMOD *International Conference on Management of Data*, pages 23–33, San Francisco, California, May 1987.

[102] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, New York, 1979.

[103] Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theoretical Computer Science*, 26(1):149–195, 1983.

[104] Seymour Ginsburg and Richard Hull. Sort sets in the relational model. *Journal of the* ACM, 33(3):465–488, July 1986.

[105] Robert Godin and Rokia Missaoui. Semantic query optimization using inter-relational functional dependencies. In Jr. Jay F. Nunamaker, editor, *Proceedings of the 24th Annual Hawaii International Conference on Systems Sciences*, volume 3, pages 368–375. IEEE Computer Society Press, January 1991.

[106] Piyush Goel and Bala[krishna] Iyer. SQL query optimization: Reordering for a general class of queries. In ACM SIGMOD *International Conference on Management of Data*, pages 47–56, Montréal, Québec, June 1996. Association for Computing Machinery.

[107] Goetz Graefe. Query evaluation techniques for large databases. ACM *Computing Surveys*, 25(2):73–170, June 1993.

[108] Goetz Graefe. Volcano, an extensible and parallel query evaluation system. IEEE *Transactions on Knowledge and Data Engineering*, 6(1):120–135, January 1994.

[109] Goetz Graefe and Richard L. Cole. Fast algorithms for universal quantification in large databases. ACM *Transactions on Database Systems*, 20(2):187–236, June 1995.

[110] G[oetz] Graefe, R[ichard] L. Cole, D[iane]. L. Davison, W. J. McKenna, and R. H. Wolniewicz. Extensible query optimization and parallel execution in VOLCANO. In Johann Christoph Freytag, David Maier, and Gottfried Vossen, editors, *Query Processing For Advanced Database Systems*, pages 305–335. Morgan-Kaufmann, San Mateo, California, 1994.

[111] Goetz Graefe and David J. Dewitt. The EXODUS optimizer generator. In ACM SIGMOD *International Conference on Management of Data*, pages 160–172, San Francisco, California, May 1987.

[112] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings, Ninth* IEEE *International Conference on Data Engineering*, pages 209–218. IEEE Computer Society Press, April 1993.

[113] Gösta Grahne. Dependency satisfaction in databases with incomplete information. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 37–45, Singapore, August 1984. VLDB Endowment.

[114] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings, Thirteenth* IEEE *International Conference on Data Engineering*, pages 444–453, Birmingham, U. K., April 1997. IEEE Computer Society Press.

[115] John Grant. Null values in a relational database. *Information Processing Letters*, 6(5):156–157, October 1977.

[116] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[117] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 358–369, Zurich, Switzerland, September 1995. Morgan Kaufmann.

[118] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In P. Apers, M. Bouzeghoub, and G[eorges] Gardarin, editors, *Advances in Database Technology*—EDBT'96 *(Proceedings of the 5th International Conference on Extending Database Technology)*, pages 140–144, Avignon, France, March 1996. Springer-Verlag.

[119] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. IEEE *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[120] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In ACM SIGMOD *International Conference on Management of Data*, pages 157–166, Washington, D.C., May 1993.

[121] Laura M. Haas, J[ohann] C[hristoph] Freytag, G[uy] M. Lohman, and H[amid] Pirahesh. Extensible query processing in STARBURST. In ACM SIGMOD *International Conference on Management of Data*, pages 377–388, Portland, Oregon, June 1989.

[122] Patrick A. V. Hall. Optimization of single expressions in a relational data base system. IBM *Journal of Research and Development*, 20(3):244–257, May 1976.

[123] Eric N. Hanson. A performance analysis of view materialization strategies. In ACM SIGMOD *International Conference on Management of Data*, pages 440–452, San Francisco, California, May 1987.

[124] Waqar Hasan and Hamid Pirahesh. Query rewrite optimization in STARBURST. Research Report RJ6367, IBM Corporation, Research Division, San Jose, California, August 1988.

