Scalpel: Optimizing Query Streams Using Semantic Prefetching

by

Ivan Thomas Bowman

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, 2005

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Client applications submit streams of relational queries to database servers. For simple requests, inter-process communication costs account for a significant portion of user-perceived latency. This trend increases with faster processors, larger memory sizes, and improved database execution algorithms, and this trend is not significantly offset by improvements in communication bandwidth.

Caching and prefetching are well studied approaches to reducing user-perceived latency. Caching is useful in many applications, but it does not help if future requests rarely match previous requests. Prefetching can help in this situation, but only if we are able to predict future requests. This prediction is complicated in the case of relational queries by the presence of request parameters: a prefetching algorithm must predict not only a query that will be executed in the future, but also the actual parameter values that will be supplied.

We have found that, for many applications, the streams of submitted queries contain patterns that can be used to predict future requests. Further, there are correlations between results of earlier requests and actual parameter values used in future requests. We present the Scalpel system, a prototype implementation that detects these patterns of queries and optimizes request streams using context-based predictions of future requests.

Scalpel uses its predictions to provide a form of semantic prefetching, which involves combining a predicted series of requests into a single request that can be issued immediately. Scalpel's semantic prefetching reduces not only the latency experienced by the application but also the total cost of query evaluation. We describe how Scalpel learns to predict optimizable request patterns by observing the application's request stream during a training phase. We also describe the types of query pattern rewrites that Scalpel's cost-based optimizer considers. Finally, we present empirical results that show the costs and benefits of Scalpel's optimizations.

We have found that even when an application is well suited for its original configuration, it may behave poorly when moving to a new configuration such as a wireless network. The optimizations performed by Scalpel take the current configuration into account, allowing it to select strategies that give good performance in a wider range of configurations.

Acknowledgements

This document marks a milestone in a course of research that began in the fall of 2000, when I had a long conversation with Kenneth Salem about problematic client applications that had been plaguing my working days. I'd like to thank Ken for provoking me to formalize proposed solutions and for helping me find a thesis-sized research problem within an unruly cloud of ideas. I owe Ken a debt of gratitude for his guidance, encouragement, and understanding. His help has been instrumental in completing this work.

I would also like to thank the other members of my examining committee—Kostas Kontogiannis, Jeffrey Naughton, Tamer Özsu, and David Toman—for their thoughtful comments, questions, and suggestions. This final text has been improved by changes based on their comments on earlier drafts.

Over the years, I have been fortunate to have the support of many people, and I would like to thank my family, friends, and colleagues for their help and best wishes. My co-workers at iAnywhere Solutions have provided useful ideas and feedback, and helped to give me the flexibility to complete my research while working full-time. In particular, I would like to thank my manager, Glenn Paulley, for providing me with challenging problems, useful advice, interesting references, and unflagging support. I'd also like to thank the other members of the `dev_qp` team for their encouragement and understanding during periods where I was working on multiple tasks: Mike Demko, Dan Farrar, Anil Goel, Anisoara Nica, and Matthew Young-Lai were unfailing in their support.

My friends have provided a source of strength and stability in my life, and I would like to thank them for the encouragement, support, and humour they have provided over the years. The `banter` list provided welcome distraction, as did occasional `events`.

My family has also contributed in uncountably many ways to the successful completion of this work. The Bowmans, Browns, Carters, Farleys, Richardsons, and Sharpes were often in my thoughts. My grandparents, Lloyd and Winnifred Richardson, provided me with model examples. My brother, Don, also provided me with advice and feedback on early versions of this research, and I thank him for that. In particular, I would like to thank my mom, Nancy, for inspiring me by her example, working hard to achieve her goals while still finding time to appreciate the world around her.

Finally, it is with love and gratitude that I thank my wife Sherri for helping to motivate me to start this course of study, then for being supportive and tolerant through many late nights, early mornings, and working vacations.

Dedication

For Dr. Lloyd T. Richardson

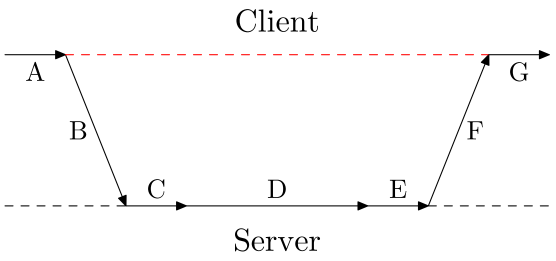
Contents

Tables

Figures

1 Introduction

Client applications submit streams of relational queries to database servers. For simple requests, inter-process communication costs account for a significant portion of user-perceived latency. This trend increases with faster processors, larger memory sizes, and improved database execution algorithms, and this trend is not significantly offset by improvements in communication bandwidth.



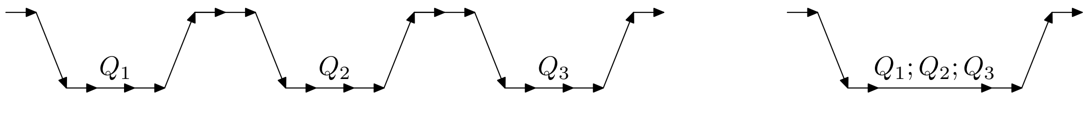
A Client prepares request Q.
B Request on wire.
C Server interprets request Q.
D Server executes Q.
E Server prepares reply R.
F Reply on wire.
G Client interprets reply R.

Figure 1.1: A sample communication trace.

Figure 1.1 shows a sample of a communication trace between a client program and a database server. Of the cost components shown, only D represents query processing costs, while the remainder consist of communication overhead. For simple requests, the time spent actually executing a request (component D) can be a relatively small portion of execution time. We have found that for simple queries, such as fetching a single row from an in-memory table, the execution time D represents less than 1.5% of the elapsed request time on a low-latency shared memory configuration; this percentage drops significantly with higher latency connections.

The problem of latency is becoming increasingly relevant due to a number of trends. We have seen exponential improvements in the number of requests that can be processed per second due to improvements in processor speed, memory size, disk throughput, network bandwidth, and query processing algorithms. During this same time period, the latency associated with these operations has not improved at anything near the same rate. Patterson [143] presents compelling evidence that latency has lagged bandwidth. Further, there is good reason to believe that this latency problem will not be solved to a significant extent by hardware advances. In addition to these trends that make latency increase relative to the other shrinking costs, new configurations such as wireless networks and high-latency WANs are increasingly being used for data access. Applications that work well in existing configurations may be unacceptably slow in these now environments merely due to the structure of their request streams.

The requests submitted by some database client applications to servers contain fine-grained access such as that shown in Figure 1.2(a). Requests that are individually cheap add up to significant exposed latency and processing costs associated with the overhead of formatting and interpreting each request. If these requests could be combined into a single request, we would eliminate the per-request costs, giving a coarse-grained access pattern as shown in Figure 1.2(b). This pattern not only reduces user-perceived latency, it can also reduce execution costs (components A, C, E and G), and it can even allow the server to choose a more efficient query execution plan. For example, the server might be able to exploit data sharing between multiple requests.



(a) Fine-Grained Access

(b) Coarse-Grained Access

Figure 1.2: The difference between fine-grained and coarse-grained communication.

Fine-grained access can be converted to coarse grained access by prefetching anticipated results before they are requested. While this runs the risk of wasting work in the case that the results are not in fact needed, it can reduce the latency and overhead associated with many small requests.

In a database setting, prefetching of future requests is not immediately possible. In general, requests are parameterized, and the values of the parameters may depend on the result of earlier requests.

Our hypothesis is that existing client applications generate stylized patterns in their request streams. We can implement an automated tool to detect these stylized patterns and prefetch anticipated future requests. The goal of this system, which we call Scalpel, is to reduce an objective function defined on a cost model. This objective could be the time an application spends waiting for database requests (exposed latency), or it could be the total time that the database server spends processing requests.

We will demonstrate techniques that can be used for this automated detection, optimizing, and rewriting system. We will evaluate the effectiveness of the techniques by considering their effect on both synthetic benchmarks and on case studies of real-world applications.

1.1 Motivation

As an illustration of the kind of optimization we hope to achieve, consider the client-side database application code shown in Figure 1.3.

```
function GET-OPEN-INVOICES(cust_id, currency)
  open c1 cursor for $Q_1$:
    SELECT id, curr, transdate
    FROM ar
    WHERE customer_id = :cust_id
    AND ar.curr = :currency
    AND NOT ar.amount = ar.paid
    ORDER BY id
  while r1 $\leftarrow$ fetch c1 do
    if currency $\neq$ defaultcurrency then
      rate $\leftarrow$ GET-EXCHANGE-RATE( r1.curr, r1.transdate )
  close c1
end
function GET-EXCHANGE-RATE(curr, transdate)
  fetch row r2 from $Q_2$:
    SELECT exchangerate FROM exchangerate
    WHERE curr=:curr AND transdate=:transdate
  return r2.exchangerate
end
```

Figure 1.3: The SQL-Ledger GET-OPEN-INVOICES function.

The code is adapted from SQL-Ledger, a web-based double-entry accounting system written in Perl (we describe this application further in Chapter 8). The GET-OPEN-INVOICES function retrieves a list of all open invoices for a given customer that were recorded with a given monetary currency. If the given currency differs from the configured system default, then the exchange rate for each invoice is retrieved using the GET-EXCHANGE-RATE function. When this application runs, it issues a series of small single-table queries ($Q_1, Q_2, Q_2, Q_2, \ldots$) to the database server. It uses the results of these queries to perform a two-way, nested loops join on the client side.

The individual requests submitted by GET-EXCHANGE-RATE may be quite cheap (for example, consisting of a single index lookup of an in-memory table). In this case, the execution time of these inner queries is likely dominated by the fixed overhead costs associated with opening a cursor. This suggests that better performance could be achieved by combining several of these small requests together, reducing the fixed overhead of many requests.

For example, if Scalpel recognizes the nested query pattern $Q_1, Q_2, Q_2, Q_2, Q_2, \ldots$ generated by the application in Figure 1.3, it can replace the entire pattern with a single, larger query similar to Q_{opt}, which is shown in Figure 1.4. Query Q_{opt} performs the join at the server and returns all of the data that would have been returned by Q_1 and the Q_2 queries.

```
SELECT id, curr, exchangerate, ...
FROM   ar LEFT JOIN exchangerate er
       ON ar.curr = er.curr AND ar.transdate=er.transdate
WHERE  customer_id = :cust_id
       AND ar.curr = :currency
       AND NOT ar.amount = ar.paid
ORDER BY id
```

Figure 1.4: The join query Q_{opt} combining Q_1 and Q_2.

```
function BATCH-EXAMPLE(cust_id, get_shipto)
  fetch row r3 from $Q_3$:
    SELECT name, tax_id, ship_id
    FROM customer c
    WHERE id = :cust_id
  if get_shipto then
    fetch row r4 from $Q_4$:
      SELECT s.addr
      FROM shipto s
      WHERE s.ship_id=:c3.ship_id AND s.default = 'Y'

  fetch row r5 from $Q_5$:
    SELECT tax_rate FROM tax t WHERE t.tax_id=:c3.tax_id
  return r5.t1
end
```

Figure 1.5: Example of a batch request pattern.

In addition to nested request patterns such as the one shown in Figure 1.3, we have found examples of what we call batch request patterns. A batch is a sequence of related queries. Figure 1.5 shows example code that generates a batch request pattern; this code is a simplified and modified version of functions we found in the SQL-Ledger application. In the BATCH-EXAMPLE function, two or three small queries are submitted to the database to retrieve different types of information about a customer. Each of these is individually cheap, leading to a fine-grained access pattern.

We could consider speculatively executing requests in order to avoid fine-grained access. For

```
SELECT c.name, c.tax_id, c.ship_id, s.addr
FROM   customer c LEFT JOIN shipto s
       ON s.ship_id = c.ship_id AND s.default = 'Y'
WHERE  c.id = :cust_id
```

Figure 1.6: Joining queries Q_3 and Q_4 with $Q_{3;4}$.

example, if we decide that query Q_4 follows Q_3 sufficiently often, we could submit a combined query $Q_{3;4}$ such as the one shown in Figure 1.6 when we see $\textsc{Open}(Q_3)$. This combined query generates the results needed for Q_3 *and* Q_4. The per-request costs are only incurred once in this case, although there is the risk that we will have executed Q_4 unnecessarily.

1.2 Using Prefetching

Scalpel's rewrites are based on *predictions* of future actions that will be performed by the client application. When Scalpel sees Q_1, it predicts that Q_1 will be followed by a series of nested Q_2 queries. Based on this prediction, it issues Q_{opt} to the server rather than Q_1. If the application then requests Q_2 as expected, Scalpel does not pass Q_2 to the server. Instead, it extracts the required data from the result of Q_{opt} and returns that to the application. If the application behaves unexpectedly, perhaps by issuing a different query Q_3, then Scalpel can simply forward Q_3 to the server for execution. In this case Scalpel has done some extra work, since Q_{opt} is a larger and more complex query than Q_1. However, Scalpel always returns correct results to the client. By replacing Q_1 with Q_{opt}, Scalpel implements a kind of prefetching. We call it *semantic prefetching* because Scalpel must understand the queries Q_1 and Q_2 in order to generate an appropriate Q_{opt}.

There are two reasons to do semantic prefetching. First, it provides the query optimizer at the server with more scope for optimization. For example, the application shown in Figure 1.3 effectively joins two tables at the client site. However, the server's optimizer will be unaware that the join is occurring. Scalpel's rewrite makes the server aware of the join, allowing its optimizer to consider alternative join methods that it may implement.

Second, by replacing many small queries with fewer larger queries, Scalpel can reduce the latency and overhead associated with the interconnection network and the layers of system interface and communications software at both ends of the connection. These costs can be quite significant. We measured the cost of fetching a single in-memory row from several commercial relational database management systems. Regardless of whether the client used local shared memory or an inter-city WAN to communicate with the server, overhead was consistently over 98.5% of the total query time. For the application shown in Figure 1.3, this means that almost all of the time spent issuing Q_2 queries to the server is overhead.

It may seem that the application developer should be responsible for avoiding nesting of the form shown in Figure 1.3. For example, the two functions shown in Figure 1.3 could be replaced by a single function that opens Q_{opt} (Figure 1.4). However, we have found that there is a place for both manual application tuning and automatic, run-time optimization of application request streams.

Manual tuning can clearly improve application performance, but run-time optimization has some strengths that application tuning does not. First, run-time optimization can take advantage of information that is not known at application development time, or that varies from installation to installation. For example, when the monetary currency of the report differs from the default currency, the implementation of the GET-OPEN-INVOICES function shown in Figure 1.3 is much worse than a revised implementation based on the join query of Figure 1.4. However, if the report currency and the default currency are the same, the implementation of Figure 1.3 will perform best. For which of these circumstances should the implementation be tuned? The programmer may not know the answer to this question; worse, the answer may be different for different instances of the program. Other examples of run-time information that may have a significant impact on the performance of the application are program parameter values, data distributions, and system parameters such as network latency. A run-time optimizer can consider these factors in deciding how best to interact with the database server.

A second argument in favor of run-time optimization is a software engineering argument. Performance is not the only issue to be considered when designing and implementing an application. For example, the SQL-Ledger application actually calls `get_exchangerate` from eight locations. Only one of these calls is shown in Figure 1.3. Rewriting `get_openinvoices` to use the join query of Figure 1.4 breaks the encapsulation of the exchange rate computation that was present in the original implementation, resulting in duplication of the application's exchange rate logic. This kind of duplication can lead to increased development cost and possible maintenance issues.

Finally, there is the issue of the time and effort required to tune applications. While we do not expect Scalpel to eliminate the need for manual application tuning, any performance tuning that can be accomplished automatically can reduce the manual tuning effort. Scalpel may be particularly beneficial for tuning automatically or semi-automatically generated application programs, for which there may be little or no opportunity for manual tuning.

1.3 Application Request Patterns

The example in Figure 1.3 shows one type of pattern that we have identified in database client code. We have found that there are several such types of patterns that we can consider optimizing. We surveyed a small set of database application programs to identify the kinds of query patterns they produce, and the prevalence of those patterns. Our sample included the following applications.

- **SQL-Ledger:** A web-based double-entry accounting system written in Perl.
- **Slaschode:** A web forum written in Perl.
- **Compière:** A Java-based ERP system.
- **TM:** A Java-based time-management GUI.
- **dbunload:** A C program that writes DDL to re-create the schema of a database.

In addition to these systems, we investigated a number of proprietary applications, ranging from on-line order processing systems to report generating systems. While non-disclosure agreements prevent us from giving details for those applications, the results of our analysis of those applications were consistent with the results from the applications listed above.

From our application sample, we identified three types of query patterns that are amenable to optimization: batches, nesting, and data structure correlations.

Batches A batch is a sequence of related queries. These individual queries could potentially be replaced with a single, larger query. For example, consider the BATCH-EXAMPLE function shown in Figure 1.5.

Nesting In batches, each query is opened, fetched, and closed independently. In the nesting pattern, one query is opened, and other queries are opened, executed, and closed for each row of the outer query. Figure 1.3 showed an example of an application that generates a nesting query pattern. The nesting pattern effectively implements a nested loops join in the application.

Data Structure Correlation In the nesting pattern, inner queries are executed while the outer query is open, expressing direct nesting in the application. In some cases, a client application opens an outer query, fetches the results into a data structure, then closes the outer query. Then, an inner query is executed for some or all of the values stored in the data structure. The performance impact of the data structure correlation pattern is similar to that of the nesting pattern. However, the pattern may be more difficult to detect due to the indirect nesting.

Batches were common in all of the applications we considered. Nesting or data structure correlation also occurred in each of the applications, although less frequently than batches. Although nesting and data structure correlation are less common than batches, the potential payoff for optimizing these patterns is greater because they usually allow more database requests to be eliminated. For example, the SQL-Ledger application in Figure 1.3 issues Q_2 once for each customer invoice. All of these queries can be replaced by a single query. Although both nesting and data structure correlation offer a large optimization payoff, nesting patterns are easier than data structure correlations to detect and optimize.

1.4 Thesis

The main hypothesis that we examine is the following. Modern database applications contain significant amounts of fine-grained access in their request streams. These fine grained requests contain dependencies that prevent existing prefetching approaches from being used. We can exploit the relational data model to efficiently recognize some of these types of patterns. Further, we can use the query processing capabilities of the DBMS to effectively execute combined requests that generate the needed results for prefetched queries. This process of semantic prefetching allows not only significant reductions in total exposed latency, but also it offers savings in the CPU costs associated with formatting and interpreting requests.

In this thesis, we present the Scalpel system, which detects patterns in the request streams of client applications and generates prefetching rules that reduce exposed latency. We use Scalpel to evaluate this main hypothesis.

Our work provides the following contributions:

1. We identify and characterize two types of fine-grained request patterns that appear in streams of requests to DBMSs.
2. We demonstrate effective techniques to automatically find these patterns.
3. We define and evaluate alternative strategies for prefetching the results of anticipated requests.

We will show in the sequel that the problem of fine-grained patterns is common, and that our automated detection and rewriting approach are effective at reducing the costs associated with this type of pattern.

1.5 Outline

The remainder of this document is organized as follows. Chapter 2 covers preliminaries of notation that will be used throughout the document. Chapter 3 describes how Scalpel detects and optimizes nested request patterns. Chapter 4 describes how Scalpel combines calibrated and observed timing information with server estimates to provide cost estimates for cost-based optimization. Chapter 5 describes how Scalpel recognizes and optimizes batch request patterns. Even though our prototype implements both nested and batch optimization, we have described them in isolation for simplicity. Chapter 6 gives details of how nesting and batch patterns could be combined in theory, and also describes the current (limited) combination used in the Scalpel prototype.

Scalpel fetches values before the client application submits an FETCH, which leads to concerns that prefetching may introduce temporal anomalies. Chapter 7 addresses this issue by showing how Scalpel ensures prefetch consistency.

Chapter 8 demonstrates the benefits of using Scalpel on real systems by presenting the results of two case studies. Finally, Chapter 9 positions our results within the body of related work, and Chapter 10 presents our conclusions and some topics for future study.

2 Preliminaries

2.1 Scalpel System Architecture

Scalpel operates in two phases. In the training phase (Figure 2.1), the Call Monitor monitors database interface calls made by the application (passing them unchanged to the DBMS-provided Interface Library). The Pattern Detector analyzes the monitored requests to build a model of the request patterns that have occurred so far. At the end of the training phase, the Pattern Optimizer uses this model to select an execution strategy for the observed requests. The Query Rewriter uses the selected strategy to generate a set of request rewrite rules, which are recorded for use at run-time. Scalpel's rewrites are condition/action rules. The conditions (called *contexts*) identify situations in which a query rewrite can be applied and the actions describe what activities Scalpel should perform at run-time when it observes a query being submitted in a particular context..

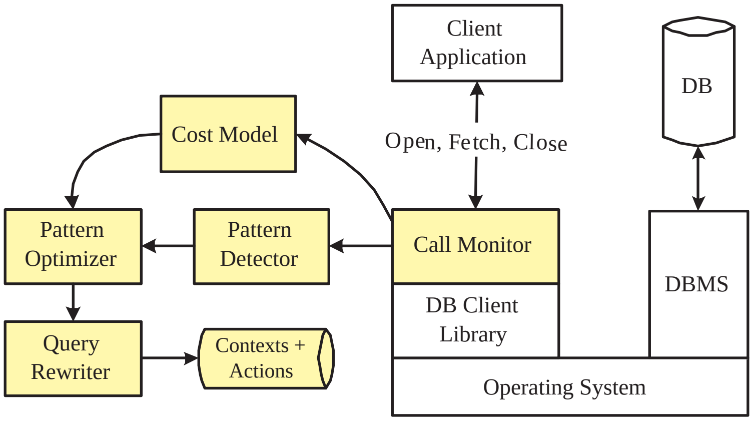


Figure 2.1: Scalpel system structure during training phase. Scalpel components are shaded.

At run-time (Figure 2.2), the Call Monitor again monitors the client request stream and tracks the current context. Each time the client opens a query, Scalpel checks the current context against the rules in the rewrite database. If the context matches a rule condition, Scalpel applies the associated action to rewrite the current query. For example, an alternate query could be submitted to fetch the original query's results and also prefetch additional results; alternatively, the query could be answered directly from prefetched data without consulting the DBMS.

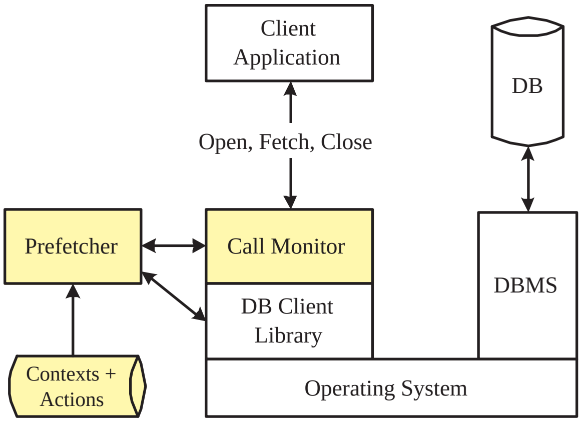


Figure 2.2: Scalpel system structure at run-time. Scalpel components are shaded.

2.2 Pseudo-Code Conventions

We use the following conventions in the pseudo-code of this document, which are generally based on the conventions used by Cormen, Leiserson and Rivest [47]:

1. Block structure is indicated by indentation.
2. The symbol ▷ is used to indicate that the rest of the line is a comment.
3. Variables are local to a procedure unless explicitly described as a global variable.
4. Arrays are accessed using the array name followed by a subscript in square brackets.
5. Lists are represented using comma-separated elements enclosed in square brackets. For example, the expression `[ a, b, c]` is a list with three elements. If an element of the square-bracket expression is a list, the contents are expanded to generate a single (flat) list. Lists are accessed using square brackets. Negative indexes are used to index from the end of the list, so `c[-1]=3` in the example below.
6. Tuples are represented with parentheses.
7. Strings are represented as double-quoted text. Concatenation is performed using the addition symbol (+). When concatenating a list to a string, the list is comma-separated.
8. Compound data types are formed of fields, and are described the using **structure** keyword. A new object of a given type is created using the **new** keyword, which can optionally specify initial values for fields in the order they are specified in the **structure** defini-

tion. If a value is not supplied in the **new** call, it has the default value that is given in the **structure** definition.

9. Fields are accessed by specifying an object name, a period (.), then the field name. For example, the expression `C.keys` represents the ‘keys’ field of object C.

```
structure SAMPLE-STRUCTURE
  flda = 0                   ▷ Field flda with default 0
  fldb = 1                   ▷ Field fldb with default 1
end
procedure SAMPLE-PROCEDURE
  ▷ This is a comment.
  a ← 1                              ▷ Set variable a to value 1
  o ← new SAMPLE-STRUCTURE           ▷ Set variable o to refer to a new object
  m ← NIL                            ▷ Set variable m to refer to no object
  if a = 1 then                      ▷ A conditional
    b ← [ a, 2 ]                     ▷ Set variable b to list [1,2]
  c ← [ b, 3 ]                       ▷ Set variable c to list [1,2,3]
  d ← “L:”+c                         ▷ Set variable d to string “L:1,2,3”
  o.flda ← d                         ▷ Set field flda of o’s object to string “L:1,2,3”
  m ← o                              ▷ Set variable m to refer to same object as o refers to
end
```

Figure 2.3: Sample pseudo-code.

2.3 Model of Request Streams

When client applications submit requests to a DBMS, they do so using API calls to a client DB library provided by the DBMS vendor. These calls are used to prepare SQL statements for later execution, to open cursors, fetch rows, and close cursors. The DB library implements these requests using inter-process communication with the DBMS. There is significant variation in the details of request streams processed by different DBMS products. Various implementations may be used even within individual products based on the settings of options. For example, prefetching of rows varies between products; within a single product, the details are affected by the selected cursor type. Some products implement a *lazy close* where a close request is not sent to the DBMS immediately, instead being tacked on to the next request. Other products close a cursor when the last row is fetched. Some products optimize queries when they are opened, others when they are prepared.

In order to make our results broadly applicable, we use a simplified representation of a request stream. We assume that the client request stream consists of the following types of requests:

CONNECT*:* A CONNECT request connects a client application to the database server. Scalpel monitors connections during training and run-time. In the training phase, Scalpel initializes data structures to monitor the application's requests in order to detect patterns. In the run-time phase, Scalpel loads stored condition/action pairs selected by an earlier training period.

DISCONNECT*:* A DISCONNECT request disconnects a client from the database server. Scalpel monitors DISCONNECT requests and releases any resources associated with the client's connection.

EXECUTE*:* An EXECUTE request is used to modify data stored in the DBMS (for example, by inserting, updating, or deleting a row). Scalpel monitors EXECUTE requests due to their impact on the consistency of prefetched results.

OPEN*:* An OPEN request is used to send a query to the database server. It returns a cursor, which is used by the application to retrieve rows from the query result. The first parameter of an OPEN request is the query text. Queries may be parameterized. If so, the OPEN request also includes a value for each query parameter.

FETCH*:* A FETCH request takes a cursor as an input parameter and either returns a single row of the query result to the application or returns EOF to indicate the end of the result set.

CLOSE*:* A CLOSE request takes a cursor as an input parameter. It is used by the application to indicate that it is finished retrieving rows from the query result.

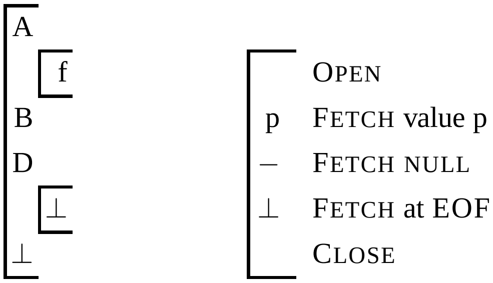
We assume that CONNECT and DISCONNECT requests are relatively infrequent and we typically do not shown them in examples unless they are relevant to a point of interest.

Figure 2.4(a) shows an example of a request stream that might be submitted by a client application. In this example, there are two different queries submitted: Q_1 and Q_2. Query Q_2 is always submitted while cursor c_1 is open over query Q_1. Only one FETCH is performed on cursor c_2, so it does not return EOF on its first invocation. On the second invocation (line 8-10), an empty result set is returned. We also use a *concise trace notation* for displaying a request trace, and an example of this is shown in Figure 2.4(b) for the requests of Figure 2.4(a). In the concise notation, we represent OPEN, FETCH and CLOSE requests with graphical symbols. Further, we show only the primary keys of the fetched results.

In some cases, we are interested only in a sequence of requests all at the same nesting level. In this case, we write a sequence such as $Q_\mathrm{a}, Q_\mathrm{b}, Q_\mathrm{c}, \ldots$. In this *sequence trace notation*, each Q_i represents OPEN(Q_i), some number of FETCH calls, then CLOSE(Q_i). Where the meaning is clear from the context, we use only the subscripts of the queries, for example writing abc to mean the sequence $Q_\mathrm{a}, Q_\mathrm{b}, Q_\mathrm{c}$.

#	Operation	Result
1	OPEN(Q_1,1)	c_1
2	FETCH(c_1)	$(A, 1)$
3	OPEN(Q_2,1)	c_2
4	FETCH(c_2)	(f)
5	CLOSE(c_2)	
6	FETCH(c_1)	$(B, 1)$
7	FETCH(c_1)	$(D, 1)$
8	OPEN(Q_2,1)	c_2
9	FETCH(c_2)	EOF
10	CLOSE(c_2)	
11	FETCH(c_1)	EOF
12	CLOSE(c_1)	

(a) Example trace (full)



(b) Example trace (concise)

(c) Legend

Figure 2.4: Example of a request trace in both (a) full form and (b) concise form.

2.4 Notation for Strings and Sequences

We use a notation for strings and sequences based on the usage of Hopcroft and Ullman [93]. This section outlines that notation.

A *sequence* is a string of atomic symbols juxtaposed. The meanings of the symbols vary between problem domains, and could be, for example, characters, page requests, or musical notes. Regardless of the actual symbols used in a problem domain, we represent symbols with the letters a, b, c, and d, with or without subscripts, while we use letters such as w, x, y, and z to denote strings. For example, $w = abca$ is a string. The length of a string w, denoted by $|w|$, is the number of symbols composing the string. The empty string is represented by ϵ, and it is the string consisting of zero symbols.

A *prefix* of a string w is any number of leading symbols from w, and a *suffix* is any number of trailing symbols. For example, the string abc has prefixes ϵ, a, ab, and abc and suffixes abc, bc, c, and ϵ.

The *concatenation* of two strings x, y is the string of length $|x| + |y|$ formed by using x as a prefix and y as a suffix. This operation is denoted by juxtaposition, so that xy represents the concatenation of x followed by y.

A finite set of symbols is called an *alphabet*, usually denoted by Σ. We use Σ^* to represent the (infinite) set of all possible strings formed using symbols from Σ, and Σ^n to represent the (finite) set of all strings of length n over Σ.

3 Nested Request Patterns

One of the patterns we have observed in request streams is *nesting*. In a nesting pattern a client application submits a database request while processing the rows of another cursor. By recognizing this pattern of requests, Scalpel can avoid the overhead of many fine-grained queries.

In this chapter, we describe how Scalpel detects, optimizes, rewrites, and prefetches nested request patterns. Figure 3.1 shows Scalpel's components that are used for nesting detection (including both those components used during training (Figure 2.1) and run-time (Figure 2.2). At present, we discuss nested request patterns in isolation; in Chapter 5 we describe how Scalpel detects batch request patterns, and in Chapter 6 we describe how Scalpel combines detection of nested and batch request patterns.

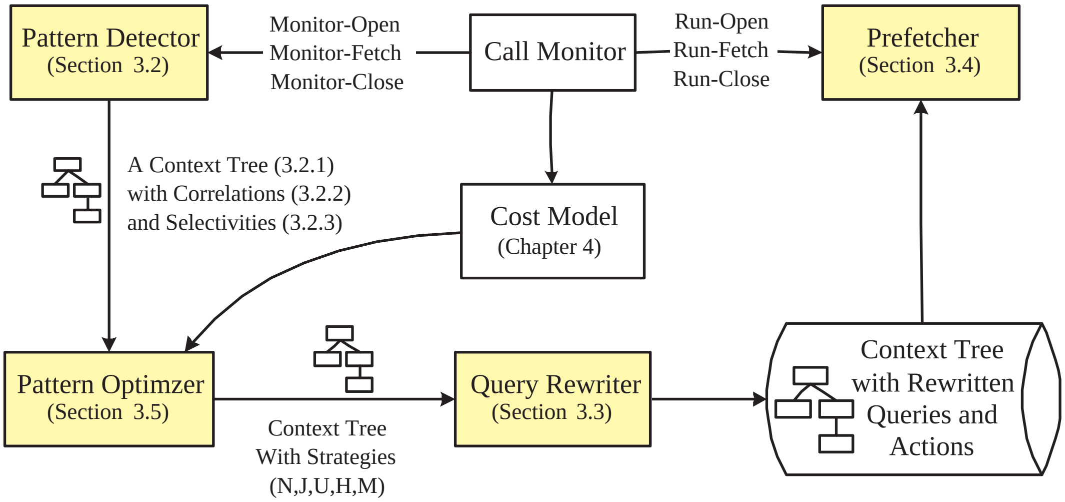


Figure 3.1: Scalpel components used for nested request patterns. Shaded components are described in this chapter.

During the training period, Scalpel's Call Monitor component intercepts OPEN, FETCH, and CLOSE calls from the client application. The Call Monitor component calls the MONITOR-OPEN, MONITOR-FETCH, and MONITOR-CLOSE functions, which are implemented by the Pattern Detector (Section 3.2). The Pattern Detector builds a *context tree* data structure to represent the structure of nesting that has been observed, predicted correlations between input parameters and earlier values, and selectivity estimates.

After the training period is over, the Pattern Optimizer uses this context tree to decide on an execution strategy for all of the nested request patterns found during the training period. The Pattern Optimizer annotates the context tree with a selected strategy for each of the queries observed in a nested pattern, then invokes the Query Rewriter. The Query Rewriter uses the strategies selected by the Pattern Optimizer to generate rewritten, combined queries that will be submitted at run-time in place of the original query. Further, the Query Rewriter adds ACTION objects as annotations to the context tree. These ACTION objects are used at run-time to inform Scalpel of how each request should be answered. Finally, at the end of the training period, the context tree is stored persistently for use at run-time.

At run-time, the Prefetcher loads the context tree from storage when it starts. As the application submits requests, the Call Monitor component calls the RUN-OPEN, RUN-FETCH, and RUN-CLOSE functions. These functions are defined by the Prefetcher component using the context tree structure.

This chapter is organized as follows. Section 3.1 gives an example program that generates a nested request pattern. This example will be used throughout the chapter. Section 3.2 describes how Scalpel's Pattern Detector observes a request stream to detect nested requests that are candidates for rewriting. Section 3.3 describes execution alternatives that the Query Rewriter can generate for the candidates identified during the training period. Section 3.4 discusses how the Prefetcher implements the alternative strategies that are generated by the Query Rewriter. Section 3.5 describes how Scalpel's Pattern Optimizer chooses between alternative valid execution strategies. Finally, Section 3.6 provides experimental results illustrating the strengths and weaknesses of the various strategies.

3.1 Example of Nesting

Figure 1.3 showed an example of nesting that we found in the SQL-Ledger application. Nesting occurs in that example because database access is encapsulated in functions, and these functions are called while processing the rows returned from another request. While only two cursors are involved in the nesting of Figure 1.3, generally an arbitrary tree can appear. Figure 3.3 shows an artificially constructed example of nesting that demonstrates specific features of our approach. Figure 3.2 shows sample data and corresponding output for the example in Figure 3.3.

There are three distinct functions in Figure 3.3, each opening a separate query (Q_1,Q_2, and Q_3). A call to the outer-most function $F_1()$ produces a two-dimensional chart. For each row that $F_1()$ fetches from Q_1, it outputs a row to the chart. For most rows, $F_1()$ calls $F_2()$, which outputs a single character enclosed in parentheses. Next, $F_1()$ outputs a colon (':'), then calls $F_3()$. $F_3()$

$S(s_1, s_2, s_3)$		
A	1	1
B	2	1
C	3	1
D	4	1
E	5	2

$T(t_1, t_2)$	
f	1
g	10
h	2
i	12
j	3
k	13
l	14

$V(v_1, v_2, v_3)$		
V	1	10
W	1	11
X	2	12
Y	3	13
Z	3	14

Output of $F_1(1)$
A(f): [V(g) W(-)]
B#
C(j): [Y! Z(l)]
D(-): []

Figure 3.2: Sample data and sample output for Figure 3.3. The first three tables show the contents of tables S, T, and V. The last table shows the chart generated by the call $F_1(1)$.

outputs a string enclosed in square brackets (‘[’,‘]’) with an entry for each row returned from Q_3. For each row returned from Q_3, $F_3()$ outputs the attribute `r3.v1` then calls $F_2()$ which again outputs a single character enclosed in parentheses. Function $F_2()$ is thus used in two different contexts. $F_2()$ is called from $F_1()$, with parameter x supplied with the value of `r1.s2`; it is also called by $F_3()$ with x supplied with the value `r3.v2`.

Figure 3.4 shows a trace of requests submitted by the program in Figure 3.3 using the sample data of Figure 3.2. The ‘Context’ column shows the ordered list of queries that are open before each of the application requests in the sample stream. The ‘Request’ column shows the request and actual parameters submitted by the sample program, and the ‘Result’ column shows the result of executing the request. The last two columns are discussed in Section 3.2.2.

3.2 Pattern Detector

The requests in Figure 3.4 are an example of a nesting pattern. If Scalpel can recognize this pattern, it can rewrite it to avoid the nesting. The Scalpel Pattern Detector monitors the stream of requests presented during the training period and constructs a model of this stream.

3.2.1 Context Tree

Notice that query Q_2 is opened in two different ways. It is opened as a result of a function call $F_1 \hookrightarrow F_2$, and it is also opened by the call $F_1 \hookrightarrow F_3 \hookrightarrow F_2$. The behaviour of these two calls is different: in the first case, the parameter x is correlated to attribute `r1.s2`, while in the second case, x is correlated to attribute `r3.v2`. If we are to prefetch results for Q_2, we must distinguish between the two ways that it is invoked in order to correctly predict the parameter values that will be used.

```
function $F_1$(w)
  open c1 cursor for $Q_1$:
    SELECT s1, s2 FROM S WHERE s3=:w ORDER BY s2
  while r1 $\leftarrow$ fetch c1 do
    PRINT( r1.s1 )
    if r1.s1 $\neq$ 'B' then
      $F_2$( r1.s2 )
      PRINT( ':' )
      $F_3$( r1.s2 )
    else
      PRINT( '#' )
    PRINT-NEWLINE( )
  close c1
end
function $F_2$(x)
  $\triangleright$ Query $Q_2$ outputs at most one row
  open c2 cursor for $Q_2$:
    SELECT t1 FROM T WHERE t2=:x
  r2 $\leftarrow$ fetch c2
  if r2 then
    PRINT( '(', r2.s1, ')' )
  else
    PRINT( '(-)' )
  close c2
end
function $F_3$(y)
  PRINT( '[' )
  open c3 cursor for $Q_3$:
    SELECT v1, v2 FROM V WHERE v3=:y
  while r3 $\leftarrow$ fetch c3 do
    PRINT( r3.v1 )
    if r3.v1 $\neq$ 'Y' then
      f2 $\leftarrow$ $F_2$( r3.v2 )
    else
      PRINT( '!' )
  close c3
  PRINT( ']' )
end
```

Figure 3.3: An example of a nested request pattern.

#	Context	Request	Result	Correlations
1	$C_0 : /$	Open(Q_1,1)	c_1	$\{\ \langle 1\vert \mathrm{C}, 1\rangle\ \}$
2	$C_1 : /Q_1$	Fetch(c_1)	$(A, 1)$	
3	$C_1 : /Q_1$	Open(Q_2,1)	c_2	$\{\langle 1\vert \mathrm{C}, 1\rangle, \langle 1\vert \mathrm{I}, C_1, 1\rangle, \langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
4	$C_2 : /Q_1/Q_2$	Fetch(c_2)	(f)	
5	$C_2 : /Q_1/Q_2$	Close(c_2)		
6	$C_1 : /Q_1$	Open(Q_3,1)	c_3	$\{\langle 1\vert \mathrm{C}, 1\rangle, \langle 1\vert \mathrm{I}, C_1, 1\rangle, \langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
7	$C_3 : /Q_1/Q_3$	Fetch(c_3)	$(V, 10)$	
8	$C_3 : /Q_1/Q_3$	Open(Q_2,10)	c_2	$\{\langle 1\vert \mathrm{C}, 10\rangle, \langle 1\vert \mathrm{O}, C_3, 2\rangle\}$
9	$C_4 : /Q_1/Q_3/Q_2$	Fetch(c_2)	(g)	
10	$C_4 : /Q_1/Q_3/Q_2$	Close(c_2)		
11	$C_3 : /Q_1/Q_3$	Fetch(c_3)	$(W, 11)$	
12	$C_3 : /Q_1/Q_3$	Open(Q_2,11)	c_2	$\{\langle 1\vert \mathrm{O}, C_3, 2\rangle\}$
13	$C_4 : /Q_1/Q_3/Q_2$	Fetch(c_2)	EOF	
14	$C_4 : /Q_1/Q_3/Q_2$	Close(c_2)		
15	$C_3 : /Q_1/Q_3$	Fetch(c_3)	EOF	
16	$C_3 : /Q_1/Q_3$	Close(c_3)		
17	$C_1 : /Q_1$	Fetch(c_1)	$(B, 2)$	
18	$C_1 : /Q_1$	Fetch(c_1)	$(C, 3)$	
19	$C_1 : /Q_1$	Open(Q_2,3)	c_2	$\{\langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
20	$C_2 : /Q_1/Q_2$	Fetch(c_2)	(j)	
21	$C_2 : /Q_1/Q_2$	Close(c_2)		
22	$C_1 : /Q_1$	Open(Q_3,3)	c_3	$\{\langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
23	$C_3 : /Q_1/Q_3$	Fetch(c_3)	$(Y, 13)$	
24	$C_3 : /Q_1/Q_3$	Fetch(c_3)	$(Z, 14)$	
25	$C_3 : /Q_1/Q_3$	Open(Q_2,14)	c_2	$\{\langle 1\vert \mathrm{O}, C_3, 2\rangle\}$
26	$C_4 : /Q_1/Q_3/Q_2$	Fetch(c_2)	(l)	
27	$C_4 : /Q_1/Q_3/Q_2$	Close(c_2)		
28	$C_3 : /Q_1/Q_3$	Fetch(c_3)	EOF	
29	$C_3 : /Q_1/Q_3$	Close(c_3)		
30	$C_1 : /Q_1$	Fetch(c_1)	$(D, 4)$	
31	$C_1 : /Q_1$	Open(Q_2,4)	c_2	$\{\langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
32	$C_2 : /Q_1/Q_2$	Fetch(c_2)	EOF	
33	$C_2 : /Q_1/Q_2$	Close(c_2)		
34	$C_1 : /Q_1$	Open(Q_3,4)	c_3	$\{\langle 1\vert \mathrm{O}, C_1, 2\rangle\}$
35	$C_3 : /Q_1/Q_3$	Fetch(c_3)	EOF	
36	$C_3 : /Q_1/Q_3$	Close(c_3)		
37	$C_1 : /Q_1$	Fetch(c_1)	EOF	
38	$C_1 : /Q_1$	Close(c_1)		

Figure 3.4: Trace example for Figure 3.3. The first column gives the number of the request. The second gives the context named C_i, which is the ordered list of open queries using path-separator notation. The third column gives the request being processed and its parameters. The fourth column gives the rows returned by Fetch, and the last column is described in Section 3.2.2.

Scalpel uses *contexts* to distinguish different uses of a query. A context is an abstraction of conditions within a request stream. By recording observations about application behaviour in specific contexts, we can distinguish between multiple uses of a query. All observations about a request stream are applied to the appropriate context within the stream, and at run-time, the context is used to decide what action to take when a request is submitted.

Our definition of context should be efficient to compute as we need to monitor the context at run-time. Further, contexts should be sufficiently detailed to distinguish between requests that will have distinct usage patterns.

We choose to use query nesting to identify contexts. We identify a context by the ordered list of queries open when a request is submitted. We write contexts using path-separator notation to emphasize the hierarchical nature of the list. For example, on line 3 of Figure 3.4, Q_2 is opened in context C_1, which has the path $/Q_1$; on line 10, it is opened in context C_4 with the path $/Q_1/Q_3$.