[125] Joseph M. Hellerstein. Practical predicate placement. In ACM SIGMOD *International Conference on Management of Data*, pages 325–335, Minneapolis, Minnesota, May 1994.

[126] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. ACM *Transactions on Database Systems*, 23(2):113–157, June 1998.

[127] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In ACM SIGMOD *International Conference on Management of Data*, pages 423–434, Montréal, Québec, June 1996. Association for Computing Machinery.

[128] Peter Honeyman. Testing satisfaction of functional dependencies. *Journal of the* ACM, 29(3):668–677, July 1982.

[129] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. ACM *Transactions on Database Systems*, 9(3):482–502, September 1984.

[130] T[oshihide] Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16:95–97, February 1983.

[131] IBM Corporation, San Jose, California. IMS/ESA *Version 3 General Information*, first edition, June 1989. IBM Order Number GC26–4275–0.

[132] IBM Corporation, San Jose, California. IMS/ESA *Version 3 Database Administration Guide*, second edition, October 1990. IBM Order Number SC26–4281–1.

[133] IBM Corporation, San Jose, California. IMS/ESA *Version 3 Application Programming:* DL/I *Calls*, third edition, February 1993. IBM Order Number SC26–4274–2.

[134] Tomasz Imielinski and Witold Lipski, Jr. Incomplete information in relational databases. *Journal of the* ACM, 31(4):761–791, October 1984.

[135] Ingres Corporation, Alameda, California. INGRES/GATEWAY *to* IMS *User's Guide*, April 1993. Ingres document number 530–40–17804.

[136] International Standards Organization. *Information Technology—Database Language* SQL 2 *Draft Report*, December 1990. ISO Committee ISO/IEC JTC1/SC21.

[137] International Standards Organization. (ANSI/ISO) *Working Draft*, SQL *Foundation*, April 1997. ISO Committee ISO/IEC JTC1/SC21/WG3.

[138] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In ACM SIGMOD *International Conference on Management of Data*, pages 312–321, Atlantic City, New Jersey, May 1990.

[139] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. ACM *Transactions on Database Systems*, 20(3):288–324, December 1995.

[140] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In ACM SIGMOD *International Conference on Management of Data*, pages 9–22, San Francisco, California, May 1987.

[141] G[iuseppe] F. Italiano. Amortized efficiency of a path retrieval structure. *Theoretical Computer Science*, 48(2–3):273–281, 1986.

[142] Matthias Jarke, Jim Clifford, and Yannis Vassiliou. An optimizing PROLOG front-end to a relational query system. In ACM SIGMOD *International Conference on Management of Data*, pages 296–306, Boston, Massachusetts, June 1984.

[143] Matthias Jarke and Jürgen Koch. Query optimization in database systems. ACM *Computing Surveys*, 16(2):111–152, June 1984.

[144] Mattias Jarke. Common subexpression isolation in multiple query optimization. In Kim et al. [159], pages 191–205.

[145] D. S. Johnson and A[nthony] Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 164–169, Los Angeles, California, March 1982. Association for Computing Machinery.

[146] Yahiko Kambayashi, Masatoshi Yoshikawa, and Shuzo Yajima. Query processing for distributed databases using generalized semi-joins. In ACM SIGMOD *International Conference on Management of Data*, pages 151–160, Orlando, Florida, June 1982.

[147] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 154–163, Austin, Texas, May 1985.

[148] Arthur M. Keller. Updating relational databases through views. Technical Report CS–85–1040, Stanford University, Palo Alto, California, February 1985.

[149] Arthur M. Keller. The role of semantics in translating view updates. IEEE *Computer*, 19(1):63–73, January 1986.

[150] Alfons Kemper, Christoph Kilger, and G[uido] Moerkotte. Function materialization in object bases. In ACM SIGMOD *International Conference on Management of Data*, pages 258–267, Denver, Colorado, May 1991. Association for Computing Machinery.