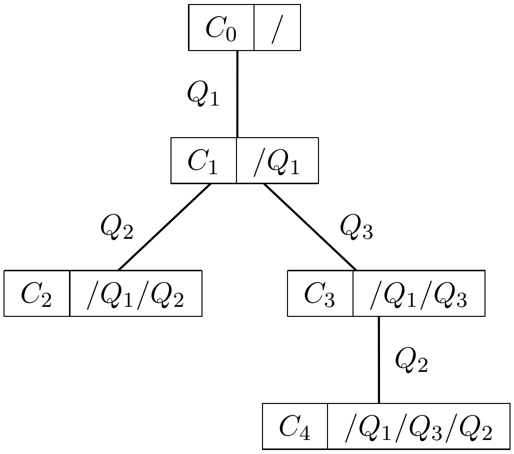


Figure 3.5: Query contexts of Figure 3.4.

Figure 3.5 shows the contexts corresponding to the program of Figure 3.3. The root context represents is an empty sentinel node. Each of the descendant nodes of the root corresponds to a context of open queries that was observed during the training period. Each node has an associated identifier ($C_0, C_1, \ldots C_4$ in Figure 3.3) as well as the list of open queries representing the context. An edge labelled Q_a between nodes C_x and C_y corresponds to an OPEN(Q_a) request that was observed in context C_x.

Figure 3.6 shows how the Pattern Detector builds the context tree by monitoring requests. The `treeroot` variable points to the root of the tree being created, and `currctxt` represents the current context of the request stream so far. When the client application submits request OPEN`(Q,parms)`, the Call Monitor calls MONITOR-OPEN`(Q, parms)`. This proce-

```
structure CONTEXT              ▷ Represent a condition within a request stream
  parent = NIL                 ▷ The context this query is opened in
  path = []                    ▷ The list of open queries
  lastinput = []               ▷ The most recent values of input parameters
  lastoutput = []              ▷ The most recent values of fetched columns
  scope = ∅                    ▷ Possible sources of input values                 (Section 3.2.2)
  correlations = ∅             ▷ Set of Correlation objects that have always held  (Section 3.2.2)
  counts = NIL                 ▷ Information to estimate selectivities            (Section 3.2.3)
  children = ∅                 ▷ A map of child contexts indexed by query
  alt = 'N'                    ▷ Alternative to use at run-time                   (Section 3.3)
  action = ∅                   ▷ List of actions to perform at run-time           (Section 3.4)
end

treeroot ← new CONTEXT                 ▷ The root of the context tree
currctxt ← treeroot                    ▷ The current context
procedure MONITOR-OPEN( Q, parms )
  child ← find( currctxt.children, Q )
  if child = NIL then
    child ← new CONTEXT( currctxt, [currctxt.path, Q] )
    add( currctxt.opens, Q, child )
    INIT-CORRELATIONS( child, parms )                              ▷ (Section 3.2.2)
    INIT-COUNTS( child )                                           ▷ (Section 3.2.3)
  else
    VERIFY-CORRELATIONS( child, parms )                            ▷ (Section 3.2.2)
    UPDATE-COUNTS( child )                                         ▷ (Section 3.2.3)
  child.lastinput ← parms
  currctxt ← child
end

procedure MONITOR-FETCH( row )
  currctxt.lastoutput ← row
  currctxt.counts.numrows ← currctxt.counts.numrows + 1
end

procedure MONITOR-CLOSE()
  currctxt ← currctxt.parent
end
```

Figure 3.6: Pattern Detector methods to build context tree.

dure searches for the query Q in the children of the current context (line 105). If it doesn't find it, then this is the first time the query has been observed in the context. MONITOR-OPEN creates a new context and adds it to the children of the current context. If the child has previously been seen, MONITOR-OPEN sets its `lastinput` field to the current parameters and updates bookkeeping information.

When the client application submits a FETCH request, the Call Monitor calls procedure MONITOR-FETCH(`row`). This procedure updates the `lastrow` field of the current context. Finally, when the client application submits a FETCH request, the Call Monitor calls procedure MONITOR-CLOSE, which moves the current context to the parent.

The structure of the context tree identifies the nesting present in a request stream. However, Scalpel requires additional information if it is to be able to rewrite these patterns. Queries are often parameterized, and Scalpel must predict the actual values that will be submitted for future requests if they are to be prefetched. The INIT-CORRELATIONS and VERIFY-CORRELATIONS procedures are used to track possible correlations between input parameters and other known quantities.

3.2.2 Tracking Parameter Correlations

When function $F_1()$ is called on line 56 of Figure 3.3, it calls $F_2()$ (line 57), leading to nested queries. We would like to rewrite this pattern (for example using a join) in order to eliminate the nesting. Since Q_2 is parameterized, Scalpel needs to predict not only that Q_2 will be executed, but also the values of the input parameters that will be supplied. Otherwise, Scalpel will not be able to derive an appropriate join query with which to replace Q_1. This prediction problem is similar to that faced by hardware speculative execution [72, 181]. With speculative execution (discussed further in Chapter 9), processor instructions can be speculatively executed based on predictions of values that registers are likely to hold.

To accomplish this prediction, Scalpel tries to identify nested queries whose input parameter values are *correlated* to values that are available to Scalpel before the query is submitted. Specifically, Scalpel considers input parameters that have the same value as an input or output parameters of the queries in the current context. In the sample program of Figure 3.3, the input parameter (x) of Q_2 matches the second result column (`s2`) of Q_1 when it is called from line 57.

In principle, we can observe this correlation by inspecting the source code of the application. For a human, it is relatively easy to recognize the correlation in this simple example; further, the tools of program analysis also offer a hope to find such correlations automatically. Data flow analysis can identify cases where one variable is guaranteed to have the same value as another. However, there are practical and theoretical limitations with a source-level analysis. Implementing a source-level analysis for complex object-oriented systems is relatively difficult [21], and such an analysis would necessarily be conservative. Even with a very careful data analysis implementation, decidability results tell us that there exist programs with a correlation that holds in all possible executions of a program yet cannot be detected. Further to this theoretical objection, systems can contain correlations that hold only in particular installations (including install-time parameters and user behaviour). A source-level analysis would likely not be able to detect these correlations.

For these reasons, we do not use a source-level analysis of the client program to find correlations. Instead, we infer the presence of correlations by observations of the values in the request stream presented during the training period. For example, consider the request trace of Figure 3.4. In each call OPEN(Q_3, y), the value of y matches the value of the second column of the row returned by the most recent FETCH(Q_1). By monitoring input and output parameter values, Scalpel learns to predict correlations between the input parameters of the inner query and the input and output parameters. In contrast to a source-level analysis, Scalpel finds all correlations that will always hold in all future executions (as we will see, it also may make some incorrect predictions). However, Scalpel does not detect correlations that are the result of functional dependencies. For example, we could have an input parameter x that is always supplied the value $w + 1$ where w is a prior output parameter. A source-level analysis might find such a correlation (with the limitations discussed previously). We could combine trace-based detection with source-level analysis; however, we have not observed such functional correlations in the systems we examined, so we expect the benefits of combination to be relatively slight.

Scalpel uses an object of type CORRELATION to represent the fact that a parameter has always been equal to a particular *correlation source*. We write a correlation for parameter i as $l = \langle i|\mathrm{T}, c, p\rangle$ to indicate that parameter i is being predicted, where T represents a particular type of correlation and c and p identify the particular source.

At present, we consider three types of CORRELATION objects (each identified by a single-character *correlation type*):

Input Parameter (I) The value of an input parameter to an outer OPEN request.

Output Parameter (O) The value of a column in the most recently fetched row of an enclosing context.

Constant (C) A constant value that is the same every time the query is opened in the given context.

Figure 3.4 shows the possible correlations for each input parameter of an OPEN request that are generated after the request is processed by Scalpel. For example, on line 3 we show the set $\{\langle 1|\mathrm{C}, 1\rangle, \langle 1|\mathrm{I}, C_1, 1\rangle, \langle 1|\mathrm{O}, C_1, 2\rangle\}$. The meaning of this is that Scalpel currently guesses that the formal parameter at index 1 of query Q_3 in context C_1 is always equal to all of the following values:

- The constant value 1 ($\langle 1|\mathrm{C}, 1\rangle$)
- The first input parameter of Q_1 ($\langle 1|\mathrm{I}, C_1, 1\rangle$)
- The second column `c1.s2` from the most recently fetched row of Q_1 ($\langle 1|\mathrm{O}, C_1, 2\rangle$)

On line 19, Scalpel has reduced its guess to $\{\langle 1|\mathrm{O}, C_1, 2\rangle\}$ because the actual value of x on line 19 does not match the other two correlation sources.

The Pattern Detector maintains a *parameter context* in the context tree data structure so that Scalpel can identify these parameter correlations. For each CONTEXT object C, Scalpel records the most recent input parameters observed in the `lastinput` field and the most recently fetched row in the `lastoutput` field. Together, the `lastinput` and `lastoutput` fields form a parameter context.

We can find the the parameter context at each point in the request stream of Figure 3.4 by considering the last fetched result (column 4) and the input parameters (column 3) for each cursor that is currently open. A CORRSOURCE object is used to identify each value in the parameter context by specifying the context from which the values are drawn, a `type` of I or O to indicate input or output parameters respectively, and a `parameter` field that gives the index of the parameter of interest.

The Pattern Detector observes the values of input parameters to OPEN requests. Figure 3.7 shows the code used to track correlation values. When a new context C is first created, the INIT-SCOPE procedure initializes the `scope` field of C. This scope contains the set of all possible CORRSOURCE objects to which Scalpel considers the input parameters of C could be correlated. At present, Scalpel initializes the scope of C to include the scope of its parent context P in combination with the the parameter context of P. This definition overloads the correlation detection (scope) with the structural detection (nesting). In principle, these do not need to be the same.

After initializing the scope of the new context, INIT-CORRELATIONS builds the initial set of CORRELATION for each input parameter i based on the CORRSOURCE objects in `scope` that have the same current value as the input parameter i (line 154). In addition to elements of the `scope` field, INIT-CORRELATIONS also adds a constant CORRELATION object of type “C” initialized with the current value of the parameter. This constant CORRELATION will detect if a parameter is always supplied with the same constant value within the context.

The INIT-CORRELATIONS procedure initialized the `correlations` field of a context to reflect the set of all correlations that held on the first OPEN call. The VERIFY-CORRELATIONS procedure is called on subsequent OPEN calls to find correlation values that no longer hold. For each stored CORRELATION object `c`, VERIFY-CORRELATIONS compares the current value of the input parameter `parms[c.inparam]` to the current value of the source predicted by `c`. If the supplied value of the input parameter does not match the current value of the CORRELATION object, `c` is removed from the `correlations` set.

The parameter correlations in Figure 3.4 are the manifestations of actual parameter correlations in the application code of Figure 3.3. In general, correlations observed by Scalpel during training may be the result of actual variable correlations in the application. However, they may also be mere coincidence. If we use a sufficiently long training period, most such coincidences should be discovered and eliminated from consideration. However, there is no guarantee that correlations inferred by the pattern detector will actually hold at run-time. This may cause Scalpel to generate semantic prefetches that are not useful. Since Scalpel can recognize such prefetches at run-time, this may impact the system's performance but it will not cause Scalpel to return incorrect query results to the application.

The `correlations` set for a context records correlations for all input parameters (identified by the `inparam` value). When we are interested in the correlations in set C that apply to a particular input parameter i, we use the CORRFORPARM(C,i) function.

DEFINITION 3.1 (CORRFORPARM(C,I))
If C is a set of correlations for request a, and i identifies a parameter of a, then we define CORRFORPARM(C,i) as follows:

$$\text{CORRFORPARM}(C,i) = \{c \in C \mid c.\texttt{inparam} = i\} \tag{3.1}$$

Function CORRFORPARM(C,i) identifies the subset of correlations within C that apply to parameter i of a.

We say that a query Q is *fully predicted* in context C if we have a non-empty correlation set for each input parameter of C_2, the Q-child of C. That is, for each parameter i, $\text{CORRFORPARM}(C_2.\texttt{correlations}, i) \neq \emptyset$. At the conclusion of the training phase, the Pattern Detector passes a set of candidate context/query pairs to the Pattern Optimizer. A context/query pair (C,Q) is a rewrite candidate if Q is fully predicted in context C. Thus, the pattern detector reports (C,Q) as a candidate if Scalpel can predict the future values of all input parameters to Q based on correlations that have always held during the training period when Q is opened in context C. The FULLY-PREDICTED(C) procedure reports TRUE for all contexts that are fully predicted.

3.2.2.1 Overhead of Correlation Detection

In the worst case, every input parameter may be correlated to each input parameter or output column of every outer query. If we have a nesting depth of D with C output columns and P input

```
structure CorrSource
  type = ?            ▷ The type of correlation: I for input, O for output, or C for constant
  context = Nil       ▷ The source context or Nil
  parameter = 0       ▷ The parameter number for I and O types
end
structure Correlation
  inparam = ?         ▷ The input parameter that is being predicted
  type = ?            ▷ The type of correlation: I for input, O for output, or C for constant
  context = Nil       ▷ The source context or Nil
  parameter = 0       ▷ The parameter number for I and O types
  value = Nil         ▷ The constant value for C types
end

▷ Initialize the scope of a new context to include the parameters of all enclosing contexts
procedure Init-Scope(C)
  P ← C.parent
  scope ← [ P.scope ]
  for i ← 1 to P.lastinput.length do
    scope ← [ scope, new CorrSource("I", P, i) ]
  for i ← 1 to P.lastoutput.length do
    scope ← [ scope, new CorrSource("O", P, i) ]
  C.scope ← scope
end

procedure Init-Correlations(C, parms)
  Init-Scope(C)
  scope ← C.scope
  for i = 1, parms.length do
    for s ∈ C.scope where  Curr-Value(s) = parms[i] do
      n ← new Correlation( i, s.type, s.context, s.parameter )
      corrs ← corrs ∪ n
    corrs ← corrs ∪ new Correlation( i, "C", parms[i])
  C.correlations ← corrs
end

procedure Verify-Correlations(C,parms)
  corrs ← C.correlations
  valid ← { c ∈ corrs | Curr-Value(c) = parms[ c.inparam ] }
  C.correlations ← validcorrs
end
```

Figure 3.7: Tracking query parameter correlations

parameters per query, this gives $D(C + P)$ possible correlation sources; each of the input parameters of the innermost query may be correlated to all of these. Overall, we have $O(DP)$ input parameters in the entire nest, giving $O(D^2P(C + P))$ correlations we must consider.

The correlation detection algorithm we have presented is polynomial. A slightly more complicated linear algorithm exists. However, we have found that even the polynomial approach has reasonable overhead for the actual systems we examined. Table 3.1 shows the open-time for a query with varying numbers of outer queries opened (Depth), varying number of columns for each query (Cols), and varying number of parameters for the innermost query (Parms). The time for the unmodified program is shown (Original) compared to the time during training (Training). Testing was performed with a local client (configuration LCL, defined in Section 3.6).

Depth	**Cols**	**Parms**	**Original (ms)**	**Training (ms)**
1	10	1	2.64	3.15
10	10	1	2.62	3.31
10	10	10	2.88	3.34
40	100	100	14.95	198,368.17

Table 3.1: Training overhead.

The overhead of the correlation detection algorithm is reasonable so long as the product $D(C+P)$ is not too high. For the configuration we tested, results were good so long as this product did not exceed 1000. In the actual systems we examined, we did not find any cases where this product exceeded 100. However, the poor scalability of the polynomial algorithm indicates that a linear algorithm may be preferable in a practical implementation.

3.2.3 Client Predicate Selectivity

If Scalpel's Pattern Detector produces a candidate context/query pair (C, Q), this means that whenever Q occurs within context C, the input parameters of Q can be predicted using values available in C. However, this does not imply that Q occurs every time C occurs. Predicates within the client application may dictate that Q occurs in some cases but not in others. For example, in Figure 1.3, an application predicate on line 10 determines whether the correlated inner query will occur inside the outer query. Similarly, predicates in Figure 3.3 on line 56 and 82 control whether nested queries are submitted. The selectivity of client predicates is important to Scalpel because it affects the costs of the various semantic prefetching strategies that Scalpel's optimizer will consider. During the training phase, Scalpel estimates client predicate selectivities and then uses these estimates during cost-based optimization.

Scalpel counts the number of rows fetched from the outer query and the number of times that the inner query is opened. The ratio EST-P is computed as an estimate of the probability that the inner query will be opened for each out row at run-time. For example, if 200 rows are fetched from the outer query and the inner query is opened 120 times, we will use the estimate EST-P $= 0.6$. At present, we do not consider the case where an inner query is opened more than one time for a particular outer row (this is addressed in Chapter 6); therefore, we are assured that the ratio is a reasonable probability estimate (EST-P $\in [0, 1]$).

There is a difference between the predicates of Figure 1.3 and Figure 3.3. The predicates on line 56 and 82 of Figure 3.3 depend on values returned from the outer query. These predicates may have different values for different rows of Q_1 and Q_3 respectively. In contrast, the predicate `currency` $\neq$ `defaultcurrency` (line 10, Figure 1.3) depends only on an input parameter to the `get_openinvoices` function. This predicate will have the same result for all calls to OPEN(Q_2) corresponding to a single open of Q_1. For some instances of Q_1, the inner query will not be opened at all for any row. This situation could also occur if the predicate on line 82 of Figure 3.3 return false for every row of Q_3 associated with one particular call to $F_3()$. For example, this can occur if a particular instance of the outer query returns no rows.

Two of the execution strategies that we consider (described in Section 3.3) cost less to execute if the inner query is not opened at all for a single instance of the outer query. We use a second parameter EST-P0 to estimate the probability that the inner query will be executed for a particular instance of the outer query. We maintain a count `C.numprtopens` for each context `C`; this counts the number of instances of the parent query for which the inner query is opened at least once. We compute the ratio of `C.numprtopens` to the number of instances of the parent query and we use this ratio as the value of EST-P0. For example, if the outer query is opened 5 times and the inner query is submitted for only 2 of these instances, Scalpel will use the estimate EST-P0 $= 0.4$.

Figure 3.8 shows routines that are used to maintain counts in order to estimate the EST-P and EST-P0 selectivities. Section 3.5.2 describes how Scalpel uses the EST-P and EST-P0 estimates for each context/query pair in order to estimate the costs of various execution strategies.

3.2.4 Summary of Pattern Detection

All of information gathered during the training period is combined into the context tree data structure. Figure 3.9 shows the context tree for the program of Figure 3.3. As described in Section 3.2.1, this model shows the structural nesting of queries (via the `children` field). Further, Section 3.2.2 described how the `correlations` field is maintained by the Pattern Detector during the training period to predict the values that will be used in future executions of the associated queries. Finally, Section 3.2.3 described how the EST-P and EST-P0 fields are maintained

```
structure COUNTS
  lastprtopen = 1     ▷ The last open # of parent for which this context opened
  numprtopens = 1     ▷ The number of opens of parent for which this context opened
  numrows = 0         ▷ The number of rows fetched in this context
  numopens = 1        ▷ The number of times this context was entered
end

procedure INIT-COUNTS(C)
  P ← C.parent
  C.counts ← new COUNTS( P.counts.numopens )
end

procedure UPDATE-COUNTS(C)
  pcnt ← C.parent.counts
  cnt ← C.counts
  cnt.numopens ← cnt.numopens + 1
  if pcnt.numopens ≠ C.lastprtopen then
    C.lastprtopen ← pcnt.numopens
    cnt.numprtopens ← cnt.numprtopens + 1
end

function EST-P(C)
  pcnt ← C.parent.counts
  cnt ← C.counts
  return cnt.numopens / pcnt.numrows
end

function EST-P0(C)
  pcnt ← C.parent.counts
  cnt ← C.counts
  return cnt.numprtopens / pcnt.numopens
end
```

Figure 3.8: Monitoring counts to estimate selectivity of local predicates

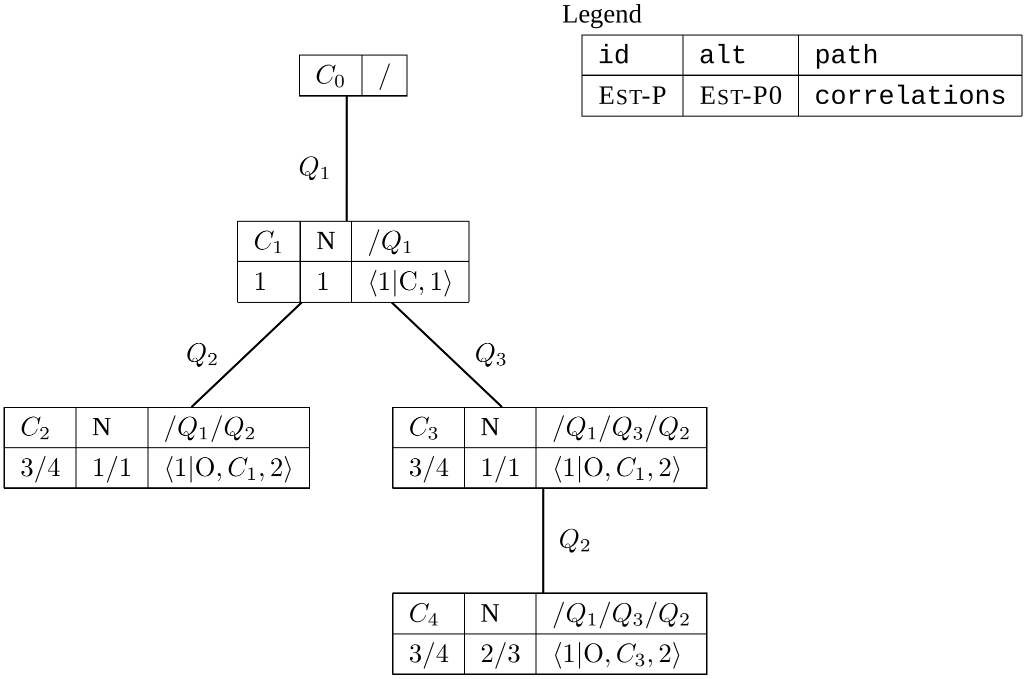

Legend
id | alt | path
Est-P | Est-P0 | correlations
C_0 | /
Q_1
C_1 | N | $/Q_1$
1 | 1 | $\langle 1|\mathrm{C}, 1\rangle$
Q_2
Q_3
C_2 | N | $/Q_1/Q_2$
3/4 | 1/1 | $\langle 1|\mathrm{O}, C_1, 2\rangle$
C_3 | N | $/Q_1/Q_3/Q_2$
3/4 | 1/1 | $\langle 1|\mathrm{O}, C_1, 2\rangle$
Q_2
C_4 | N | $/Q_1/Q_3/Q_2$
3/4 | 2/3 | $\langle 1|\mathrm{O}, C_3, 2\rangle$

Figure 3.9: Context tree constructed from Figure 3.4. Nodes show an identifier for each context (`id`), execution alternative (`alt`), query path (`path`), estimates Est-P and Est-P0, and the set of correlations for each input parameter (`correlations`).

to estimate how often a nested query will be executed. These estimates are used to choose an appropriate execution strategy.

The context tree data structure represents the culmination of all of the information gathered by the Pattern Detector. The context tree provides the link between the Pattern Detector and Pattern Optimizer, which selects the execution method that will be used at run-time. The selected method is shown in Figure 3.9 using the `alt` field; in this example, all nodes are annotated with ‘N’ to indicate a nested strategy. After optimization, the context tree contains everything that is needed by the Prefetcher to execute the appropriate action at run-time when a query is submitted.

3.3 Query Rewriter

Given a context tree, Scalpel’s optimizer selects an execution strategy for each of the nodes in the tree. Figure 3.10(a) shows the request trace of Figure 3.4 (submitted by the program in Figure 3.3). The trace is shown in the concise trace notation described in Section 2.3. This trace includes 10 cursor opens, returning a total of 12 rows (and 3 empty result sets). Each of the result sets returned in this sample trace is small, and the cost of each request is dominated by the interprocess communication cost. Further, the number of cursor opens depends on the number of rows returned by Q_1 and Q_2. While the number of opens is small with this tiny data set, it could grow significantly for larger examples. In order to provide better performance, Scalpel considers alternate execution strategies that submit fewer OPEN requests to the database server.

There are three fundamental approaches to the execution of these nested queries: nested, partitioned, or unified execution:

Nested Execution The nested query can be executed in a nested fashion, as it was originally executed by the client program. Figure 3.10(a) is an example of nested execution.

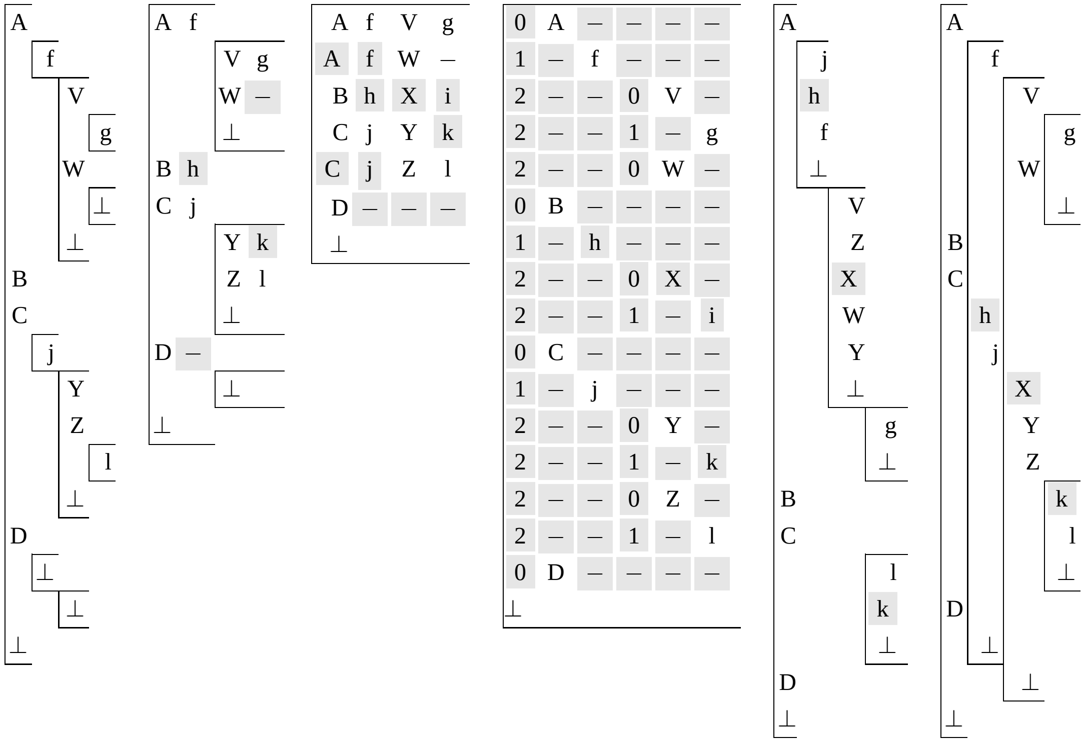
Unified Execution The second approach, called *unified* by Fernández, Morishima and Suciu [67], combines the inner and outer queries into a single query. The result of this combined query encodes the rows of both the outer and inner queries. Figures 3.10(c) and (d) are example of unified execution: in this example, a single cursor is opened over a result encoding all values needed by the application.

Partitioned Execution Partitioned execution combines all of the executions of the inner query (within a particular outer query) into a single query. A rewritten version of the inner query is submitted once to the server. The results of the rewritten inner query are merged at the client with the results for the outer query. This effectively executes the nested query like a distributed join in which the inner table is moved to the outer table's location and joined there. Figure 3.10(e) and (f) are example of partitioned execution. A rewritten query is submitted to the server at most once for each time that the immediately enclosing query is submitted. This contrasts with the nested execution strategy, where the number of OPEN requests is proportional to the number of rows returned from the outer query.

In the remainder of this section, we describe these execution strategies in more detail.

3.3.1 Nested Execution

Figure 3.10(a) illustrates the nested execution strategy for the application fragment of Figure 3.3. Although fixed per-request costs are associated with each invocation of the inner query, the nested execution strategy is appropriate when the selectivity of local predicates guarding the inner query is expected to be very low, i.e., when the inner query is often not executed at all. For example, this will be true for the `get_openinvoices` function in Figure 1.3 if the report currency is usually the same as the default currency.

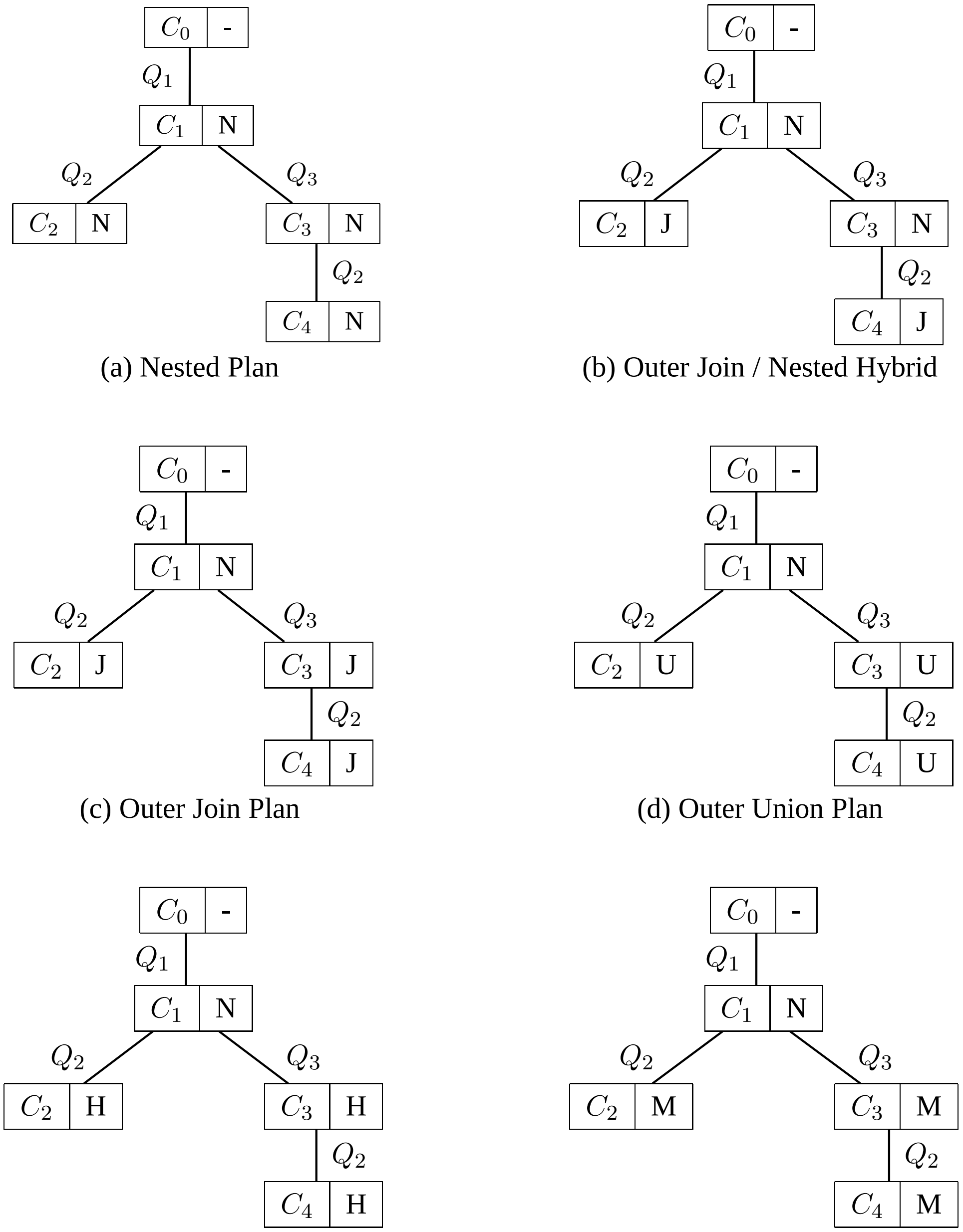


(a) Nested (b) Hybrid (c) Outer Join (d) Outer Union (e) Hash (f) Merge

Figure 3.10: Fetch traces for alternate executions of Figure 3.4. The following strategies are shown: (a) nested execution, (b) a hybrid of nested execution and outer join, (c) outer join, (d) outer union, (e) client hash join, and (f) client merge join. Attributes in the combined queries that were not returned in the original strategy are shaded (these are redundant attributes). The context tree for each of these traces is shown in Figure 3.11.

3.3.2 Query Rewrite Preliminaries

If the nested execution strategy is used, Scalpel does not rewrite the application's queries. However, the partitioned and unified strategies do require query rewrites, and these rewrites share common requirements, which are described in this section. Section 3.3.2.2 describes a SQL construct (lateral derived tables) that is quite helpful in these rewrites. Section 3.3.2.3 discusses how input parameter markers are rewritten.



(a) Nested Plan

(b) Outer Join / Nested Hybrid

(c) Outer Join Plan

(d) Outer Union Plan

(e) Client Hash Join Plan

(f) Client Merge Join Plan

Figure 3.11: Context trees corresponding to traces in Figure 3.10. Only the `id` and `alt` fields are shown.

3.3.2.1 Algebraic Notation

The following notation is used to express rewrites and demonstrate their correctness. For a full definition of the constructs used, see the definitions used by Paulley [145] and Galindo-Legaria [75].

Symbol	**Definition**
$r[A]$	Tuple formed by projecting tuple r on attributes A
(r, s)	Tuple formed by using values of tuple r followed by s
$\pi^{\textsc{All}}[A](R)$	Project relation R onto attributes A; preserve duplicates
$\pi^{\textsc{Dist}}[A](R)$	Project relation R onto attributes A; eliminate duplicates
$\sigma[p](R)$	Select rows of R that satisfy predicate p
$R\,\mathcal{A}^{\times}_{\bar{x},M}\,E(\bar{x})$	Apply operator with mapping $\bar{x} = M$
$R\,\mathcal{A}^{\text{LOJ}}_{\bar{x},M}\,E(\bar{x})$	Outer apply operator with mapping $\bar{x} = M$
$R \uplus T$	Duplicate-preserving union of relations R and T
$\mathit{sch}(R)$	Attributes of R
$\textsc{Key}[R]$	A key of R
$\textsc{Nulls}(A)$	A tuple formed with NULL for each attribute in A
$\textsc{rowid}(r)$	A unique identifier for tuple r, possibly using virtual attributes

Table 3.2: Algebraic notation for query rewrites.

The projection operators $\pi^{\textsc{All}}[A](R)$ and $\pi^{\textsc{Dist}}[A](R)$ take a relational argument R and a tuple $A = (a_1, a_2, \ldots, a_n)$ of attributes. The elements a_i can be simple attributes ($a_i \in \mathit{sch}(R)$). We also permit a_i to be scalar functions of these attributes and constant values; this extension allows us to better represent SQL queries, where the SELECT list allows not only attributes but scalar expressions. The results of projecting a tuple r on attributes A is given by $r[A] = (a_1(r), a_2(r), \ldots, a_n(r))$. The expression $\pi^{\textsc{All}}[A]R$ represents the projection of relation R on attributes A, and this is the result of performing the tuple-wise projection on each tuple in relation R.

The apply operator $R\,\mathcal{A}^{\times}_{\bar{x},M}\,E(\bar{x})$ takes a relational argument R, a parameterized relational expression $E(\bar{x})$ with parameter tuple $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$, and a list of attributes $M = (M_1, M_2, M_n)$. As with the projection operations, we allow the elements of M to refer to attributes of R or scalar functions of those attributes. The result is formed by evaluating expression E for each row $r \in R$, using the substitution values $\bar{x} = r[M]$. Formally, we have the following:

$$R\,\mathcal{A}^{\times}_{\bar{x},M}\,E(\bar{x}) = \biguplus_{r \in R} \left[\ \{r\} \times E(\bar{x} = r[M])\ \right] \tag{3.2}$$

Here, $E(\bar{x} = s)$ represents the results of relational expression E evaluated under the parameter binding $\bar{x}_i = s_i, i \in 1, \ldots, n$.

The outer apply operator $R\,\mathcal{A}^{\text{LOJ}}_{\bar{x},M}\,E(\bar{x})$ is defined similarly, except that it retains rows $r \in R$ where the expression $E(\bar{x} = r[M])$ is empty. This construct is similar in effect to an outer join, and concatenates NULL values to tuple r in place of the attributes of E.

$$R\,\mathcal{A}^{\text{LOJ}}_{\bar{x},M}\,E(\bar{x}) = R\,\mathcal{A}^{\times}_{\bar{x},M}\,E(\bar{x}) \biguplus \left(\biguplus_{r \in R \,\mid\, E(\bar{x}=r[M])=\emptyset} \{(r, \textsc{Nulls}(\mathit{sch}(E)))\} \right) \tag{3.3}$$

As with outer joins, we note that the result of the outer apply operator $R\,\mathcal{A}^{\text{LOJ}}_{\bar{x},M}\,E(\bar{x})$ includes every tuple of R at least one time (more often if there are multiple joining tuples).

3.3.2.2 Lateral Derived Tables

Figure 3.12 illustrates one way to combine query Q_1 and Q_2 from the example application of Figure 3.3. The query is expressed using a join in an unnested style, and is likely to execute efficiently. However, the algorithm for combining the two queries requires a flattening of the nesting relationship that is present in the client application. This is a non-trivial procedure that has been studied extensively for the case when the nesting is present within a single query.

```
SELECT S.s1, S.s2, T.t1
FROM   S LEFT JOIN T ON T.t2 = S.s2
WHERE  S.s3=:w
```

Figure 3.12: Manually combined query joining Q_1 and Q_2

If we could combine the two queries into a single nested query, we could rely on rewrite optimizations implemented in the DBMS optimizer to unnest the query and generate an efficient access plan. For example, if we could write a query such as the one in Figure 3.13, we could easily express the application's query nesting, allowing the DBMS query optimizer to select the best execution strategy. Unfortunately, the query in Figure 3.13 is not legal according to SQL/2003. The reference to `P.s2` within query `I` is an *outer reference*. The outer reference is not within the scope of `P`, and is therefore disallowed.

Shanmugasundaram et al. [166] used a clever approach to combine queries, relying on the fact that the scope clause of a table reference includes all of the join conditions for joins containing the table reference. Their approach would result in the query in Figure 3.14. This approach works adequately for queries consisting of select-project-join (SPJ), where the correlation value

```
SELECT P.s1, P.s2, I.t1
FROM   ( SELECT s1, s2
         FROM   S
         WHERE  s3=:w ) P,
       ( SELECT t1
         FROM   T
         WHERE  t2=P.s2 ) I
```

Figure 3.13: Combined query joining Q_1 and Q_2 (not legal SQL/2003)

is used only in predicates in the WHERE clause of the inner query. However, this approach does not work with cases where the correlations appear in more complex contexts such as an ON condition of an outer join in the FROM clause or in an expression in the SELECT list. The approach does not work at all for grouped, distinct, or unioned queries. Even in the cases where this approach does work, the resulting queries will not be executed efficiently unless the DBMS simplifies the generated ON condition.

```
WITH P AS ( SELECT s1, s2
            FROM   S
            WHERE  s3=:w ) P,
     I AS ( SELECT t2, t1
            FROM   T ) I
SELECT P.s1, P.s2, I.t1
FROM   P LEFT JOIN I ON (I.t2 = P.s2)
```

Figure 3.14: Combined query joining Q_1 and Q_2 using outer references in the ON condition (Shanmugasundaram et al.'s Approach [166])

Fortunately, SQL/99 [13] introduced a new construct called a *lateral derived tables,* and this feature makes it easy to express nesting in the FROM clause. Using the keyword LATERAL we are able to write a query in the style of Figure 3.13 that is legal. The LATERAL keyword signals that the inner derived table contains outer references.

The syntax:

```
FROM  <table reference list>,
      LATERAL (<query expression>) <correlation name>
```

has the following semantics. Let TRL be the <table reference list>. Let QE be the <query expression>. The SQL within QE can contain references to attributes of TRL; these are called *outer references*. Let TRLR be the multiset of rows resulting from TRL. Let QE(r) represent the multiset resulting from evaluating QE with attributes of r supplied as actual parameters to the corresponding outer references. The result of the FROM clause above is the following multiset:

$$\{\!\mid (r,q) \mid r \in \texttt{TRLR}, q \in \texttt{QE(r)} \mid\!\} \tag{3.4}$$

This definition is equivalent to the algebraic apply operator TRLR $\mathcal{A}^{\times}_{\bar{x},M}$ QE$(\bar{x})$ defined by Galindo-Legaria [75] and outlined in Section 3.3.2.1. Figure 3.15 shows how the illegal query of Figure 3.13 can be corrected using the LATERAL keyword.

```
SELECT P.s1, P.s2, I.t1
FROM   ( SELECT s1, s2
         FROM   S
         WHERE  s3=:w ) P,
       LATERAL
       ( SELECT t1
         FROM   T
         WHERE  t2=P.s2 ) I
```

Figure 3.15: Combined nested query joining Q_1 and Q_2 using LATERAL

The lateral derived table construct allows Scalpel to generate a single SQL query that directly matches the application semantics that it infers from monitoring the request stream. By using lateral derived tables, Scalpel allows the DBMS optimizer to select the best execution strategy. Unfortunately, the LATERAL construct is not yet widely supported in commercial DBMSs. Of the three commercial systems we considered in our experiments, one system supported the LATERAL construct directly while the other two supported the same semantics using (distinct) vendor-specific syntax.

The query in Figure 3.15 has a shortcoming: rows from the outer query will not be included in the result unless there is at least one row from the inner with matching currency and transaction date. If these rows must be included, an equivalent of an outer join must be used to preserve the rows from the outer queries (the table references that precede the lateral derived table in the FROM clause). The SQL/2003 standard does not provide support for directly expressing a lateral derived table operating in an outer join with outer references to the preserved side of the outer join. It is possible to simulate this behaviour using the existing syntax by introducing an additional outer join, but the result does not directly express the desired behaviour, and the additional complexity may lead to more optimization costs or poor execution plans. Two of the three commercial DBMSs we considered do support using LATERAL-style derived tables in outer joins.

Although there is existing commercial support for expressing nesting in the `FROM` clause which preserves rows in the face of an empty inner table expression, the syntax is product-dependent. We define an extension to SQL/2003, `LEFT OUTER LATERAL`, which provides the required outer join semantics:

```
FROM <table reference list>,
     LEFT OUTER LATERAL (
         <query expression>
     ) <correlation name>
```

The result of the `FROM` clause above is the following multiset:

$$\{\!|\, (r, Q) \mid r \in \texttt{TRLR}, q \in \texttt{QE(r)} \,|\!\} \biguplus \{\!|\, (r, \textsc{Nulls}(QE)) \mid r \in \texttt{TRLR}, \texttt{QE(r)} = \emptyset \,|\!\} \qquad (3.5)$$

Again, this definition is equivalent to the algebraic outer apply operator `TRLR` $\mathcal{A}^{\text{LOJ}}_{\bar{x},M}$ `QE`$(\bar{x})$ defined by Galindo-Legaria [75] and outlined in Section 3.3.2.1. The corresponding RIGHT OUTER LATERAL and FULL OUTER LATERAL are not meaningful because the right side of the construct can not be evaluated without a row from the left side to provide outer bindings.

We translate this syntax to vendor specific language dialects as necessary. For systems that do not support a vendor-specific outer join variant of `LATERAL` derived tables, we can translate a `LEFT OUTER LATERAL` construct to standards compliant SQL.

We may be curious as to whether the `LATERAL` construct is really needed, or whether it is merely a syntactic convenience that simplifies the rewriting. Galindo-Legaria [75, 76] answered this question by showing that any query containing a `LATERAL` construct can be rewritten into an equivalent query with no `LATERAL` constructs. This shows that `LATERAL` merely provides a syntactic convenience, and does not extend the class of queries that can be posed. In principle, this result shows that the approach used by Shanmugasundaram et al. [166] could be extended to handle arbitrary queries. In practice, the rewriting defined by Galindo-Legaria [75] generates queries that are much less amenable to optimization. Contemporary query optimizers are not, in general, able to recover the original nested query from the flattened variant. For this reason, the `LATERAL` construct is more than syntactic sugar in practice: it allows Scalpel to provide a query to the DBMS optimizer in a form that allows cost-based rewrite decisions. Scalpel uses the `LATERAL` construct to directly encode the nesting detected in the application code, leaving all rewrite optimizations to the DBMS query optimizer.

3.3.2.3 Choosing The Best Correlation Value

The combined query in Figure 3.15 replaced the parameter x of query Q_3 with the outer reference `P.s2`. This replacement was based on the predicted parameter correlations found by the

Pattern Detector and shown in Figure 3.4. In the case that the `correlations` set contains only a single entry for a parameter i, there is only one choice for replacement. In other cases, however, multiple sources are predicted as possible parameter correlations. During the training period, each of these sources always had the same value as the actual parameter value supplied when opening the associated query. For example, on line 3 the formal parameter x of query Q_3 is believed to be always equal to $\{$`1, w, c1.s2`$\}$ (expressed with the `correlations` set $\{\langle 1|\mathrm{C},1\rangle, \langle 1|\mathrm{I},C_1,1\rangle, \langle 1|\mathrm{O},C_1,2\rangle\}$).

During training, all of the candidate correlation sources had the same value each time the associated query was opened, so there is no reason to believe that any of them will provide better predictive ability at run time. That leaves us free to choose any of these sources. Our choice of predictor has an impact on the cost of combined queries. If we choose an output attribute of the outer query, the combined query will have an outer reference. Alternatively, if we select a CORRELATION of a constant literal or input parameter, then the value is available when we open the combined query and does not require an outer reference in the combined query.

In order to minimize the cost of combined queries, we select an CORRELATION object in the following priority order:

1. Constants (type 'C')
2. Input parameters (type 'I')
3. Output attributes (type 'O'), ordered from highest ancestor down).

The CHOOSE-CORRELATIONS procedure is shown in Figure 3.16. This procedure is used to select the appropriate CORRELATION for each input parameter of context C. After making its selections of parameters, CHOOSE-CORRELATIONS modifies the query associated with the context by replacing parameter markers. For correlations of type constant, the marker is replaced with the literal constant value. For input parameters, the marker is replaced with an updated marker that will select the value from the appropriate input parameter of an opened cursor. For output parameters, the CHOOSE-CORRELATIONS replaces the original parameter marker with an outer reference to an attribute of the derived table representing the predicted source query.

3.3.3 Unified Execution

Under the unified execution strategy, Scalpel uses the `LATERAL` construct to combine the inner and outer queries into a single query which returns all of the rows that would have been returned by the individual outer and inner queries. When the Prefetcher observes the application opening the outer query (in the correct context), it submits instead the combined query. The Scalpel system then uses the cursor opened over the combined query to respond to the application's requests to fetch rows from the original inner and outer queries that were combined.

```
procedure Choose-Correlations( C )
  correlations ← C.correlations
  sql ← C.query
  for i ← 1 to correlations.length do
    corrs ← CorrForParm(correlations,i)
    cs ← Choose-Best(corrs)         ▷ Select the best source by priority
    sql ← Replace-Parameter(i,cs)   ▷ Replace the parameter marker text
  C.query ← sql
end
```

Figure 3.16: The Choose-Correlations procedure.

Many unified strategies are possible. Scalpel's optimizer currently considers two representative unified strategies, one that combines the outer and inner queries using an outer join and another which combines them using an outer union. The optimizer (described in Section 3.5) sets the `alt` field of nodes in the context tree to 'J' to select the outer join strategy and 'U' to select the outer union strategy. We describe these two strategies next.

3.3.3.1 The Outer Join Strategy

Scalpel forms a combined join query using a lateral derived table expression. The procedure for combining the texts of the outer and inner queries is shown Figure 3.17. Given the texts of the outer and inner queries, this procedure will produce a derived table expression similar to the one shown in Figure 3.15, except that `LEFT OUTER LATERAL` is used in place of `LATERAL` to ensure that all rows of the original outer query are included in the result. In addition, the combination procedure adds an `ORDER BY` clause that matches the `ORDER BY` clause of the original outer query, if that query has one. Thus, Scalpel would add `ORDER BY S.s1` to the combined query shown in Figure 3.15 since the original outer query Q_1 (from the $F_1()$ function in Figure 3.3) was so ordered.

The Combine-Join function operates on a single node C in the context tree at a time. First, it identifies `join_children`, the child contexts of C that are annotated with alternative 'J'. Second, it calls Choose-Correlations to select what value will be used to predict the input parameter values that will be submitted in the future. Recall that Section 3.2.2 described how the Pattern Detector finds a set of Correlation objects for each input parameter. Each of these Correlation objects represents a value that could be used to predict the values of the input parameter. The Choose-Correlations procedure chooses one of these predictions for each input parameter, and rewrites the query text based on the choice. Next, Combine-Join generates the combined query text by combining the outer and inner queries using the `LEFT OUTER LATERAL` keyword and adding an `ORDER BY` clause if needed.

```
function COMBINE-JOIN( C )
  join_children ← [ child ∈ C.children | child.alt = ‘J’]
  sql = “SELECT P.*”
  for i ← 1 to join_children.length do
    sql += “, I”+i+“.*”
  sql += “FROM (”+C.query+“) P”
  for i ← 1 to join_children.length do
    child ← join_children[i]
    CHOOSE-CORRELATIONS( child )
    sql += “, LEFT OUTER LATERAL (”+child.query+“) I”+i”
  if C.query.orderby ≠ ∅ then
    sql += “ORDER BY ”+C.query.orderby
  return sql
end
```

Figure 3.17: Combine procedure for outer join.

The join strategy can significantly reduce per-query overhead as compared to the nested execution strategy. It also increases the scope of the server's optimizer by exposing join operations to it. For the example of Figure 3.3, Scalpel can combine Q_2 with its parents Q_1 and, separately, with Q_3. Figure 3.10(b) illustrates the execution trace that would result if these two combinations were performed while Q_3 was executed in its original nested form (the associated context tree is shown in Figure 3.11(b)). As compared to Figure 3.3, in which all queries are nested, the number of query invocations is reduced from ten to four, and the server's optimizer is explicitly made aware that the results of Q_1 are being joined to the results of Q_2 on `t2=C1.s2` while the results of Q_3 are also being joined to the results of Q_2 on `t2=P.v2`. These joins would otherwise have been hidden in the application's code. Figure 3.18 shows the combined queries returned by COMBINE-JOIN for this example.

At present, Scalpel considers the join strategy only when it can determine that the inner query (or queries) will return at most one row. Scalpel's query analyzer implements a support routine, AT-MOST-ONE(Q) to identify queries that return at most one row. For example, if the `t2` attribute is a candidate key for the `T` table, Scalpel can conclude that the query Q_2 will return at most one row [145]. This function acts as an oracle that must be correct when it returns true, but is allowed to return false for queries that can only return one row. These errors lead to missed cases where a join strategy would have been possible, but they do not pose a correctness problem.

The at-most-one condition, together with the use of `LEFT OUTER LATERAL`, ensures that the combined query will return exactly as many rows as the original outer query. The `ORDER BY` clause ensures that these rows will be returned in the order in which they would have been returned by the original outer query. Thus, decoding the result of the combined join query is

```
SELECT   P.*, I1.*
FROM     ( SELECT s1, s2
           FROM   S
           WHERE  s3=:w ) P,
         LEFT OUTER LATERAL
         ( SELECT t1
           FROM   T
           WHERE  t2=P.s2 ) I1
ORDER BY P.s2
```

(a) Combination of Q_1 and Q_2

```
SELECT P.*, I1.*
FROM   ( SELECT v1, v2
         FROM   V
         WHERE  v3=:y ) P,
       LEFT OUTER LATERAL
       ( SELECT t1
         FROM   T
         WHERE  t2=P.v2 ) I1
```

(b) Combination of Q_3 and Q_2

Figure 3.18: Outer join queries generated by (a) COMBINE-JOIN(C_1) and (b) COMBINE-JOIN(C_3) for the context tree in Figure 3.11(b).

straightforward. When the application performs a FETCH on the outer query, Scalpel consumes the next row from the combined query's cursor and extracts the values that correspond to the outer query's columns. When the application performs a FETCH on an inner query, Scalpel simply extracts the attributes required by that query from the current row of the combined query. Scalpel identifies the case where all attributes of the inner query are NULL because there is no matching inner row for the outer row, and returns an empty result set in this case. This case is distinguished from a case where all attributes happen to be NULL by requiring that at least one non-null attribute of the inner query is included in the combined result (for example, a key column).

It is possible to use join-based strategies under less restrictive circumstances than those considered by Scalpel. However, doing so may introduce substantial additional redundancy into the the results of the combined query, which will increase its cost. Figure 3.10(c) illustrates the result of executing all four queries from the example of Figure 3.3 as a single combined outer join query. The shaded parts of the join query result indicate those portions of the result that are redundant. These redundant data are computed by the combined join query, but they should not be returned to the application when it performs FETCH operations on its open cursors. Redundancy generated by the unrelated inner queries can generate excessive overhead due to the duplication of attributes and rows. This redundancy is not present to a great extent in this example since, although Q_2 in C_2 is unrelated to Q_3, Q_2 returns at most one row. In general, context C_2 and C_5 could both return many rows for a single row of C_1, and the combined result set would encode the cross product of these two result sets. This led Shanmugasundaram et al. [166] to label this approach "redundant relations", and they stopped considering it after finding that the performance was poor.

Not only does such redundancy add processing and data redundancy, it also complicates the decoding procedure, since it must now determine which values from the result set are duplicates. In general, it is also possible that Scalpel would have to fetch backwards on the combined query's cursor in order to provide the correct return value for an application's FETCH. This would require either buffering or a scrollable cursor for the combined query. Where supported, scrollable cursors are often more expensive than forward-only cursors. Furthermore, fetching backward may reintroduce some of the the per-request latency that the unified strategy is designed to avoid, since fetching backward may require that rows be re-fetched from the server. By restricting the join strategy to situations in which the inner queries satisfy the AT-MOST-ONE(Q) predicate, Scalpel avoids all of this complexity and cost.

LEMMA 3.2 (OUTER JOIN STRATEGY IS SOUND)
The Outer Join strategy returns the correct multiset of rows to the outer and inner queries.

PROOF. Let Q_1 be the outer query, and $Q_2(\bar{x})$ be an inner query with parameters $\bar{x}$ that returns at most one row. Our correlation detection gives us a mapping M that gives the values of $\bar{x}$ given a row $r \in Q_1$. Then, we define a combined query Q_C as follows:

$$Q_C = Q_1 \, \mathcal{A}^{\text{LOJ}}_{\bar{x},M} \, Q_2(\bar{x}) \tag{3.6}$$

The combined query is submitted to the database server, and the outer query is answered using the following projection:

$$Q_1' = \pi^{\text{ALL}}[\mathit{sch}(Q_1)] \left(\, Q_C \, \right) \tag{3.7}$$

Since $Q_2(\bar{x})$ returns at most one row, Q_C has exactly the same number of rows as Q_1, corresponding to tuples of Q_1 extended either with a single matching row from $Q_2(\bar{x})$ or with NULL values. Therefore, $Q_1' = Q_1$ and the correct rows are returned for the inner query.

When an instance of the the inner query $Q_2(x)$ is submitted while the outer query is positioned on a row r identified by $\text{ROWID}(r) = r_1$, Scalpel first checks that $x = r[M]$; if not, the original query is submitted unmodified. If the predicted correlations match, then the inner query is answered with the following:

$$Q_2' = \pi^{\text{ALL}}[\mathit{sch}(Q_2)] \left(\sigma[\text{ROWID}(Q_1) = r_1 \wedge \neg\text{NULL-SUPPLIED}(Q_2)] \left(\, Q_C \, \right) \right) \tag{3.8}$$

The selection and projection operations used to define query Q_2' give the following:

$$Q_2' = Q_2(r[M]) \tag{3.9}$$

Since we have checked that $x = r[M]$, therefore the returned result $Q_2' = Q_2(x)$ and the inner query returns the desired multiset of rows.

The above argument shows the outer join strategy is sound for joining a single inner query; the extension to two or more inner queries is straightforward. □

3.3.3.2 The Outer Union Strategy

The outer union strategy is illustrated in Figure 3.10(d). Each query is represented by distinct columns in the result of the combined query. Each row corresponds to a tuple that would have been returned by one of the original queries, with NULL values supplied for the columns corresponding to the other queries.

```
function Combine-Union( C )
  union_children ← [ child ∈ C.children | child.alt = ‘U’]
  parent_order_key ← [ C.orderby, C.keys ]
  numcols ← Count-Columns( C, union_children )
  sql = “SELECT ”+[parent_order_key,“U.*”]
  sql += “FROM (”+C.query+“) P”

  inner_order ← []
  type ← 0
  offset ← Count-Columns( C )
  sql += “, LATERAL (”
  sql += “SELECT ”+[type,“P.*”, Null-List(numcols - offset)]
  sql += “FROM ( VALUES( 1 ) ) DT_OneRow”
  for i ← 1 to union_children.length do
    child ← union_children[i]
    Choose-Correlations( child )
    type ← type + 1
    sql += “UNION ALL SELECT ”+[type,Null-List(offset)]
    offset ← offset + Count-Columns( child )
    sql += [“I”+i+“.*”, Null-List( numcols - offset)]
    sql += “FROM (”+child.query+“) I”+i
    inner_order ← [inner_order, child.orderby]
  sql += “) U( type, ”+ [ “u”+i for i = 1 to numcols ]+ “)”
  sql += “ORDER BY ”+[parent_order_key,“U.type”,inner_order]
  return sql
end

function Null-List(n)  ▷ Generate a list of n NULL values
  nulls ← []
  for i ← 1 to n do
    nulls ← [ nulls, NULL ]
  return nulls
end
```

Figure 3.19: Combine procedure for outer union.

```
SELECT    P.s2 AS orderkey1, P.s1 AS orderkey2,
          U.*
FROM      ( SELECT s1, s2
            FROM S
            WHERE s3 = :w
          ) P,
LATERAL
          ( SELECT 0, P.*, NULL
            FROM ( VALUES(1) ) DT_OneRow
          UNION ALL
            SELECT 1, NULL, NULL, I1.*
            FROM   ( SELECT t1
                     FROM   T
                     WHERE  t2 = P.s2 ) I1
          ) U(type, u1, u2, u3)
ORDER BY 1, 2, U.type
```

(a) Combined query

s2	s1	type	u1	u2	u3
A	1	0	A	1	–
A	1	1	–	–	f
B	2	0	B	2	–
B	2	1	–	–	h
C	3	0	C	3	–
C	3	1	–	–	j
D	4	0	D	4	–

(b) Sample output

Figure 3.20: Queries Q_3 and Q_2 combined using outer union.

The procedure for combining outer and inner queries using outer union is given in Figure 3.19. Figure 3.20 shows the result of applying the combination procedure to queries Q_1 and the nested Q_3 query of our running example. The first two columns of the combined query's result (`orderkey1` and `orderkey2`) are the ordering column and key of Q_1 respectively. The third column is a *type field*, which is used to ensure that Scalpel can unambiguously determine which of the original queries a particular row of the outer union result is associated with. In the example from Figure 3.20, rows resulting from the original outer query are tagged with type `0`, while those from the inner query have type `1`. In the more general case, Scalpel assigns a distinct type field value to each of the inner queries. The `ORDER BY` clause is used to ensure that resulting rows appear in the order in which they will be required the application. The rows are ordered first by the ordering attributes of the outer query (`o1` in this case, an alias for `s2`), then by a candidate key of the outer query (`k1` here, an alias for `s1`), then by the type field. This ordering prefix ensures that all of the inner query tuples that correspond to a particular outer tuple are grouped together in the result. Any ordering attributes of the inner queries are then appended so that rows within each group are relatively ordered as specified by the original `ORDER BY` clause. For some DBMSs, it may be possible to eliminate the type field by relying on the sort order of NULL [166].

The first branch of the generated `UNION` construct is responsible for generating the rows corresponding to the outer query. This is accomplished by selecting the attributes of the outer derived table as outer references from `DT_OneRow`, a specially constructed table that returns a single row. A `VALUES` clause is used to generate the required single row; for DBMSs that do not support the `VALUES` construct, a missing `FROM` clause can be used instead. The effect of the first branch of the `UNION` is to include a tuple in the union for each row of the outer query. This ensures that there is exactly one encoded outer row for each row returned from the outer query. For this strategy, an outer join is not needed because of the `DT_OneRow` construct.

The `parent_order` ordering attributes must be taken from the outer query derived table (`P`), not the union (`U`) because the attributes of the union are supplied with NULL values when encoding inner rows. Putting these ordering attributes in the select list of the combined query leads to data redundancy: the ordering attributes of an outer row are duplicated for all corresponding inner rows. Worse, with our current formulation the ordering attributes are duplicated in the rows encoding an outer result row if they were already selected as attributes. Some of the DBMS systems we tested support an extension to SQL/2003 that allows a result to be ordered by attributes that are not in the select list. This approach avoids the data redundancy associated with returning these duplicated values. Scalpel could exploit this vendor-specific capability. Alternatively, Scalpel could be extended to avoid duplicating ordering attributes in the encoded inner rows. These changes would provide better performance, especially when the ordering attributes are a significant contributor to the row size. At present, we have not implemented these extensions.

Unlike the join strategy, the outer union strategy can be applied regardless of the number of rows returned by the inner query. When the application performs a FETCH on either the outer query or the inner query, Scalpel obtains the next row from the combined query's cursor and extracts the values that correspond to the original query's columns. A change in the value of the type field (for example, from `1` to `0`) indicates that there are no more inner query tuples for the current row of the outer.

LEMMA 3.3 (OUTER UNION STRATEGY IS SOUND)
The Outer Union strategy returns the correct multiset of rows to the outer and inner queries.

PROOF. Let Q_0 be the outer query, and $Q_1, \ldots Q_n$ be the n inner queries. Let $\text{UNULLS}(j,k)$ be defined as follows:

$$\text{UNULLS}(j,k) = (\text{NULLS}(\mathit{sch}(Q_j)), \text{NULLS}(\mathit{sch}(Q_{j+1})), \ldots \text{NULLS}(\mathit{sch}(Q_k))) \tag{3.10}$$

The $\text{UNULLS}()$ construct allows us to represent supplying NULL for all of the attributes of a range of queries.

Let $\text{USCHEMA}(i)$ be defined as follows:

$$\text{USCHEMA}(i) = \big(i, \text{UNULLS}(0, i-1), \mathit{sch}(Q_i), \text{UNULLS}(i+1, n)\big) \tag{3.11}$$

The $\textsc{USchema}()$ construct defines the schema of branch i of the combined query. It contains a type field with value i, NULL values for all queries that precede Q_i in the encoded result set, the attributes of Q_i, and, finally, NULL values for all queries that follow Q_i.

The combined query Q_C is formed as follows:

$$Q_C = Q_0\,\mathcal{A}^{\times}_{\bar{x},I}\left(\{(0,\bar{x},\textsc{UNulls}(1,n)\} \biguplus_{i=1,\ldots,n} \pi^{\textsc{All}}[\textsc{USchema}(i)]\,Q_i\big(\bar{x}[M_i]\big)\right) \tag{3.12}$$

where $I = \mathit{sch}(Q_0)$ is used to represent the identity mapping $r[I] = r$.

When the original outer query Q_0 is submitted, Scalpel instead returns the following:

$$Q_0' = \pi^{\textsc{All}}[\mathit{sch}(Q_0)]\left(\sigma[\text{type} = 0](Q_c)\right) \tag{3.13}$$

Applying the projection and selection criteria simplifies this to:

$$\begin{aligned} Q_0' &= Q_0\,\mathcal{A}^{\times}_{\bar{x},I}\left(\{(\bar{x})\}\right) && (3.14)\\ &= \biguplus_{r \in Q_0} \{r\} && (3.15) \end{aligned}$$

The result of Q_0' contains exactly one row r corresponding to each row r in Q_0. Thus, $Q_0' = Q_0$, and Scalpel returns the correct multiset of rows for the outer query.

When the outer cursor is positioned on a row r_1 identified by $r_1[\textsc{Key}[Q_0]] = k_1$ and the application submits an inner query $Q_i(x)$, Scalpel first checks that the predicted correlations $x = r_1[M_i]$ hold; if not, the original query is submitted unmodified. If so, Scalpel answers the inner query with the following:

$$Q_i' = \pi^{\textsc{All}}[\mathit{sch}(Q_i)]\left(\sigma[\text{type} = i \wedge \textsc{Key}[Q_0] = k_1](Q_c)\right) \tag{3.16}$$

Applying the selection predicates simplifies the expression as follows:

$$Q_i' = \pi^{\textsc{All}}[\mathit{sch}(Q_i)]\left(\{r_1\}\,\mathcal{A}^{\times}_{\bar{x},I}\left(\pi^{\textsc{All}}[\textsc{USchema}(i)]\,Q_i\big(\bar{x}[M_i]\big)\right)\right) \tag{3.17}$$

If we apply the projections and expand the apply operator, we obtain the following simplified form:

$$Q_i' = Q_i\big(r_1[M_i]\big) \tag{3.18}$$

Since we have already checked that $x = r_1[M_i]$, this gives $Q_i(x) = Q_i'$, and the correct multiset of rows is also returned for each inner query Q_i. □

3.3.4 Partitioned Execution

Under the unified execution strategies the combined query is issued when the application first opens the original *outer* query. In contrast, under a partitioned strategy, the rewritten, combined query is issued when the application first opens the original *inner* query of a query/context pair. There are many possible partitioned strategies, of which Scalpel's optimizer currently considers two: the *client hash join* strategy and the *client merge join* strategy. The optimizer (described in Section 3.5) sets the `alt` field of nodes in the context tree to 'H' to select the client hash join strategy and 'M' to select the client merge join strategy. We describe these two strategies next.

3.3.4.1 The Client Hash Join Strategy

Under this strategy, the inner query is combined with the outer using a lateral derived table like the one shown in Figure 3.21. This gives a single statement that retrieves *all* of the desired rows from the inner query for all possible outer rows. The first time that the inner query is executed by the application, the combined query is submitted instead to the server. All result rows are fetched and stored in a hash table at the client using the parameters of the inner query from the result set (`PD.s2` in Figure 3.21) as the hash key. When the application opens the inner query, Scalpel searches the hash table using the inner query parameter values as the lookup key to determine the tuples that should be returned to the application. When the outer query is closed, the hash table is discarded.

```
SELECT PD.*, I.*
FROM   ( SELECT DISTINCT s2
         FROM   ( SELECT s1, s2
                  FROM   S
                  WHERE  s3=:w ) P ) PD,
       LATERAL
       ( SELECT t1
         FROM   T
         WHERE  t2=PD.s2 ) I
```

Figure 3.21: Queries Q_1 and Q_2 combined using client hash join.

Figure 3.22 shows the procedure for combining queries under the client hash join strategy. This is similar to the procedure that is used under the unified outer join strategy (Figure 3.17). However, there are a few important differences. First, the partitioned approach rewrites a single CONTEXT by combining it with its parent. Second, only the attributes of the outer query that provide parameter values to the inner query are included in the result for partitioned execution. In

Figure 3.21, only `s2` is needed, not `s1`. Also, each distinct combination of inner query parameter values need only be included once in the outer table. If there are several rows with the same `s2` value, they will all generate the same inner rows. Figure 3.21 shows how a `DISTINCT` keyword can be used to achieve avoid duplicating inner rows. Finally, `LEFT OUTER LATERAL` is not needed, since any correlation values that result in an empty inner query can be left out of the client hash table.

```
function COMBINE-HASH( C )
  P $\leftarrow$ C.parent
  corrs $\leftarrow$ CHOOSE-CORRELATIONS C
  sql  = “SELECT PD.*, I.*”
  sql += “FROM ( SELECT DISTINCT ”+corrs
  sql += “       FROM (”+P.query+“) P )”
  sql += “) PD, LATERAL (”+C.query+“) I”
  if C.orderby $\neq \emptyset$ then
    sql += “ORDER BY ”+C.orderby
  return sql
end
```

Figure 3.22: Combine procedure for client hash join.

Figure 3.10(e) illustrates the the situation in which the client hash join strategy is used for all four of the queries nested under Q_1. While the nested strategy opens 10 cursors, the partitioned client hash join strategy only opens 5 cursors. Furthermore, the number of opened cursors in the partitioned execution strategy does not depend on the number of rows returned from outer queries. However, this strategy does require sufficient memory at the client to hold the hash table, and the CPU of the client machine may make the hash lookups slower than the original, nested strategy.

In the example of Figure 3.10(e), Scalpel combines inner query Q_2 with outer query Q_3. The combined query is executed at most once per *instance* of the parent query Q_3. Scalpel does not need to execute the combined query if the application never opens Q_2 under a particular instance of Q_3. Thus, in the example, the combined query is executed twice, because the application opens Q_3 three times, but in one of those cases it never opens the nested query Q_2 because Q_3 returns no rows. In general, it would be possible to combine Q_2 with *both* its parent Q_3 and grand-parent Q_1 so that the combined query would have to be opened (at most) once per instance of Q_1. Whether this strategy is preferable to combining only with the immediate parent depends on costs and query selectivities. Although it would certainly be possible to consider these alternatives, at present Scalpel only considers the hash join combination of a query with its immediate parent from the context tree.

LEMMA 3.4 (CLIENT HASH JOIN STRATEGY IS SOUND)
The client hash join strategy returns the correct multiset of rows for the outer and inner query.

PROOF. Let Q_1 be an outer query and $Q_2(\bar{x})$ be an inner query with parameters $\bar{x}$. The client hash join strategy does not modify the outer query. However, it computes the results for all invocations of the inner query with the following combined query Q_C:

$$Q_C = \big(\pi^{\text{DIST}}[M](Q_1) \big) \ \mathcal{A}^{\times}_{\bar{x},M} \ Q_2(\bar{x}) \tag{3.19}$$

The combined query Q_C returns the results of $Q_2(\bar{x})$ evaluated with each distinct set of outer bindings from Q_1.

When an instance of the inner query $Q_2(x)$ is submitted while the outer cursor is positioned on a row r, Scalpel first checks that the predicted correlations $x = r[M]$ hold; if not, the original query is submitted unmodified. If the predictions do hold, then Scalpel answers the instance of the inner query using the hash table that was filled from the combined query Q_C as follows:

$$\begin{aligned} Q_2' &= \pi^{\text{ALL}}[sch(Q_2)] \ \big(\sigma[r[M] = x](Q_C) \big) && (3.20) \\ &= \pi^{\text{ALL}}[sch(Q_2)] \ \Bigg(\sigma[r[M] = x]\Big(\big(\pi^{\text{DIST}}[M](Q_1) \big) \ \mathcal{A}^{\times}_{\bar{x},M} \ Q_2(\bar{x})\Big) \Bigg) && (3.21) \end{aligned}$$

Since Scalpel has checked that $x = r[M]$ for the current outer row r, we know that there is at least one row r in Q_1 such that $x = r[M]$. Applying the distinct projection $\pi^{\text{DIST}}[M](Q_1)$ and the selection criteria $\sigma[r[M] = x]$ therefore gives the following simplification:

$$\begin{aligned} Q_2' &= \pi^{\text{ALL}}[sch(Q_2)] \ \big(\{r\} \ \mathcal{A}^{\times}_{\bar{x},M} \ Q_2(\bar{x}) \big) && (3.22) \\ &= Q_2(r[M]) && (3.23) \\ &= Q_2(x) && (3.24) \end{aligned}$$

The set Q_2' that Scalpel returns for an invocation of $Q_2(x)$ contains exactly the desired multiset of rows.

□

3.3.4.2 The Client Merge Join Strategy

The client hash join strategy amounts to a distributed hash join executed at the client. Similarly, the client merge join strategy amounts to a distributed merge join implemented at the client. For this to work properly, Scalpel must ensure that the inner and outer tuples arrive at the client in the proper order for merging.

The merge join approach consists of opening rewritten versions of both the outer query and the inner query. The outer query is rewritten so that the result has a known total ordering, and so that it includes those attributes that we guess will be used as correlation parameters to the inner queries (based on our training period). The inner query is rewritten by combining it with the original outer query so that it returns matching inner rows for *all* of the rows of the outer query. This is similar to the rewriting that is done to the inner query under the client hash join strategy. However, under the client merge join strategy, the rewritten inner query is ordered to match the known ordering that we imposed on the outer query results, as well as any order requirements specified in the original inner query.

```
function Combine-Merge( C )
  P ← C.parent
  corrs ← Choose-Correlations C
  parent_order ← [ P.orderby, P.keys ]
  sql  = “SELECT ”+parent_order+“, C.*”
  sql += “FROM ( (”+P.query+“) P )”
  sql += “, LATERAL (”+C.query+“) C”
  sql += “ORDER BY ”+[parent_order, C.orderby]
  ▷ The parent query will be modified to be ordered by parent_order (Figure 3.25 line 294)
  return sql
end
```

Figure 3.23: Combine Procedure for Client Merge Join

Figure 3.23 shows the procedure for producing the combined inner query, and Figure 3.24 shows the query that would result from combining the inner and outer queries from our running example. In this case, the rewritten inner query is ordered by `P.Id`, which is the sort order of the outer query. No additional ordering constraints are imposed by the original inner query. In Section 3.3.5, we show how the parent query is is modified to include the ordering attributes needed to totally order the results (Figure 3.25 line 294).

The first time that the inner query is opened, Scalpel submits instead the combined query. In response to a Fetch on the inner query, Scalpel first checks the values of the sorting and key attributes of the current row of the outer query. It then advances the cursor of the combined inner query to the first row for which the corresponding attributes do not exceed the current values from the outer. If combined query’s sorting attributes match those of the current outer tuple, Scalpel returns the values of the inner query attributes. If they exceed those of the current outer tuple, this indicates the end of the application’s inner query’s result set. Scalpel closes the combined inner query’s cursor when the outer query’s cursor is closed.

Figure 3.10(f) illustrates the the situation in which the client merge join strategy is used for all four of the queries nested under Q_1. As was the case for the client hash join, Scalpel only considers the merge join combination of a query with its immediate parent from the context tree, e.g., Q_2 is combined with Q_3 but not with Q_1. Thus, the resulting pattern of query instances is

```
SELECT   P.s2 o1, P.s1 k1, I.*
FROM     ( SELECT s1, s2
           FROM   S
           WHERE  s3=:w ) P,
         LATERAL
         ( SELECT t1
           FROM   T
           WHERE  t2=P.s2 ) I
ORDER BY o1, k1
```

Figure 3.24: Combined inner query for the client merge join strategy.

almost the same as that of the client hash join strategy, except for the ordering of the result of the combined inner queries. Unlike the hash join strategy, the merge join strategy does not require that the result set of the combined inner query be stored at the client. However, the merge join strategy does impose an additional sorting burden on the server. Scalpel's optimizer uses its cost model to choose between these alternatives.

Lemma 3.5 (Client Merge Join Strategy is Sound)
The client merge join strategy returns the correct multiset of rows for both the outer and inner queries.

Proof. Let Q_1 be an outer query and Q_2 an inner query that is executed using the client merge join strategy.

The client merge join strategy modifies only the ordering specification of the outer query, by appending additional attributes to ensure a total ordering. Therefore, the multiset of rows returned for the outer query is correct.

A combined inner query Q_C is formed to return all inner rows as follows:

$$Q_C = Q_1 \; \mathcal{A}^{\times}_{\bar{x},M} \; Q_2(\bar{x}) \tag{3.25}$$

When the outer cursor is positioned on row r identified by $r[\textsc{Key}[Q_1]] = k_1$ and an instance of the inner query $Q_2(x)$ is submitted, Scalpel first checks that the predicted correlations $r[M] = x$ hold. If so, then Scalpel answers the query with the following results from the combined query:

$$\begin{aligned}
Q_2' &= \pi^{\textsc{All}}[\mathit{sch}(Q_2)] \Big(\sigma[\textsc{Key}[Q_1] = k_1] \; (\; Q_C \;) \Big) && (3.26)\\
&= \pi^{\textsc{All}}[\mathit{sch}(Q_2)] \Big(\sigma[\textsc{Key}[Q_1] = k_1] \; (\; Q_1 \; \mathcal{A}^{\times}_{\bar{x},M} \; Q_2(\bar{x}) \;) \Big) && (3.27)\\
& && (3.28)
\end{aligned}$$

We know that exactly one row r exists in Q_1 with $r[\textsc{Key}[Q_1]] = k_1$. If we push the selection criteria down, we can simplify to the following:

$$
\begin{aligned}
Q'_2 &= \pi^{\textsc{All}}[\mathit{sch}(Q_2)]\ \big(\ \{r\}\ \mathcal{A}^{\times}_{\bar{x},M}\ Q_2(\bar{x})\ \big) && (3.29)\\
&= \pi^{\textsc{All}}[\mathit{sch}(Q_2)]\ \Big(\ \{r\} \times Q_2\ \big(\ r[M]\ \big)\Big) && (3.30)\\
&= Q_2\ \big(\ r[M]\ \big) && (3.31)\\
&= Q_2(x) && (3.32)
\end{aligned}
$$

Therefore, Scalpel returns the desired multiset of rows for the outer and inner queries of the client merge join strategy. □

3.3.5 Rewriting a Context Tree

The preceding section described rewrite procedures that generate a rewritten query for each of the execution strategies we consider. The unified rewrite methods combine a context with its unified children while the partitioned rewrites combine a context with its parent. Each of these rewrite procedures operates on an individual Context node at a time. In this section, we describe how Scalpel rewrites an entire context tree by operating on one context at a time. First, Section 3.3.5.1 describes how Scalpel uses Action objects to represent steps the Prefetcher should follow to implement the prefetch strategies. Next, Section 3.3.5.2 describes how the Rewrite-Tree procedure (Figure 3.25) rewrites an entire context tree. Finally, Section 3.3.6 provides a summary of rewriting a context tree.

3.3.5.1 Representing Run-Time Behaviour With Action Objects

In addition to the rewritten query text, Rewrite-Tree generates information that is used at run-time to direct the Prefetcher component how to respond to requests from the client application. This information is recorded in Action objects. Each Action object contains an `acttype` field that indicates how the action should be applied. Further, the Action object contains a `resultquery` field that contains the query text that defines the result set returned when the action is used, and a `submitquery` that contains the rewritten combined query. For example, for context C_2 in Figure 3.11(c) we would have an Action object of type Submit-Hash. The `resultquery` field would be the text of query Q_2, and the `submitquery` field would be the text of the combined query shown in Figure 3.21.

The Action objects encapsulate the operations that will be performed on behalf of a single execution strategy for a context `C`. The `C.resultquery` field specifies the query text that defines the result set returned by the Action object. When the client submits `resultquery`, the

Prefetcher uses internal bookkeeping in the ACTION to find if the desired result is already available. For example, for the client hash join strategy, the Prefetcher searches in a hash table of prefetched results. When the prefetched results are not available, the `submitquery` is passed on to the DBMS to retrieve the results for the current request and to prefetch additional results. For example, in the client hash join case the combined query is submitted and the results are used to fill a hash table.

S	C	Action Type	Description
‘J’	P	INTERPRET-JOIN	Pass combined query to next action in `actions`. Return only original attributes at the parent context.
‘J’	C	DECODE-JOIN	Read attributes from current parent row. If this row was null-supplied, return an empty result set.
‘U’	P	INTERPRET-UNION	Pass combined query to next action in `actions`. Return rows to parent that have type attribute of 0.
‘U’	C	DECODE-UNION	Move forward in parent’s result set until a row with the appropriate type field is found; return all such rows.
‘M’	P	INTERPRET-MERGE	Pass ordered query to next action in `actions`. Return only original attributes.
‘M’	C	SUBMIT-MERGE	Submit combined, ordered query to DBMS. Return all rows from the combined result set that match the ordering attributes of the current parent row.
‘H’	C	SUBMIT-HASH	Submit the combined query to the DBMS, and populate a hash table with the results. Satisfy OPEN requests from the hash table.
‘N’	C	SUBMIT-NEST	Submit the query to the DBMS.

Table 3.3: Types of ACTION objects. The first column is the execution strategy for which the type is used, and the second column is C if the action type applies to the context itself or P if it applies to the parent context. The third column gives the `acttype` value, and the last column describes how it is used.

The client hash join strategy generates an ACTION object only for the child context that is rewritten. In contrast, the client merge join needs to change not only the query that is submitted at the child context, but also the query for the parent. The parent query is altered to add ordering attributes that ensure a total ordering permitting merging of result rows. For the client merge join, an ACTION of type SUBMIT-MERGE is associated with the child, while an ACTION of type

INTERPRET-MERGE is used for the parent context. The SUBMIT-MERGE indicates that the combined, ordered query should be submitted for the child, and result sets are interpreted by advancing on the combined, ordered result set. The INTERPRET-MERGE specifies that a variant of the parent query should be submitted, rewritten to provide a total ordering. The result of this rewritten query is interpreted by returning a result set to the client containing only the original attributes.

The SUBMIT-HASH and SUBMIT-MERGE types of action submit a query directly to the DBMS. In contrast, actions of type INTERPRET-MERGE are associated with the parent context, which will also have an execution strategy of its own. For example, in Figure 3.11 the context C_3 will have an action of type INTERPRET-MERGE corresponding to C_4, and it will also have an action of type SUBMIT-MERGE resulting from its own annotation with strategy 'M'. Actions are maintained in an ordered list `actions` for each context.

The `actions` list may have multiple ACTION objects with a type prefixed by INTERPRET- before the final ACTION object. Each of these INTERPRET- actions has the effect of interpreting the result set that answers the associated `resultquery` from a result set that answers the associated `submitquery`. For example, a subset of the columns may be returned (INTERPRET-MERGE, INTERPRET-JOIN) or a subset of the rows and columns (INTERPRET-UNION). The Prefetcher never directly submits a query for a INTERPRET- action. Instead, the `submitquery` is used when constructing the next action in the list. The `submitquery` of the preceding ACTION object always matches the `resultquery` of this next object. At the end of the list is an ACTION object with a type prefixed with SUBMIT- or DECODE- (described in the sequel).

The partitioned strategies create a SUBMIT- action for the associated context, leading to a combined query being submitted for the child at run-time. In contrast, the unified strategies do not submit any query at the child context, instead submitting a single unified query at the parent context. This is represented in the context tree by associating DECODE-JOIN or DECODE-UNION as the last element in the `actions` list for the child objects and associating INTERPRET-JOIN or INTERPRET-UNION as a non-final element of the parent's `actions` list. The REWRITE-TREE procedure calls APPEND-ACTION to add ACTION objects to the `actions` field of a context as it processes the tree. The APPEND-ACTION procedure also modifies the text of the `query` field of the context to reflect the `submitquery` of the most recently added action. In this way, the `query` field of the context is maintained with the currently needed query as the rewrite proceeds.

Table 3.3 summarizes the ACTION types supported by Scalpel and outlines how they are used at run-time. In the next section, we describe how the REWRITE-TREE procedure builds the ACTION objects for the entire context tree.

3.3.5.2 The REWRITE-TREE Procedure

Figure 3.25 shows the REWRITE-TREE procedure. Scalpel calls REWRITE-TREE for the root of the context tree (`treeroot`) to generate the combined queries for all nodes in the context tree. First, REWRITE-TREE(C) recursively rewrites all child nodes (line 291). This recursive rewriting will modify the `query` field of the child contexts, complete their `actions` lists, and possibly add ACTION objects to the list `C.actions`. Next, if any child uses the client merge join strategy, REWRITE-TREE adds an ACTION object of type INTERPRET-MERGE to `C.actions`.

Next, if there are any 'J'-annotated children, REWRITE-TREE calls COMBINE-JOIN`(C)` to generate a combined query `nsql` that retrieves the original result set required by `C` and the attributes needed for all 'J'-annotated children. A call to APPEND-ACTION`( C,` INTERPRET-JOIN`, nsql)` adds a new ACTION object `A` that will be used to pass on the combined query `nsql` and interpret the results for the context `C`. Further, the `C.query` field is updated to contain the combined query text `nsql`.

Similarly, if there are any 'U'-annotated children of `C`, then REWRITE-TREE calls COMBINE-UNION`(C)`. In this case, however, the `C.query` field used in COMBINE-UNION`(C)` (line 226) will be the modified query that was returned by COMBINE-JOIN(C) (if there are 'J'-annotated children). In this way, the query combine procedures build on the results of prior calls as the context tree is modified in place.

After combining any unified children of `C`, REWRITE-TREE modifies `C` to reflect the strategy `C.alt` that was selected for it. If `C` is annotated with a unified strategy, then REWRITE-TREE adds an action of type DECODE-JOIN or DECODE-UNION. No query will be submitted for `C` at run-time; instead, the results will be interpreted from the combined results of `C`'s parent context. If, on the contrary, `C` is annotated with a partitioned strategy, then REWRITE-TREE calls COMBINE-HASH or COMBINE-MERGE to generate a combined query `nsql`. These calls combine the current `C.query` with the parent query. If the unified rewrite procedures modified `C.query`, then the modified query is used when generating the combined partitioned query. This combined query will be submitted to the DBMS at run-time when no prefetched results are available, and this fact is recorded by adding an ACTION object of type SUBMIT-HASH or SUBMIT-MERGE with the new combined query text. Finally, if the context `C` is annotated with strategy 'N', then an ACTION object of type SUBMIT-NEST is added to `C.actions`. This action will submit the current query stored in `C.query`. This could be the original query text, or the result of a unified combined procedure.

Figure 3.21 shows a sample trace of REWRITE-TREE when processing the context tree shown in Figure 3.27(a), and Figure 3.27 shows the state of the context tree after steps in the sample trace.

```
structure Action
  acttype = “”              ▷ The type of action to perform
  resultquery = Nil         ▷ The query defining the result set
  submitquery = Nil         ▷ The combined query that will be submitted instead
  ...                       ▷ Additional bookkeeping information is omitted
end

procedure Append-Action( C, type, submitquery )
  A ← new Action( type, C.query, submitquery )
  C.query ← submitquery                       ▷ Change the current query
  C.actions ← [C.actions, A]                  ▷ Append the new action
end

procedure Rewrite-Tree( C )
  for child ∈ C.children do
    Rewrite-Tree(child)
  if ∃ { child ∈ C.children | child.alt = ‘M’}
    ▷ If any child uses strategy ‘M’, submit a rewritten query that is totally ordered
    nsql ← Add-Keys-To-Order-By( C.query )
    Append-Action( C, Interpret-Merge, nsql )
  if ∃ { child ∈ C.children | child.alt = ‘J’}
    nsql ← Combine-Join( C )
    Append-Action( C, Interpret-Join, nsql )
  if ∃ { child ∈ C.children | child.alt = ‘U’}
    nsql ← Combine-Union( C )
    Append-Action( C, Interpret-Union, nsql )
  case C.alt
    when ‘J’ then Append-Action( C, Decode-Join, C.query )
    when ‘U’ then Append-Action( C, Decode-Union, C.query )
    when ‘H’ then
      nsql ← Combine-Hash( C )
      Append-Action( C, Submit-Hash, nsql )
    when ‘M’ then
      nsql ← Combine-Merge( C )
      Append-Action( C, Submit-Merge, nsql )
    when ‘N’ then Append-Action( C, Submit-Nest, C.query )
end
```

Figure 3.25: Combine procedure to rewrite an entire context tree.

```
Rewrite-Tree( $C_1$ )
  Rewrite-Tree( $C_2$ )
    Append-Action( $C_2$, Decode-Join, $Q_2$ )
  Rewrite-Tree( $C_3$ )
    Rewrite-Tree( $C_4$ )
      Append-Action( $C_4$, Decode-Join, $Q_2$ )
    $Q_{3b} \leftarrow$ Combine-Join( $C_3$ )
    Append-Action( $C_3$, Interpret-Join, $Q_{3b}$ )
    Append-Action( $C_3$, Decode-Union, $Q_{3b}$ )
  $Q_{1b} \leftarrow$ Combine-Join( $C_1$ )
  Append-Action( $C_1$, Interpret-Join, $Q_{1b}$ )
  $Q_{1c} \leftarrow$ Combine-Union( $C_1$ )
  Append-Action( $C_1$, Interpret-Union, $Q_{1c}$ )
  Append-Action( $C_1$, Submit-Nest, $Q_{1c}$ )
```

Figure 3.26: Steps of Rewrite-Tree.

3.3.6 Summary of Query Rewriter

When Scalpel detects nested request patterns in a client's request stream, it can choose between alternative execution strategies, some of which prefetch the results for future inner queries based on predictions made using a context tree.

Unified execution strategies provide a modified query that is submitted at the root of a nested pattern. The modified query combines the result set for the requested root query, and also encodes the results of inner queries that are predicted to be executed in the future. The outer join unified strategy uses a `LEFT OUTER LATERAL` derived table construct to join the results request for the parent query with the inner query results. As this strategy is only used when the inner queries return at most one row, the combined query returns exactly the same number of rows as the original root query.

The outer union strategy is another unified strategy that combines the parent query with an outer union of the inner queries, augmented with a type attribute that represents the query associated with each row. Again, the `LATERAL` derived construct is used to combine the outer and inner queries. In this case, a `LEFT OUTER LATERAL` is not needed as the outer union returns at least one row for each row of the outer query.

In contrast to the unified execution strategies, the partitioned execution strategies operate with a single context. The client hash join strategy submits a combined query that returns all of the rows needed for the inner query associated with all rows of the outer query. These rows are added to a hash table, and individual Open requests for the inner query are satisfied from the hash table.