[151] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases: Design, realization, and evaluation. IEEE *Transactions on Knowledge and Data Engineering*, 6(4):587–608, August 1994.

[152] Alfons Kemper and Guido Moerkotte. Access support in object bases. In ACM SIGMOD *International Conference on Management of Data*, pages 364–374, Atlantic City, New Jersey, May 1990. Association for Computing Machinery.

[153] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 290–301, Brisbane, Australia, August 1990. Morgan Kaufmann.

[154] Alfons Kemper and Guido Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[155] Werner Kiessling. On semantic reefs and efficient processing of correlation queries with aggregates. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 241–249, Stockholm, Sweden, August 1985.

[156] Won Kim. A new way to compute the product and join of relations. In ACM SIGMOD *International Conference on Management of Data*, pages 179–187, Santa Monica, California, May 1980. Association for Computing Machinery.

[157] Won Kim. On optimizing an SQL-like nested query. Research Report RJ3083, IBM Corporation, Research Division, San Jose, California, February 1981. See also ACM *Transactions on Database Systems*, 7(3), September 1982.

[158] Won Kim. On optimizing an SQL-like nested query. ACM *Transactions on Database Systems*, 7(3), September 1982.

[159] Won Kim, David S. Reiner, and D. S. Batory, editors. *Query Processing in Database Systems*. Springer-Verlag, Berlin, Germany, 1985.

[160] Jonathan J. King. QUIST–A system for semantic query optimization in relational databases. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 510–517, Cannes, France, September 1981. IEEE Computer Society Press.

[161] Jonathan J. King. *Query Optimization by Semantic Reasoning*. UMI Research Press, Ann Arbor, Michigan, 1984.

[162] Anthony Klug. Calculating constraints on relational expressions. ACM *Transactions on Database Systems*, 5(3):260–290, September 1980.

[163] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the* ACM, 29(3):699–717, July 1982.

[164] Anthony Klug and Rod Price. Determining view dependencies using tableaux. ACM *Transactions on Database Systems*, 7(3):361–380, September 1982.

[165] Robert Philip Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserve University, Cleveland, Ohio, September 1980.

[166] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, August 1986. Morgan Kaufmann.

[167] Sukhamay Kundu. An improved algorithm for finding a key of a relation. In *Proceedings,* ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 189–192, Austin, Texas, May 1985.

[168] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Kloster Banz/Staffelstein, Germany, June 1987. Springer-Verlag.

[169] Rom Langerak. View updates in relational databases with an independent scheme. ACM *Transactions on Database Systems*, 15(1):40–66, March 1990.

[170] Per-Åke Larson. *Relational Access to* IMS *Databases: Gateway Structure and Join Processing.* University of Waterloo, Waterloo, Ontario, Canada, December 1990. Unpublished manuscript, 70 pages.

[171] Per-Åke Larson. *Query Optimization for* IMS *Databases: General Approach.* University of Waterloo, Waterloo, Ontario, Canada, January 1991. Unpublished manuscript, 15 pages.

[172] Per-Åke Larson. SQL *Gateway for* IMS, *Version 1.6: User's Guide.* University of Waterloo, Waterloo, Ontario, Canada, October 1991. Unpublished manuscript, 36 pages.

[173] Per-Åke Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical report, Microsoft Corporation, Redmond, Washington, January 1998.

[174] Per-Åke Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 259–269, Stockholm, Sweden, August 1985.

[175] Per-Åke Larson and H. Z. Yang. Computing queries from derived relations: Theoretical foundation. Research Report 87–35, University of Waterloo, Waterloo, Ontario, Canada, August 1987.

[176] Mark Levene. A lattice view of functional dependencies in incomplete relations. *Acta Cybernetica*, 12:181–207, 1995.

[177] Mark Levene and George Loizou. The additivity problem for functional dependencies in incomplete relations. *Acta Informatica*, 34(2):135–149, 1997.