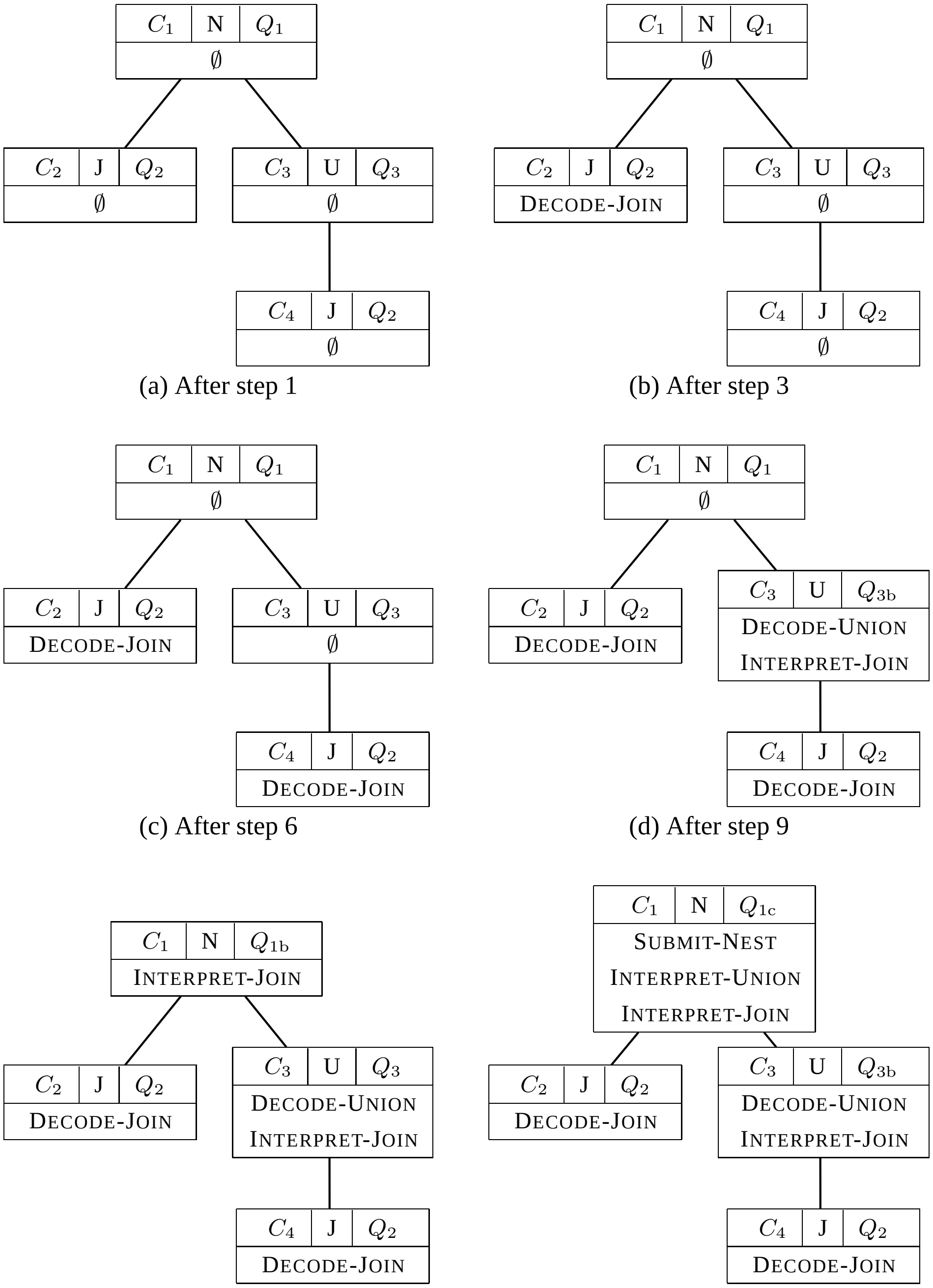


Figure 3.27: Context tree after executing steps of Figure 3.26. Nodes show the context, execution strategy, current query, and list of actions (`C.actions`) ordered from bottom to top.

The client merge join strategy is similar to the client hash join strategy, except that it uses ordering on the server instead of a hash table on the client. The parent query is modified to guarantee a total ordering, and a combined version of the inner query is submitted matching the parent's order. Individual OPEN requests for the inner query are satisfied by fetching forward on the combined result set. A comparison of the current value of ordering attributes from the parent row is used to determine where the prefetched result set for the inner query begins and ends within the combined result.

Scalpel uses the REWRITE-TREE to prepare a context tree for execution. This procedure traverses the tree in depth-first order. For each context `C` in the tree, it fills in the field `C.actions` with a list of ACTION objects. These objects describe how the Prefetcher should respond to OPEN, FETCH, and CLOSE requests for the context. After rewriting the tree, the contexts and associated `actions` lists provide all of the information needed to execute the selected execution strategies at run time. The results of REWRITE-TREE are stored persistently in the context tree for use at run-time, and the Prefetcher is responsible for performing these specified actions.

3.4 Prefetcher

When the client application submits OPEN requests to the database server, Scalpel intercepts these requests and tracks the current context. If the context and query matches an edge in the optimized tree, Scalpel executes an alternative query instead and responds to the application's OPEN request by decoding the alternate result set. Figure 3.28 gives a simplified sketch of how the alternate strategies are executed by the Prefetcher.

First, on line 319 Scalpel verifies that all correlation predictions made by the Pattern Detector (Section 3.2.2) hold for this OPEN request. If the correlations do not hold, Scalpel's correlation prediction has failed, and the stored execution strategy cannot be used.

If the predicted parameters match the actual values supplied, then the stored execution strategy is executed. The activities required by each ACTION objects in the `actions` list are performed in order until the last action, which always returns a result set. For a child context assigned a unified strategy, the last action is prefixed with DECODE-, meaning that the result set is decoded from a combined query submitted by the context's parent. For contexts assigned a partitioned strategy, the last action in the list is prefixed with SUBMIT-, meaning that a combined query is submitted and used to prefetch the needed data.

Each list may have one or more INTERPRET- actions before the final action (SUBMIT- or DECODE-). Any INTERPRET- actions control how the result set is interpreted for the client (for example, by removing columns or skipping rows from other branches of a union strategy). Table 3.3 summarizes the ACTION types supported by Scalpel and outlines how they are used at run-time.

```
currctxt ← treeroot                ▷ The current context
function Run-Open( Q, parms )
  C ← find( currctxt.children, Q )
  C.lastinput ← parms
  currctxt ← C

  if not Check-Corr-Predictions(action,parms) then
    ▷ Correlation prediction failed: submit Q to DBMS unmodified

  for A ∈ C.actions do
    assert Q = A.resultquery
    case A.acttype
      when Interpret-Join then ...
      when Interpret-Union then ...
      when Interpret-Merge then ...
      when Decode-Join then
        if Null-Supplied(...) then ▷ Return a cursor over an empty result set
        else ▷ Return an interpreted cursor on parent's current row
      when Decode-Union then
        ▷ Move parent's cursor forward until its type field matches A.type.
        if Not-Found then ▷ Return a cursor over an empty result set
        else ▷ Return an interpreted cursor on the parent rows matching A.type.
      when Submit-Hash then
        if A.cachedresults = Nil then
          ▷ Submit Q to DBMS and load the A.cachedresults hash table with the results
        rs ← find( A.cachedresults, parms )
        if rs = Nil then ▷ Return a cursor over an empty result set
        else return rs
      when Submit-Merge then
        if A.cursor = Nil then
          ▷ Submit Q to DBMS and set A.cursor to the result set
        ▷ Move forward in combined cursor to first row matching parent key
        if Not-Found then ▷ Return a cursor over an empty result set
        else ▷ Return an interpreted result over combined rows matching the parent key
      when Submit-Nest then
        ▷ Submit t.combinedqueryo DBMS
    Q ← A.submitquery
  ▷ This point is not reached; a result set is returned from within the above loop.
end
```

Figure 3.28: Sketch of execution of alternate strategies.

As RUN-OPEN processes the `C.actions` list, it considers the value of `A.acttype` field. For types INTERPRET-JOIN, INTERPRET-UNION, and INTERPRET-MERGE, no query is submitted immediately (lines 325–327). Instead, these ACTION objects change the way a result set is interpreted, for example by restricting the attributes returned. The details for these types are omitted.

Action objects of type DECODE-JOIN and DECODE-UNION are associated with contexts that have been assigned strategy 'J' or 'U'. In this case, no query is submitted. Instead, the result set for the OPEN request is generated by interpreting the combined result set that was opened for the parent context. It may be that the combined query contains no row corresponding to the query `Q`. In this case, an empty result set is returned. Otherwise, a result set is returned that generates its rows based on the cursor in the parent context.

The partitioned strategies have a single action associated with each partitioned child (annotated with SUBMIT-HASH or SUBMIT-MERGE). These actions submit a combined query the first time that the partitioned child's query is submitted for an instance of the outer query.

The SUBMIT-HASH action loads a hash table with the results of its combined query. The hash table is indexed by the parameter values that are predicted to be submitted on future opens of the inner query, and the entries in the hash table are buffered result sets. The hash table is loaded by finding or adding the appropriate result set for each row of the combined query, then adding the row to the buffered result set. The SUBMIT-HASH action satisfies OPEN requests using buffered result sets from the hash table (if found) or an empty result set (if no match is found).

The SUBMIT-MERGE action submits a combined query that is ordered to match the ordering specification and key of the parent query. Recall that the parent query was modified to be totally ordered (line 294). The SUBMIT-MERGE action satisfies OPEN requests by moving forward in the combined result set until either a row is found with order/key attributes matching the parent query's current order/key or until a row is found beyond these attributes. In the first case, the action returns an interpreted result set that returns rows from the combined results set with the current order/key values; in the second case, an empty result set is returned.

Scalpel uses an ACTION object of type SUBMIT-NEST to indicate that the optimizer has selected a nested execution strategy for a context. Every time an OPEN request is submitted for the context, Scalpel sends an OPEN request to the DBMS for the `submitquery` associated with the action object. This is either the original query or a combined query generated by unified children of the context.

The RUN-OPEN function returns a result set at run-time. Further, Scalpel's Prefetcher implements RUN-FETCH and RUN-CLOSE (not shown). The RUN-FETCH function generates rows based on the selected strategy, and also maintains the `currctxt.lastoutput` field with the most recently fetched row, allowing the predicted correlations to be verified. The RUN-CLOSE function updates the `currctxt` global variable to point to the parent context. Further, RUNCLOSE releases all prefetched results for the inner queries nested as children of the close query.

3.4.1 Mistaken Predictions

All of the unified and partitioned rewriting strategies that Scalpel considers depend on the query attribute correlations that it learns during its training phase. If these learned correlations hold during the run-time phase, then Scalpel can use its rewritten queries to return the appropriate values in response to the application's FETCH requests as we have already described. However, Scalpel must be prepared to cope with the possibility that its predictions will be wrong.

At run-time, whenever the application opens an inner query for which Scalpel has determined to use a unified or partitioned execution strategy, Scalpel first checks whether the expected attribute correlations actually hold (line 319). That is, it compares the inner query's parameter values with the values from the current rows of the outer query (or queries) with which the inner query is expected to be correlated. If they all match, then the correlation has held as expected and Scalpel can proceed to answer the application's subsequent FETCH requests using its rewritten query. If any correlations do not hold, then Scalpel cannot use the rewritten query as planned. Instead, Scalpel submits the *original, unmodified inner query* to the server and uses the results of that query to provide values to the application.

Since a single rewritten query normally takes the place of many instances of the original inner query, it may be the case that Scalpel issues *both* the rewritten inner query and one or more instances of the original query to the server. For example, under the partitioned strategies, Scalpel issues the rewritten, combined query the first time the application attempts to open the original inner query. If that first inner query instance is properly correlated, Scalpel will issue its rewritten query. However, the next instance of the inner query may not be properly correlated, and Scalpel will be unable to use the already-opened rewritten query to answer it as planned. In this case, it must issue the original query to obtain a correct answer for the application.

These cases, when they occur, constitute failures of Scalpel's semantic prefetching strategies. Such failures cause Scalpel to do extra work, since it may execute queries that are wholly or partially unnecessary. However, since Scalpel is always free to issue the application's original, unmodified query, they do not lead to incorrect answers.

3.5 Pattern Optimizer

Section 3.3 described five alternative strategies that can be employed to execute a nested query pattern. After the training phase has built a context tree identifying rewrite candidates, an optimization step is used to determine which of these execution strategies will be used for each context in the tree.

3.5.1 Valid Execution Plans

Consider a node T that appears in the context tree (for example, node C_2 in Figure 3.5). If the FULLY-PREDICTED procedure returns TRUE for T, then the parameters of the query Q associated with T can be predicted based on the context of execution and we can consider executing Q using other strategies than the original (nested) strategy. Scalpel can execute Q using the nested ('N'), outer join ('J'), outer union ('U'), client hash-join ('H') or client merge-join ('M') execution strategy. A plan is described by a context tree in which each context is annotated with 'N', 'J', 'U', 'H' or 'M'. For example, Figure 3.11 shows six plans for the context tree example of Figure 3.3. These strategies correspond to the execution traces shown in Figure 3.10.

With 5 possible annotations per node, there are up to 5^n possible execution strategies for a tree with n contexts. Of these 5^n strategies, some are not permitted. Section 3.3.3.1 described the at-most-one condition restriction for the outer join strategy. We rely on the AT-MOST-ONE`(Q)` support function to identify queries for which the join strategy is permitted. In addition, the root node of the context tree and the immediate children of the root node can only be annotated with the nested strategy as these contexts have no parent query with which to combine. A final complication results from the way that Scalpel rewrites the child contexts annotated with the outer union strategy ('U'). The encoded results for these children are ordered by increasing `type` values. The ordering matches the order that the original child queries were submitted by the client application. If the application were to submit the child queries in an alternate order, Scalpel would not be able to decode the prefetched results for all children. For this reason, Scalpel does not consider strategies where two contexts are annotated with 'U' if they were observed to be submitted in conflicting orders during the training period.

The enumeration algorithm iterates through all of the plans that are permitted by the above rules and the optimizer selects the plan with the lowest estimated cost.

3.5.2 Ranking Plans

We rank execution plans using estimates of the response time (in seconds) experienced by the client application. Scalpel's Cost Model is responsible for estimating cost parameters needed for optimization. The implementation of the Cost Model is described in Chapter 4. We describe our ranking algorithm based on the following support routines which are defined in Chapter 4.

EST-COST(Q) Estimate the total cost for query Q, including server, client, and communication cost elements based on observed costs and server estimates.

EST-ROWS(*Q*) Estimate the number of rows returned by Q.

EST-INTERPRET(`T,numopens,numrows`) Estimate the cost of processing the ACTION objects in `T.actions` to interpret the results for context T. The estimate is based on `numopens` application calls to OPEN returning a total of `numrows` rows. For example, for the client hash join strategy, estimate the cost to add `numrows` rows to a hash table and look up a result set `numopens` times.

The Cost Model provides estimates of the cost of individual requests; in order to rank strategies, we use the routine EST-COST-TREE (Figure 3.29) to estimate the cost of an entire context tree based on the costs of the requests submitted by the tree.

```
352  function EST-COST-TREE(T, prtopens)
353    prtrows ← EST-ROWS( T.parent.query )
354    qrows ← EST-ROWS( T.query )
355    nopens ← EST-P(T) × prtopens × prtrows
356    partopens ← EST-P0(T) × prtopens
357    case T.alt
358      when 'N' then cost ← nopens × EST-COST( T.query )
359      when 'J' or 'U'  then
360        ▷ No query is submitted at this node
361        ▷ The only cost is associated with interpreting the encoded results (line 364)
362      when 'H' or 'M'  then
363        cost ← partopens × EST-COST( T.query )
364    cost ← cost + EST-INTERPRET( T, nopens, partopens )
365    for child ∈ T.children do
366      cost ← cost + EST-COST-TREE( child, nopens )
367    return cost
368  end
```

Figure 3.29: Estimating the cost of a plan.

Figure 3.29 gives an overview of how Scalpel estimates the cost of a context tree with associated execution strategies. The EST-COST-TREE function estimates the cost of a tree that has execution strategies assigned to each node and a rewritten query generated by the REWRITE-TREE procedure.

The EST-COST-TREE function is called with a context T and `prtopens`, the estimated number of times that the query associated the parent context is opened. The call EST-COST-TREE(`treeroot,1`) is used to estimate the cost of one execution of the entire context tree.

If node T is annotated with a the nested strategy ('N'), then the query Q associated with T is executed every time that a row is returned from the outer query and the local predicates pass. We use `nopens` to estimate this number, calculated by multiplying `prtrows` (the estimated number of rows returned from the parent context's query) by `prtopens` and EST-P (an estimate of the probability of executing the inner query for each outer row). The cost estimate for an 'N' node is `nopens` multiplied by the expected cost of executing `T.query` one time (returned by EST-COST).

If node T is annotated with a unified strategy ('U' or 'J'), then `T.query` is not actually executed at run time. Instead, the rows for the node are decoded from a combined result set submitted by the parent context. The cost associated directly with the context T consists of the client's cost to interpret the combined result rows, and the EST-INTERPRET function provides an estimate of these costs (line 364).

The partitioned strategies do not execute their rewritten query every time the application submits the inner query. Instead, they execute the rewritten query at most once for every time that the parent context's query is opened. The variable `partopens` is initialized with the estimate (based on EST-P0(T)) of how often the rewritten query will be submitted. The cost of the partitioned strategies is estimated using the product of `partopens` and the estimated query cost. In addition, the cost of interpreting the results of the combined query is included in the estimate for these strategies (line 364).

So far, the `cost` variable has been initialized to an estimate for the costs directly associated with node T. Children of T may also introduce costs by executing other queries. The EST-COST-TREE function recurses to account for the cost of child requests.

3.5.3 Exhaustive Enumeration

Exhaustive enumeration provides one way to choose an execution plan. With this approach, Scalpel exhaustively enumerate all plans for a context tree. For each enumerated plan that is valid according the above rules, REWRITE-TREE is be used to assign rewritten queries to each node in the tree. Finally, plans are ranked by estimating the cost to execute the tree one time, using cost estimation methods provided by the Cost Model applied to the rewritten requests associated with the rewritten context tree. The plan for the tree with the lowest estimated cost is stored persistently in the 'Contexts+Rewrites' store (Figure 2.1). These stored context and actions are used at run-time to perform the actions selected by the optimizer.

3.6 Experiments

The costs associated with execution strategies depend on a number of factors described in the earlier sections. This section presents experiments that give a sense of how these factors combine to affect system performance. Table 3.4 shows the computers used in the experiments, and Table 3.5

Computer	**Processor**	**O/S**
A	1.8 GHz Pentium IV	Windows XP
B	2 × 2.2GHz Pentium XEON	Windows 2003 Server
C	3GHz Pentium IV	Windows XP
D	733MHz Pentium III	Windows 2000

Table 3.4: Available computers.

Name	**Client**	**Server**	**Communication Link**	U_0 **Overhead (ms)**
LCL	C	C	Local shared memory	0.3
LAN1	B	C	1Gbps LAN	1.1
LAN0.1	B	C	100Mbps LAN	1.4
WiFi	A	C	11Mbps 802.11b WiFi	11.6
WAN	A	D	1Mbps Cable modem + WAN	468.9

Table 3.5: Tested configurations and the per-request overhead U_0

shows the configurations of these computers. We ran our tests with three commercial DBMS products. The license agreements prevent us from identifying them. As results for all three systems were consistent (although with different constants) we show results for only one DBMS product.

We tested a sample program with a single outer query Q_0. We used a number of inner queries $Q_1, Q_2, \ldots, Q_\mathrm{F}$, with F set to 2 unless otherwise noted. All inner queries are executed for each row of the outer query that passes any local predicates. The outer query returns $2048/F$ rows from a sequential scan with a range predicate (this setup gives a constant number of inner query opens when varying fanout F). Each inner query returns R rows for each invocation using an index range scan (R is set to 1 by default so that the outer join strategy can be compared). We used an additional outer join in each query that allowed us to vary the server cost of the query without affecting the number of rows returned.

All tests were run with JDK 1.5.0 and JDBC drivers provided by the DBMS vendors. The database instance was fully cached to minimize the variance in server costs. A prototype implementation of Scalpel was used for the experiments; combined queries were automatically combined using the LATERAL keyword for one of the DBMS systems, and vendor-specific equivalents for the other two systems.

In our experiments, we vary the following factors:

Fanout (F) The number of inner queries executed for each outer row.

Selectivity (P_0) A predicate selectivity independent of values in the outer rows.

Selectivity (P_1) A predicate selectivity dependent on values in the outer rows.

Inner Rows (R) The number of rows returned from the inner queries.

Inner Columns (L) The number of integer columns in each inner query.

Outer Cost (C_O) The cost of the outer query.

Inner Cost (C_I) The cost of the inner query.

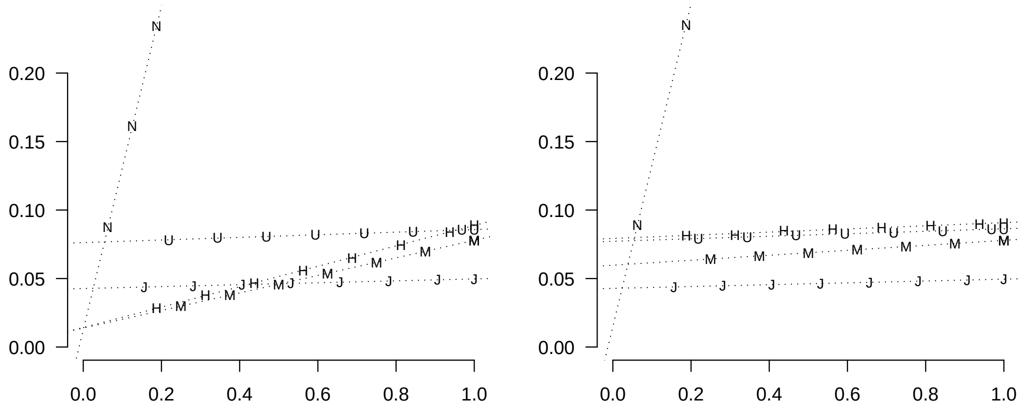
All results show the average of a number of repetitions, with the number of repetitions selected so that the standard error of the mean (σ_M) is less than 5% of the mean for each measurement. The SQL statements were prepared once for each factor combination and prepare time is not included in the reported measurements. We use slightly different settings of the independent variable for each of the strategies to reduce overlap in the resulting charts.

3.6.1 Effects of Client Predicate Selectivity

Scalpel uses two parameters, EST-P and EST-P0, to model the selectivity of client predicates, as discussed in Section 3.2.3. In our experiments, we vary the selectivities of two predicates (called P_0 and P_1) in the driver program. Predicate P_0 is evaluated once per instance of the outer query. If it is false, the inner query is not executed at all. The selectivity of P_0 corresponds to the estimate EST-P0. The second predicate, P_1, controls how many times the inner query is submitted relative to the number of rows fetched from the outer query. The estimate EST-P is equal to a combined predicate selectivity of $P = P_0 \times P_1$. Figure 3.30 shows the run-time of the nested, unified, and partitioned execution strategies with varying selectivity of the P_0 and P_1 client predicates.

In Figure 3.30(a), the selectivity of P_1 is fixed at 1.0 and the selectivity of P_0 is varied. In Figure 3.30(b), P_0 is fixed at 1 and P_1 is varied using the same values as P_0. With a selectivity of $P_1 = 0$, the inner query is never submitted and the behaviour is equivalent to the $P_0 = 0$ case: the partitioned approaches are equivalent to nested (both executing only the outer query). We avoid this discontinuity by omitting the setting $P_1 = 0$.

For the original nested execution strategy, execution time is proportional to the product $P = P_0 \times P_1$. The outer query is always executed one time, and the two inner queries are executed an average of $|Q_0|P$ times. Because the nested strategy depends on the product of P_0 and P_1, it has similar behaviour in both Figure 3.30(a) and (b). For low values of P, the nested strategy is optimal. With small P values, the inner queries are only rarely executed, and no resources



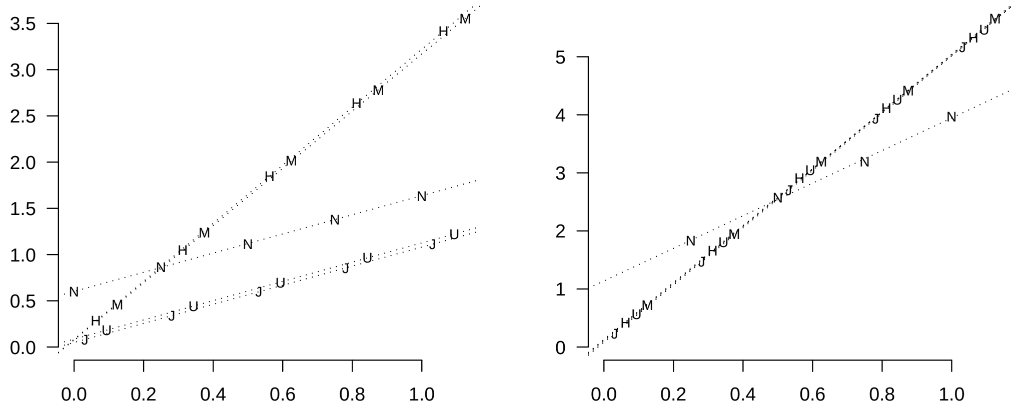
(a) Varying selectivity P_0 ($P_1 = 1$) (b) Varying selectivity P_1 ($P_0 = 1$)

Figure 3.30: Run times with varying predicate selectivity on configuration LCL.

are wasted fetching unneeded results. The run time of the nested strategy grows rapidly with increasing P, as the inner queries are executed for more and more outer rows.

For the unified execution strategy, the execution time is largely independent of the selectivities of P_0 and P_1. Regardless of the results of these predicates, the unified query is executed and all rows are fetched by the client. While most of the costs of the unified strategy are independent of P, there is a slight linear dependence resulting from the cost of decoding the attributes for the inner queries. If an inner query is not opened, its attributes are not decoded from the combined result set, leading to slightly lower run-times.

The partitioned execution strategy fetches all possible rows from the rewritten inner query when an inner query is first opened (the P_0 predicate evaluated to true). This gives the partitioned strategy a strong dependence on the P_0 selectivity. The P_1 selectivity also has a small effect: for the client hash join strategy, it changes the number of lookups performed in the hash table; for the client merge join strategy, it changes the number of rows that are interpreted from the combined result set. Figure 3.30(b) shows the relatively weak dependence of the partitioned strategies on the P_1 selectivity.



(a) Outer query cost factor (b) Inner query cost factor

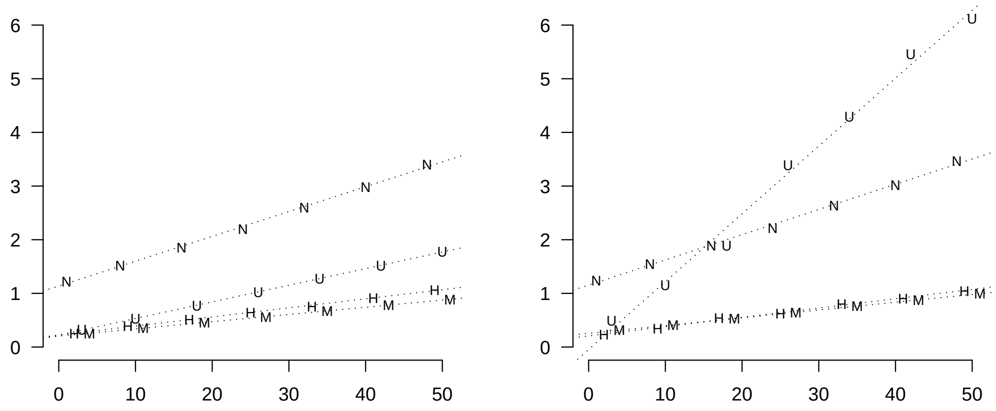
Figure 3.31: Run times with varying query cost on configuration LCL with $P_0 = 1$, $P_1 = 0.5$. Outer and inner cost factors are not directly comparable.

3.6.2 Query Costs

The execution costs of different optimization strategies depends on the cost of the outer and inner queries that are combined. Figure 3.31 shows the execution times of nested, unified, and partitioned strategies for our sample program. Predicate selectivity was fixed at $P_0 = 1$ and $P_1 = 0.5$.

The nested strategy increases linearly with the cost of the outer. The unified strategies also increases linearly with the cost of the outer, and the slope of increase is the same as the nested strategy for the cost of the outer query: both strategies execute the outer query once. In contrast, the partitioned strategies execute the outer query once for each of the child queries in addition to the original, for a total of three executions. This gives the partitioned strategies a higher dependence on the cost of the outer query. Figure 3.31(a) shows the run time for each the execution strategies when varying the cost of the outer query.

When we consider instead the cost of the inner queries, we find that all of the prefetching strategies behave in a similar way. The unified and partitioned strategies both compute the result of these inner queries for all outer rows, while the nested strategy only evaluates the inner query when the P_1 predicate passes (with selectivity 0.5 in this experiment). When P_1 is less than 1, the prefetching strategies are cheaper than nested when the inner query cost is relatively low, but become more expensive with increasing cost of the inner query. Figure 3.31(b) shows the run time



(a) Fanout $F = 2$ (b) Fanout $F = 8$

Figure 3.32: Run time (s) with varying number of columns in inner queries on configuration LCL with $P_0 = 1$, $P_1 = 1$, $R = 8$. Note that the outer join ('J') strategy could not be compared as more than one row is returned from the inner queries.

for each the execution strategies when varying the cost of the two inner queries. With this configuration, the nested strategy submits 1024 inner queries while the prefetching strategies compute the results for all 2048 predicted inner queries.

3.6.3 Number of Columns

If we vary the number of columns returned by the inner queries, the cost of executing the nested pattern increases. Figure 3.32 shows the run-time for different strategies with varying number of columns in the inner queries. Figure 3.32(a) uses a fanout of $F = 2$, while Figure 3.32(b) uses $F = 8$.

There are several effects contributing to the increasing execution cost with an increasing number of columns. First, the per-request overhead increases because more resources are needed to initialize and describe communication buffers for more columns. This increase has the most impact on the nested strategy because it submits the largest number of requests. Second, the server and client computation costs increase due to the higher number of columns that are formatted and interpreted for each row fetched. Third, the cost of sort operations grows as the size of the materialized rows grows. This factor affects the outer union and client merge join strategies as they both add ordering attributes, possibly leading to a sort being added to the execution plan used by the DBMS. In our tests, the original queries already had sort operations, so this factor did not have an effect. Finally, the client cost for hash join increases as the size of the rows stored in the hash table increase. More columns require more memory and more time spent copying the data into the hash table.

The merge, hash, and nested strategies have similar behaviour with both fanout $F = 2$ and $F = 8$. In contrast, the union strategy has a higher slope with a configuration of 8 inner queries. The union strategy encodes a combined result set into a single unioned result using NULL values to represent attributes that are not appropriate for a given row. The total number of NULL values returned grows with the product of the query fanout F and the total number of columns in all the queries that are combined.

3.6.4 Execution Costs

Figure 3.33 shows the CPU costs and total time for each of the strategies. Results are shown for a configuration with $P_0 = 1$, $P_1 = \frac{1}{8}$, and an inner cost factor $C_I = 8$. One row was returned from each invocation of the two inner queries.

Client and server CPU costs were measured using O/S functions and DBMS-specific requests respectively. The difference between elapsed execution time and the measured CPU costs is labelled *latency*. In cases where the elapsed time was less than the sum of the server costs, a negative latency is shown.

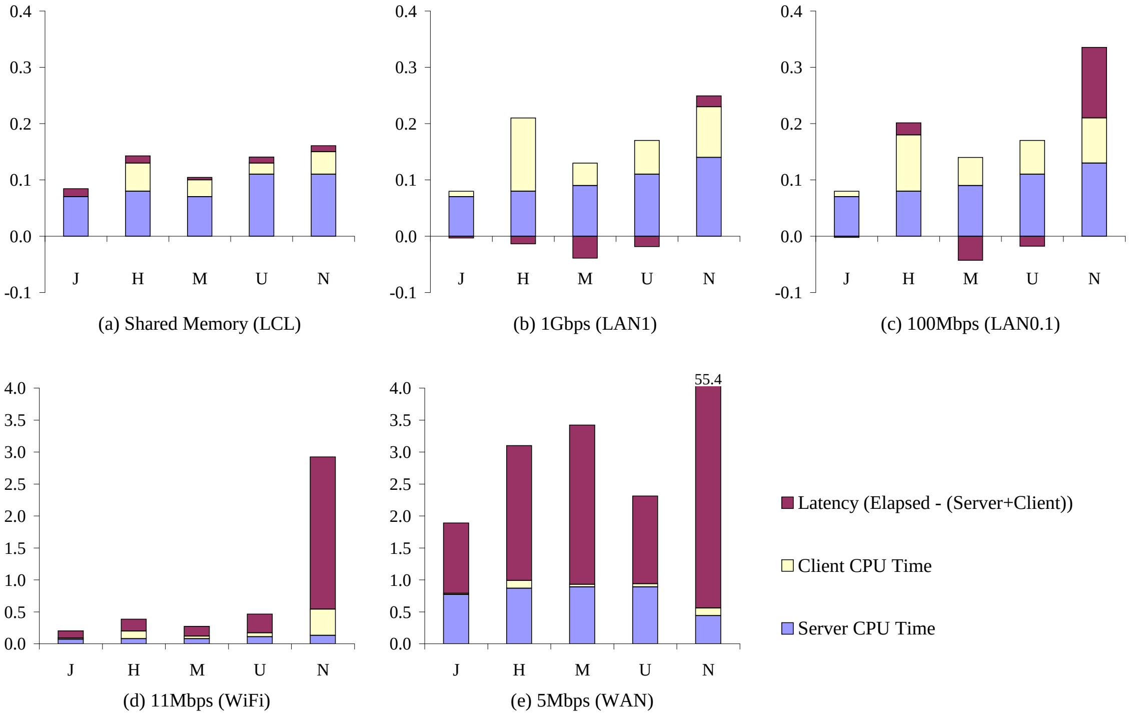
The server costs for the WAN case are higher for two reasons: the server machine is slower, and packet compression was used.

There are several interesting observations that we can draw from Figure 3.33.

First, the original nested execution strategy is not significantly slower than the optimal strategy (join) in the LCL configuration. It is reasonable for system developers to select a nested strategy for this case, especially considering the difficulty of estimating the selectivity of local predicates and the complexity of manually combining queries.

Second, we observe that the join and merge strategies are *faster* in the LAN1 and LAN0.1 configurations than in the LCL configuration. The LAN configurations allow for overlap between server processing and client processing (the machine D used in LCL is a uni-processor). The nested strategy, on the other hand, takes more than 1.5 times as long in the LAN1 configuration and more than 2 times as long in the LAN0.1 configuration when compared to the LCL configuration.

Third, the unified and partitioned execution strategies reduce not only latency but also the client and server CPU costs. This cost savings results from fewer messages that need to be for-



(a) Shared Memory (LCL)

(b) 1Gbps (LAN1)

(c) 100Mbps (LAN0.1)

(d) 11Mbps (WiFi)

(e) 5Mbps (WAN)

Figure 3.33: Run time (s) with varying network configurations and $P_1 = \frac{1}{8}$, $C_I = 8$. Note that the bar for ‘N’ is truncated in figure (e), with an actual height of 55.4s.

matted, sent, and interpreted. Even in the LCL configuration, all strategies but union use less server CPU time than the nested strategy, despite the selectivity $P_1 = \frac{1}{8}$.

Finally, while the savings for the LAN configuration are low in absolute terms (about 125ms), the savings are very significant for WiFi and WAN; for example, with a wireless configuration the joined strategy saves nearly 1.5s of elapsed time, and the savings grows to 100s with the WAN setup.

The original nested execution strategy is close to optimal for a local connection. Even if a LAN configuration is considered, absolute benefits are modest for an single execution of the outer and inner query (hundreds of milliseconds) so system developers might decide to use the simpler nested implementation. However, the server costs are 50% lower with the joined variant. Further, if other deployments with higher network latency are used in the future, the nested implementation will be unsatisfactory. Finally, the predicate selectivity $P_1 = \frac{1}{8}$ is quite low. If the value of P_1 in particular configurations is higher, then the nested strategy will be far from optimal.

3.6.5 Scalpel Overhead

The Scalpel system monitors all database requests during training. At run-time, all OPEN requests are intercepted. If a query is not being re-written, the original query is submitted to the DBMS and the result set is wrapped in a monitor object that can detect when the cursor is closed. This monitoring is needed to maintain the current context for triggering rewrites.

We measured the overhead Scalpel adds to opening a query. With a local configuration (LCL), the overhead is 27μs per query, or 7.0%. The overhead is the same for all network setups, and drops to 1.6% in the LAN configuration due to the higher base cost.

The overhead would be reduced if Scalpel were integrated with the vendor-provided JDBC driver (for example, eliminating the need to use wrapper objects to track the current context). However, the overhead seems acceptable as it is low in absolute terms, and consists entirely of client CPU time. Even with current overhead levels, significant gains can be made without significantly impacting user interaction due to the benefits of rewrites performed by the Scalpel system.

3.7 Summary of Nested Request Patterns

We have found that client applications submit request streams that contain nested requests. This nesting appears when an outer query is opened and an inner query is submitted while the outer query is still open. Typically, the inner query has input parameter values that are supplied with values fetched from the database. In this way, we can view a nested request pattern as a type of distributed join implemented in the client code.

While input parameters to inner requests are typically drawn from output columns of the enclosing query, we have also found that, in some cases, they are equal to an input parameter of an enclosing query. In some instances, the input parameter is always equal to a constant value. We monitor correlations between input parameters and these three types of sources. At the end of a training period, we predict future parameter values based on correlations that have always held so far.

From a performance standpoint, it would appear that a join in client application code is not a good idea. There are software engineering reasons that might make it difficult to avoid such a join without destroying desirable properties such as encapsulation. In other cases, we have found that the nested execution strategy is, in fact, optimal (at least in particular configurations). This optimality can arise as a result of local predicates in the client application that limit the number of inner requests to a fraction of the rows returned by the outer query. We use two parameters to estimate the effects of local predicates: EST-P0 estimates the probability that the inner query will be executed at all for an instance of the outer query, while EST-P estimates the proportion of rows of the outer for which the inner query is submitted.

After a training period, Scalpel identifies a context tree that represents the nested structure we observed. An optimization step is used to select an execution strategy for each node in the tree. There are three broad classes of strategy that we consider: nested execution, unified execution, and partitioned execution. Nested execution corresponds to the original strategy, where an inner request is submitted up to once per row of the outer. If a node is executed with a unified strategy, then at run-time a single request is submitted to retrieve the results for the parent node and the unified child node. In contrast, with partitioned execution a separate cursor is submitted for the parent node and child node. However, unlike the nested execution strategy, the partitioned execution strategy opens at most one inner cursor per instance of the outer, instead of for each row of the outer. Table 3.6 summarizes the strategies that we consider.

Strategy	**Symbol**	**Adds** `ORDER BY`	**Probability**
Nested Execution	N	No	EST-P(C)
Outer Union Strategy	U	Yes	1
Outer Join Strategy	J	No	1
Client Hash Join	H	No	EST-P0(C)
Client Merge Join	M	Yes	EST-P0(C)

Table 3.6: Summary of rewrite strategies. The first column gives the name of the strategy, and the second gives the symbol used to represent it. The third column indicates whether the strategy requires that we add an `ORDER BY` clause to the rewritten query, and the last column shows the quantity that determines how often results are retrieved for inner queries.

The unified and partitioned execution strategies that we consider encode the desired result set as the result of a combined query. We decode this result set to retrieve the desired results. The outer union strategy encodes a parent query and one or more inner queries using an outer union rewrite, where each query is associated with a separate union branch. An `ORDER BY` clause is added to ensure that rows for inner queries come after the associated outer query and in the required order. The outer union strategy computes the result of the inner queries for each outer row. The outer union strategy can be used when the inner queries return 0 or more rows (empty result sets are detected by a missing union branch). The additional NULL values returned by the outer union strategy contribute to a higher communication cost, which is partly why the outer union strategy is out-performed by the outer join strategy when it can be used.

The outer join strategy can only be used for inner queries that return at most one row. Inner queries are combined with their parent using outer joins. There is no need to add extra `ORDER BY` elements, and, as with the outer union strategy, the result of each inner query is computed for each outer row.

In contrast to the unified strategies, the partitioned strategies use a separate cursor for the parent query and each child query; as such, they incur one extra per-request overhead U_0 per child. However, the rewritten query for child is only submitted if the P_0 predicate is true. We have observed some client applications where the inner query is either submitted for all rows of one instance of the outer, or none. The partitioned strategies typically outperform the unified strategies in this case as the rewritten, combined query is only submitted when the first inner request is submitted, meaning that the results for inner rows are actually needed

In the client hash join strategy, a rewritten query is submitted when the inner query is first observed. This rewritten query encodes the results of the inner query evaluated for all rows of the outer query. These multiple results are stored in a hash table, which is used to satisfy future request. The client hash join requires sufficient client memory to hold the combined results. Further, the approach may not reduce latency if the client CPU is not fast enough to make fetching results from the hash table faster than the per-request latency U_0. An alternative that avoids these concerns is the client merge join.

In the client merge join strategy, `ORDER BY` elements are added to the parent query when it is submitted to ensure that the outer rows are totally ordered. When an inner query is submitted, a rewritten request is submitted to retrieve the results of the inner query evaluate for all outer rows. This rewritten request has `ORDER BY` elements that return the encoded result sets in the same (total) order as the adjust parent query. In this way, inner queries are satisfied by fetching forward in the combined result set, effectively performing a distributed merge join on the client.

By using a training period, Scalpel identifies nested request patterns. Using cost-based optimizations, the execution strategy estimated to be cheapest is selected. We have found that, even where optimal, the original nested execution may perform very badly when moving to another configuration with different predicate selectivity or network latency. The context-based prefetches performed by Scalpel can significantly reduce the exposed latency of these applications; in many cases, the execution costs at the client and server are also reduced due to the lower number of messages.

4 Cost Model

In order to make effective cost-based rewrite decisions, the Pattern Optimizer needs estimates of various cost parameters. The Cost Model is responsible for providing these estimates. In general, the cost for a query can be represented as a vector, with one component for each resource being used. For example, we could have components associated with client cost, communication cost, and server cost. With the cost vector, we associate a total latency caused by a request. This latency can be estimated using the sum of estimated server, communication, and client costs. Clearly, other ranking functions can be used instead; for example, we could choose to rank based only on server execution costs, or we could attempt to create a more precise model of latency by estimating the amount of overlap for server, network, and client processing costs. In the current implementation, we concentrate only on ranking by total latency, expressed in seconds. For this reason, we do not show costs as vectors, but instead show the scalar latency.

Table 4.1 shows the quantities estimated by the Cost Model. The $\textsc{Est-Cost}(Q)$ and $\textsc{Est-Rows}(Q)$ functions are estimates for queries which have either been observed during a training period or are a combination of such queries formed by the Query Rewriter. The value U_0 is an estimate of per-request overhead. The $\textsc{Est-Interpret}(C, p, r)$ function estimates the cost of interpreting the results of an encoded result set for context C.

Quantity	**Description**
$\textsc{Est-Cost}(Q)$	Estimated cost for Q
$\textsc{Est-Rows}(Q) = \lvert Q \rvert$	Estimated rows returned by Q
U_0	Overhead of a single request
$\textsc{Est-Interpret}(C, p, r)$	Cost of interpreting results for context C with p opens, r rows

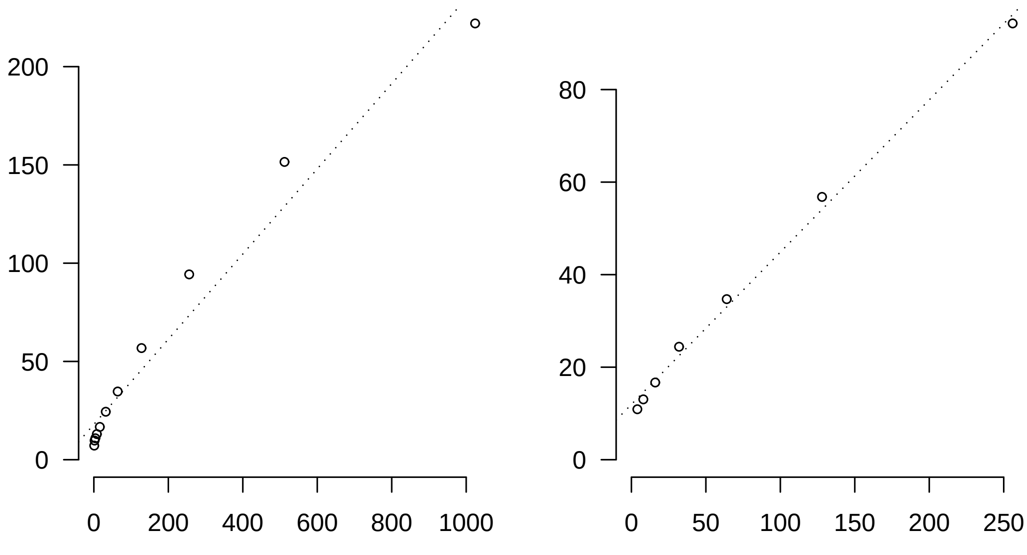
Table 4.1: Estimated quantities.

Scalpel's Cost Model uses a combination of calibration, observed quantities, and support routines provided by the DBMS in order to provide the estimates in Table 4.1.

4.1 Estimating Per-Request Overhead U_0

When we submit a request to a database server, the total latency is a result of several components: formatting and transmitting the request via inter-process communication, finding the appropriate execution plan for the request (either by optimizing or finding a stored plan), initializing the execution data structures, and, finally, executing the selected plan. All but the last of these costs represents a fixed per-request overhead. This overhead is independent of the cost of executing the selected plan (although it may depend on factors such as the complexity of the query).

We can assess the impact of this per-request overhead using an experiment inspired by Bernstein, Pal and Shutt [19]. We create a table T_1 containing 1024 rows of rows, each with an integer primary key and 100 integer columns. We fetch all 1024 rows from the table by submitting N range queries, each of which retrieves $1024/N$ rows using a range predicate on the primary key.



(a) $T(N) \approx 0.217N + 18$ (b) $T(N) \approx 0.329N + 12$

Figure 4.1: Run times (ms) with varying number of requests to the server. Dashed lines show the results of linear regression; however, a linear regression is not well justified by these results.

Figure 4.1 shows the result of this experiment on the LCL configuration (described in Table 3.5). In principle, the server performs the same amount of useful work (fetching rows), while the per-request overhead is performed N times. This would lead us to expect a linear relationship between run-time and N, where the slope of the relationship is U_0 and the intercept is the server cost to fetch the desired rows. Figure 4.1(a) shows the results of N requests with $N = 2^i$ for $i \in [0, 10]$. Clearly the run time $T(N)$ is proportional to N, but it does not appear to be a linear relationship: compare the points to the linear regression shown with a dashed lines. There is more changing in this experiment than just the number of times the per-request costs are incurred. Figure 4.1(b) removes the top two and bottom two data points providing a region that is more closely approximated by a linear model (having a coefficient of determination $R^2 = 0.995$ compared to $R^2 = 0.965$ for Figure 4.1(a)), but the use of a linear model to estimate U_0 using the slope does not appear to be well justified.

We take the results shown in Figure 4.1 as confirmation that there are per-request overheads, but the approach does not provide a good estimate of this cost. Instead, we time the cost of a query that we expect to have very low server execution costs. We time the execution of a query that fetches a single row with a constant value:

```
SELECT T.x
FROM   ( VALUES (1) ) T(x)
```

For this simple query, the server execution costs (apart from per-request overhead) are negligible. When Scalpel is configured for a particular installation, it uses a calibration step to estimate U_0. Scalpel executes the above request a number of times and uses the average execution time as the estimate of U_0 for the configuration. Table 4.2 shows the calibrated value for each of the configurations that we have tested (the configuration details are given in Table 3.5).

Configuration	U_0 **(ms)**
LCL	0.27
LAN1	1.1
LAN0.1	1.4
WiFi	11.6
WAN	468.9

Table 4.2: Tested configurations and the per-request overhead U_0

4.2 Estimating the Cost of Interpreting Results

When we prefetch, we submit a combined query that returns an encoding of multiple result sets. At run-time, we interpret this encoded result set to return the desired result sets. This interpretation adds extra costs in the client process. We calibrate this cost by comparing the execution time when interpreting the multiple result sets to the run time if we just fetch the rows of the combined query without interpretation.

Type	T	Estimated Overhead (μs)		
Nested Execution	N	$18.1p$	$+$	$0.07r$
Outer Union	U	$32.4p$	$+$	$11.08r$
Outer Join	J	$27.0p$	$+$	$0.00r$
Client Hash Join	H	$26.5p$	$+$	$8.68r$
Client Merge Join	M	$22.7p$	$+$	$4.63r$

Table 4.3: Estimated overhead EST-INTERPRET(C, p, r) of interpreting combined result sets.

As with U_0, Scalpel uses a calibration step when it is configured for a particular installation to estimate the overhead of each execution strategy. Table 4.3 shows the estimated linear combination EST-INTERPRET(C, p, r) for each type T of interpretation that could be used for C. The p parameter represents the number of encoded result sets, and r represents the number of rows. The values in Table 4.3 are the result of a linear regression based on 100 iterations of each type with a combination of opens and fetches.

The results in Table 4.3 represent the overhead of interpreting result sets from an encoded result set, except for the first row. The Nested Execution results represent the overhead of keeping track of query open and close requests when no prefetching is performed. As such, this is an estimate of the cost of the Call Monitor overhead.

4.3 Estimating the Cost of Queries

In addition to estimates of U_0 and EST-INTERPRET(C, p, r), the Pattern Optimizer needs to estimate the cost of queries, along with the number of rows they will return. Estimates are needed not only for queries that we have observed during the training period, but also for composite queries generated by the Query Rewriter.

For some DBMS products, we can define functions SRV-COST(Q) and SRV-ROWS(Q) that give the server's estimate of the cost and cardinality of a query. We could use these functions to implement EST-COST(Q) and EST-ROWS(Q) regardless of whether we have previously observed Q. In principle, this approach has several appealing qualities. Unfortunately, it also has significant limitations. First, the DBMS query optimizer's cost estimates typically don't include communication latency (as these costs are incurred by each of the plans the optimizer considers). This latency is an important cost that Scalpel seeks to minimize, and it is therefore important to have a reasonable estimate of this latency in the same units as the cost estimate. Second, the query optimizer typically does not have the values of host variables when generating a plan, and therefore can only give a generalized estimate for any parameter values. In contrast, Scalpel has observed actual parameter values from the application. If the training period is representative of run-time, these observed costs will be more accurate than the server's estimates. Finally, the estimates used by the DBMS are intended only to rank plans in relative order. These estimates often do not translate well to linear approximations of cost that can be combined with external calibrated values such as latency.

We have found that, where possible, it is better to estimate the cost of individual queries by monitoring the execution costs and row counts. We loosely follow an approach suggested by Zhu [209] and Rahal, Zhu and Larson [147]. During the training period, the Cost Model observes for each submitted query Q the number of rows returned by ($\textsc{Avg-Rows}(Q)$) and the average running time ($\textsc{Avg-Cost}(Q)$). For a query Q that has been observed during training, we use the averages as our estimate (Equation 4.1 and Equation 4.2).

$$\textsc{Est-Cost}(Q) \equiv \textsc{Avg-Cost}(Q) \tag{4.1}$$

$$\textsc{Est-Rows}(Q) \equiv \textsc{Avg-Rows}(Q) \tag{4.2}$$

In addition to estimates for queries observed during training, Scalpel needs estimates for queries that are generated by the Query Rewriter. We have never observed these combined queries being executed, so we cannot base estimates on past behaviour. Instead, we use an analytical model to estimate these cost attributes. This model estimates the parameters of a combined query using estimates of the components of the query. Further, if a $\textsc{Srv-Cost}$ function can be defined, Scalpel incorporates predictions made by the DBMS so that the Pattern Optimizer can detect opportunities where the DBMS chooses a strategy that performs better than the naïve approach.

The Query Rewriter combines queries using two basic constructs: lateral derived tables and outer unions. We represent these constructs using symbols as follows. Query $Q_i \otimes Q_j$ is the query formed by using an outer query Q_i joined to Q_j with a lateral derived table as described in Section 3.3.2.2. If we are using instead a left outer lateral derived table, we write $Q_i \hat{\otimes} Q_j$. We use the notation $Q_i \hat{\uplus} Q_j \hat{\uplus} Q_k \hat{\uplus} \ldots \hat{\uplus} Q_m$ to represents the outer union of queries $Q_i, Q_j, \ldots Q_m$ as described in Section 3.3.3.2.

4.3.1 Estimating the Cost of a Lateral Derived Table

Given a combination $Q_i \otimes Q_j$, we would like to produce an estimate $\textsc{Est-Cost}(Q_i \otimes Q_j)$ and also $\textsc{Est-Rows}(Q_i \otimes Q_j)$. Recall that, by the definition of the semantics of lateral derived tables, the combined result set contains each row of Q_i concatenated with the result of Q_j evaluated under the outer bindings supplied by that row. A straightforward implementation technique for this operator is based on nested-loops joins. We use $\text{JNL-}\textsc{Cost}(Q_i, Q_j)$ and $\text{JNL-}\textsc{Rows}(Q_i, Q_j)$ to estimate the attributes of $Q_i \otimes Q_j$ using the nested-loops model:

$$\text{JNL-}\textsc{Cost}(Q_i \otimes Q_j) \equiv \textsc{Est-Cost}(Q_i) + \textsc{Est-Rows}(Q_i) \times (\textsc{Est-Cost}(Q_j) - U_0) \quad (4.3)$$

$$\text{JNL-}\textsc{Rows}(Q_i \otimes Q_j) \equiv \textsc{Est-Rows}(Q_i) \times \textsc{Est-Rows}(Q_j) \quad (4.4)$$

Equation 4.3 uses a nested-loops model to estimate the cost of $(Q_i \otimes Q_j)$ based on estimates for the component queries Q_i and Q_j. Note that we subtract U_0 for the inner because it is not submitted from the client application, and therefore we assume that we save U_0. The number of rows in the result is the product of the outer and inner table. To estimate the parameters for a left outer lateral derived table, we modify the above row count estimate using $\max$ to account for the outer join preserving all outer rows as shown in Equation 4.2. We use the same estimate for the cost of an inner and outer lateral derived table (Equation 4.1). Although there is a difference in the communication costs for NULL-supplied values, we do not currently include estimates for this difference.

$$\text{JNL-}\textsc{Cost}(Q_i \hat{\otimes} Q_j) \equiv \text{JNL-}\textsc{Cost}(Q_i \otimes Q_j) \quad (4.5)$$

$$\text{JNL-}\textsc{Rows}(Q_i \hat{\otimes} Q_j) \equiv \max(\text{JNL-}\textsc{Rows}(Q_i \otimes Q_j), \textsc{Est-Rows}(Q_i)) \quad (4.6)$$

We assume that the nested loops cost estimate $\text{JNL-}\textsc{Cost}(Q_i \hat{\otimes} Q_j)$ provides an upper bound on the execution time. The nested loops strategy is available to the server, and we assume that the DBMS query optimizer does not choose a more expensive strategy by mistake. However, the DBMS may in fact be able to use a more efficient join strategy. We can use SRV-COST to find such cases, but the earlier limitations we noted make it difficult to incorporate the results of SRV-COST directly. Instead, we define a function SRV-SAVINGS (Equation 4.7) that estimates the relative savings the server predicts for executing two queries using a better strategy that the naïve nested loops join.

$$\textsc{Srv-Savings}(Q_i \otimes Q_j) = \frac{\textsc{Srv-Cost}(Q_i \otimes Q_j)}{\textsc{Srv-Cost}(Q_i) + \textsc{Srv-Rows}(Q_i) \times \textsc{Srv-Cost}(Q_j)} \quad (4.7)$$

The value $\textsc{Srv-Savings}(Q_i \otimes Q_j)$ is the ratio of the server's estimated cost for joining Q_i and Q_j to the cost of performing a nested loops join when we use the server's cost estimates instead of Scalpel's predictions. The value of $\textsc{Srv-Savings}(Q_i \otimes Q_j)$ is close to 1 when the server chooses a straightforward nested loops strategy (or one with a similar cost). If the server discovers a better strategy (for example, by finding a better join algorithm or by exploiting shared subexpressions as suggested by Sellis [160]), the ratio $\textsc{Srv-Savings}(Q_i \otimes Q_j)$ will be less than 1.

Equation 4.7 gives the relative benefit estimated by the DBMS query optimizer for combining the outer and inner query into one request. This quantity will vary between 0 and 1, with 1 meaning there is no benefit to the server and values close to 0 meaning that there is substantial benefit. If we maintained separate cost components in a vector, we would apply the SRV-SAVINGS multiplier to the server cost component of EST-COST. As we represent cost as a scalar representing a combination of server, communication, and client costs, it is not possible to apply the reduction only to the server components. Instead, we apply the reduction to the entire estimated cost. We use a configuration parameter K with $0 \leq K \leq 1$ to form a weighted average between Scalpel's unmodified prediction (JNL-COST$(Q_i \otimes Q_j)$) and using the full effect of the estimated savings (SRV-SAVINGS$(Q_i, Q_j) \times$ JNL-COST$(Q_i \otimes Q_j)$). With a K value near 0, we pay little attention to the server's predicted savings; near 1, we apply most of the effect of the savings. Equations 4.10 shows how we define the weighted average. One complication is the cost of U_0. The savings in server cost resulting from combining Q_i and Q_j into one query does not reduce the per-request overhead U_0. Therefore, we remove this quantity before forming the weighted average.

$$\begin{aligned} J &= \text{JNL-COST}(Q_i \otimes Q_j) - U_0 && (4.8) \\ S &= \text{SRV-SAVINGS}(Q_i \otimes Q_j) && (4.9) \\ \text{EST-COST}(Q_i \otimes Q_j) &\equiv KSJ + (1-K)J + U_0 && (4.10) \end{aligned}$$

The estimate provided by EST-COST$(Q_i \otimes Q_j)$ is a blending of estimates from the DBMS query optimizer and an estimate Scalpel makes assuming a nested loops strategy. The K parameter controls how much credence is given to the server's estimates.

4.3.2 Estimating the Cost of an Outer Union

The query $Q_i \hat{\uplus} Q_j \hat{\uplus} Q_k \hat{\uplus} \ldots \hat{\uplus} Q_m$ represents an outer union. The result will have a `type` column, and a distinct set of columns for each branch of the union. The number of columns in the result of each query affects the overall cost due to the introduction of NULL values. Section 3.6.3 provided results exploring the effects of increasing the number of columns involved in an outer union. The costs do increase with an increasing number of columns, but this increase is relatively insignificant when considering the other errors present in the cost estimation. Therefore, we elect to ignore this parameter, and we estimate the cost of $Q_i \hat{\uplus} Q_j \hat{\uplus} Q_k \hat{\uplus} \ldots \hat{\uplus} Q_m$ to be the same as the (inner) union $Q_i \uplus Q_j \uplus Q_k \uplus \ldots \uplus Q_m$. We compute this pairwise as follows:

$$\text{EST-COST}(Q_i \uplus Q_j) \equiv \text{EST-COST}(Q_i) + \text{EST-COST}(Q_j) - U_0 \quad (4.11)$$

$$\text{EST-ROWS}(Q_i \uplus Q_j) \equiv \text{EST-ROWS}(Q_i) + \text{EST-ROWS}(Q_j) \quad (4.12)$$

As with lateral derived tables, we could consider using the estimates from the DBMS optimizer to reduce this cost estimate in cases where the DBMS is able to do something clever, such as exploiting common sub-expressions in the combined query. This approach would allow Scalpel to favour combining queries when multi-query optimizations make it favourable, but a naïve implementation does not. At present, we do not include this complication as the DBMS products we tested do not appear to exploit such opportunities.

4.4 Summary of Cost Model

The Pattern Optimizer needs to rank alternative execution strategies based on estimates of their cost. The Cost Model uses a combination of explicit calibration, measurements of queries during training, and DBMS-provided support routines to estimate the cost parameters needed by the Pattern Optimizer.

When estimating the cost of a query that was submitted by the client application, Scalpel uses measurements of the run-time during the training period. In principle, we could use server support routines, but we have found that such an approach omits estimates for quantities such communication costs that are important to Scalpel but not important to the DBMS.

We cannot use prior behaviour to estimate the cost attributes of the queries generated by the Query Rewriter. Instead, we estimate the cost of these queries by considering the operations that we used to generate them, namely lateral derived tables and outer unions. An upper bound for the cost of a lateral derived table can be found using the traditional formula for nested loops joins (augmented to account for the reduction in U_0). This estimate will be too high in the case that the DBMS is able to choose a better join strategy. We use a function SRV-COST if it is available to define SRV-SAVINGS, an estimate of the relative savings achieved by combining the queries that were separate in the application into a single join. We use SRV-SAVINGS to reduce our nested-loops cost. This reduction applies to all elements of the cost, including client, server, and communication cost; the reduction should only be applied to the server component, but we do not have access to this separate value, although we do avoid applying the reduction to U_0. We use a configuration parameter K to control the weight we give to the estimates provided by the DBMS.

The Query Rewriter also combines queries using outer union operations. At present, Scalpel ignores the cost of supplying redundant NULL values in outer unions. This cost is non-zero, but our experiments show that it is relatively insignificant compared to both other cost attributes and the quite high errors associated with other estimates. With this change, we estimate the cost of outer unions as we would for (inner) unions. While it is possible to define a SRV-SAVINGS adjustment for unions, we did not do so because we have found that the DBMS products we study do not exploit opportunities in this case.

The Cost Model is used by Scalpel's Pattern Optimizer to rank various strategies based on an estimate of the total latency associated with each strategy. While the estimates contain inaccuracy due to sampling error and approximations, this approach allows the Pattern Optimizer to select a strategy that we can expect to perform well relative to the rejected strategies.

5 Batch Request Patterns

Chapter 3 described how Scalpel can detect and optimize nested patterns within requests submitted by an application to the database server. While nesting patterns offer substantial opportunities for improvement, they occur relatively infrequently in the database applications that we studied. In these applications, we found that there are sequences of queries which allow us to predict the most likely queries that will be submitted in the future. We call these sequences of queries *batches*. By identifying batches, Scalpel can prefetch future queries before they are submitted, reducing the overhead associated with per-request overhead. This reduction in overhead results in a reduced response time for users, and also can lead to lower execution costs.

Figure 1.5 (page 4) shows function BATCH-EXAMPLE, an example of a function that generates a batch of requests. After query Q_3 is submitted, query Q_4 is submitted if a local predicate is TRUE (line 25). If we could recognize this pattern and find that Q_4 is submitted sufficiently often following Q_3, then we could prefetch the results for Q_4 when we recognize query Q_3. When Q_3 is submitted, we would instead send a modified query $Q_{3;4}$ that generates the results required for both queries.

If we prefetch the results for Q_4 and the application subsequently submits it, we would save the latency associated with one database request, U_0. This type of savings is less than we achieved with the nesting rewrites, where we can save several or even hundreds of times the per-request latency. However, these batch patterns occur frequently in the client applications we investigated. Rewriting these batch patterns can also save a significant amount of exposed latency, and can even reduce server costs due to the smaller number of messages that are interpreted and formatted.

It may be the case that the local predicate evaluates to FALSE and Q_5 is submitted instead of Q_4; then, we have wasted the work of evaluating results for Q_4 in the combined query $Q_{3;4}$. We need to predict the probability that Q_4 will follow Q_3 and weigh this against the cost of Q_4 to decide whether it is worthwhile prefetching the results for Q_4 because the expected savings are greater than the expected increase in cost due to wasted prefetching.

In order to make decisions about what queries to prefetch, we need to predict the sequence of queries that can follow a given request, and also have an estimate of both the cost of each request and the likelihood that it will be executed. This information is not enough, though, to prefetch the results of future queries. As with nested request patterns, queries are parameterized; in general, the actual values used for parameters can depend on the results of earlier queries. If we are to prefetch Q_4 when Q_3 is submitted, we need to be able to predict the actual value used for the formal parameters of the query. In this example, the `ship_id` column returned from Q_3 is used as the actual parameter value when opening Q_4. We do not know this quantity when Q_3 is opened, so we cannot supply it as a parameter from the client when we submit the modified $Q_{3;4}$. Instead, we use a join to combine query Q_3 and Q_4 into one request, where the `ship_id` attribute of Q_3 is used in place of the formal parameter of Q_4. In this example, we could use a combined request such as the one shown in Figure 1.6.

Figure 5.1 shows the components of the Scalpel system that are used to detect, optimize, rewrite, and prefetch batch request patterns. Scalpel uses the same system structure to detect batch request patterns as was used for nested request patterns (Figure 3.1). For ease of exposition, we have presented nesting and batch patterns separately as if they are implemented in isolation. In fact, the two approaches are combined in our prototype (as described in Chapter 6).

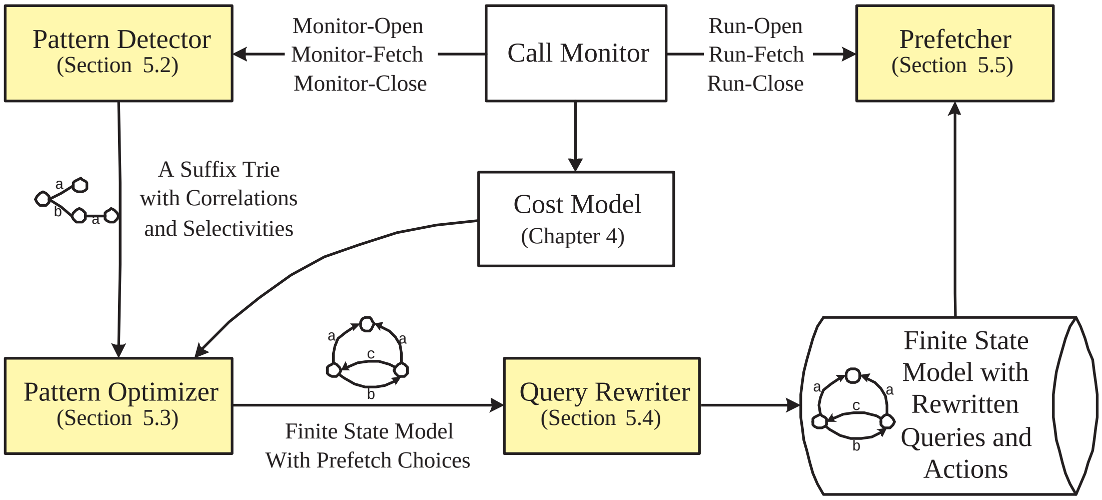


Figure 5.1: Scalpel components used for batch request patterns. Shaded components are described in this chapter.

During the training phase (Figure 2.1), the Pattern Detector component (Section 5.2) identifies batch request patterns. These patterns are encoded in a suffix trie data structure, annotated with probability estimates and predicted correlations between query input parameters and previously observed values. The suffix trie structure corresponds to the context tree structure used for nested request patterns. After the training period has completed, the Pattern Optimizer (Section 5.3) uses the patterns detected by the Pattern Detector component to choose prefetches that will be executed at run-time. The Pattern Optimizer removes the significant redundancy that is present in the suffix trie data structure, producing a compact finite state model that contains a set of states and edges corresponding to predicted requests. For each edge in the model, the Pattern Optimizer associates a list of anticipated future queries that should be prefetched. The Query Rewriter (Section 5.4) combines these lists of queries to build a single query that prefetches the result for the original query and all prefetched queries. These combined queries are stored persistently with the finite state model. At run-time, the Prefetcher (Section 5.5) tracks the current state within the model and executes the prefetches associated with edges in the tree (selected by the Pattern Optimizer) using the combined queries generated by the Query Rewriter.

In addition to the sections shown in Figure 5.1, Section 5.1 gives an example program that generates batches of queries; this example will be used throughout the chapter. Section 5.6 provides experimental results illustrating the strengths and weaknesses of the various strategies. Finally, Section 5.7 summarizes the results for batch request patterns.

In summary, batch patterns are common in the client applications that we studied. We can reduce the latency exposed to users of the application and reduce server costs by recognizing opportunities where we can prefetch the results of queries in anticipation of their execution. In order to prefetch effectively, we need to be able to predict a likely sequence of requests that follow the current request. We need to estimate the probability that each request in the sequence will be executed; combined with an estimate of the cost of the query and the per-request latency, this allows us to choose whether to prefetch a request. Finally, we need to predict the source of parameter values for all prefetched queries. We use these correlation predictions to generate a rewritten query that fetches not only the results for the immediately requested query, but also the results for anticipated future queries.

5.1 Example of Batch Pattern

Figure 5.2 shows a subset of a database application that issues a series of small queries to a relational database server. This particular example is a simplified, artificially constructed application designed to show particular features of our approach. However, its features are a composite of some of those that we observed in a set of database applications that we studied (described in Chapter 8).

The GETCUSTOMER function takes a partially-filled customer structure (`cust_info`) as input, and retrieves additional customer information from the database. It first issues query Q_{a} to retrieve the customer name and account number. If the application does not already have shipping information for the customer, it calls the GETDEFAULTSHIPTO function to obtain the customer's default shipping address. Finally, the application checks the customer's outstanding balance (Q_{c}) and uses that information to determine the available credit.

In addition to the GETCUSTOMER function that retrieves information about customers, Figure 5.2 contains the GETVENDORORDER function that is used to generate a parts order for a ven-

```
function GetCustomer(cust_info)
  fetch row r1 from a:
    SELECT name, accno
    FROM customer c
    WHERE c.id = :cust_info.id
  cust_info.name ← r1.name
  if not cust_info.shipto then
    cust_info.shipto ← GetDefaultShipTo( cust_info )
  fetch row r3 from c:
    SELECT SUM(amount - paid) AS balance
    FROM ar a
    WHERE a.accno = :r1.accno
  cust_info.balance ← r3.balance
end

function GetDefaultShipTo(info)
  fetch row r2 from b:
    SELECT addr
    FROM shipto s
    WHERE s.shipid = :info.id AND s.default='Y'
  return r2.addr
end

function GetVendorOrder(vendor_info)
  fetch row r4 from d:
    SELECT name
    FROM vendor v
    WHERE v.id = :vendor_info.id
  vendor_info.name ← r4.name
  vendor_info.mailto ← GetDefaultShipTo( vendor_info )
  ▷ Find parts supplied by the vendor that need re-stocking.
  open c5 cursor for e:
    SELECT partname, invlevel - onhand AS qty
    FROM part p
    WHERE p.vendor_id = :vendor_info.id AND p.onhand < p.invlevel
    ORDER BY partname
  while r5 ← fetch c5 do
    AddOrder( vendor_info, r5.partname, r5.qty )
  close c5
end
```

Figure 5.2: Application generating a query batch.

#	Query	Input	Output	Possible Correlations
1	Q_{x}	(42)	(501)	$\langle 1\|\mathrm{C}, 42\rangle$
2	Q_{a}	(101)	(‘Alice’, 501)	$\langle 1\|\mathrm{C}, 101\rangle$
3	Q_{b}	(101)	(‘1500 Robie St.’)	$\langle 1\|\mathrm{C}, 101\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$
4	Q_{c}	(501)	($400.00)	$\langle 1\|\mathrm{C}, 501\rangle$, $\langle 1\|\mathrm{O},-2, 2\rangle$, $\langle 1\|\mathrm{O},-3, 2\rangle$
5	Q_{d}	(201)	(‘Mary’)	$\langle 1\|\mathrm{C}, 201\rangle$
6	Q_{b}	(201)	(‘1400 Barrington St.’)	$\langle 1\|\mathrm{C}, 201\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$
7	Q_{e}	(201)	{ (‘Bell’,3), (‘Tire’,6) }	$\langle 1\|\mathrm{C}, 201\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$, $\langle 1\|\mathrm{I},-2, 2\rangle$
8	Q_{a}	(121)	(‘Bob’, 537)	$\langle 1\|\mathrm{C}, 121\rangle$
9	Q_{c}	(537)	($0.00)	$\langle 1\|\mathrm{C}, 537\rangle$, $\langle 1\|\mathrm{O},-1, 1\rangle$
10	Q_{x}	(43)	(31337)	$\langle 1\|\mathrm{C}, 43\rangle$
11	Q_{a}	(107)	(‘Cindy’, 523)	$\langle 1\|\mathrm{C}, 107\rangle$
12	Q_{b}	(107)	(‘1100 Sackville St.’)	$\langle 1\|\mathrm{C}, 107\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$
13	Q_{c}	(523)	($800.00)	$\langle 1\|\mathrm{C}, 523\rangle$, $\langle 1\|\mathrm{O},-2, 2\rangle$
14	Q_{y}	(189)	(‘Elbereth’)	$\langle 1\|\mathrm{C}, 189\rangle$
15	Q_{d}	(255)	(‘Ned’)	$\langle 1\|\mathrm{C}, 255\rangle$
16	Q_{b}	(255)	(‘1200 Weber St.’)	$\langle 1\|\mathrm{C}, 255\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$
17	Q_{e}	(255)	{ (‘Pedal’,7), (‘Seat’,3) }	$\langle 1\|\mathrm{C}, 255\rangle$, $\langle 1\|\mathrm{I},-1, 1\rangle$, $\langle 1\|\mathrm{I},-2, 1\rangle$
18	Q_{z}	(42)	(‘Xyzzy’)	$\langle 1\|\mathrm{C}, 42\rangle$, $\langle 1\|\mathrm{I},-17, 1\rangle$

Figure 5.3: An example trace containing query batches. Each row in the table represents a complete query sequence including Open,Fetch, and Close. The Query column gives the name of a query from Figure 5.2, Input gives the values of query parameters, and Output gives the row returned by the Fetch calls. Query Q_{e} returns multiple rows, shown as a set; other queries are shown with a single tuple. The last column will be explained in Section 5.2.2.3.

dor. Function GetVendorOrder is passed a partially-filled vendor structure (`vendor_info`) as input, which it then fills in with additional information fetched from the database. First, it retrieves the vendor’s name by submitting query Q_{d}; next, it calls GetDefaultShipTo to find the default mailing address for the vendor. Finally GetVendorOrder builds a list of parts supplied by the vendor that need to be ordered because there are fewer on hand than the desired inventory level.

Figure 5.3 shows a sample trace generated by a program that contains the code in Figure 5.2, as well as other code that we have not shown. Each row of the trace table in Figure 5.3 represents a complete query subsequence (Open, Fetch, and Close). This sample trace shows examples of calls to GETCUSTOMER (positions 2-5, 8-9, 11-13) and GETVENDORORDER (positions 5-7, 15-17). In addition, other queries may be submitted from portions of the batch application outside of the code shown in Figure 5.2. Queries Q_{x}, Q_{y}, and Q_{z} are examples of this at trace positions 1,10,14, and 18.

5.2 Pattern Detector

In the sample program of Figure 5.2, query Q_{b} is issued after Q_{a} if the predicate on line 375 is true. If this predicate is true sufficiently often, then it would be more efficient to prefetch the results for Q_{b} when Q_{a} is submitted by the application. The Pattern Detector monitors the client application during the training phase to build a model that is used by the Pattern Optimizer to make prefetching decisions. Figure 5.4 shows the structure of the Pattern Detector used for batch request patterns.

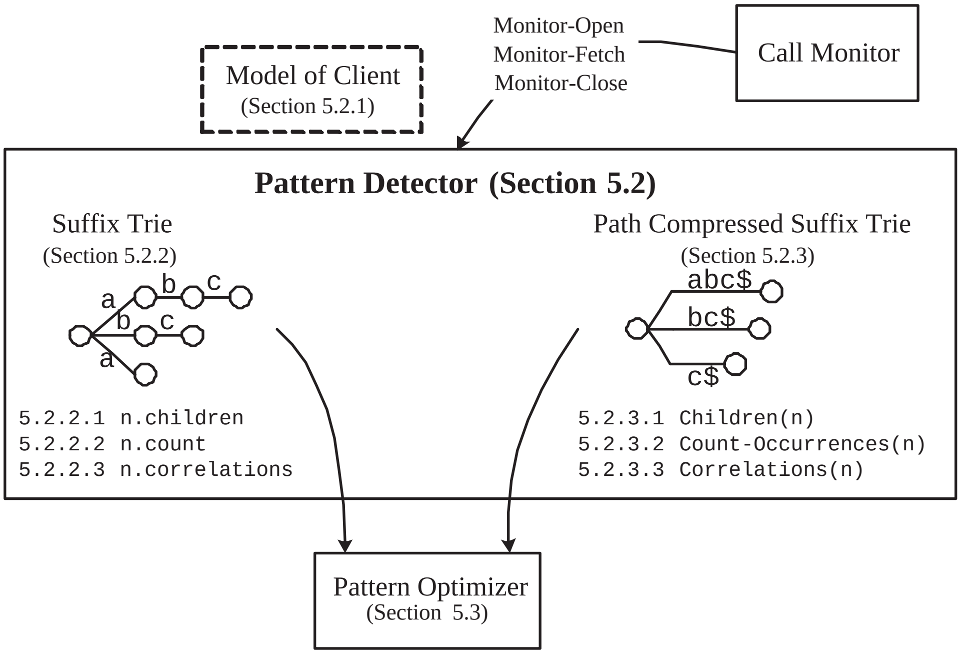


Figure 5.4: Overview of the Pattern Detector.

In order to make effective prefetching decisions, we need to predict the probability of future requests based on the history of requests we have observed. Section 5.2.1 describes how we can construct a model of a request stream that will allow us to estimate the probability of future requests. In particular, Section 5.2.1 describes the order-k model, which uses k previous requests to predict the next request.

As we will see, it is difficult to choose an appropriate k value before training commences. Instead of choosing a particular k value, Scalpel builds a trie-based data structure that contains all order-k models that can be built for a trace. After the training is over, the Pattern Optimizer can use this data structure to select a locally good k value for different portions of the trie. Section 5.2.2 describes this suffix trie data structure. Section 5.2.2.1 describes how Scalpel builds the structure, and Section 5.2.2.2 shows how Scalpel uses the structure to estimate the probability of future requests. In addition to probabilities, the suffix trie structure is used to maintain information about correlations that have always held during the training period. This correlation detection is described in Section 5.2.2.3.

The suffix trie described in Section 5.2.2 requires quadratic space and time. More efficient linear algorithms have been available for some time [80, 132, 180, 191]. In particular, Ukkonen [180] provides a novel approach that builds a space-efficient suffix trie on-line as requests are observed. In Section 5.2.3, we show how this algorithm can be extended for our purposes. Section 5.2.3.1 shows how this compressed trie can be used to provide counts for probability estimation. Section 5.2.3.2 demonstrates how the compressed trie data structure can be used maintain the correlation information needed to make the rewrites needed for prefetching without disturbing the line space/time bound.

Finally, Section 5.2.4 summarizes the Pattern Detector and describes how it passes information to the Pattern Optimizer (Section 5.3).

5.2.1 Models of Request Streams

We can consider the client application and its inputs to be a *stochastic process* with an unknown structure. If we use the sequence trace notation described in Section 2.3, then each trace of requests is a sequence of queries (ignoring for now the details of OPEN, FETCH, CLOSE used for each query). Each query in this trace is associated with a *random variable* X_i. We use Σ as the set of all possible queries, so $X_i \in \Sigma$. The stochastic process is characterized by the joint probability mass functions shown in Equation 5.1.

$$p(x_1 x_2 \ldots x_n) = \Pr\{X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n\} \quad n = 0, 1, 2, \ldots \tag{5.1}$$

Equation 5.1 defines an infinite set of joint probability mass functions, each associated with n, the length of strings described by the function. The function $p(x_1 x_2 \ldots x_n)$ gives the probability that a request trace generated by the client application will begin with the specific sequence of requests $x_1x_2\ldots x_n$. The joint probability mass functions are related by the rules shown in Equation 5.2 and Equation 5.3.

$$
\begin{aligned}
p(\epsilon) &= 1 && (5.2)\\
\forall\, \sigma \in \Sigma^* \quad p(\sigma) &= \sum_{a\in\Sigma} p(\sigma a) && (5.3)
\end{aligned}
$$

For prefetching, we are particularly interested in the conditional probability $p(a|x_1x_2\ldots x_n)$ defined in Equation 5.4.

$$p(a|x_1x_2\ldots x_n) = \frac{p(x_1x_2\ldots x_n a)}{p(x_1x_2\ldots x_n)} \tag{5.4}$$

This conditional probability gives the probability that the query a will be submitted after observing the query sequence $x_1x_2\ldots x_n$. This conditional probability is exactly the quantity that we need to decide if it is worthwhile to prefetch request a after observing the sequence $x_1x_2\ldots x_n$.

The probability distribution of the stochastic process associated with the client application is not available to Scalpel directly; instead, Scalpel builds a *model* of the request source based on observations of sequences of requests the application generates during the training period. This model approximates the conditional probability function based on the training. In order to explain how Scalpel models the client application, we begin with a restricted class of models, the class of order-k models.

5.2.1.1 Order-k Models

A stochastic process can have a special structure where each random variable X_n depends only on the k preceding variables, and is conditionally independent of all preceding random variables. In this case, we say the stochastic process is order-k.

DEFINITION 5.1 (ORDER-k STOCHASTIC PROCESS)
A discrete stochastic process is said to be order-k if Equation 5.5 holds for all $n = 1, 2, \ldots$ and for all $x_i \in \Sigma$:

$$
\begin{aligned}
&\Pr\{X_{n+1} = x_{n+1}|X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_1 = x_1\}\\
&\quad = \Pr\{X_{n+1} = x_{n+1}|X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_{n-k+1} = x_{n-k+1}\}
\end{aligned}
\tag{5.5}
$$

For an order-k stochastic process, the probability that the next query is a depends only on the k previous requests. We can build a graph-based model of an order-k stochastic process as follows.

DEFINITION 5.2 (ORDER-k MODEL)
Let Σ be the set of all possible queries extended with #, an out-of-band value. Let $S = \Sigma^k$ be a set of states, and δ be a transition function defined by $\delta(x_1x_2 \ldots x_k, a) = x_2 \ldots x_k a$. That is, $\delta(x_1x_2 \ldots x_k, a)$ is the state associated with the string formed by the concatenation of the last $k - 1$ queries of the previous state followed by a. Let $s_0 = \#^k$ be the initial state and let $\hat{p}(x_i|s)$ be a conditional probability mass function giving the estimated probability of observing x_i when in state $s \in S$. Then, $M = \langle S, \Sigma, \delta, s_0, \hat{p}\rangle$ is an order-k model of request sequences.

An order-k state $s \in S = \Sigma^k$ is a string of k queries. We say that the model is in state s after observing a trace $t = rs$ that begins with any string r and ends with s. When processing a trace from the beginning, there are $k - 1$ queries observed before we enter a 'real' state. These are handled by padding the trace on the left with k copies of an out-of-band value #.

Figure 5.5 shows the order-k contexts that we observe in the trace of Figure 5.3 (k between 0 and 5). Nodes in these graphs are labelled with strings of length k representing contexts that we have observed in the trace. Edges are annotated with the query that leads to transitions between the contexts, and with a count of how many times the transition was observed. It is important to note that the nodes shown in Figure 5.5 are *sparse* in the sense that not all of the states of the order-k model are shown. Only those states that were actually observed are shown, while the full model contains Σ^k states.

Figure 5.5 suggests how we can define $\hat{p}(x_i|s)$, the estimated probability that query x_i will be submitted when the model is in state s. If we maintain a count COUNT-OCCURRENCES(σ, t) of how many times string σ occurred in trace t, then we can estimate $\hat{p}(x_i|s)$ as shown in Equation 5.6:

$$\hat{p}(x_i|s) = \frac{\text{COUNT-OCCURRENCES}(sx_i)}{\text{COUNT-OCCURRENCES}(s)} \tag{5.6}$$

That is, we estimate the probability that query x_i will be submitted when the model is in state s based on the proportion of times in the training period that we observed x_i submitted when the model was in state s.

The order-0 model has a single context, ϵ. This model does not consider any previous requests, and it estimates the probability that the next request will be x_i based on the relative frequency of x_i in the training period. For example, query `a` was observed 3 times in the trace of length 18, giving $\hat{p}(\mathtt{a}|\epsilon) = \frac{1}{6}$. In contrast, query `'b'` was observed 4 times, giving $\hat{p}(\mathtt{b}|\epsilon) = \frac{2}{9}$. There may be specialized situations where an order-0 model is helpful for prefetching. In general, we need to consider longer contexts in order to make more specific predictions.

In the order-1 model, context `'b'` is observed 4 times, followed 2 times by a `'c'` and 2 times by `'e'`. This gives $\hat{p}(\mathtt{c}|\mathtt{b}) = 0.5$ and $\hat{p}(\mathtt{e}|\mathtt{b}) = 0.5$. In contrast, the order-2 model has two distinct contexts `'ab'` and `'db'`. Context `'ab'` is observed 2 times and followed each time by `'c'`,

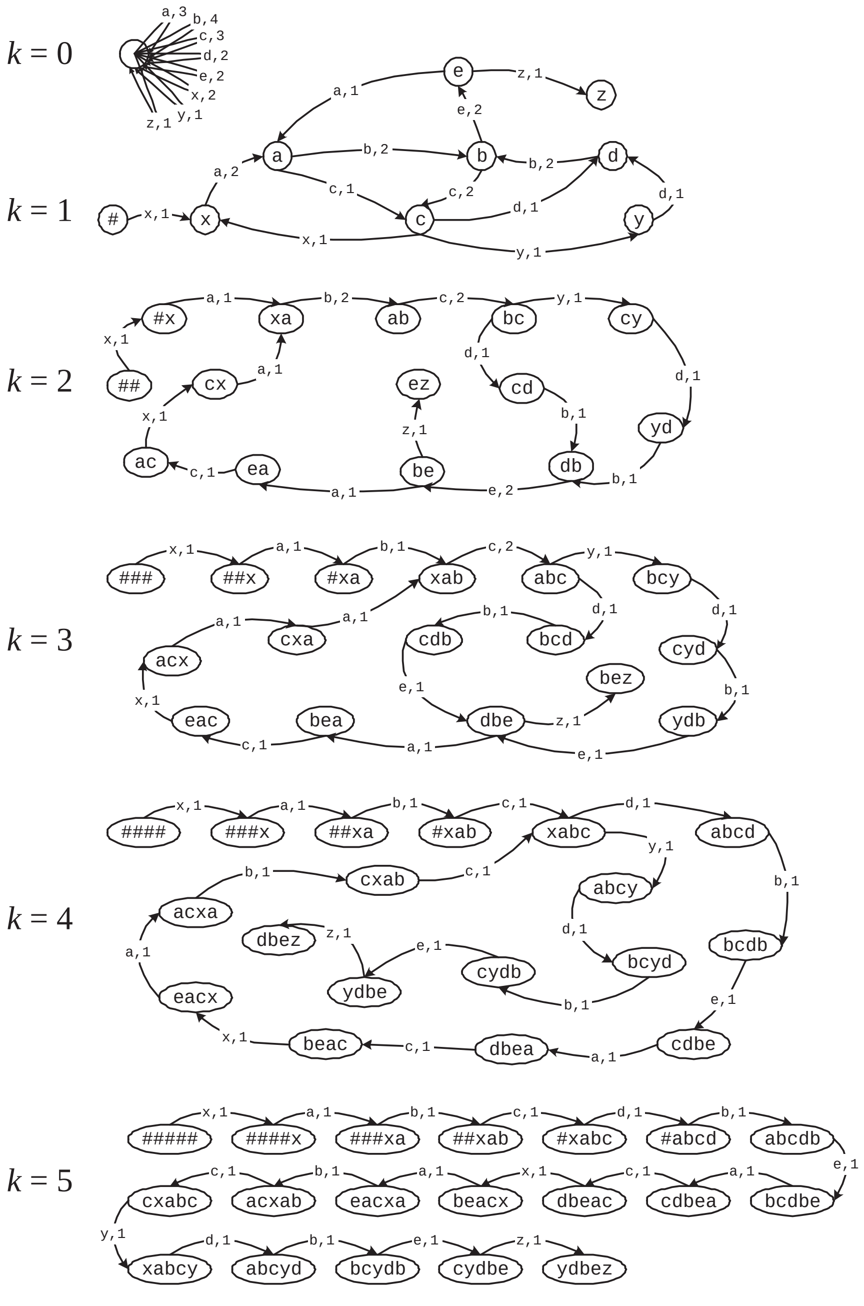


Figure 5.5: Order-k models for trace of Figure 5.3: x a b c d b e a c x a b c y d b e z. Edges are labelled with the query subscript that causes the transition between contexts (nodes) and with a count of how many times that transition was observed in the sample trace.

and `'db'` is also observed 2 times and followed each time by `'e'`. This gives $\hat{p}(\texttt{c}|\texttt{ab}) = 1.0$ and $\hat{p}(\texttt{e}|\texttt{db}) = 1.0$. It appears that the order-2 model is better than the order-1 and order-0, at least for predicting the request that will follow Q_{b}. This raises the question of how we should choose an appropriate context length k.

5.2.1.2 Choosing a Context Length

We might assume that larger contexts provide better predictions. With a sufficiently long training period, this is true (with the 'sufficiently' depending on k, being at least $O(|\Sigma|^k)$). However, the larger k value means that individual contexts are observed less frequently in a trace of finite size. This low frequency is most obvious for those contexts that did not occur at all in the training period: the order-k model can provide no estimate for the probability of future actions in these missing contexts. For example, consider the sequence `'yab'`. This sequence did not occur in the training period, so an order-3 model cannot make a prediction for the next request. An order-2 model, on the other hand, provides a prediction that the next request will be `'c'`; in this case, that prediction turns out to be correct due to the structure of GETCUSTOMER. For this instance, an order-2 model is preferable to a model of order 3 or higher. This problem can also occur when predicting the probability of a query x_i that has never been observed when the model was in state s, even if s has been observed during training.