[178] Mark Levene and George Loizou. Null inclusion dependencies in relational databases. *Information and Computation*, 136(2):67–108, 1997.

[179] Mark Levene and George Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theoretical Computer Science*, 206(1–2):283–300, 1998.

[180] Mark Levene and George Loizou. Database design for incomplete relations. ACM *Transactions on Database Systems*, 24(1):80–126, 1999.

[181] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 96–107, Santiago, Chile, September 1994. Morgan Kaufmann.

[182] Leonid Libkin. *Aspects of Partial Information in Databases.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, 1994.

[183] Y. Edmund Lien. Multivalued dependencies with null values in relational databases. In *Proceedings of the 5th International Conference on Very Large Data Bases*, pages 61–66, Rio de Janeiro, Brazil, October 1979. IEEE Computer Society Press.

[184] Y. Edmund Lien. On the equivalence of database models. *Journal of the* ACM, 29(2):333–362, April 1982.

[185] Tok-Wang Ling. *Improving Data Base Integrity Based on Functional Dependencies.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1978.

[186] Guy Lohman, C. Mohan, Laura Haas, et al. Query processing in R*. In Kim et al. [159], pages 31–47. Also published as IBM Research Report RJ4272.

[187] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In ACM SIGMOD *International Conference on Management of Data*, pages 18–27, Chicago, Illinois, June 1988.

[188] Guy M. Lohman, Dean Daniels, Laura M. Haas, et al. Optimization of nested queries in a distributed relational database. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 403–415, Singapore, August 1984. VLDB Endowment. Also published as IBM Research Report RJ4760.

[189] James J. Lu, Guido Moerkotte, Joachim Schue, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In ACM SIGMOD *International Conference on Management of Data*, pages 340–351, San Jose, California, May 1995.

[190] Wei Lu and Jiawei Han. Distance-associated join indices for spatial range search. In *Proceedings, Eighth* IEEE *International Conference on Data Engineering*, pages 284–292, Tempe, Arizona, February 1992. IEEE Computer Society Press.

[191] Cláudio L. Lucchesi and Sylvia L. Osborn. Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–279, October 1978.

[192] David Maier. Minimum covers in the relational database model. *Journal of the* ACM, 27(4):664–674, October 1980.

[193] David Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[194] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. ACM *Transactions on Database Systems*, 4(4):455–469, December 1979.

[195] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *Data & Knowledge Engineering*, 12(1):83–99, February 1994.

[196] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. ACM *Computing Surveys*, 20(3):191–221, September 1988.

[197] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 316–323, Stockholm, Sweden, August 1985.

[198] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1(3):337–360, 1986.

[199] Claudia Maria Bauzer Medeiros. A validation tool for designing database views that permit updates. Technical Report CS–85–44, University of Waterloo, Waterloo, Ontario, Canada, November 1985.

[200] Jim Melton and Alan R. Simon. *Understanding the New* SQL*: A Complete Guide.* Morgan-Kaufmann, San Mateo, California, 1993.

[201] Alberto O. Mendelzon and David Maier. Generalized mutual dependencies and the decomposition of database relations. In *Proceedings of the 5th International Conference on Very Large Data Bases*, pages 75–82, Rio de Janeiro, Brazil, October 1979. IEEE Computer Society Press.

[202] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[203] Priti Mishra and Margaret H. Eich. Join processing in relational databases. ACM *Computing Surveys*, 24(1):63–113, March 1992.

[204] Roika Missaoui and Robert Godin. The implication problem for inclusion dependencies: A graph approach. ACM SIGMOD *Record*, 19(1):36–40, March 1990.

[205] R[okia] Missaoui and R[obert] Godin. Semantic query optimization using generalized functional dependencies. Rapport de Recherche 98, Université du Québec à Montréal, Montréal, Québec, September 1989.

[206] John C. Mitchell. Inference rules for functional and inclusion dependencies. In *Proceedings*, ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 58–69, Atlanta, Georgia, March 1983. Association for Computing Machinery.