The difficulty of providing a probability estimate for a situation that has not previously occurred has long been recognized as an issue of philosophical and practical importance. Hume [96] noted that there is no rational basis for using empirical observations of the past to make predictions of future conditions that do not match what we have measured, and Kant [100] expanded on this topic in his *Critique of Pure Reason*. In practice, we may need to assign a specific probability to events that have not previously been observed (even though there is no purely rational basis for making such an assignment). This problem affects fields such as data compression and information theory, where it is usually known as the *zero frequency problem*. Bell, Cleary and Witten [16] provide a description of the problem as it applies to the data compression field and Witten and Bell [195] summarize solutions that have been applied for data compression. In arithmetic compression, a non-zero probability must be selected for every possible symbol; solutions to the zero-frequency problem in this field consist of choosing a specific probability estimate for each of the symbols that may occur next. In contrast to the compression field, the specific probability values are not important to Scalpel. When we have not observed a situation during our training period, Scalpel can merely select to not prefetch in that case, without deciding on any particular probability estimate for any unobserved contexts.

The problem of low frequencies is most obvious when we have not observed a context at all during a trace; however, even when we do observe a context, it is likely that we observe it less frequently in a model with higher order k. For the contexts that we have observed, we are less confident in the inferences provided by the context. For example, consider a trace ending in `'cdb'`. In an order-2 context, we have observed `'db'` two times, and both times it was followed by `'e'`. This gives $\hat{p}(\texttt{e}|\texttt{db}) = 1$. In an order-3 context, we have observed `'cdb'` only one time, again followed by `'e'`. This result also gives $\hat{p}(\texttt{e}|\texttt{cdb}) = 1$. Both the order-2 and order-3 have the same estimate for the probability of the next request being `'e'`, but we might have more confidence in the order-2 prediction as it has been tested one more time. The difference for this short sample trace is slight, but we can imagine much longer cases where an order-k_1 model makes a prediction based on 1000 observations while an order-k_2 model has only one relevant observation to form a prediction. In a very real sense, we have less confidence in the prediction made by the order-k_2 model. We can formalize this doubt using a *confidence interval* for the predicted probability $\hat{p}(x_i|s)$.

Definition 5.3 (Confidence Interval for $p(x_i|s)$)
Let n be the count of times that state s was observed during training, and X be the count of how many times x_i was observed when the trace was in state s. Let α be a pre-specified significance level, and let $\kappa = z_{\alpha/2}$ be the $100(1-\alpha/2)$-th percentile of the standard normal distribution. Let $\hat{p}(x_i|s) = X/n$. Let $\tilde{n} = n + \kappa^2$ and $\tilde{X} = X + \kappa^2/2$. Let $\tilde{p} = \tilde{X}/\tilde{n}$. We estimate $p(x_i|s)$ as $\hat{p}(x_i|s)$ and use the following confidence interval:

$$CI = \textsc{Confidence-Interval}(X, N) = \tilde{p} \pm \kappa\sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}} \tag{5.7}$$

The confidence interval CI is based on a proposal by Agresti and Coull [6]. The interval is not centered about $\hat{p}$ (although it does contains $\hat{p}$). Instead, it is centered about a point $\tilde{p}$ that is closer to 0.5; the movement toward the center reduces with increasing n. For example, with a 95% confidence interval, we have $\kappa = 1.96$. For the order-3 prediction of $\hat{p}(\texttt{e}|\texttt{cdb})$, this gives an interval $CI = [0.17, 1.04]$. The order-2 prediction is slightly better at $CI = [0.29, 1.05]$ and our hypothetical example with a longer trace giving 1000 observations is tightened to $CI = [0.995, 1.001]$. The definition of interval CI may include values that are greater than 1 or less than 0. The Agresti-Coull definition that we use above is conservative in that it gives intervals that are wider than the Wilson intervals [194] they approximate, but we prefer this definition due to its simplicity. Appendix A provides more background on this choice of interval.

When making a decision about prefetching request `'e'`, we must consider the confidence interval, the payoff if we guess correctly, and the penalty for guessing wrong. This topic is explored further in Section 5.3.

If k is too small, Scalpel may miss valuable special cases and parameter correlations. On the other hand, unnecessarily large values of k can lead to overly specific predictions. With longer contexts, each specific context will only be observed infrequently. Therefore, much longer training periods are needed to avoid missing predictions or excessively wide confidence intervals Clearly there is some relationship between the number of states in the 'true' model of the application and the choice of k that will give the best predictions. However, the stochastic process associated with the client application is is not available to Scalpel, and it can therefore not be used to select an appropriate k value.

For these reasons, Scalpel defers choosing context lengths until the end of its training period, at which point it needs to make effective prefetching decisions. During training, Scalpel builds a trie-based data structure to capture all of the relevant information about the training trace of n requests for all possible context lengths $0 \leq k < n$. After training is complete, the Pattern Optimizer uses the trie to select contexts that are sufficiently long to provide predictions of parameter correlations and predicate selectivity, but as short as possible in order to be generally applicable. Scalpel may choose a different context length for each individual prefetching rule. The resulting model is no longer an order-k model, as individual states may correspond to different context lengths. We use a more general finite-state model to represent the results of the optimization.

5.2.1.3 Finite State Models

The class of finite state models is an extension of the class of order-k models to a setting where the states in the model do not need to be identified with any particular string of queries.

DEFINITION 5.4 (FINITE STATE MODEL)
A *finite-state model* (FSM) M of requests is a five-tuple $M = \langle S, \Sigma, \delta, s_0, \hat{p} \rangle$, where S is a finite set of states, Σ is the set of all possible queries, $\delta : S \times \Sigma \mapsto S$ is a transition function, and s_0 is an initial state. The function $\hat{p} : S \times \Sigma \mapsto [0, 1]$ is an estimated conditional probability mass function where $\hat{p}(x_i|s)$ estimates the probability of observing request $x_i \in \Sigma$ when the model is in state $s \in S$.

Scalpel's Pattern Optimizer generates a finite state model, with edges in the model annotated with prefetch actions that the Prefetcher should apply. The Prefetcher also uses a finite state model to track the current state at run-time and execute the selected prefetch activities.

5.2.1.4 Summary of Request Models

We consider a client application to be a stochastic process. At each position i in its request sequence, we have a random variable X_i. We are particularly interested in predicting the probability that the next request will be a particular query x_i, in order to assess whether it is worthwhile prefetching x_i. The stochastic process has an associated probability distribution $\Pr\{X_1 = x_1, X_2 = x_2, \ldots, X_{i-1} = x_{i-1}\}$ that assigns a probability to every sequence of requests. If we knew this distribution, we could estimate the probability that the next request is x_i based on the conditional probability mass function $p(x_i|x_1x_2 \ldots x_{i-1})$.

The probability distribution $\Pr$ is not known to Scalpel, so we build a model during a training period. We use this model to estimate the conditional probability mass function as $\hat{p}(x_i|x_1x_2 \ldots x_{i-1})$. Scalpel bases its model on a class called order-k models. These predict the probability of the next query based only on the previous k queries.

There does not appear to be a good reason to select any particular k value. Choosing a k that is too small risks missing important special cases, while large k values require long training periods in order to make useful predictions. Instead of selecting a single k value, Scalpel maintains information that constructs order-k models for all k values in parallel. At optimization time, an appropriate context length is selected for each prefetch decision. This choice is guided by a consideration of the expected benefits of the prefetch (if it is correct), the costs (if it is mistaken), and a confidence interval for the prediction that is constructed in a way that controls the number of mistakes we expect Scalpel to make.

In order to construct all order-k models in parallel, Scalpel builds a suffix-trie data structure. This suffix trie encodes all of the statistical information that Scalpel needs to build the order-k models.

5.2.2 Suffix Trie Detection

A *suffix trie* for a string T of length m is a *trie* data structure [73] that contains the set of m words that are suffixes of T.

DEFINITION 5.5 (SUFFIX TRIE)
A suffix trie τ for string T of length m is a rooted, directed tree with exactly m leaves. Each edge in the tree is labelled with a character from T, and no two edges out of a node have the same label. Every node n is identified by the string corresponding to the edge labels on the path from the root to n. The string of each leaf L_i is equal to the suffix $T[i..m]$ of string T.

Figure 5.6 provides an algorithm that builds a suffix trie data structure as queries are submitted. If we use the sequence trace notation described in Section 2.3, then each query is considered to be a character in the sequence T of requests contained in the trie.

When the client application submits an OPEN(Q,parmvals) request, the Call Monitor component intercepts the call and passes the corresponding query Q and input parameter value list to MONITOR-OPEN. Similarly, when the application calls FETCH() to return a row from the query result, the Call Monitor invokes MONITOR-FETCH, passing it a copy of the returned row. Scalpel records the input parameters in the OPEN call and output parameters returned by the FETCH call in order to perform correlation detection (as we did when recognizing nested request patterns). This correlation detection is described in Section 5.2.2.3.

Figure 5.7 shows the suffix trie built after the each of the first few queries of Figure 5.3 have been submitted. Nodes in the trie represent the contexts that have been observed within the trace, and are labelled with unique identifiers. Edges are labelled with the subscripts of queries from the trace. Each node represents the context consisting of the queries labelled on the path from the root to that node. Thus, node 1 represents the context `'a'` while node 2 represents the context `'ab'`. The root node, labelled Λ, represents the single context of length $k = 0$. Each node has a suffix link, which is shown with a dashed link in Figure 5.7. Suffix links represent context generalization. For example, the suffix link for node 2, which represents the context `'ab'`, points to node 3, which represents the more general context `'b'`. Figure 5.8 shows the suffix trie built for the entire trace of Figure 5.3. We have left the suffix links out of Figure 5.8 to reduce clutter. Figure 5.8 contains `$` characters; the purpose of these is described next in Section 5.2.2.1.

5.2.2.1 Implicit Suffix Tries

The algorithm in Figure 5.6 builds an *implicit suffix trie*. The tree does not necessarily contain a leaf for every suffix of the trace as required by Definition 5.5. For example, in Figure 5.7 (e) we have observed a string of length 6, but there are only 5 leaves in the trie. The problem is that there is a suffix, `'b'`, of the observed trace that is a prefix of another suffix (`'bcdb'`). If we wish to convert an implicit suffix trie into an explicit suffix trie, we do so by adding an *out-of-band character* to the end of the observed trace. This character (represented as `$` in keeping with convention) does not appear elsewhere in the trace so it avoids this prefix problem.

The implicit suffix trie captures all of the information observed during training. In principle, it would be possible to define all of our algorithms to work with this implicit structure. However, we require explicit suffix trees to implement pattern detection efficiently (discussed further in Section 5.2.3). For consistency, we use explicit suffix tries in the sequel. We add an out-of-band character at the end of a trace to ensure this property.

5.2.2.2 Estimating the Probability of a Future Request

If node n is identified by a string of length k, then we say that n is of order k. The order-k nodes of the suffix trie correspond to the nodes in the order-k models defined in Section 5.2.1.1. For example, node 6 of Figure 5.7 corresponds to context `'b'` while node 9 corresponds to `'bc'`.

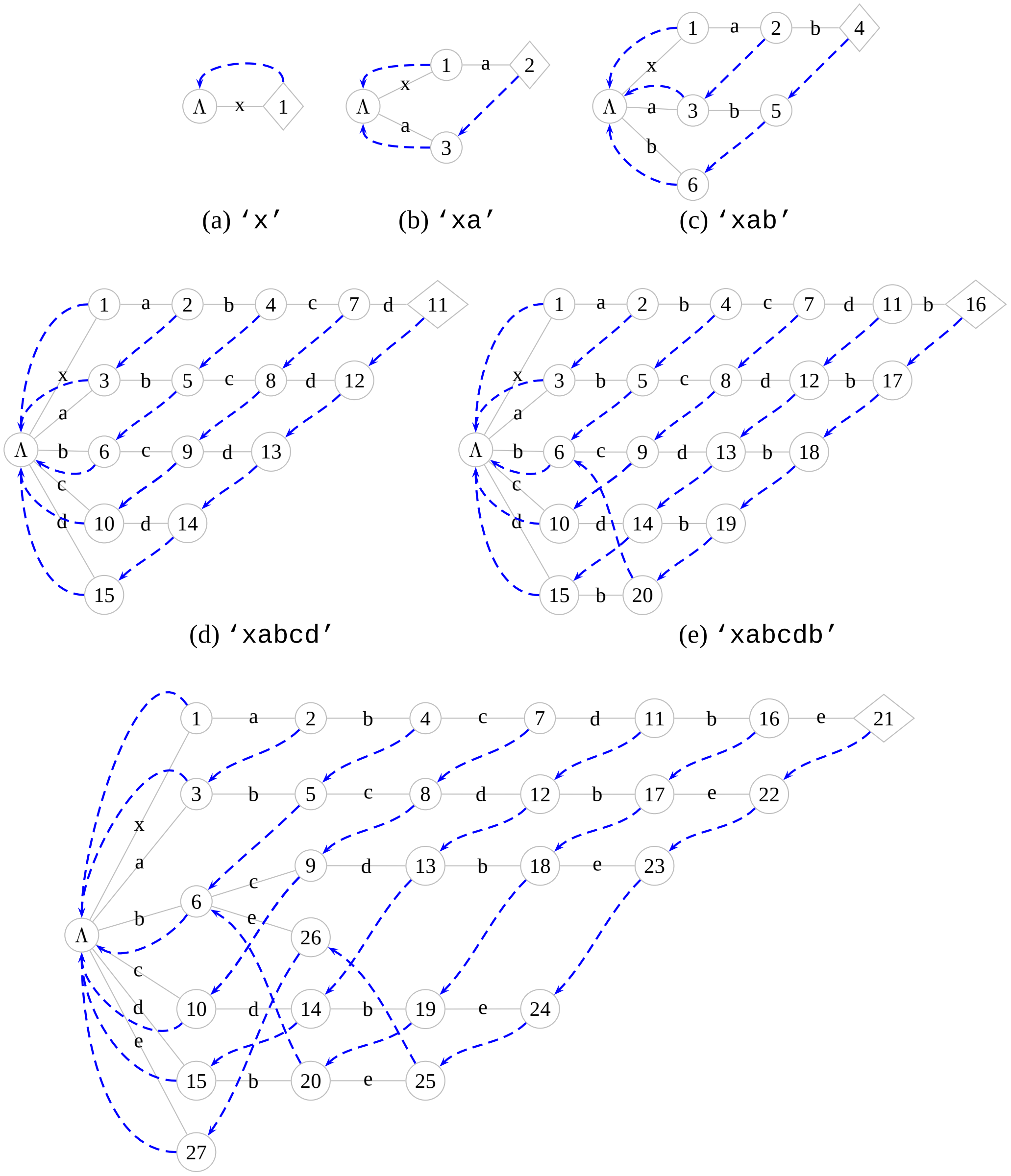
In the order-k models shown in Figure 5.5, edges are annotated with the number of times that the associated request was observed following the context of the node. These counts can be used to estimate the future probability of observing the request when in the context. We maintain similar information in the suffix trie by updating a `count` field on line 436. This count allows Scalpel to estimate the probability of future requests.

```
structure NODE
  count = 0
  parent = NIL
  suffix = NIL
  lastIn[i] = NIL                          ▷ Last input param values
  lastOut[j] = NIL                         ▷ Last fetched row
  correlations = ∅                         ▷ Set of possible correlations
end

▷ Record query Q, with input parameter values InVals, in suffix trie T
procedure MONITOR-OPEN( T, Q, InVals )
  ▷ start with node of longest suffix in T
  curr ← LONGESTSUFFIX( T )
  lastnewchild ← NIL
  while curr ≠ NIL do
    n ← FindChild( curr, Q )               ▷ get query Q child of curr
    if n = NIL then
      n ← NEW-CHILD( T, curr, Q )          ▷ make new Q child of curr in T
      n.parent ← curr
      n.correlations ← FIND-CORRELATIONS( n, InVals )
      newchild ← n
    else
      VERIFY-CORRELATIONS( n, InVals )
      newchild ← NIL
    if lastnewchild ≠ NIL then lastnewchild.suffix ← n
    lastnewchild ← newchild
    n.lastIn ← InVals
    n.count ← n.count + 1
    prev ← n
    curr ← curr.suffix
  ▷ The suffix of the last new node is the root
  if lastnewchild ≠ NIL then lastnewchild.suffix ← GetRoot(T)
end

procedure MONITOR-FETCH( T, fetchVals ) ▷ Record fetched values in trie T
  curr ← LONGESTSUFFIX( T )
  while curr ≠ NIL do
    curr.lastOut ← fetchVals
    curr ← curr.suffix
end
```

Figure 5.6: Code to build a suffix trie.



(a) 'x' (b) 'xa' (c) 'xab'

(d) 'xabcd' (e) 'xabcdb'

(f) 'xabcdbe'

Figure 5.7: Suffix trie after first 7 queries in trace. A diamond represents the longest suffix within the trie.

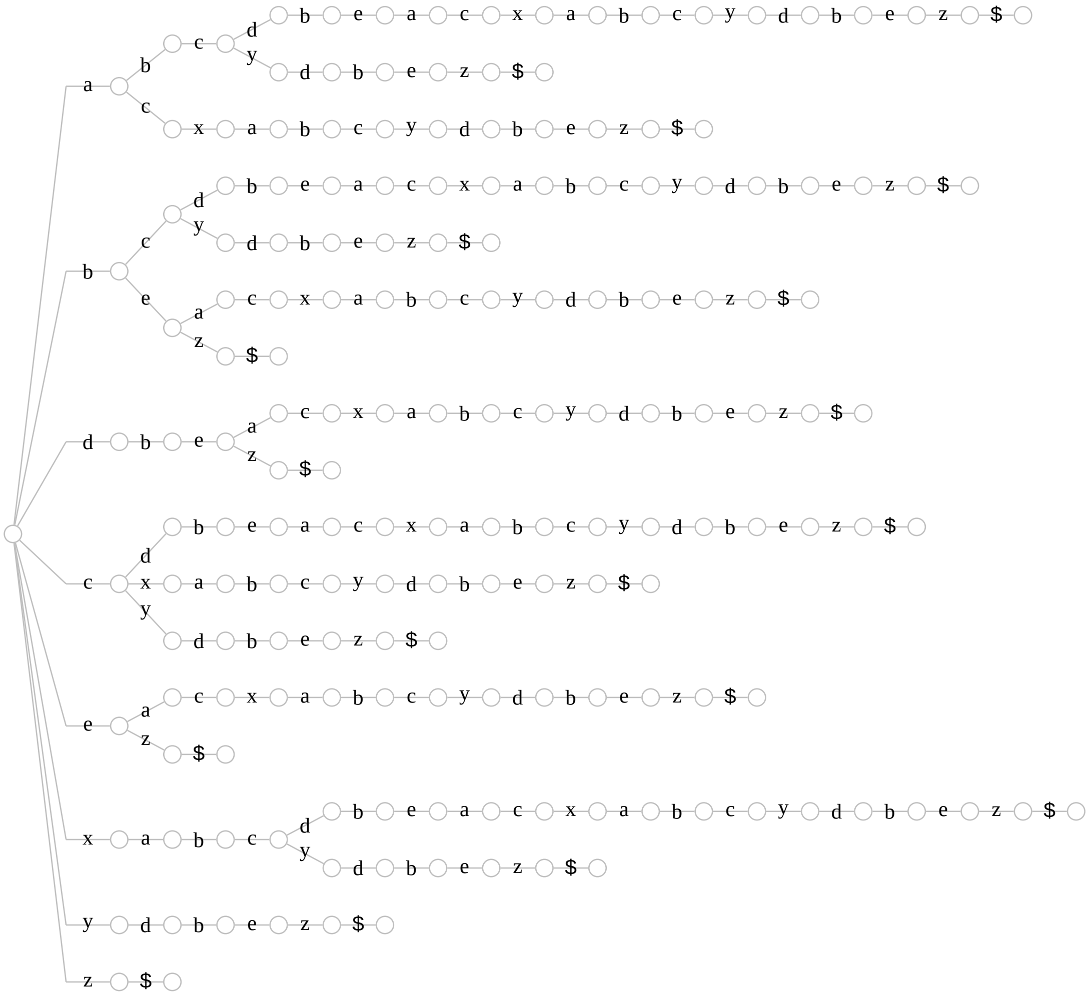


Figure 5.8: Suffix trie for trace of Figure 5.3. Suffix links are omitted.

LEMMA 5.6 (CONTEXT FREQUENCY IS MEASURED BY THE `count` FIELD)
Let w be a string associated with a node in the implicit trie for string T of length m. The `count` field of the node associated with w gives the count of the number of occurrences of word w in T.

PROOF. Let y be the prefix of w and a the last character of w so that $w = ya$. For each instance of w in T, MONITOR-OPEN is invoked with a after processing αy for some string α of length l. The LONGESTSUFFIX function returns the node associated with αy, and the MONITOR-OPEN procedure visits each suffix $\alpha[1..l]y$, $\alpha[2..l]y$, $\ldots\alpha[l..l]y$, eventually reaching node y. Either it finds a pre-existing node $ya = w$ and increments its `count` field, or it creates a new node and initializes `count` to 1. The only way to reach line 436 with node w is after finding w as the a child of y when αy was the result of LONGESTSUFFIX for some α. Therefore, the `count` of w is updated exactly once for every occurrence of w in T, which is the desired property. $\square$

Lemma 5.6 tells us that the `count` field of node w counts how many times w is observed in the trace T observed during the training period. We can estimate the probability of observing a request a after a sequence w by considering the ratio of the `count` of wa and w.

5.2.2.3 Tracking Parameter Correlations

The suffix trie structure that we have described so far can be used to predict the likelihood of future requests. In addition to this probabilistic information, we must be able to predict the actual parameter values that would be used if the request were submitted. In addition to structural information, Scalpel maintains information about parameter correlations.

We use an approach similar to the one we used for nested request patterns (Section 3.2.2). We record the values of input and output parameters in the suffix trie data structure, and look for correlations that have always held throughout the trace. In Section 3.2.2, we considered three possible predictors of future actual parameter values. We consider these same three predictors for batch query patterns; however, we consider previous queries that have already been closed instead of the currently open outer queries.

Constants ('C'*)* If a query parameter is always supplied with the same value in every instance of a query within our training period, we may conclude that the parameter is a "variable that won't".[1]

Input Parameters ('I'*)* A query parameter p_i may always have the same value as a parameter p_j of some previously submitted query. For example, in Figure 5.2, the `vendor_info.id` parameter of query Q_e on line 404 is always equal to the `vendor_info.id` parameter of query Q_d on line 396. At run-time, after observing Q_d, we can predict the value that will be used for the parameter of Q_e.

Output Parameters ('O'*)* An input parameter may instead be correlated to a value returned by a previous query. For example, the `r1.accno` parameter of Q_c (line 380) is always equal to the second column of the preceding Q_a (line 373). At run-time, we can use the results of the previous query to predict the value used in a future request.

Recall that for nested request patterns, we maintained a `scope` field for each node in the context tree. This scope listed a set of correlation source objects that were possible predictors of

[1] From Osborn's Law: "Variables won't, constants aren't", Don Osborn

```
 structure CORRELATION
   inparam = ?             ▷ The input parameter predicted by this object
   type = ?                ▷ The type of correlation: C-constant, I-input, or O-output
   value = NIL             ▷ For type C, the constant value
   prevcnt = NIL           ▷ For type I or O, the distance to the source query
   param = NIL             ▷ For type I or O, the parameter number
 end

 ▷ Find possible correlations for node n and input parameter values InVals
 function FIND-CORRELATIONS( n, InVals )
   for i ← 1 to InVals.length do
     prevcnt ← 0
     curr ← n.parent
     corrs ← new CORRELATION(i, C, InVals[i])
     while curr ≠ NIL do
       prevcnt ← prevcnt - 1
       for j ← 1 to curr.lastIn.length do
         if InVals[i] = curr.lastIn[j] then
           corrs ← corrs ∪ new CORRELATION(i, I, prevcnt, j)
       for j ← 1 to curr.lastOut.length do
         if InVals[i] = curr.lastOut[j] then
           corrs ← corrs ∪ new CORRELATION(i, O, prevcnt, j)
       curr ← curr.parent
   return corrs
 end
 ▷ Verify correlations for node n and input parameter values InVals
 procedure VERIFY-CORRELATIONS( n, InVals )
   for c ∈ n.correlations do
     if InVals[ c.inparam ] ≠ CURR-VALUE( c ) then
       n.correlations ← n.outCorr[i] \ c
 end
 ▷ Find the current value of a correlation source c relative to node n
 function CURR-VALUE( n, c )
   if c.type = C then return c.value
   else
     for prevcnt ← c.prevcnt to 0 do
       n ← n.parent
     if c.type = I then return n.lastIn[ c.param ]
     else return n.lastOut[ c.param ]
end
```

Figure 5.9: Code to monitor parameter correlations.

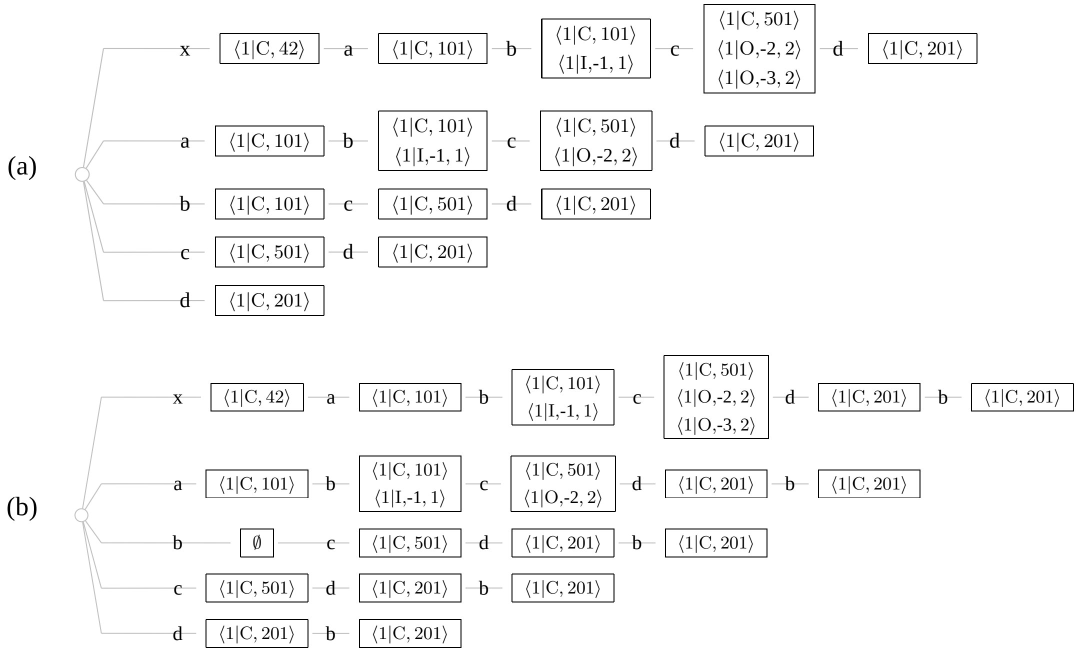

(a)
x ⟨1|C, 42⟩ a ⟨1|C, 101⟩ b ⟨1|C, 101⟩ ⟨1|I,-1, 1⟩ c ⟨1|C, 501⟩ ⟨1|O,-2, 2⟩ ⟨1|O,-3, 2⟩ d ⟨1|C, 201⟩
a ⟨1|C, 101⟩ b ⟨1|C, 101⟩ ⟨1|I,-1, 1⟩ c ⟨1|C, 501⟩ ⟨1|O,-2, 2⟩ d ⟨1|C, 201⟩
b ⟨1|C, 101⟩ c ⟨1|C, 501⟩ d ⟨1|C, 201⟩
c ⟨1|C, 501⟩ d ⟨1|C, 201⟩
d ⟨1|C, 201⟩
(b)
x ⟨1|C, 42⟩ a ⟨1|C, 101⟩ b ⟨1|C, 101⟩ ⟨1|I,-1, 1⟩ c ⟨1|C, 501⟩ ⟨1|O,-2, 2⟩ ⟨1|O,-3, 2⟩ d ⟨1|C, 201⟩ b ⟨1|C, 201⟩
a ⟨1|C, 101⟩ b ⟨1|C, 101⟩ ⟨1|I,-1, 1⟩ c ⟨1|C, 501⟩ ⟨1|O,-2, 2⟩ d ⟨1|C, 201⟩ b ⟨1|C, 201⟩
b ∅ c ⟨1|C, 501⟩ d ⟨1|C, 201⟩ b ⟨1|C, 201⟩
c ⟨1|C, 501⟩ d ⟨1|C, 201⟩ b ⟨1|C, 201⟩
d ⟨1|C, 201⟩ b ⟨1|C, 201⟩

Figure 5.10: Suffix trie with possible correlations after (a) `'xabcd'` and (b) `'xabcdb'`.

input parameter values. For batch request patterns, the scope for a node in the suffix trie contains all of the input and output parameters of queries that are ancestors of the node in the trie. In this way, we allow a node in the trie to represent correlations to any query on a path from the node to the root. For batch request patterns, we don't explicitly maintain the `scope` field (in order to allow a linear-space implementation, described in Section 5.2.3).

The input parameter values for the most recent instance of a node's query are recorded in the `lastIn` field, and the values returned by the most recent FETCH are recorded in the `lastOut` field. The `correlations` field of a node records possible sources of values that could be used to predict the value of each input parameter. This field is maintained by the FIND-CORRELATIONS and VERIFY-CORRELATIONS procedures

Scalpel uses an object of type CORRELATION (line 449) to represent an observed correlation between an input parameter (identified by index in `inparam`) and a source of values that may predict actual parameters at run-time. This structure is similar to the structure of the same name used for nested request patterns (Figure 3.7). In Chapter 6, we describe how the two types of correlations are maintained in parallel; for now, we present results as if CORRELATION objects are used only for batch request patterns. For nested request patterns, we recorded the source of an input or output correlation in the `context` field using a reference to the appropriate context that was the source of values. For batch request patterns, we instead use an integer value in the `prevcnt` field. This alternate representation is needed to allow for an efficient implementation of the suffix trie correlation detection (described in Section 5.2.3.2).

For objects of type 'I' and 'O', the `prevcnt` field gives the distance to the preceding query that contains the value of interest. For example, the immediately preceding query has `prevcnt` of -1, while the query preceding that is recorded as -2. The `param` field of the CORRELATION object records the parameter number that is predicted to be the source of future actual values. For CORRELATION objects of type 'C', the `value` member records the associated constant value; the `prevcnt` and `param` fields are unused.

The last column in Figure 5.3 shows the CORRELATION objects that Scalpel finds for each request in the trace. For example, in position 3 request $Q_{\rm b}$ has input parameters (101). The value 101 matches the first input parameter of the immediately preceding query, $Q_{\rm a}$. Scalpel therefore finds that the first parameter of $Q_{\rm b}$ could be correlated to the constant value 101, recorded as $\langle 1|\mathrm{C}, 101\rangle$. In this representation, the first number (1) is the `inparam` index of the input parameter being predicted, the 'C' indicates a correlation to a constant, and 101 is the constant value. Alternatively, Scalpel also finds the first parameter of $Q_{\rm b}$ could also be correlated to to input parameter 1 of the immediately preceding query, and this is recorded as $\langle 1|\mathrm{I}$,-1$, 1\rangle$. The first number (1) again indicates the `inparam` index, the 'I' indicates this is an input parameter correlation, the -1 indicates the source is the immediately preceding query, and the last value indicates it is the first input parameter of the preceding query we are interested in.

The correlations between the input parameters of a request and previously observed values depend on the context in which the request is submitted. In the context tree structure used to detect nested request patterns (Section 3.2.2), each node in the tree represented a context. When a new node was created, we initialized the `correlations` field of the node with a set of correlations that held at that time. Every subsequent time the context was matched during the training period, we re-checked each of the correlations stored in the `correlations` set, removing any that no longer held. For batch request patterns, each context is represented by a node in the suffix trie structure. The `correlations` field of node n is initialized by FIND-CORRELATIONS with a set of CORRELATION objects, and this set is re-checked by VERIFY-CORRELATIONS every time the context associated with the node is subsequently observed. In this way, the `correlations` field of a node is maintained with the set of correlations that have always held when observing the query leading into n.

Figure 5.10 shows the set of CORRELATION objects maintained for each node after processing (a) `'xabcd'` and (b) `'xabcdb'`. In Figure 5.10 (a), each request has only been seen one time. As each node is added, MONITOR-OPEN calls FIND-CORRELATIONS (line 428). The `correlations` set for each newly created node is based on the parameter values for the current request and preceding request values. Because of this, the `correlations` set is the same for each of the nodes created in response to a single request, except that shorter contexts eliminate some of the correlations that are longer than the context. For example, contexts `'xab'` and `'ab'` both have their set initialized to $\{\langle 1|\mathrm{C},101\rangle, \langle 1|\mathrm{I},\text{-}1,1\rangle\}$, while the set for context `'b'` is initialized to $\{\langle 1|\mathrm{C},101\rangle\}$ because there is no preceding query in that context.

The correlation set $\{\langle 1|\mathrm{C},101\rangle\}$ stored for context `'b'` represents the fact that the first parameter of Q_{b} has *always* been equal to the value 101 every time it has been observed so far in the trace. This gives us a prediction that the parameter will always have this value in the future. In this case, the prediction turns out to be false, and Scalpel detects this after observing the last request in `'xabcdb'`. The MONITOR-OPEN function finds that node `'b'` already exists in the trie, and calls VERIFY-CORRELATIONS (line 431). The result of CURR-VALUE($\langle 1|\mathrm{C},101\rangle$) does not match the current parameter value of 201, so the correlation source is removed as a possible predictor.

5.2.2.4 Summary of Suffix Trie Detection

At the end of the training period, the suffix trie structure contains a node for every context (of any length k) that was observed during the trace. The contexts contain a `count` field that is used to estimate the likelihood of future requests, and they contain a `correlations` field that records correlations that have always held between input parameters and a known value every time that the context was observed. Scalpel uses this information to select appropriate rewrite rules that will be used at run-time

LEMMA 5.7 (SUFFIX TRIE BUILDS $O(n^2)$ NODES)
A suffix trie τ for string T of length n contains at most $1 + n(n+1)/2$ nodes.

PROOF. When processing character i, we have the longest suffix as $T[1..i]$ of length i. Every iteration k (line 423) moves `curr` to a suffix node $T[k..i]$. There can be at most i such suffixes before we reach the root, so at most i nodes are added when processing character i of string T. Summing (including the root node), we get:

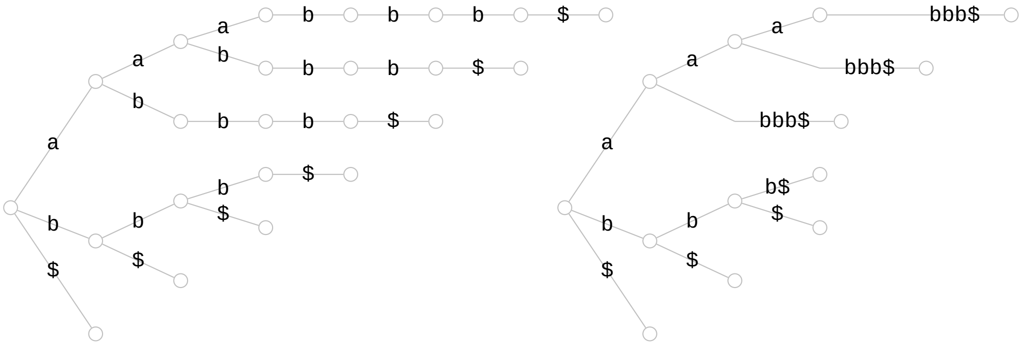
$$|T| \leq 1 + \sum_{i \leftarrow 1 \ldots n} i = 1 + \frac{n(n+1)}{2} \tag{5.8}$$

Further, this worst case bound is achievable. Consider the string $a_1 a_2 \ldots a_m$ where each character is novel. The FINDCHILD call (line 424) will never find an existing child, so i nodes are added on each step. □

We can implement GETCHILD in amortized $O(1)$ time by either using an array of size $|\Sigma|$ or a dictionary such as a hash table (using the RAM model). In addition to the $O(n^2)$ time and space needed to build the nodes of the suffix trie, the correlation detection requires additional space and time. In the worst case, every input parameter is equal to every previous input or output parameter. This work increases the overall time bound to $O(n^3m^2)$, where m is the number of parameters per query.

5.2.3 A Path Compressed Suffix Trie

For traces of significant size, the asymptotic complexity of MONITOR-OPEN is likely to be impractical. The asymptotic $O(n^2)$ bound can be achieved even with small alphabets $|\Sigma| \ll n$. For example, consider the string a^jb^j, which gives the implicit suffix trie shown in Figure 5.11(a).



(a) Suffix trie (b) Path-compressed suffix trie

Figure 5.11: Atomic suffix trie and path-compressed suffix trie for $\mathtt{a}^j\mathtt{b}^j$, $j = 3$

Figure 5.11(a) provides a suggestion for reducing the overall size of the trie. In Figure 5.11(a), a number of nodes have only one child, and this was also true in Figure 5.8. We can combine these branchless nodes, giving a *path compressed suffix trie* as shown in Figure 5.11(b). Path compression is a general approach for reducing the size of a trie. It was first introduced by Morrison [136] as the *Patricia tree* data structure, and this term is often used in the literature to describe this type of structure. In the case of suffix tries, the term *suffix tree* is used to indicate a suffix trie which is encoded with path compression. It would perhaps be better to following Nilsson and Tikkanen [141] and use the term path compressed suffix trie. This term is verbally distinguishable from suffix trie (according to an editor's note, Fredkin apparently used "Trie" as a derivation of reTRIEval; this would lead to 'tree' and 'trie' being homonyms). Further, 'path compressed suffix trie' emphasizes both the underlying trie-based nature of the structure and its compact encoding. However, we accept current practice and refer to a suffix tree when we mean a path compressed suffix trie, and we pronounce trie as 'try' to avoid verbal ambiguity.

DEFINITION 5.8 (SUFFIX TREE)
A suffix tree $\mathcal{T}$ for string T of length m is a rooted, directed tree with exactly m leaves L_i labelled with $i = 1 \ldots m$. Every internal node (excluding the root) has at least two children. Each edge in the tree is labelled with a nonempty substring of T, and no two edges out of a node have the same first character. Every node n is identified by the string corresponding to the edge labels on the path from the root to n. Each leaf L_i is identified by the suffix $T[i..m]$ of string T.

As with the suffix trie, we run into difficulty if a suffix of T is a prefix of another suffix of T. We avoid this difficulty using the out-of-band character `$`, and define an *implicit suffix tree* built from a prefix $T[1..i]$ to be a suffix tree that may not have the desired number of leaves. An (implicit) suffix tree $\mathcal{T}_i$ for a prefix $T[1..i]$ can be created from the (implicit) suffix tree τ_i by combining all branch-free nodes into a single edge.

Path compression for a trie with m leaves gives a structure with at most $2m + 1$ nodes, which is an asymptotic improvement over the original definition. We could build a suffix tree by first building the associated suffix trie structure then merging paths of branch-free nodes into a single edge. This approach would not, however, improve our asymptotic bound. Instead, we implement an $O(n)$ algorithm introduced by Ukkonen [180] (using a slightly different presentation that more closely matches our existing MONITOR-OPEN procedure). Figure 5.12 shows the implementation of MONITOR-OPEN-COMP, which builds a suffix tree for a trace while monitoring correlations that have always held in, all in $O(n)$ time and space. Figure 5.13 shows a trace of the steps taken by MONITOR-OPEN-COMP when processing `'aaabbb'`.

In order to meet the $O(n)$ requirements, we need to be careful about our representation and implementation. First, although the definition of a suffix tree suggests that edges are labelled by strings, we cannot store distinct copies of these strings in $O(n)$ space: there are $n(n+1)/2$ characters in the n suffixes of an n-character string. Instead, we use an object of type STRPTR that represents a substring of T by storing the index `first` of its first character and `last` of the last character (exclusive).

We use an object of type NODEPTR to refer to *virtual nodes* within the suffix tree. These virtual nodes are positions where the suffix trie has a branch-free node. The NODEPTR object consists of field `s`, an *explicit node* (one that is actually present in the suffix tree) and a string `path` represented by a STRPTR object. This string specifies a path to follow from node `s` to the implicit node, which may either be an explicit node or a position 'within an edge'. For example, in Figure 5.13 step 2:1, we have a NODEPTR of $(0,$ `'a'`$)$ that refers to a virtual node. There are multiple NODEPTR values that refer to the same node: Figure 5.13 step 6:1, NODEPTR values

```
structure StrPtr                       ▷ Represent a substring of T.trace
  first = 0                            ▷ The index of first character
  last = 0                             ▷ The index of last character (not inclusive)
end
structure NodePtr
  s = Nil                              ▷ The Node starting the path
  path = new StrPtr(0,0)               ▷ The string on the path leading out of s
end
structure Tree
  trace = ϵ                              ▷ The string of queries so far
  tracker = new CorrTracker              ▷ $S$-sized sliding dictionary for correlation detection
  correlations = [ ]                     ▷ A list of sets of Correlation objects
  root = new Node                        ▷ The root of the tree
end
▷ Record query Q, with input parameter values InVals, in suffix tree T
procedure Monitor-Open-Comp( T, Q, InVals )
  ▷ Start with node of longest suffix in T that has been observed at least twice
  curr ← LongestSuffix2( T )
  lastnewchild ← Nil; end_point ← False
  ▷ Add request Q to the trace
  T.trace ← T.trace + Q
  ▷ Append the set of correlations that currently hold to list correlations
  corrs ← Find-Correlations-Comp( T, InVals )
  T.correlations ← T.correlations + corrs
  while curr ≠ Nil and not end_point do
    n ← FindChild( T, curr, Q )                ▷ get query Q child of curr
    if n = Nil then
      n ← New-Child( T, curr, Q )              ▷ make new Q child of curr in T
      newchild ← n.s
    else
      end_point ← Verify-Correlations-Comp( n, InVals )
      newchild ← Nil
    if lastnewchild ≠ Nil then lastnewnchild.s.suffix ← n.s
    lastnewchild ← newchild
    curr ← GetSuffix( curr )
  ▷ The suffix of the last new node is the root
  if lastnewchild ≠ Nil and lastnewchild.s ≠ T.root then
    lastnewchild.suffix ← T.root
end
```

Figure 5.12: Code to build suffix tree

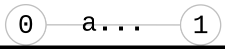
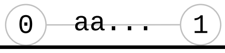
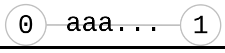
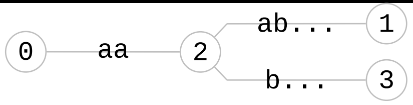
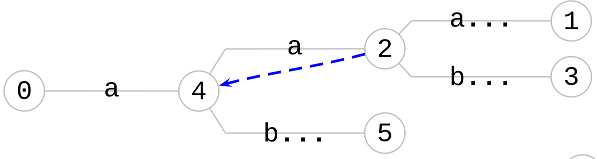
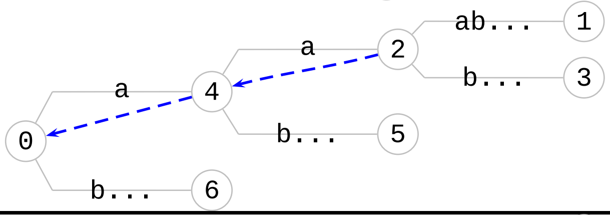
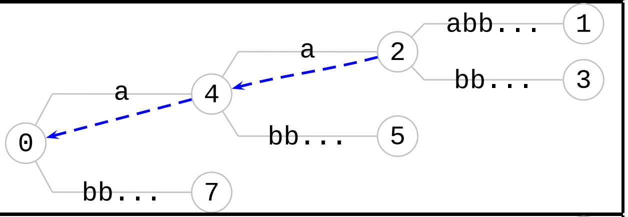
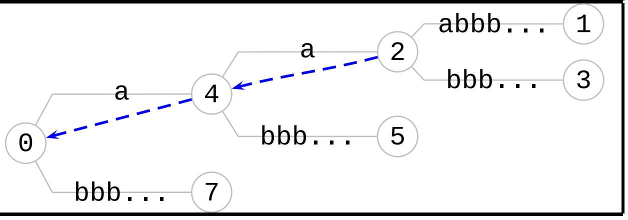
Step	`curr`	`n` (517)	`n` (520)	Tree
1:1	$(0, \epsilon)$	NIL	$(0, \mathtt{a})$	
2:1	$(0, \epsilon)$	$(0, \mathtt{a})$	NIL	
3:1	$(0, \mathtt{a})$	$(0, \mathtt{aa})$	NIL	
4:1	$(0, \mathtt{aa})$	NIL	$(2, \mathtt{b})$	
4:2	$(0, \mathtt{a})$	NIL	$(4, \mathtt{b})$	 0 a 4 a 2 a... 1 b... 3 b... 5
4:3	$(0, \epsilon)$	NIL	$(0, \mathtt{b})$	 0 a 4 a 2 ab... 1 b... 3 b... 5 b... 6
5:1	$(0, \epsilon)$	$(0, \mathtt{b})$	NIL	 0 a 4 a 2 abb... 1 bb... 3 bb... 5 bb... 7
6:1	$(0, \mathtt{b})$	$(0, \mathtt{bb})$	NIL	 0 a 4 a 2 abbb... 1 bbb... 3 bbb... 5 bbb... 7

Figure 5.13: Steps of building a suffix tree for `'aaabbb'`. The first column shows the current step i:j where i is the number of the call to MONITOR-OPEN-COMP and j is the number of the iteration within one call. The next column shows the value of the `curr` variable on line 514. The next two columns show the value of the `n` variable after creating a new child (line 517) or finding an existing one (line 520). The last column shows the structure of the tree at line 522. Nodes are labelled in order of creation, with `0` used for the root node.

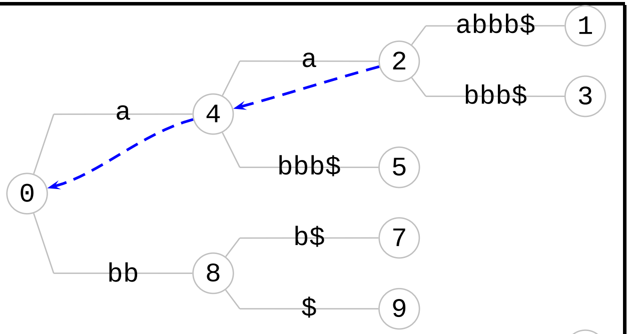
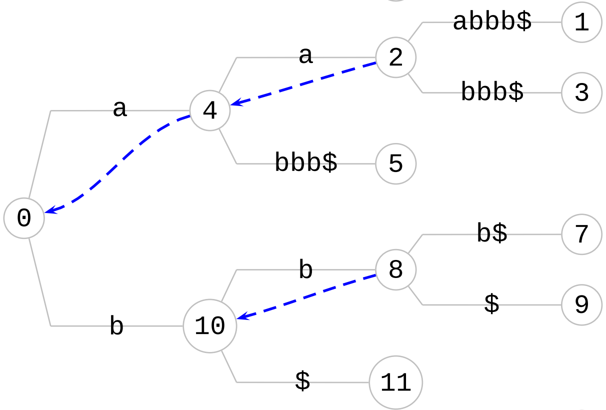
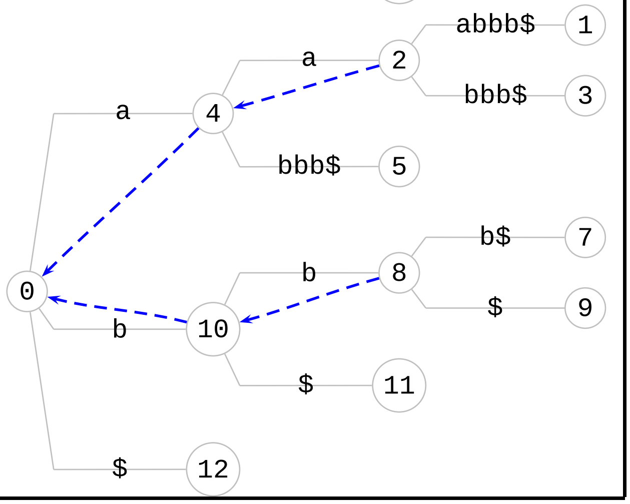
Step	`curr`	n (517)	n (520)	Tree
7:1	$(0, \texttt{bb})$	Nil	$(8, \$)$	 0 a 4 a 2 abbb$ 1 bbb$ 3 bbb$ 5 bb 8 b$ 7 $ 9
7:2	$(0, \texttt{b})$	Nil	$(10, \$)$	 0 a 4 a 2 abbb$ 1 bbb$ 3 bbb$ 5 b 10 b 8 b$ 7 $ 9 $ 11
7:3	$(0, \epsilon)$	Nil	$(10, \$)$	 0 a 4 a 2 abbb$ 1 bbb$ 3 bbb$ 5 b 10 b 8 b$ 7 $ 9 $ 11 $ 12

Figure 5.14: Steps to convert the implicit suffix tree into an explicit suffix tree. Columns have the same meaning as in Figure 5.13. In the explicit suffix tree, there is a distinct leaf node for each suffix of the trace, and we can count the occurrences of any substring by the number of leaf nodes in the sub-tree below the associated node.

of $(0, \texttt{'aa'})$, $(4, \texttt{'a'})$, and $(2, \epsilon)$ all refer to the same explicit node. We say that a NODEPTR $n = (s, x)$ is in *canonical form* if there are no explicit nodes on the path from s following x.

The insight Ukkonen used in his presentation is that we can exploit the NODEPTR representation of edges to have all of the leaf nodes in the implicit suffix tree $\mathcal{T}_i$ automatically extended as we add new characters. When creating a new leaf node, the algorithm labels the node with a STRPTR object with a `last` value of ∞. The interpretation of this is that the string extends to the right including all new characters processed so far. When we add a new character to `T.trace`, we extend all of the edges to these leaf nodes with the new character.

This trick takes care of adding virtual nodes at the end of every branch-free edge to a leaf node, and this is fundamental to the asymptotic improvement in running time. The original suffix trie algorithm started updating at LONGESTSUFFIX and added new nodes by following suffix links. For the first part of this update step, existing leaf nodes were extended to form new leaf nodes, until eventually the suffix link traversal reached an internal node. With the ∞ trick used by Ukkonen, all of the leaf nodes that are to be updated are implicitly updated as soon as we add a new character. Instead of starting the loop at LONGESTSUFFIX, we instead will start at the first internal node that was encountered in the original traversal. This internal node is the first node that is not implicitly updated by the ∞ trick, and it is the first place that a new explicit node might need to be created.

We use LONGESTSUFFIX2 to identify this internal node that represents the new update point. In the suffix tree structure, the internal node could be explicit or virtual. As we show in the following lemma, if we define LONGESTSUFFIX2$(\mathcal{T}_i)$ to be the node in tree $\mathcal{T}_i$ that is the longest context that has been observed at least two times in the trace $T[1..i]$, then we identify the appropriate update position that is the first location in the suffix-link traversal where a new explicit node might be created when generating suffix tree $\mathcal{T}_{i+1}$.

LEMMA 5.9 (LONGESTSUFFIX2 IDENTIFIES THE UPDATE POINT)
If x is an explicit node in implicit suffix tree $\mathcal{T}_{i+1}$, then x has been observed at least twice in $\mathcal{T}_i$.

PROOF. Since x is explicit in $\mathcal{T}_{i+1}$ for $T[1..i+1]$, it is branching. Therefore, there must be $y = xa$ and $z = xb$ for $a \neq b$ in $T[1..i+1]$. Hence, x must occur at least twice in $T[1..i]$. □

By Lemma 5.9, the LONGESTSUFFIX2 identifies the most specific position where we might need to create an explicit node. The `FindChild` function checks if there is already a `Q`-labelled edge from `curr`. If `curr` is an explicit node, this is checked by looking in a dictionary of child nodes. Alternatively, if `curr` is a virtual node, then `curr` has a `Q` child `n` if `Q` matches the next request on the branch-free edge from `curr`. If `Q` does match, then the child `n` may either be another virtual node, or an explicit node.

If `FindChild` returns NIL, then a new child is created by a call to NEW-CHILD (line 516). We create a new leaf node child of `curr` starting with `Q`. In the case that `curr` is already an explicit node, NEW-CHILD adds a new leaf node as a direct child of `curr`, and returns $(\texttt{curr}, \texttt{Q})$ as the new (virtual) child node. In step 1:1 of Figure 5.13, a new leaf node is created as a child of the root (node 0), and the `newchild` variable is set to $(0, \texttt{a})$. Alternatively, if the NODEPTR `curr` is a virtual node, then we must first split the edge that `curr` is on, creating a new internal node. This is performed in each of the iterations 4:1, 4:2, 4:3 when request `'b'` is observed. The NEW-CHILD returns a NODEPTR object in canonical form. The implementation of NEW-CHILD is relatively straightforward, and is omitted here.

When a new child node is created, the `suffix` field is NIL. We never set this field for leaf nodes (nor do we attempt to follow it). The `suffix` field of newly created internal nodes is set on the next loop iteration (or after the loop has terminated).

In Figure 5.6 (line 438) `curr` is moved to its suffix using the `suffix` field. In Figure 5.12 (line 523) the GETSUFFIX function is used instead. If `curr` has the value $(\texttt{s}, ax)$, then GETSUFFIX`(curr)` returns $(\texttt{s}, x)$ if `s` is the root node; otherwise, $(\texttt{s.suffix}, ax)$.

5.2.3.1 Estimating the Probability of a Future Request

A final structural item we must account for is the `count` fields. In the original MONITOR-OPEN algorithm (Figure 5.6), we updated the `count` field for each context on the suffix path from the longest suffix to the root. That update gives the desired frequency property that we need to predict future queries (Lemma 5.6). In the path compressed suffix trie, we cannot perform such updates in linear time. Instead, we defer calculating the `count` for a node until after we have fully processed string T and, further, added the out-of-band character `$`.

LEMMA 5.10 (FREQUENCY IS MEASURED BY NUMBER OF LEAVES)
Let $w = xy$ be a sub-string of T with canonical NODEPTR of (x, y) in the explicit suffix tree $\mathcal{T}$ for $T\$$. Let c be the child node of x found by following y (if $y = \epsilon$, then $c = x$). The number of occurrences of w in T is given by the number of leaf nodes in the sub-tree rooted at node c.

PROOF. Let k_i, $i = 1 \ldots l$ be the first character position in T of each of the l occurrences of w. Let z be the string on the path from (x, y) to c. Because the edge $x \xrightarrow{yz} c$ is non-branching, we know that xy has always been followed by z in T. Further, by the properties of suffix trees, the suffix $T[k_i..m]$ is a leaf in $\mathcal{T}$ with a label that starts with xy. Since the label starts with xy, xyz must also be a prefix. Therefore, the leaf $T[k_i..m]$ is in the sub-tree rooted at $c = xyz$. Finally, if any leaf is in the sub-tree rooted at c, the path to the leaf starts with $w = xy$ so it must be one of the k_i occurrences accounted for. Therefore, the number of leaf nodes in the sub-tree rooted at c is exactly the number of occurrences of string w. □

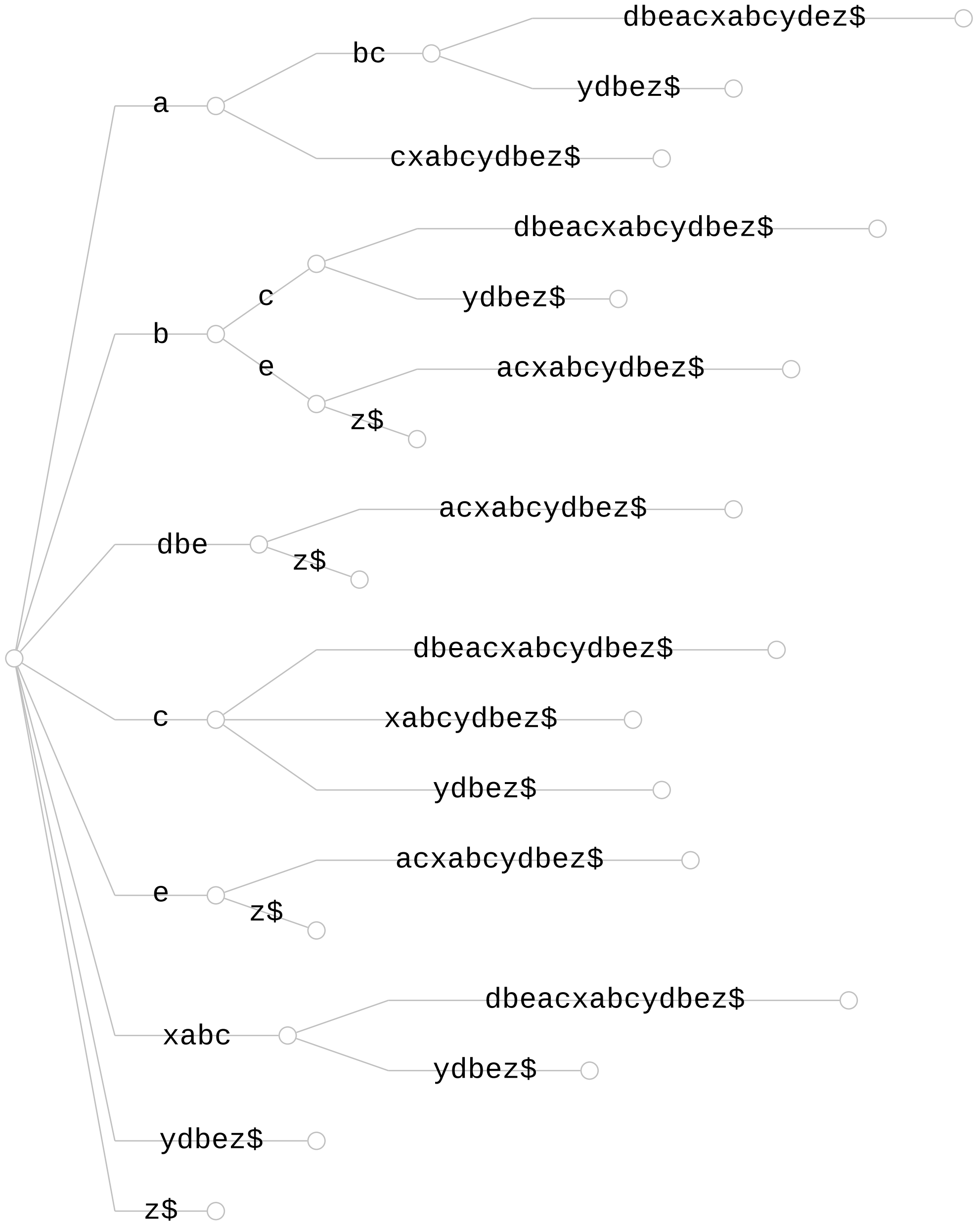


Figure 5.15: Suffix tree for trace of Figure 5.3. Suffix links are omitted to avoid clutter.

By using the approach of Ukkonen, we have shown an algorithm, MONITOR-OPEN-COMP that builds a suffix tree for string T in $O(|T|)$ time and space. Figure 5.15 shows the suffix trie that is built for the trace of Figure 5.3. In addition to the structural changes that give linear complexity, we have adapted our correlation detection to run in linear space and time.

5.2.3.2 Tracking Correlations

In the original MONITOR-OPEN algorithm (Figure 5.6), the FIND-CORRELATIONS procedure for a node n considers all of the parent nodes of n when finding initial correlations; this leads to $O(m^2)$ time and space complexity in m the length of the trace. Further, when request $k+1$ is observed, the k generalizations of the longest suffix are adjusted with VERIFY-CORRELATIONS-COMP. We cannot support these operations directly in linear time and space.

The problem of space complexity is inherent in our definition of the correlations we are willing to consider. In the suffix trie, we considered that each of the parameters of request $k+1$ can be correlated to any of the parameters of any of the k previous requests. Instead of allowing correlations of unbounded length, in the suffix tree representation we restrict the sources we consider to the S previous requests, where S is a configuration parameter used to indicate the *scope length* that we consider. With this change, the size of the `correlations` sets for each input parameter are constant with respect to the trace length m. There is still a quadratic dependence on r, the number of parameters per query: in the worst case, each input parameter can be correlated to all of the parameters of prior queries. This leads to $O(Sr^2)$ space needed for correlation sources for each of the nodes.

We use an object named `tracker` (line 499) to maintain a dictionary over a sliding window of the S previous requests. This dictionary maps a value to a set of `(type, prevcnt, param)` triples that identify prior input and output parameter values equal to the given value.

While the introduction of the scope length S makes the number of correlation sources for each node constant with respect to trace size, there is another source of quadratic space complexity. In the original implementation of MONITOR-OPEN, we stored correlations for each of the $O(n^2)$ contexts encountered during the trace. The path compression reduced the number of physical nodes in the tree to $\Theta(n)$, but it is not clear that path compression is helpful in reducing the recorded correlation sources. In fact, the very branch-free paths compressed in the suffix tree may be exactly the sequences that we would like to prefetch, provided that we can predict the actual parameter values that will be used.

The algorithm of Ukkonen represents a suffix trie compactly by storing edge labels as indexes into the string of requests. Many edges use the same character for their label, which gives the asymptotic space savings. We use a similar approach for storing correlation information. We store a single sequence `T.correlations` for the suffix tree `T`. Each element in the sequence is a set of CORRELATION objects. For each string s that appears in trace T, we define an index in this sequence that holds the CORRELATION objects for string s; we call this position the *correlation home* for string s.

DEFINITION 5.11 (CORRELATION HOME)
The correlation home for a string s in trace T is the index of the last character of the first occurrence of s in T. For example, in the trace shown in Figure 5.16, the correlation home for string `'b'` and is 3, while the correlation home for string `'db'` is 6.

More than one string may have the same correlation home. For example, in Figure 5.16 the strings `'b'`, `'ab'`, and `'xab'` all have the same correlation home (position 3). We would like to store the CORRELATION objects for all of these strings in one set. However, each of these strings may have a distinct set of CORRELATION objects. If we store these distinct sets separately, we haven't saved any space. Fortunately, we are able to exploit a trick that relies on a subset relationship between the stored sets.

LEMMA 5.12
Let s_1 and s_2 be two sub-strings of trace T where s_1 and s_2 have the same correlation home position. Without loss of generality, let s_1 be the shorter of these two strings. Then $\text{CORRELATIONS}(s_1) \subseteq \text{CORRELATIONS}(s_2)$, where $\text{CORRELATIONS}(s)$ is the set of CORRELATION objects that are associated with s.

PROOF. Since s_1 and s_2 have the same correlation home position, we know they both end at the same character position in T. This tells us that s_1 must be a suffix of s_2 Therefore, every time that we have observed string s_2, we have necessarily observed its suffix s_1. Hence, there cannot be a correlation that has always held after observing s_1 but has not held at least once when observing s_2. Thus, there cannot be a CORRELATION object $c \in \text{CORRELATIONS}(s_1)$ where $c \notin \text{CORRELATIONS}(s_2)$. This gives the stated subset relationship. □

The subset relationship described in Lemma 5.12 tells us that we can store a single set C at each correlation home position m. If this set C contains all of the CORRELATION objects for the longest string s associated with position m, then it necessarily contains all of the CORRELATION objects for all strings s_i associated with position m. It remains only to define the function $\text{CORRELATIONS}(T, s_i)$ that identifies the subset of C that applies to each string s_i.

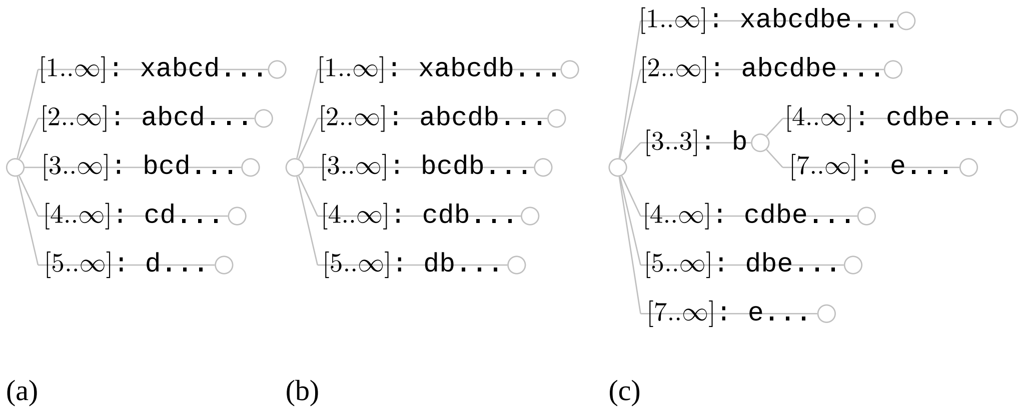
We accomplish this by augmenting the CORRELATION object with a `minorder` field. The $\text{CORRELATIONS}(T, s)$ function interprets the set of stored CORRELATION objects, returning only those whose `minorder` field is less than the length $|s|$. When we wish to remove a CORRELATION object `c` from the set associated with a string s, we do so by increasing the `c.minorder` value to $|s|$. In this way, we remove c from the results of $\text{CORRELATIONS}(\mathtt{T}, s)$ without affecting the stored set for any longer strings that have the same correlation home. The `minorder` trick allows us to represent the correlations for multiple strings in a single location.

Figure 5.16 shows the implicit suffix tree after processing (a) `'xabcd'`, (b) `'xabcdb'`, and (c) `'xabcdbe'` (the first two correspond to Figure 5.10(a) and (b) respectively). The edges in these trees are represented by STRPTR objects (indexes shown), and this allows sharing between edges. For example, the `'a'` at position 2 is used in edge `'xabcd'` and `'abcd'`. Figure 5.16 (e) shows the contents of the `T.correlations` list of sets after processing `'xabcdbe'`. The elements of these sets are CORRELATION objects extended with the `minorder` field.

As each request in the trace is processed, the `correlations` list is extended (line 512) with the correlations found by FIND-CORRELATIONS-COMP. The elements in the set appended at step i correspond to the result of FIND-CORRELATIONS in Figure 5.6 (line 428) on the first iteration with `curr` equal to the longest suffix in the existing tree. This generates all correlations that hold when considering the entire list of preceding queries as the context. The corresponding set of CORRELATION objects apply in the most specific context, but they do not apply for some generalizations. For example, Figure 5.10 showed that the correlation $\langle 1|\text{O,-2}, 2\rangle$ is included in the set for character `'c'` when it is observed in context `'xabc'` and `'abc'`, but not when it is executed in `'bc'`. In this last context, there is only one preceding query to consider, so we cannot include correlations to earlier requests. The `minorder` field of this correlation is initialized to 2. The CORRELATIONS function therefore excludes this correlation when considering context `'bc'` because $|\texttt{bc}| = 2$.

The position 4 in the `correlations` list represents the request `'c'`, and in the path-compressed suffix trie it is used for multiple edges. In Figure 5.16(a) it is used for the 4 longest edges to represent information about request `'c'` when observed in the contexts `'xab'`, `'ab'`, `'b'`, and ϵ. The set of CORRELATION objects shown in position 4 is correct for the first context `'xab'`, but it contains a superset of the desired elements when we consider the generalization contexts `'ab'`, `'b'`, and ϵ. We encode this subset property with the associated `minorder` field. The `minorder` field gives the minimum context length for which the correlation has always held. We use a `minorder` value of 2 in the CORRELATION object $\langle 1|\text{O,-2}, 2|2\rangle$. The value 2 indicates that the correlation at position 4 has held every time we have observed request `'c'` when the trace is in context $T[4-2..4-1]$ (that is, when the trace is in context `'ab'`).

When we create a new CORRELATION object for an input or output type correlation, we initialize the `minorder` field to the value of `prevcnt`. In addition to setting the `minorder` for new objects, we must also be able to remove a CORRELATION object associated with a particular edge without affecting the more specific edges sharing the same list position. For example, after processing the second `'b'` request, we recognize that the first parameter is not always equal to the constant 101. We would like to remove the CORRELATION object $\langle 1|\text{C}, 101|0\rangle$ from context ϵ, while retaining it for context `'a'`. We accomplish this removal operation by incrementing the `minorder` field, giving $\langle 1|\text{C}, 101|1\rangle$, which is shown in Figure 5.16. As we check correla-



(d)	x	a	b	c	d	b	e
(e)	$\langle 1\mid C, 42\mid 0\rangle$	$\langle 1\mid C, 10\mid 0\rangle$	$\langle 1\mid C, 101\mid 1\rangle$ $\langle 1\mid I,-1, 1\mid 1\rangle$	$\langle 1\mid C, 501\mid 0\rangle$ $\langle 1\mid O,-2, 2\mid 2\rangle$ $\langle 1\mid O,-3, 2\mid 3\rangle$	$\langle 1\mid C, 201\mid 0\rangle$	$\langle 1\mid C, 201\mid 0\rangle$ $\langle 1\mid I,-1, 1\mid 1\rangle$	$\langle 1\mid C, 201\mid 0\rangle$ $\langle 1\mid I,-1, 1\mid 1\rangle$ $\langle 1\mid O,-2, 2\mid 2\rangle$
(f)	1	2	3	4	5	6	7

Figure 5.16: Steps of tracking correlations for a suffix tree. Figures (a), (b), and (c) show the tree structure after processing `'xabcd'`, `'xabcdb'`, and `'xabcdbe'` respectively. Edges are labelled with the StrPtr indexes and the associated sub-string. Figure (d) shows the `trace` list of characters processed in the trace, (e) shows the `correlations` list of sets of Correlation objects that are stored in each position, and (f) gives the numerical index of each trace position.

tions with the Verify-Correlations-Comp procedure, we increase the `minorder` value to effectively 'remove' a correlation that no longer holds in a particular context.

The Find-Correlations-Comp procedure (Figure 5.17) generates the set of Correlation objects that hold for an observed request. For each input parameter i, it adds a new Correlation object of type C (line 531) to represent the possibility that the request always has the same value for the parameter. Next, it calls `CurrSources` to find all of the correlation sources within the S-length window of previous requests that match the current input value. The `CurrSources` method is implemented as a map from a value to a set of Correlation objects. By using techniques such as hashing, it can be implemented in amortized $O(Sr)$ time—proportional to the number of matching sources. Each triple `(type,prevcnt,param)` is used to generate a new Correlation object. The `minorder` of the new object is initialized to `-prevcnt` (recall that `prevcnt` is negative). The Correlation objects generated consist of all predictions that can be made considering the current constant value of each input parameter and a window of S prior

```
▷ Find possible correlations for node n and input parameter values InVals
procedure FIND-CORRELATIONS-COMP( T, InVals )
  for i ← 1 to InVals.length do
    corrs ← new CORRELATION(i, C, InVals[i], 0)
    for (type,prevcnt,param) ∈ CurrSources(T.tracker, InVals[i]) do
      nc ← new CORRELATION(i, type, prevcnt, param, -prevcnt)
      corrs ← corrs ∪ nc
  ADDINPUT(T.tracker,InVals)
  return corrs
end

▷ Record fetched tuple fetchVals in suffix trie T
procedure MONITOR-FETCH( T, fetchVals )
  ADDOUTPUT(T.tracker,fetchVals)
end

▷ Identify a 'home' location for reference pair n $= (s, x)$, used to store all correlations for n
function CORR-HOME(T,n)
  ▷ home identifies the last character of the first occurrence of $n = sx$ in T.trace
  return home
end

▷ Return the set of correlations that have always held at node n
function CORRELATIONS(T,n)
  homecorrs ← T.correlations[ CORR-HOME(T,n) ]
  corrs ← { c ∈ homecorrs | STRINGORDER(n) > c.minorder }
  return corrs
end

▷ Verify correlations for node n and input parameter values InVals
function VERIFY-CORRELATIONS-COMP( T, n, InVals )
  changed ← FALSE
  corrs ← CORRELATIONS( T, n )
  for c ∈ corrs do
    if InVals[ c.inparam ] ≠ CURR-VALUE( c ) then
      c.minorder = STRINGORDER(n)+1
      changed ← TRUE
  return not changed
end
```

Figure 5.17: Code to monitor parameter correlations in linear space/time.

requests. This set is assigned to the `n.correlations` field.

Finally, the FIND-CORRELATIONS-COMP procedure adds the current `InVals` to the dictionary with a call to ADDINPUT (line 535). These will be considered as possible correlation sources for future requests. Likewise, after a successful fetch, MONITOR-FETCH calls ADDOUTPUT (line 541) to add the most recent fetched values to the dictionary. The ADDINPUT and ADDOUTPUT methods remove values associated with a request more than S in the past. In this way, the size of the dictionary does not exceed $O(S)$ requests or $O(Sr)$ values. Overall, the amortized time complexity does not exceed $O(Sr)$.

The result of FIND-CORRELATIONS-COMP is a set of CORRELATION objects representing the equality relationships that hold for the current OPEN request. For request k, the MONITOR-OPEN-COMP adds this set at position k in list `T.correlations` (line 512). Each node n (virtual or explicit) that is created at step k corresponds to a string $T[i..k]$ that has not appeared earlier in the trace. Therefore, the correlation home for each newly created node n is k. The set of CORRELATION objects is the same for all newly created nodes, except possibly where the `prevcnt` of a CORRELATION exceeds the length of the string associated with n. We represent the length of this string with STRINGORDER(n), and use the `minorder` field of the CORRELATION to exclude the correlation from the set for nodes with a context that is too short.

The FIND-CORRELATIONS-COMP works with MONITOR-OPEN-COMP to find the initial set of correlations for each newly created node n (virtual or explicit). The other facet of correlation detection is the process of removing correlations that no longer hold. This is implemented in the VERIFY-CORRELATIONS-COMP procedure.

First, we need a way to associate a node in the suffix tree (either explicit or virtual) with its correlation home position. Recall that each node n can be expressed in canonical form as $n = (s, x)$ where s is an explicit node and x is a string (possibly empty) leading out of s. The correlation home for n is the last position of the first occurrence of string sx in the trace. We use function CORR-HOME(T, n) to find this position. We can implement CORR-HOME efficiently in $O(1)$ time by using the STRPTR representation of x (we omit the details). The CORRELATIONS(n) returns the set of CORRELATION objects for node n based on its correlation home position (given by CORR-HOME(n)).

Consider a NODEPTR n with value (s, xa) found by `FindChild` (line 514). We have observed sxa before, and a previous call to MONITOR-OPEN-COMP has initialized the correlations for node n as CORRELATIONS(n). It may be that some of these correlations no longer hold. The VERIFY-CORRELATIONS-COMP function considers the set CORRELATIONS(n) of CORRELATION objects. For each CORRELATION object c, we compare the current value of c to the associated input value of the current query. If the values do not match, this represents a case where a correlation that previously held in context n no longer holds, hence a guessed true correlation has been shown to be false. In this case, we increase the c.minorder field to be $|sx|+1$. This increase effectively removes c from the set of possible correlations associated with n.

If any of the CORRELATION objects is removed from a set of possible correlations by increasing the minorder field, then we set changed to TRUE. If we reach a point in the iteration of MONITOR-OPEN-COMP where FindChild returns an existing child n and no correlation sets change values (line 519), then we can halt the iteration. Any suffix of curr will also have a Q child, so the original algorithm by Ukkonen [180] terminated as soon as child n is found. If we find that all of the guessed correlation sources for an existing child n still hold, then there will be no changes in generalization nodes, which contain a subset of the guesses for n.

We have modified the original algorithm [180] to continue processing generalizations of curr until no further changes are made by VERIFY-CORRELATIONS-COMP. We should be suspicious that this modification has affected our asymptotic complexity. Fortunately, we can see that the additional cost of the extra iterations cannot exceed $O(mSr^2)$ for a trace of length m with scope length S and maximum parameters r per request. We continue iterating beyond the original algorithm every time that we remove at least one CORRELATION from node n. Since we initially create n with at most Sr^2 of these objects, we will introduce at most $O(mSr^2)$ extra work.

5.2.3.3 Summary of Suffix Tree Detection

Together, the above alterations allow us to build a suffix tree (or path compressed suffix trie) for a length m trace of requests with at most r parameters in $O(mSr^2)$ time and space. We convert the implicit suffix tree into an explicit suffix tree by appending $ (without affecting our asymptotic bound). The frequency of each context within the tree can be assessed by counting the number of leaf nodes in the sub-tree rooted at the appropriate node. For each edge $n \xrightarrow{a} na$, we maintain a set of CORRELATION objects that represent correlations that have always held when request a was submitted while the trace was in context n. This set is accessed with CORRELATIONS(T, na). Thus, the path compressed suffix trie (or suffix tree) can be used to satisfy all of the requirements of the atomic suffix trie, all in time and space that is linear in the number of requests.

5.2.4 Summary of Pattern Detector

In order to make semantic prefetching choices, we must predict the probability of future requests and the parameter values that will be used if they are submitted. We have decided to limit our choices to finite state models, which predict future behaviour based on a finite set of states.

A particular subset of finite state models is the order-k model, which makes these predictions by defining contexts of the k preceding requests and using these to condition probability predictions and predictions of parameter correlations. This approach runs into difficulty when selecting an appropriate value of k. There is no a-priori information that we can use to give a strict bound on k. Choosing a low value of k will miss special cases that have unique probability or correlation behaviour, while a high value of k reduces the generalization performed during training. We introduced confidence intervals of the probability estimates given by the order-k model in order to help with selecting an appropriate k value.

The suffix trie data structure provides a mechanism that lets us simultaneously maintain order-k models for all $k = 1 \ldots n$ for a trace of length $n + 1$. After we have observed a trace, we can make a post-hoc decision of the best k value. Because we have the model for all k values, we can select a variable-order order model that uses different k values at different positions in the tree, depending on the estimated query costs and correlation results.

The suffix trie structure gives us all that we need to make effective cost-based semantic prefetching decisions. Unfortunately, the asymptotic complexity is $O(m^2)$ for a trace of length m. It is possible to use the structure in practical implementations, for example by choosing an upper limit for k, say 100. Fortunately, we can avoid these ad-hoc limits and achieve better performance by using a path-compressed suffix trie, which we can build in linear space and time.

In a path-compressed suffix trie, branch-free paths are compressed into string-labelled edges. Strings are represented by pointers into the single string that represents all requests observed so far. We define a configuration parameter S that limits the scope length of possible sources of correlation values that we consider. A dictionary structure is used to map an input parameter value v to the set of CORRELATION objects that represent parameters within scope S that match v, and we extend the suffix tree algorithm to initialize and maintain the set of correlations that have always held for a request Q in context c.

The suffix trie or suffix tree provides all the information that we need to make prefetching decisions, and we describe that process in Section 5.3. After optimization, the Query Rewriter is used to generate an alternate query that will fetch the results of the submitted query and also prefetch the results specified by the optimizer.

5.3 Pattern Optimizer

Section 5.2 described how Scalpel builds a trie-based data structure during the training period. The suffix trie summarizes frequency information and correlations that were observed during the training period, and the Pattern Optimizer uses this information to make prefetching decisions. Figure 5.18 gives an overview of the implementation of the Pattern Optimizer.

The Pattern Optimizer uses a cost-based decision process to decide what queries to prefetch. Section 5.3.1 shows how we determine if a query is likely to be submitted sufficiently often to be worth prefetching. The Pattern Detector supplies a function COUNT-OCCURRENCES(n) that reports how many times a node n was observed during the training period. The Pattern Optimizer uses this frequency information to derive estimates (with confidence intervals) of the probability

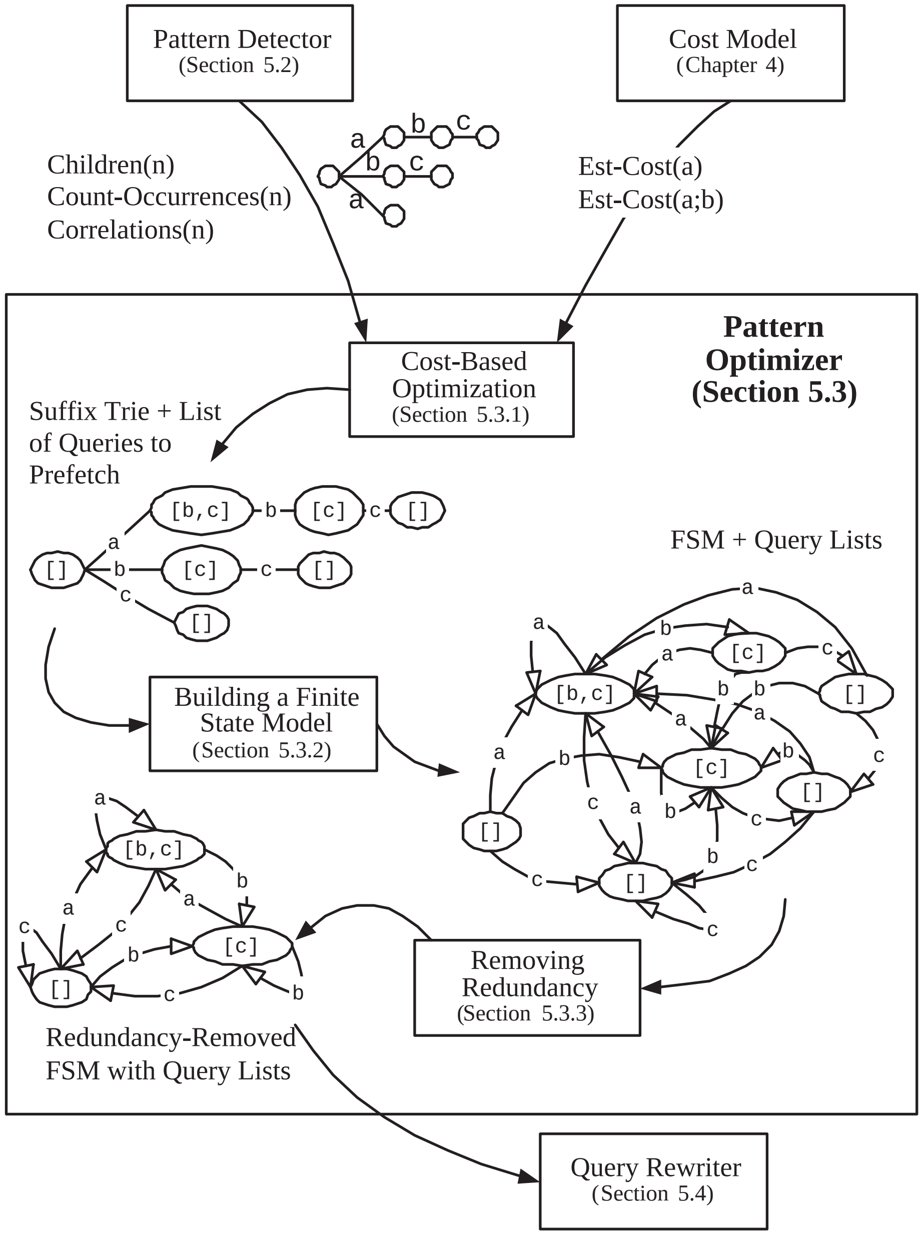


Figure 5.18: Overview of the Pattern Optimizer.

of future requests. The Pattern Optimizer combines these probability estimates with a cost estimate function EST-COST provided by the Cost Model in order to decide what queries to prefetch when a demand fetch is sent to the DBMS. Scalpel is not able to prefetch a query if it cannot predict what values will be used for its parameters. The suffix trie built by the Pattern Detector provides a CORRELATIONS function that gives the predict correlation values (if any) for each parameter. The selected queries are stored as a list for each node in the suffix trie.

The suffix trie does not directly provide all of the details needed at run-time. For example, it is not clear what the Prefetcher should do when a request is seen that has never been seen at a particular node in the suffix trie. Section 5.3.2 describes how the Pattern Optimizer uses the suffix trie to build a finite state model. This finite state model retains the lists of queries that were selected for each node in the suffix trie.

The finite model generated directly from the suffix trie contains significant redundancy. Section 5.3.3 shows how this finite state model can be simplified by removing this redundancy, generating a redundancy-removed finite state machine annotated with lists of queries to prefetch.

Finally, Section 5.3.4 summarizes the operation of the Pattern Optimizer. After the Pattern Optimizer has built its simplified FSM, the Query Rewriter component (Section 5.4) modifies the FSM by generating combined queries that implement the selected prefetches.

5.3.1 Cost Based Optimization

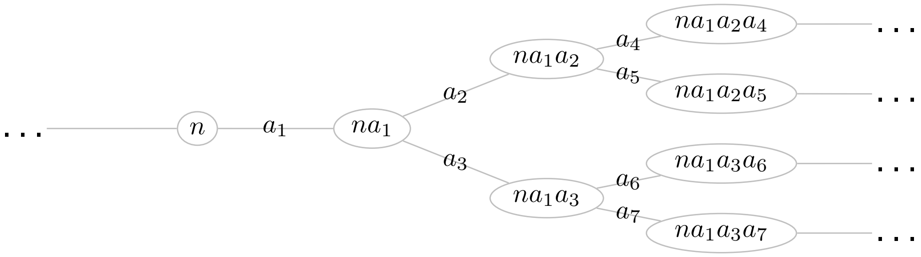


Figure 5.19: Example for choosing prefetches.

Figure 5.19 shows a fragment of a suffix trie that we might find after the training period. Node n represents a context associated with string $n = n_1n_2 \ldots n_k$ of requests, and the edge from n to na represents the fact that we have observed query a in the past when the trace was in context n. In an atomic suffix trie, n is a physical node; in a path compressed suffix trie, $n = (s, x)$ may be a physical node ($x = \epsilon$) or virtual node.

We are interested in using cost estimates to decide what queries Scalpel should prefetch when the application submits request a_1 after submitting the query sequence n. We could choose to submit a_1 to the server unmodified. Alternatively, we could elect to prefetch a request we anticipate might be submitted in the future; for example, a_2. In this case, we would submit a modified query to the server that would fetch the results of a_1 and the predicted results for a_2. If the application then submitted a_2 with parameter values that match our predictions, Scalpel would use the results of the combined query to answer a_2 and would have saved the overhead of having to issue a_2 to the server as a separate query. Clearly, the costs of these two alternatives will depend on the execution costs of the various queries involved, and on the likelihood that a_2 will actually occur following request a_1 in context n. We represent this likelihood with the conditional probability mass function $p(a_2|na_1)$, which gives the conditional probability that query a_2 will occur next, given that we have previously observed the sequence na_1 of requests.

We can use the EST-COST function (defined in Chapter 4) to estimate the total execution cost associated with a_1 and a_2 when they are executed unoptimized. This expected cost is given by Equation 5.9:

$$\text{EST-COST}(a_1) + p(a_2|na_1) \times \text{EST-COST}(a_2) \tag{5.9}$$

If Scalpel does prefetch a_2, the total cost will be the cost of executing the larger combined query which prefetches results for a_2 in addition to a_1. In Section 5.4, we will describe how we form this combined queries using strategies similar to the outer union and outer join strategies we used for nested patterns of requests. The combined query is implemented using an outer join strategy if a_1 returns at most one row; otherwise, an outer union approach is used. For now, we will denote the cost of this combined query by EST-COST$(a_1; a_2)$, ignoring the details of which of these approaches is used.

With these estimates of costs, we define PREFETCH-BENEFIT(n, a_1, a_2) as an estimate the savings achieved by prefetching query a_2 when query a_1 is submitted after sequence n.

$$\begin{aligned} \text{PREFETCH-BENEFIT}(n, a_1, a_2) \quad = \quad & \text{EST-COST}(a_1) + p(a_2|na_1) \times \text{EST-COST}(a_2) \\ & - \text{EST-COST}(a_1; a_2) \end{aligned} \tag{5.10}$$

With the estimate of prefetching benefit given in Equation 5.10, it is worthwhile to prefetch a_2 when a_1 is submitted in context n if the following holds:

$$p(a_2|na_1) > \frac{\text{EST-COST}(a_1; a_2) - \text{EST-COST}(a_1)}{\text{EST-COST}(a_2)} = P_{\text{min}}(n, a_1, a_2) \tag{5.11}$$

The value $P_{\text{min}}(n, a_1, a_2)$ gives the minimum probability for which it is worthwhile to prefetch query a_2 when a_1 is submitted in context n. If we think the probability of observing a_2 is greater than $P_{\text{min}}(n, a_1, a_2)$, then we expect that it would be cheaper to prefetch a_2.

The value of P_{min} is defined by the expected benefit when prefetched results are useful and the cost when the results of a_2 are not useful. In fact, we can provide a lower bound on P_{min} by considering the cost of a_2 relative to the per-request overhead U_0. Recall that the EST-COST function (Chapter 4) provides Scalpel's estimate of the latency associated with a request; the estimate is derived using observations of costs during the training period and a cost model based on either a nested loops join or union. Further, the estimate is combined with SRV-SAVINGS, an estimate of the benefit the server can achieve if requests are submitted together instead of as separate queries. The outer join strategy is used only if the outer query returns at most one row ($|a_1| \leq 1$), otherwise the outer union strategy is used. Therefore, we can bound the cost of the combined query as follows:

$$\text{EST-COST}(a_1; a_2) \leq \text{EST-COST}(a_1) + \text{EST-COST}(a_2) - U_0 \tag{5.12}$$

Unless the database server is able to exploit sharing to execute the combined query more efficiently than the two separate queries, the bound in Equation 5.12 is tight. Equation 5.12 provides an upper bound for $P_{\min}$:

$$P_{\min} \leq 1 - \frac{U_0}{\text{EST-COST}(a_2)} = P_{\min 0} \tag{5.13}$$

The two sides of Equation 5.13 are equal unless the server is able to exploit sharing (corresponding to SRV-SAVINGS < 1). If $p(a_2|na_1) > P_{\min 0}$, it is worth prefetching; otherwise, it might still be worth prefetching if SRV-SAVINGS < 1, in which case $P_{\min} < P_{\min 0}$.