[207] C. Mohan, Don Haderle, Yun Wang, and Josephine Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology*—EDBT'90 *(Proceedings of the 2nd International Conference on Extending Database Technology)*, pages 29–43. Springer-Verlag, Venice, Italy, March 1990.

[208] Shinichi Morishita. Avoiding Cartesian products for multiple joins. *Journal of the* ACM, 44(1):57–85, January 1997.

[209] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In ACM SIGMOD *International Conference on Management of Data*, pages 247–258, Atlantic City, New Jersey, May 1990. Association for Computing Machinery.

[210] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 264–277, Brisbane, Australia, August 1990. Morgan Kaufmann.

[211] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 77–85, Amsterdam, The Netherlands, August 1989. Morgan Kaufmann.

[212] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 91–102, Vancouver, British Columbia, August 1992. Morgan Kaufmann.

[213] Ryohei Nakano. Translation with optimization from relational calculus to relational algebra having aggregate functions. ACM *Transactions on Database Systems*, 15(4):518–557, December 1990.

[214] M. Negri, G. Pelagatti, and L. Sbattella. The effect of three-valued predicates on the semantics and equivalence of SQL queries. Rapporto Interno 85–27, Politecnico di Milano, Milan, Italy, 1985.

[215] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. Rapporto Interno 89–069, Politecnico di Milano, Milan, Italy, 1989.

[216] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. ACM *Transactions on Database Systems*, 16(3):513–534, September 1991.

[217] Wilfred Ng. Ordered functional dependencies in relational databases. *Information Systems*, 24(7):535–554, 1999.

[218] J. M. Nicolas. First-order logic formalization for functional, multivalued, and mutual dependencies. In ACM SIGMOD *International Conference on Management of Data*, pages 40–46, Austin, Texas, May 1978.

[219] Patrick O'Neil. *Database: Principles, Programming, Performance.* Morgan-Kaufmann, San Francisco, California, 1994.

[220] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. ACM SIGMOD *Record*, 24(3):8–11, September 1995.

[221] K. Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very*

Large Data Bases, pages 314–325, Brisbane, Australia, August 1990. Morgan Kaufmann.

[222] Oracle Corporation, Belmont, California. SQL*CONNECT *to* IMS *Installation and System Administration Guide*, July 1991. Oracle part number 5324–V1.0.

[223] G[ultekin] Ozsoyoğlu, Z. M[eral] Ozsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. ACM *Transactions on Database Systems*, 12(4):566–592, December 1987.

[224] M. Tamer Ozsu and David J. Meechan. Finding heuristics for processing selection queries in relational database systems. *Information Systems*, 15(3):359–373, 1990.

[225] M. Tamer Ozsu and David J. Meechan. Join processing heuristics in relational database systems. *Information Systems*, 15(4):429–444, 1990.

[226] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[227] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings, Fourth* IEEE *International Conference on Data Engineering*, pages 311–319, Los Angeles, California, 1988. IEEE Computer Society Press.

[228] G. N. Paulley and Per-Åke Larson. Exploiting uniqueness in query optimization. In *Proceedings, Tenth* IEEE *International Conference on Data Engineering*, pages 68–79, Houston, Texas, February 1994. IEEE Computer Society Press.

[229] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 306–315, Athens, Greece, August 1997. Morgan-Kaufmann.

[230] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in STARBURST. In ACM SIGMOD *International Conference on Management of Data*, pages 39–48, San Diego, California, June 1992. Association for Computing Machinery.

[231] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. A rule engine for query transformation in STARBURST and IBM DB2 C/S DBMS. In *Proceedings, Thirteenth* IEEE *International Conference on Data Engineering*, pages 391–400, Birmingham, U. K., April 1997. IEEE Computer Society Press.

[232] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, Boston, Massachusetts, 1998.

[233] Darrell R. Raymond. Partial order databases. Technical Report CS-96-02, University of Waterloo, Waterloo, Ontario, Canada, February 1996.

[234] Daniel J. Rosenkrantz and Harry B. Hunt, III. Processing conjunctive predicates and queries. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 64–72, Montréal, Québec, October 1980. IEEE Computer Society Press.

[235] Arnon Rosenthal and César Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In ACM SIGMOD *International Conference on Management of Data*, pages 291–299, Atlantic City, New Jersey, May 1990. Association for Computing Machinery.

[236] Arnon Rosenthal and David Reiner. An architecture for query optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 246–255, Orlando, Florida, June 1982.

[237] Arnon Rosenthal and David S. Reiner. Extending the algebraic framework of query processing to handle outerjoins. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 334–343, Singapore, August 1984. VLDB Endowment.

[238] Doron Rotem. Spatial join indices. In *Proceedings, Seventh* IEEE *International Conference on Data Engineering*, pages 500–509, Kobe, Japan, April 1991. IEEE Computer Society Press.

[239] Nicholas Roussopoulos. View indexing in relational databases. ACM *Transactions on Database Systems*, 7(2):258–290, June 1982.

[240] Nicholas Roussopoulos. An incremental access method for VIEWCACHE: Concept, algorithms, and cost analysis. ACM *Transactions on Database Systems*, 16(3):535–563, September 1991.

[241] Nicholas Roussopoulos, Nikos Economou, and Antony Stamenas. ADMS: A testbed for incremental access methods. IEEE *Transactions on Knowledge and Data Engineering*, 5(5):762–774, October 1993.

[242] Nick Roussopoulos. Overview of ADMS: A high performance database management system. In *Proceedings,* IEEE *Fall Joint Computer Conference*, pages 452–460, Dallas, Texas, 1987. IEEE Computer Society Press.

[243] Fereidoon Sadri and Jeffrey D. Ullman. A complete axiomatization for a large class of dependencies in relational databases. In *Proceedings, Twelfth Annual* ACM *Symposium on the Theory of Computing*, pages 117–122, Los Angeles, California, April 1980. Association for Computing Machinery.

[244] Y. Sagiv. Quadratic algorithms for minimizing joins in restricted relational expressions. SIAM *Journal on Computing*, 12(2):316–328, May 1983.

[245] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the* ACM, 27(4):633–655, October 1980.

[246] Hossein Saiedian and Thomas Spencer. An efficient algorithm to compute the candidate keys of a relational database schema. *The Computer Journal*, 39(2):124–132, April 1996.

[247] P[atricia] Griffiths Selinger, M. M. Astrahan, D[onald] D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In ACM SIGMOD *International Conference on Management of Data*, pages 23–34, Boston, Massachusetts, May 1979.

[248] Timos K. Sellis. Global query optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 191–205, Washington, D.C., May 1986.

[249] Timos K. Sellis. Efficiently supporting procedures in relational database systems. In ACM SIGMOD *International Conference on Management of Data*, pages 278–291, San Francisco, California, May 1987.

[250] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–185, 1988.

[251] Timos K. Sellis. Multiple-query optimization. ACM *Transactions on Database Systems*, 13(1):23–52, March 1988.

[252] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. IEEE *Transactions on Knowledge and Data Engineering*, 2(2):262–266, June 1990.

[253] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In ACM SIGMOD *International Conference on Management of Data*, pages 435–446, Montréal, Québec, June 1996. Association for Computing Machinery.

[254] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *Proceedings, Twelfth* IEEE *International Conference on Data Engineering*, pages 450–458, New Orleans, Louisiana, February 1996. IEEE Computer Society Press.

[255] Shashi Shekhar, Jaideep Srivastava, and Soumitra Dutta. A formal model of trade-off between optimization and execution costs in semantic query optimization. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 457–467, New York, New York, August 1988. Morgan Kaufmann.

[256] Eugene Shekita. High-performance implementation techniques for next-generation database systems. Technical Report 1026, University of Wisconsin, Madison, Wisconsin, May 1991.

[257] Sreekumar T. Shenoy and Z. Meral Ozsoyoğlu. A system for semantic query optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 181–195, San Francisco, California, May 1987.

[258] Sreekumar T. Shenoy and Z. Meral Ozsoyoğlu. Design and implementation of a semantic query optimizer. IEEE *Transactions on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[259] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, March 1994.

[260] Michael Siegel, Edward Sciore, and Sharon Salveter. A method for automatic rule derivation to support semantic query optimization. ACM *Transactions on Database Systems*, 17(4):563–600, December 1992.

[261] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 57–67, Montréal, Québec, June 1996. Association for Computing Machinery.

[262] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In P. Apers, M. Bouzeghoub, and G[eorges] Gardarin, editors, *Advances in Database Technology*—EDBT'96 *(Proceedings of the 5th International Conference on Extending Database Technology)*, pages 625–628, Avignon, France, March 1996. Springer-Verlag.

[263] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Communications of the* ACM, 18(10):568–579, October 1975.

[264] Rolf Socher. Optimizing the clausal normal form transformation. *Journal of Automated Reasoning*, 7(3):325–336, September 1991.

[265] Hennie J. Steenhagen, Peter M. G. Apers, and Henk M. Blanken. Optimization of nested queries in a complex object model. In Mattias Jarke, Janis Bubenko, and Keith Jeffery, editors, *Advances in Database Technology—*EDBT'94 *(Proceedings of the 4th International Conference on Extending Database Technology)*, pages 337–350. Springer-Verlag, Cambridge, United Kingdom, March 1994.

[266] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The* VLDB *Journal*, 6(3):191–208, August 1997.

[267] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1st International Conference on Very Large Data Bases*, pages 65–78, San Jose, California, May 1975. IEEE Computer Society Press.

[268] Michael Stonebraker and Joseph Kalash. TIMBER: A sophisticated relation browser. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 1–10, Mexico City, Mexico, September 1982. VLDB Endowment.

[269] Wei Sun and Clement T. Yu. Semantic query optimization for tree and chain queries. IEEE *Transactions on Knowledge and Data Engineering*, 6(1):136–151, February 1994.

[270] Arun Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In ACM SIGMOD *International Conference on Management of Data*, Portland, Oregon, June 1989.

[271] Arun Swami and Anoop Gupta. Optimization of large join queries. In ACM SIGMOD *International Conference on Management of Data*, pages 8–17, Chicago, Illinois, June 1988.

[272] Arun Swami and Bala[krishna] Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings, Ninth* IEEE *International Conference on Data Engineering*, pages 345–354. IEEE Computer Society Press, April 1993.

[273] V[u] D[uc] Thi. Minimal keys and antikeys. *Acta Cybernetica*, 7(4):361–371, August 1986.

[274] Frank Wm. Tompa and José A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[275] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *The* VLDB *Journal*, 5(2):101–118, April 1996.

[276] D. C. Tsichritzis and F. H. Lochovsky. Hierarchical data-base management: A survey. ACM *Computing Surveys*, 8(1):105–123, March 1976.

[277] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, Rockville, Maryland, 1988.

[278] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2.* Computer Science Press, Rockville, Maryland, 1989.

[279] Patrick Valduriez. Optimization of complex database queries using join indices. IEEE *Data Engineering Bulletin*, 9(4):10–16, December 1986.

[280] Patrick Valduriez. Join indices. ACM *Transactions on Database Systems*, 12(2):218–246, June 1987.

[281] M. F. van Bommel and G[rant] E. Weddell. Reasoning about equations and functional dependencies on complex objects. IEEE *Transactions on Knowledge and Data Engineering*, 6(3):455–469, June 1994.

[282] H. J. A. van Kuijk. The application of constraints in query optimization. Memoranda Informatica 88–55, Universiteit Twente, Enschede, The Netherlands, 1988.

[283] H. J. A. van Kuijk, F. H. E. Pijpers, and P. M. G. Apers. Semantic query optimization in distributed databases. In S. G. Akl, F. Fiala, and W. W. Koczkodaj, editors, *International Conference Proceedings of Advances in Computing and Information*—ICCI *'90*, pages 295–303, Niagara Falls, Ontario, 1990. Springer-Verlag.

[284] Bennet Vance and David Maier. Rapid bushy join-order optimization with Cartesian products. In ACM SIGMOD *International Conference on Management of Data*, pages 35–46, Montréal, Québec, June 1996. Association for Computing Machinery.

[285] Brad T. Vander Zanden, Howard M. Taylor, and Dina Bitton. Estimating block accesses when attributes are correlated. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 119–127, Kyoto, Japan, August 1986. Morgan Kaufmann.

[286] Yannis Vassiliou. Null values in data base management–A denotational semantics approach. In ACM SIGMOD *International Conference on Management of Data*, pages 162–169, Boston, Massachusetts, May 1979.

[287] Yannis Vassiliou. Functional dependencies and incomplete information. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 260–269, Montréal, Québec, October 1980.

[288] Günter von Bültzingsloewen. Translating and optimizing SQL queries having aggregates. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 235–243, Brighton, England, August 1987. Morgan Kaufmann.

[289] Min Wang, Jeffery Scott Vitter, and Bala[krishna] Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proceedings, Thirteenth* IEEE *International Conference on Data Engineering*, pages 169–180, Birmingham, U. K., April 1997. IEEE Computer Society Press.

[290] Yalin Wang. Transforming normalized Boolean expressions into minimal normal forms. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1992.

[291] Eugene Wong and Karel Youssefi. Decomposition—A strategy for query processing. ACM *Transactions on Database Systems*, 1(3):223–241, September 1976.

[292] Zhaohui Xie and Jiawei Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 522–533, Santiago, Chile, September 1994. Morgan Kaufmann.

[293] G. Ding Xu. Search control in semantic query optimization. Research Report TR–83–09, University of Massachusetts, Amherst, Massachusetts, 1983.

[294] Weipeng P. Yan. *Query Optimization Techniques for Aggregation Queries.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, September 1995.

[295] Weipeng P. Yan and Per-Åke Larson. Performing group by before join. In *Proceedings, Tenth* IEEE *International Conference on Data Engineering*, pages 89–100, Houston, Texas, February 1994. IEEE Computer Society Press.

[296] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 345–357, Zurich, Switzerland, September 1995. Morgan Kaufmann.

[297] H. Z. Yang and Per-Åke Larson. Query transformation for PSJ-queries. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 245–254, Brighton, England, August 1987. Morgan Kaufmann.

[298] S. B[ing] Yao. Approximating block accesses in database organizations. *Communications of the* ACM, 20(4):260–261, April 1977.

[299] Clement T. Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Applications.* Morgan-Kaufmann, San Francisco, California, 1998.

[300] Carlo Zaniolo. Database relations with null values. In *Proceedings,* ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems*, pages 27–33, Los Angeles, California, March 1982. Association for Computing Machinery.

[301] Carlo Zaniolo. The database language GEM. In ACM SIGMOD *International Conference on Management of Data*, pages 207–218, San Jose, California, May 1983. Association for Computing Machinery.

[302] Carlo Zaniolo. Database relations with null values. *Journal of Computer and System Sciences*, 28(1):142–166, February 1984.

[303] Fred Zemke. Cleanup of functional dependencies. Unpublished manuscript, 50 pages. ANSI ISO/IEC JTC1/SC32 WG3 change proposal FRA-036R1., November 1998.

List of Notation

Italicized page numbers indicate the location of definitions.

Index

Italicized page numbers indicate the location of definitions.

T

U

V

W