In order to assess whether it is worthwhile prefetching a query, we need an estimate of $p(a_2|na_1)$ to compare to $P_{\min}$. Equation 5.6 showed how we can compute $\hat{p}(x_i|s)$, an estimate of the probability of request x_1 after the string s of requests. The definition of $\hat{p}(x_i|s)$ in Equation 5.6 is based on COUNT-OCCURRENCES(σ, t), a count of how many times substring σ occurs in string t. The COUNT-OCCURRENCES function can easily be defined for a suffix trie data structure based on the `count` field (as shown by Lemma 5.6). The COUNT-OCCURRENCES function can also be defined for a path-compressed suffix trie based on the number of leaves below the associated node (shown in Lemma 5.10).

As discussed in Section 5.2.1.1, the point estimate $\hat{p}(x_i|s)$ does not tell the whole story in that it does not reflect how confident we are in the estimate. Equation 5.7 provides us with a confidence interval $CI = [p_0, p_1]$ that expresses the amount of confidence we place in the estimate $\hat{p}(x_i|s)$.

We use the confidence interval to provide a prefetching decision as follows. If the low-point of the confidence interval (p_0) is greater than $P_{\min}$, then we will prefetch as the estimated value $\hat{p}(a_2|na_1)$ is significantly greater than $P_{\min}$ (at the α level of confidence). If the high-point of the confidence interval (p_1) is less than $P_{\min}$, we will not prefetch as the estimate $\hat{p}(a_2|na_1)$ is significantly lower than $P_{\min}$ (at the α level). In the remainder of the cases, $P_{\min}$ is inside the confidence interval for $\hat{p}(a_2|na_1)$. In this case, we do not have enough information to decide whether to prefetch or not. No matter how long the training period, there will be some contexts where we have not made enough observations to make a definite decisions: this is a consequence of growing the context length on each request. In these cases, we have a context n that is too specific to provide a definite choice; however, there may be a shorter context n' that has been observed sufficiently often to be definite. We can use the suffix links in the suffix trie data structure to explore information about more general contexts that are suffixes of n.

5.3.1.1 Feasible Prefetches

Before Scalpel decides whether it is beneficial to prefetch query a_2 after observing a_1 in context n, it must first decide whether such a prefetch is *feasible*. It is feasible to prefetch a_2 if we can make a prediction of the actual values that would be used for each parameter of a_2 if it were submitted. The Pattern Detector (Section 5.2) described how we maintain sets of correlations that have always held. In the atomic suffix trie data structure, these are stored in a `correlations` field of the node associated with na_1a_2. In the compressed suffix trie, the set is encoded in the list element `T.correlations[i]` where i is the correlation home for string na_1a_2, and the encoded set is interpreted based on the `minorder` field of the CORRELATION objects and the length $|na_1|$. The $\textsc{Correlations}(x)$ function was defined to find the set of CORRELATION objects for the node identified with string x (Figure 5.17). We extend $\textsc{Correlations}(x)$ to (atomic) suffix tries by returning the `correlations` field of the associated node..

The $\textsc{Correlations}(na_1)$ set contains correlations that apply to all of the parameters of query a_1. We use the function $\textsc{CorrForParm}(C, i)$ to find the subset of correlations in set C that apply to parameter i, following the same approach we used for nested request patterns (Definition 3.1, page 27). Armed with these two functions, we can define what it means for a request to be feasible as follows.

DEFINITION 5.13 (FEASIBLE PREFETCH)
Request a_2 is a *feasible prefetch* when query a_1 is submitted in context n if there is a predicted correlation for each of the r parameters of a_2. Let $C = \textsc{Correlations}(na_1a_2)$. Then $\textsc{Feasible}(na_1, a_2)$ is defined as follows:

$$\textsc{Feasible}(na_1, a_2) \iff \textsc{CorrForParm}(C, i) \neq \emptyset \quad \forall\, i \in [1, r] \tag{5.14}$$

5.3.1.2 Choosing the Best Feasible Prefetch

There may be more than one feasible, beneficial prefetch from a given context. Thus, Scalpel must decide which, if any, of these prefetches to make. When query a_1 is submitted, Scalpel can choose to prefetch 'wide' by prefetching several possible subsequent queries in the hope that one will actually be requested after a_1. For example, Scalpel could prefetch the results for both a_2

and a_3 in Figure 5.19. Alternatively, Scalpel could prefetch ‘deep’ by prefetching a chain of candidates, each of which is predicted to follow it’s predecessor. For example, Scalpel could prefetch both a_2 and a_4. Finally, we could also consider some combination of these two. For example, we could prefetch query a_2, a_3, and a_6. At present, the Scalpel prototype considers only prefetching ‘deep’; that is, chains of queries, each of which is predicted to be the best prefetch candidate.

```
▷ Choose queries to prefetch when $a_1$ submitted after $n$
procedure CHOOSE-PREFETCH( $n$, $a_1$ )
  $x \leftarrow \epsilon$                          ▷ The list of queries to prefetch
  while $|x|$ < $M$ do
    ▷ Choose the best feasible prefetch candidate
    best $\leftarrow$ BEST-PREFETCH( $n$, $a_1$, $x$ )
    if best $=$ NIL then break
    $x \leftarrow x$ + best                     ▷ Append to list of queries to prefetch
  ▷ Annotate the node $na_1$ with the queries to prefetch when $a_1$ is submitted after sequence $n$
  SET-PREFETCH( $na_1$, $x$ )
end
```

Figure 5.20: Choosing a list of queries to prefetch.

Figure 5.20 shows the CHOOSE-PREFETCH procedure that is used to choose the list of queries that should be prefetched when a_1 is submitted after the query sequence n. Procedure CHOOSE-PREFETCH uses a loop to find a chain of prefetch queries, which are stored for use at run-time. We use a configuration parameter M to limit the maximum number of queries that Scalpel will prefetch.

At each step of the loop, CHOOSE-PREFETCH uses the BEST-PREFETCH function (Figure 5.21) to find the best feasible candidate that should be prefetched after the query sequence na_1x has been observed. The BEST-PREFETCH function considers each query b that was observed following na_1x during the training period. The $\text{CHILDREN}(na_1x)$ function is used to find these queries, abstracting the details of atomic or path compressed suffix trie. For each child b, BEST-PREFETCH uses the SHOULD-PREFETCH function to decide whether b is both feasible and beneficial.

Once a best prefetch candidate `best` is found, the loop proceeds by assuming that `best` is executed next. In this way, a chain of prefetch queries is selected, conditionally assuming that each selected query is actually used. We must also consider the conditional probability that the entire chain is followed. SHOULD-PREFETCH accomplishes this by using $N = \text{COUNT-OCCURRENCES}(na_1)$ (the count of the originating context na_1) as the denominator when forming the estimate $\hat{p}$ and its confidence interval (line 598). The numerator uses the full context length $X = \text{COUNT-OCCURRENCES}(na_1xb)$. This choice of denominator accounts

```
▷ Find the best request b to prefetch if $a$ follows sequence $na_1x$
function BEST-PREFETCH( $n$, $a_1$, $x$ )
  best_b ← NIL                        ▷ Best prefetch candidate
  best_benefit ← 0                    ▷ Expected benefit for prefetching best_b
  for b ∈ CHILDREN( $na_1x$ ) do
    benefit ← SHOULD-PREFETCH( $n$, $a_1$, $x$, $b$ )
    if benefit > best_benefit then
      best_b ← b
      best_benefit ← benefit
  return best_b
end

▷ Return the expected benefit of prefetching $a_2$ when $a_1$ is submitted after sequence $n$
function SHOULD-PREFETCH( $n$, $a_1$, $x$, $b$ )
  ▷ We should not attempt to prefetch $b$ unless it is feasible after $na_1x$
  if not FEASIBLE$(na_1x, b)$ then
    ▷ It is not feasible to prefetch $b$; use benefit of $-\infty$
    return $-\infty$
  N ← COUNT-OCCURRENCES$(na_1)$
  X ← COUNT-OCCURRENCES$(na_1xb)$
  CI ← CONFIDENCE-INTERVAL( X, N )                ▷ Equation 5.7
  pmin ← $P_{\min}(n, a_1x, b)$                   ▷ Equation 5.11
  if CI.high < pmin then
    ▷ Prefetching is significantly worse (at the $\alpha$ level)
    return $-\infty$
  else if Ci.low > pmin then
    ▷ Prefetching is significantly better (at the $\alpha$ level)
    savings ← PREFETCH-BENEFIT$(n, a_1x, b)$      ▷ Equation 5.10
    return savings
  else
    ▷ Not enough information to decide; consider generalization
    if GETSUFFIX(n) ≠ NIL then
      return SHOULD-PREFETCH( GETSUFFIX(n), $a_1$, $x$, $b$ )
    else
      return $-\infty$
end
```

Figure 5.21: Choosing the best prefetch query.

for the conditional probability that all links along the chain na_1x are followed. If SHOULD-PREFETCH finds that there is insufficient evidence to form a firm decision about prefetching, it returns the result for the generalization given by GETSUFFIX(n) (if it exists). When a context is reached where there are no feasible prefetch candidates that are beneficial, the loop in CHOOSE-PREFETCH terminates.

The CHOOSE-PREFETCH procedure identifies a list of queries to prefetch for each edge $n \xrightarrow{a_1} na_1$ in the tree. This prefetch list represents additional queries that should be prefetched when we observe a query a_1 being submitted at run-time after observing the string of queries n. The SET-PREFETCH function is used to associate this list with node na_1. The Query Rewriter uses these lists to generate combined queries that fetch the results for a_1 and all queries in the prefetch list. First, however, we define how the Pattern Optimizer uses the suffix trie data structure to build a finite state model.

5.3.2 Building a Finite-State Model

After selecting queries to prefetch, the result of pattern detection consists of a suffix trie (possibly path compressed), where each node is annotated with a list of queries to prefetch. This structure can be used as a form of finite state model during run-time. In this model, the nodes of the trie form the states and edges are used to provide the transition function.

While the suffix trie does give a kind of finite state model, a transition function defined using only the edges observed during a finite training period cannot be considered to be complete. We will certainly encounter novel requests in some context. This will occur due to the low number of observations for long contexts, particularly those associated with leaf nodes. When we encounter a novel request in a context, there is no edge to follow. We have no prefetching decisions available, and no next state to move to.

We could implement a heuristic that adds an implicit transition to the root when we encounter a novel request in a context. This approach is sound, but it ignores the fact that, while the current request is novel in the specific context of the tree, it may have been observed in a more general context that considers fewer of the preceding queries as a conditioning context. Instead, we exploit the suffix links. When we encounter a query a that is novel at node n, we consider GETSUFFIX(n) (if it exists). If there is a query a that was not observed at all during training, we will not have an edge for it anywhere in the trie. Therefore, we use a transition to the root of the trie. Our transition function δ_τ is defined for suffix trie T as follows:

$$\delta_\tau(x,a) = \begin{cases} xa & a \in \textsc{Children}(x) & (C_1) \\ \delta_\tau(y,a) & \neg C_1 \wedge y = \textsc{GetSuffix}(x) \wedge y \neq \textsc{Nil} & (C_2) \\ s_0 & \neg C_1 \wedge \neg C_2 & (C_3) \end{cases} \tag{5.15}$$

In the first case (C_1), there is an edge $x \xrightarrow{a} xa$ recorded in the trie, and δ_τ uses a transition based on that edge. In the second case (C_2), query a is novel at node x; we use the transition defined by $\textsc{GetSuffix}(x)$. In the final case (C_3), the query a has not been seen at all during the training phase. In this case, we use a transition to s_0, the root node of the trie. The δ_τ definition in Equation 5.15 allows us to define a finite state model M_τ for suffix trie τ as follows.

DEFINITION 5.14 (FINITE STATE MODEL FOR SUFFIX TRIE τ)
Let Σ be the set of all possible requests. Let $S_\tau = \{x \in \Sigma^* \mid x \in \textsc{Nodes}(\tau)\}$ be a set of states corresponding to the nodes of τ, and let $s_0 = \epsilon \in S_\tau$ be an initial state corresponding to the root node of τ. Let δ_τ be defined as in Equation 5.15, and let $\hat{p}(a|x)$ be a conditional probability mass function given by the counts of τ. With these settings, then $M_\tau = \langle S_\tau, \Sigma, \delta, s_0, \hat{p} \rangle$ is the *finite state model for suffix trie* τ.

With δ_τ defined as shown in Equation 5.15, all of the transitions into node $s \in S \setminus s_0$ are caused by the same query. We use function $\textsc{Get-Incoming}(q)$ to identify this query. At run-time, we enter state q when we observe a call to $\textsc{Open}(a)$ for query $a = \textsc{Get-Incoming}(q)$.

Definition 5.14 gives a model that can be used at run time. At each point in the trace, the current state q of model M_τ is the longest element of τ that matches the current sequence of requests. When processing request a, M_τ moves to a new state by following $q' = \delta_\tau(q, a)$. This new state q' has a list of queries that should be prefetched when query a is submitted to the DBMS. While the definition of M_τ is sound, M_τ may be substantially larger than an equivalent model due to redundancy introduced while building the suffix trie. In the next section, we describe how this redundancy arises and how we can remove it.

5.3.3 Removing Redundancy

The finite state machine M_τ that is build directly from a trie contains significant redundancy resulting from having multiple context lengths that predict the same future behavior. To understand this issue, consider the following contexts from Figure 5.15 (page 119): `'b'`, `'ab'`, and `'xab'`. This is a sequence of successively less general contexts, each of which ends with `'b'`. Suppose that, using the procedures described in Section 5.3.1.2, Scalpel has decided not to prefetch from context `'b'`, but to prefetch query Q_c in context `'ab'`. This may happen because Scalpel observes that the conditional probability of `'c'` after the sequence `'ab'` is higher than the conditional probability of `'c'` after `'b'` alone. This is an example of Scalpel avoiding prefetching decisions based on contexts that are too short.

As we consider successively more specific contexts, such as `'xab'`, there are several possibilities. The procedures from Section 5.3.1.2 may also decide that it is worthwhile to prefetch Q_c in context `'xab'`. Such a prefetching decision is redundant, because whenever the system is in context `'xab'`, it is also in the more general context `'ab'` for which the same prefetching decision has been made. Second, Scalpel may have found that context `'xab'` was not observed enough times to make a definite prefetching decision. In that case, CHOOSE-PREFETCH selects to prefetch Q_c in keeping with the generalized context `'ab'`; again, the prefetch is redundant. The final possibility is that prefetching `'c'` is definitely rejected for `'xab'` in favour of either prefetching a more beneficial query or not prefetching at all. In this case, `'xab'` represents a special case of `'ab'`, and the prefetching recommendation at `'xab'` is not redundant.

```
▷ Determine whether context n is redundant
function IS-REDUNDANT( n )
  if GETSUFFIX( n ) = NIL then
    ▷ The root node has no suffix, and it is not redundant.
    return FALSE
  if GET-PREFETCH( n ) $\neq$ GET-PREFETCH( GETSUFFIX( n ) ) then
    return FALSE
  else
    for child $\in$ CHILDREN(n) do
      if $\neg$IS-REDUNDANT( child ) then return FALSE
end
```

Figure 5.22: Marking redundant nodes.

Scalpel uses the IS-REDUNDANT(n) function shown in Figure 5.22 to determine whether or not a given context node n is redundant. Node n is redundant if it has the same prefetch set as its suffix, and, further, all of its child nodes are redundant. This definition of redundancy lets us define a finite state machine M for suffix trie τ that avoids redundant nodes.

DEFINITION 5.15 (REDUNDANCY-REMOVED FINITE STATE MODEL FOR TRIE τ)
Let Σ be the set of all possible possible requests. Let Q be a set of states corresponding to the non-redundant nodes of τ, defined as follows:

$$S = \{x \in \Sigma^* \mid x \in \tau \wedge \neg\textsc{Is-Redundant}(x)\} \tag{5.16}$$

Let $s_0 = \epsilon \in S$ be an initial state corresponding to the root node of τ. Let δ be a modified definition of transition function modified from the definition of δ in Definition 5.14 as follows:

$$\delta(x,a) = \begin{cases} xa & a \in \textsc{Children}(x) \wedge \neg\textsc{Is-Redundant}(xa) \quad (C_1) \\ \delta(y,a) & \neg C_1 \wedge y = \textsc{GetSuffix}(x) \wedge y \neq \textsc{Nil} \quad (C_2) \\ s_0 & \neg C_1 \wedge \neg C_2 \quad (C_3) \end{cases} \tag{5.17}$$

Then we say that $M = \langle S, \Sigma, \delta, s_0, \hat{p} \rangle$ is the redundancy-removed finite state model for suffix trie τ, where $\hat{p}$ is a conditional probability mass function derived from τ.

The redundancy-reduced finite state machine M has the same behaviour as M_τ, although it has fewer nodes. In particular, model M_τ contains states corresponding to the leaf nodes of trie τ, even though these are never useful.

5.3.4 Summary of the Pattern Optimizer

After a training period that observes a trace T of requests, the Pattern Detector generates a suffix trie τ that encodes a summary of the information gathered during the trace. The correlations that were found to always hold during the training period are used to find what prefetches are *feasible* in that we can predict actual values that would be used if the predicted request were submitted by the application. Further, frequency information in the trie is used to generate confidence intervals (at a pre-configured level α) for the probability of executing a request a when in a context n based on the ratio of COUNT-OCCURRENCES(na) to COUNT-OCCURRENCES(n).

These confidence intervals are used to determine whether it is worthwhile to prefetch a request a. By comparing the estimated cost of a to the per-request overhead U_0, the confidence interval is used to find if prefetching a is a) definitely beneficial; b) definitely *not* beneficial; or, c) indeterminate. The last case occurs in a trie built even for a very long training period, as it is a consequence of the low frequency components associated with the longest contexts in the trie. When it is indeterminate whether it is prudent to prefetch a in context n, we use suffix links in the trie to use the prefetching decision in a generalization of n where we have enough observations to make a definite decision.

The Pattern Optimizer finds a prefetch list for every context in the suffix trie, whether it is an explicit node or a virtual node. Therefore, the Pattern Optimizer uses $O(n^2)$ space in n the number of requests observed during the training period. Further, as currently stated the algorithm could take up to $O(n^3)$ running time due to the traversal of suffix links; however, an $O(n^2)$ implementation can be achieved with a little care to re-use previous results. The redundancy in the atomic suffix trie suggests that a linear algorithm can be developed. However, such an approach would also need a way to control the size of the finite state machine generated by the Pattern Optimizer. At present, the Pattern Optimizer could select up to $O(n^2)$ states with distinct prefetching choices. Implementing a linear complexity Pattern Optimizer is an important topic for future study. At present, we have found that the current optimization time is not excessive for the systems we have tested.

After using CHOOSE-PREFETCH to identify a list of queries to prefetch for the contexts in the suffix trie, the Pattern Optimizer generates a redundancy-removed finite state model M for the trie based on τ. Each node n in this trie is annotated with a list of queries that should be prefetched when the machine transitions into the node. The Query Rewriter (Section 5.4) combines this list of queries with the query on the edge entering n in order to fetch the originally request results and all prefetched results.

5.4 Query Rewriter

The result of the Pattern Optimizer is a finite state model M. Each state n in the model represents a context that the model might be in at run-time, and each state has an associated list of queries that should be prefetched. The Query Rewriter constructs combined queries that can be used to execute these prefetches, and it also builds a list of ACTION objects that describe how the Prefetcher should execute the prefetches. Figure 5.23 shows an overview of the input the Query Rewriter takes from the Pattern Optimizer and the output that it saves persistently for use at run-time.

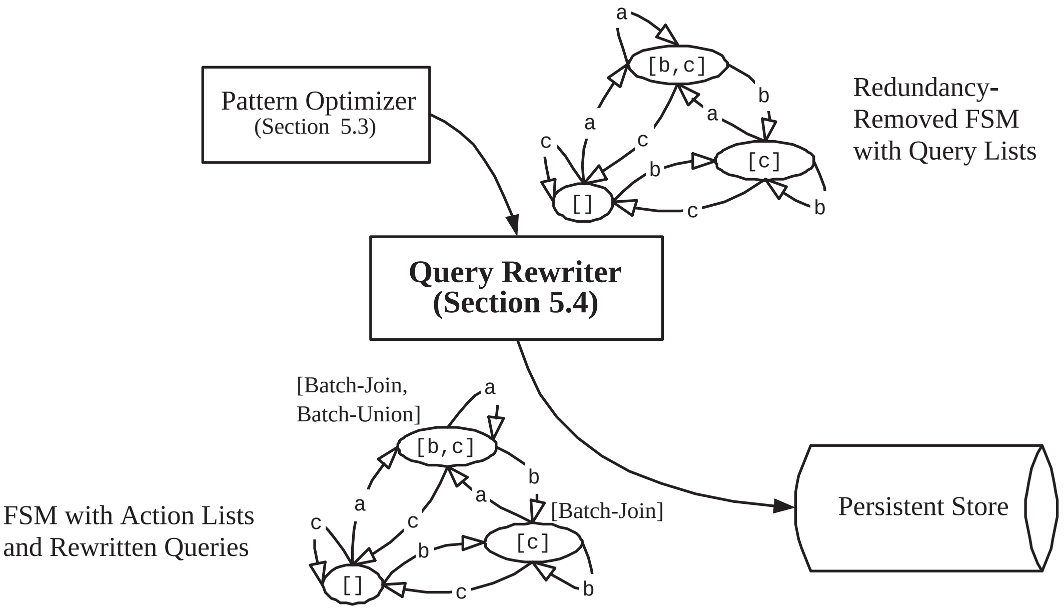


Figure 5.23: Overview of the Query Rewriter

Every state n except the initial state s_0 has the same label a on any edges entering the state, and we represent this with GET-INCOMING(n). Further, the Pattern Optimizer has associated a list of queries to prefetch with each state n; we represent this with GET-PREFETCH(n). Finally, each state n has an associated set of correlations that are predicted to hold, given by CORRELATIONS(n).

The Query Rewriter uses this information to generate (for each state n) a combined query Q' that encodes the results for GET-INCOMING(n) and also for the list GET-PREFETCH(n) of queries that are to be prefetched. The combined query is built using rewrite rules and the correlation predictions stored in CORRELATIONS(n). Further, the Query Rewriter adds ACTION objects as annotations to each state in the model. These ACTION objects are similar to those used for nested request patterns (Section 3.3.5). These ACTION objects are used at run-time to inform Scalpel of how each request should be answered. Finally, at the end of the training period, the finite state model with its ACTION annotations is stored persistently for use at run-time.

5.4.1 Alternative Prefetch Strategies

There are several mechanisms that we could consider for prefetching the results of future queries. We evaluated the following approaches:

Batch requests Several DBMS products allow a request OPEN(Q) to consist of a *batch request* such as $Q = q_1; q_2; \ldots; q_k$. The OPEN($Q$) request returns a list of cursors, and the database-access API (such as JDBC) provides a mechanism to move sequentially through the list of cursors.

Stored procedure As well as batches of queries, several DBMS products support *stored procedures*. These allow procedural code to be executed by the DBMS. As with batch requests, multiple cursors can be returned. While a batch request is specified in an OPEN call, a stored procedure is created persistently in the database schema.

Join As with the nested patterns of Chapter 3, we can combine queries using an outer join.

Union A join approach uses separate columns for each of the original queries. If the original result sets are union-compatible then we can use a UNION to prefetch the desired results. Even if the results are not union-compatible, we could use a construct similar to the outer-union used for nested request patterns described in Section 3.3.3.2.

We evaluated the above approaches to prefetching by using each approach to fetch a list of queries `TQ(`i`)` for $i = 1 \ldots k$. Each query `TQ(`i`)` fetches the single row from table `T` with primary key equal to i as follows:

```
SELECT x FROM T WHERE pk = :i
```

In addition to the above 4 prefetch approaches, we considered a sequential strategy that executes the k queries one at a time without prefetching. The sequential strategy corresponds to an unoptimized sequence. We also include an IN-list query that exploits the special structure of our query list to fetch all of the k rows with a single scan of `T` driven by an IN-list as follows:

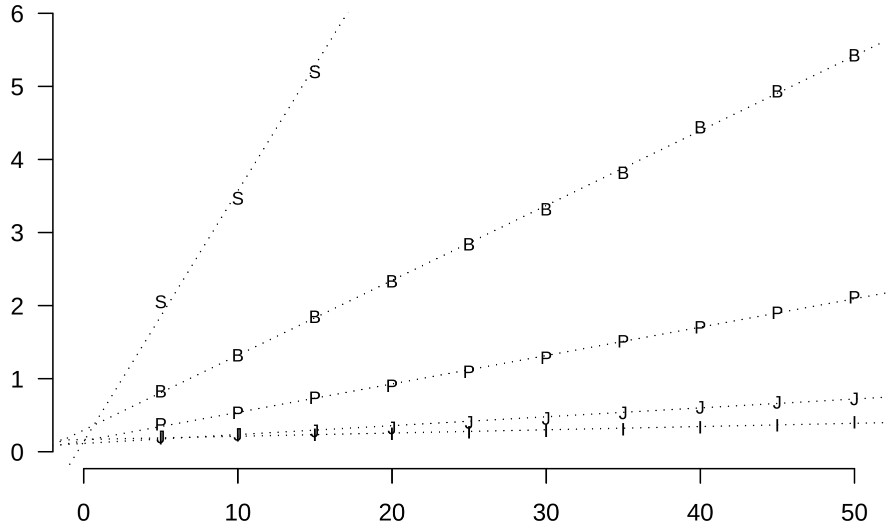


Figure 5.24: Run-time (ms) to execute k sequential queries using a sequential (S), batch request (B), stored procedure (P), join (J), or in-list (I) strategy. Each point is the average of 1000 iterations. Note: the union (U) strategy is not shown as it is very similar to the join approach (J). All results are for configuration LCL.

```
SELECT x FROM T WHERE pk IN (1,2,...,:k)
```

The IN-list query is only possible because of the way we generate our test queries TQ(i), and we do not consider generating these types of prefetches. We include the IN-list query only because it essentially provides a lower bound to the cost of getting our k result rows.

Figure 5.25 shows the code we used to perform timings, Figure 5.24 shows the measured results for the alternatives and Table 5.1 summarizes the results of a linear regression for each alternative. All results were measured on configuration LCL (described in Section 3.6).

The batch request (B) and stored procedure (P) approaches eliminate the per-request communication overhead. Further, their ability to use procedural code would make it convenient to generate a prefetch request that uses parameters from earlier queries as input parameters to later queries. Unfortunately, these approaches do not eliminate the costs associated with crossing the procedural/relational implementation boundary. The run-times for these approaches improve on the sequential times (and the improvement increases if we move to configurations with higher communication latency). However, the run-times are not very close to the lower bound we measured with the IN-list query.

```
function TQ(i) return “SELECT x FROM T WHERE pk=”+i end
function BatchQuery(k)
  sql ← “BEGIN ”
  for i ←1 to k do sql ← sql + TQ(i) + “; ”
  sql ← sql + “ END”
  return sql
end

▷ Open a cursor over Q and fetch all of the rows (timing code not shown)
function FAll(Q)
  open c cursor for Q ; do r ← fetch c until r = Nil; close c
end
function TimeSequential(k)
  for i ←1 to k do FAll( TQ(i) )
end
function TimeBatchRequest(k)
  FAll( BatchQuery(k) )
end
function TimeStoredProcedure(k)
  execute ( “CREATE PROCEDURE P() AS ”+ BatchQuery(k) )
  FAll( “CALL P()” )
  execute ( “DROP PROCEDURE P”)
end
function TimeJoin(k)
  sql ← “SELECT * FROM ( ”+ TQ(i) +“) DT1”
  for i ←2 to k do sql ← sql + “, (”+ TQ(i) +“) DT”+i
  FAll( sql )
end
function TimeUnion(k)
  sql ← TQ(1)
  for i ←2 to k do sql ← sql + “UNION ALL ”+ TQ(i)
  FAll( sql )
end
function TimeIN(k)
  sql ←“SELECT x FROM T WHERE pk IN ( 1 ”
  for i ←2 to k do sql ← sql + “, ”+ i
  sql ← sql + “) ”
  FAll( sql )
end
```

Figure 5.25: Code to evaluate alternative prefetching approaches.

Prefetch Approach	**Time (μs)**	**Sample**
Sequential (S)	139.8 + 343.0k	TQ(1);TQ(2);...;TQ(k)
Batch Request (B)	296.4 + 102.4k	BEGIN TQ(1); TQ(2); ... END
Stored Procedure (P)	150.4 + 38.9k	CALL P()
Join (J)	113.7 + 12.2k	SELECT * FROM TQ(1),TQ(2),...
Union (U)	119.2 + 11.7k	TQ(1) UNION ALL TQ(2) ...
IN-List (I)	167.4 + 4.4k	FAll(SELECT * FROM T WHERE pk IN (1, 2, ..., k)

Table 5.1: Run-time (μs) to execute k queries using different prefetch strategies. The last column gives a sample of the calls during the test.

In contrast, the join (J) and union (U) approaches encode all of the requests as a single query. In this way, all but one instance of the per-request costs are eliminated, including both the communication costs and the costs of crossing the relational barrier. This reduction is substantial, and these approaches are relatively close to the lower bound provided by the IN-list query. The additional cost for the join and union is caused by the cost of beginning an index scan on a quantifier. The IN-list query initializes the search strategy for a single table, while the other two must initialize k quantifiers. Further, conceivably a DBMS could implement rewrite optimizations that transform the query generated by the join or union approach into a form that is as fast as the IN-list query. This would give a SRV-SAVINGS < 1, representing the fact that the server finds a more efficient strategy for the combined query than for the sum of the individual requests. The DBMS products we tested did not benefit from such a transformation, but at least it is possible with such a combination. It seems less likely that such a transformation would be used for the batch (B) or stored procedure (P) approaches.

The batch request (B) and stored procedure (P) approaches seem be convenient targets of rewrites because the procedural capabilities of these approaches allows simple handling of the correlations between earlier parameter values and later input parameters. However, these methods are not supported by all DBMS products. Further, these methods do not (directly) allow for rewrite optimizations that can reduce the total running time to the IN-list lower bound. Finally, the savings provided by these methods is not very close to that achieved by the union and join methods. For these reasons, we consider only the union and join prefetch strategies. With these strategies, we must implement rewrites that supply values for the input parameters of prefetched queries.

5.4.2 Rewriting with Join and Union

When considering join-based rewrites for nested patterns (Section 3.3.2.2) we introduced the lateral derived table construct (and the outer-join variant thereof) for prefetching nested queries using joins. We can also use this construct when generating prefetch queries for sequences of queries. Figure 5.26 shows the result of combining query Q_{d} and Q_{b} of Figure 5.2 using a lateral derived table.

```
SELECT DT0.name, DT1.addr
FROM   ( SELECT name
         FROM   vendor v WHERE v.id = :vendor_info.id ) DT0
       LEFT OUTER LATERAL
       ( SELECT addr
         FROM   shipto s
         WHERE  s.shipid = :vendor_info.id ) DT1
```

Figure 5.26: Q_{db}: queries Q_{d} and Q_{b} combined using a lateral derived table.

The lateral derived table construct implements a form of join. If a query may return more than one row, then the join introduces data redundancy. To avoid this, we use a union-based rewriting similar to the outer union strategy we used for nested queries (Section 3.3.3.2). Query Q_{e} of Figure 5.2 can return more than one row, so we use a union-based rewrite for it. Figure 5.27 shows the union-based query we use to combine queries Q_{d}, Q_{b}, and Q_{e}.

Query Q_{dbe} (Figure 5.27) uses a union with two branches. The first branch, associated with type -1, returns the values of all at-most-one-row queries in the prefetch list (in this case, Q_{d} and Q_{b}). As in the outer-union strategy used for nested patterns, we use a `VALUES` clause to generate a derived table `DT_OneRow` that returns a single row.

The second branch of the union represents query Q_{e}, which is at position 2 in the prefetch list. Therefore, it has a `type` value of 2. This branch contains a derived table `DT2` based on query Q_{e}. The branch can return 0 or more rows. Figure 5.28 shows the result of the combined query Q_{dbe} when invoked with a `vendor_info.id` value of 201 (corresponding to line 4 of Figure 5.3).

5.4.3 Representing Run-Time Behaviour With ACTION Objects

Scalpel uses ACTION objects for batch request patterns in a way that is similar to their use for nested request patterns (Section 3.3.5.1). For each state in the finite state model, the Query Rewriter associates a list of ACTION objects. Figure 5.29 shows how Scalpel generates these lists.

The ACTION objects have an `acttype` field that represents what action should be performed at run-time. There are only two types of action used for batch request patterns: BATCH-JOIN,

```
SELECT DT_UNION.*
FROM   ( SELECT name
         FROM   vendor v WHERE v.id = :vendor_info.id ) DT0
       LEFT OUTER LATERAL
       ( SELECT addr
         FROM   shipto s
         WHERE  s.shipid = :vendor_info.id ) DT1,
       LATERAL
       ( SELECT -1, DT0.name, DT1.addr, NULL, NULL
         FROM   ( VALUES( 1 ) ) DT_OneRow
         UNION ALL
         SELECT 2, NULL, NULL, DT2.*
         FROM   ( SELECT partname, invlevel - onhand AS qty
                  FROM   part p
                  WHERE  p.vendor_id = :vendor_info.id
                         AND p.onhand < p.invlevel ) DT2
       ) DT_UNION( type, c1, c2, c3, c4)
ORDER BY DT_UNION.type, DT_UNION.c3
```

Figure 5.27: Q_{dbe}: queries Q_{d}, Q_{b}, and Q_{e} combined using union and lateral derived tables.

type	c1	c2	c3	c4
-1	'Mary'	'1400 Barrington St.'	NULL	NULL
2	NULL	NULL	'Bell'	3
2	NULL	NULL	'Tire'	6

Figure 5.28: Result of Q_{dbe} for `vendor_info.id=201` (line 4 of Figure 5.3)

which is used to decode the result set for a query that returns at most one row, and BATCH-UNION, used to decode the results of queries that might return more than one row. The GENERATE-PREFETCH-ACTIONS function (Figure 5.29) is called for each state n in the finite-state model. It generates an ACTION object for the query a that is used to enter state n (line 685). This ACTION object will be used to respond to the original OPEN request, and this is accomplished by modifying the associated `submitquery` to be a combined query that returns results for all queries. To begin with, the `submitquery` associated with this action is initialized with a; it will be changed later after calling GENERATE-PREFETCH-QUERY (line 694).

In addition to the ACTION object for query a, GENERATE-PREFETCH-ACTIONS builds an ACTION object for each query q in the prefetch list associated with node n. The `submitquery`

```
structure Action
   acttype = “”            ▷ The type of action to perform
   resultquery = Nil       ▷ The query defining the result set
   submitquery = Nil       ▷ The combined query that will be submitted instead
   ...                     ▷ Additional bookkeeping information is omitted
end

▷ Add a new Action object to state $n$
procedure Append-Action( $n$, resultquery, submitquery)
   if At-Most-One(resultquery) then acttype ← Batch-Join
   else acttype ← Batch-Union
   A ← new Action( acttype, resultquery, submitquery)
   $n$.actions ← [$n$.actions, A]                  ▷ Append the new action
end

▷ Generate a list of Action objects for state $n$
function Generate-Prefetch-Actions( $n$ )
   a ← Get-Incoming($n$)
   prefetch ← Get-Prefetch($n$)

   ▷ Add an action for the query $a$ observed in current Open call
   Append-Action( $n$, $a$, $a$ )
   for i ← 1...prefetch.length do
      ▷ Get correlations predicted to hold for prefetch query i
      corrs ← Correlations( n, i )
      q ← prefetch[i]
      submitquery ← Replace-Parameters( q, corrs )
      Append-Action( $n$, q, submitquery )

   ▷ Generate the combined query, and assign it to the action for $a$
   $n$.actions.submitquery ← Generate-Prefetch-Query( $n$ )
end
```

Figure 5.29: Building Action objects.

of these ACTION objects is initialized with a version of q that has been modified to replace all parameter references with an appropriate value source based on the correlations predicted to hold for the parameter. The modified `submitquery` is generated by REPLACE-PARAMETERS (line 690). This procedure (not shown) chooses a correlation source for each input parameter t of each query q using the set of CORRELATION objects given by CORRELATIONS(n, i). For a CORRELATION object c in CORRELATIONS(n, i), we have the following possibilities:

Constant If c.`type` = 'C', then c represents a constant. We replace parameter t with the literal value.

Input If c.`type` = 'I', then c represents the fact that parameter t has always been equal to the input parameter of a preceding query opened c.`prevcnt` requests earlier. If $i + c.\texttt{prevcnt} \leq 0$, then the value of the input parameter is available when the OPEN(q) request is submitted; we replace parameter t with a parameter filled with the value of the earlier input parameter. Otherwise, the input parameter is from a query in the prefetch list `prefetch`. The value of this input parameter is not available when the OPEN(a) request is submitted, hence we cannot use it directly. However, any such input parameter is in turn correlated to a source that *is* available (the elements of the prefetch list are feasible, as checked by the Pattern Optimizer). This correlation implies there is another correlation source available for parameter t that we can use.

Output If c.`type` = 'O', then c represents the fact that parameter t has always been equal to the output parameter of a preceding query opened c.`prevcnt` requests earlier. As with input correlations, we may have $i + c.\texttt{prevcnt} < 0$, which means the the value of the prior output parameter has been fetched before the OPEN(a) request is submitted; in this case, we replace parameter t with a parameter filled with the value of the earlier output parameter. Alternatively, for the output case we may have the input parameter t of prefetched query `prefetch[i]` being correlated to the output column p of a prior request `prefetch[j]`. In that case, we replace the parameter with the text `'DT'+`j`+'.c'+`p. This replacement text is an outer reference to a derived table representing the prior query.

If there are multiple elements for a single parameter in CORRELATIONS(n, i), we can choose any of the CORRELATION objects (except for the restriction on type 'I' objects noted above); all of the CORRELATION objects had the same current value on every execution of the context. We choose a CORRELATION from the set according to the following ordering. First, any constant (type 'C'). These appear as constant literals in the combined query, requiring no additional work at run-time. Next, we consider any input or output source (type 'I' or 'O') that has a `prevcnt` that draws a value available when OPEN(a) is submitted. These will be treated as input parameters to the query, and Scalpel will supply the input parameter with a value that it has previously observed. Finally, we use any output source (type 'O') that is an output parameter of query `prefetch[j]` for some $0 \leq j < i$. In this case, the replacement text is an outer reference to a previous prefetch query's derived table, and this is effectively a join condition.

The GENERATE-PREFETCH-ACTIONS procedure generates a list of ACTION objects for each state n. It then calls the GENERATE-PREFETCH-QUERY function (line 694) to generate the combined query that will be used at run time. This combined query is assigned to the `submitquery` field of the first ACTION object in the list for state n. Figure 5.30 shows the GENERATE-PREFETCH-QUERY procedure that builds these combined queries. The procedure generates a combined query that encodes the results of all the `submitquery` values associated with the ACTION objects for state n.

5.4.4 Summary of Query Rewriter

After training, Scalpel identifies a list of queries that should be prefetched when we see an OPEN(Q) request in a particular context. We considered four approaches to prefetching based on batch requests, stored procedures, joins, and unions. We found that the procedural approaches are useful for reducing communication latency, but they do not perform as well as the approaches that use a single query, reducing the costs associated with crossing the procedural/relational boundary. As a consequence, we considered only the union and join based approaches.

The GENERATE-PREFETCH-ACTIONS procedure is called for each state n, and builds a list of ACTION objects. Each ACTION object is initialized with a `resultquery` set to the original query text and `submitquery` initialized with a rewritten version of the query text that replaces parameters with appropriate references based on predicted correlations. The `acttype` field of each ACTION object indicates how the query for the action is encoded in the combined result set, with BATCH-JOIN used for join-encoded queries that return at most one row, and BATCH-UNION used for union-encoded queries that may return multiple rows.

After generating the list of ACTION objects for state n, GENERATE-PREFETCH-ACTIONS calls GENERATE-PREFETCH-QUERY to produce a combined query that encodes the results for all of the ACTION `submitquery` fields. The GENERATE-PREFETCH-QUERY procedure combines the queries together using joins (left outer lateral derived tables) for queries that return at most one row and outer unions for queries that may return more than one row. The procedure returns the combined query text. The details of the run-time procedure are described next in Section 5.5.

```
▷ Generate a combined query that encodes the submitquery results for all Action objects
function Generate-Prefetch-Query( n )
  actions ← n.actions
  any_union ← False
  select_onerow ← []
  from ← Nil

  ▷ Generate derived tables for the join-based queries
  for i ← 0...actions.length do
    if actions[i].acttype = Batch-Join then
      if from = Nil then from ← “FROM ( ”
      else from ← from + “ LEFT LATERAL ( ”
      from ← from + actions[i].submitquery + “ ) DT”+ i
      select_onerow ← select_onerow + (“DT”+ i + “.*”)
    else
      any_union ← True

  if not any_union then
    ▷ There are no union-based actions; generate the join based query
    sql ← “SELECT ” + select_onerow
    sql ← sql + from
  else
    ▷ Generate a union-based query, with a branch of type -1 for the join-based actions
    if from = Nil then from ← “FROM ( ”
    else from ← from + “ LEFT LATERAL ( ”
    from ← from + “ SELECT -1, ”+ select_onerow + AddNulls(...)
    for i ← 0...actions.length do
      if actions[i].acttype ≠ Batch-Union then continue
      from ← from + ‘ UNION ALL ’
      from ← from + ‘SELECT ’+i+‘ AS type ’ + AddNulls(...)
      from ← from + ‘, DT’+i+‘.*’ + AddNulls(...)
      from ← from + ‘FROM (’+actions[i].submitquery+’) DT’+i
    from ← from+‘) DT_UNION(type,c1,c2,...)’
    sql ← ‘SELECT DT_UNION.* ’+from+‘ ORDER BY DT_UNION.type’
    for i ← 0...actions.length do
      if actions[i].acttype = Batch-Union then
        sql ← sql+AdjustOrderBy( actions[i].submitquery, ...)
  return sql
end
```

Figure 5.30: Generating a batch prefetch query.

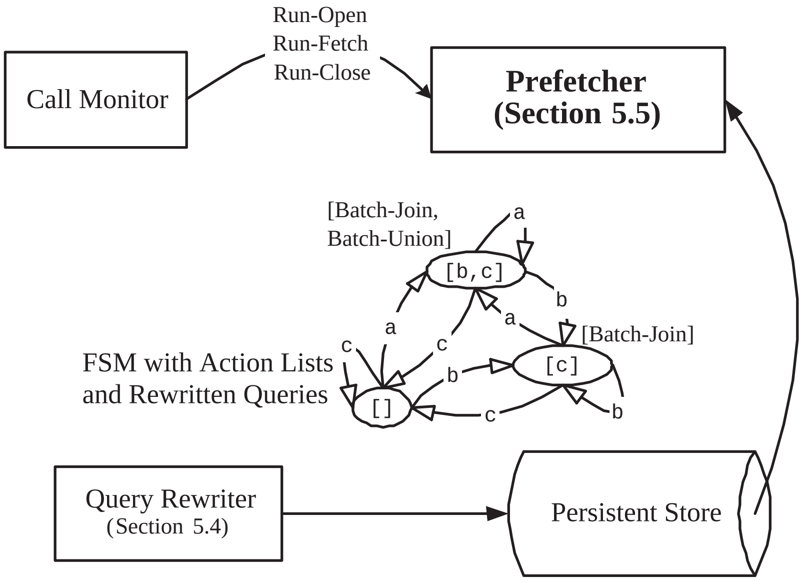


Figure 5.31: Overview of the role of the Prefetcher.

5.5 Prefetcher

Figure 5.31 gives an overview of how the Prefetcher integrates with the rest of Scalpel's batch request processing. After a training period has been completed, the Pattern Optimizer has constructed a finite-state model $M = \langle S, \Sigma, \delta, s_0, p \rangle$. Further, the Query Rewriter has annotated each state n in this model with a list n.actions that describe how queries should be processed at run-time. The first element of this list has a submitquery value that gives the query text that should be submitted to the DBMS when request OPEN(a) is submitted. The Query Rewriter stores this model persistently, and the Prefetcher loads the model when the application first uses Scalpel. The Call Monitor component monitors calls the application makes to OPEN, FETCH, and CLOSE. For each of these, the Call Monitor calls the appropriate RUN- method implemented by the Prefetcher.

Scalpel maintains run-time state, R, that allows it to respond to application requests. The run-time state includes the request model M that was generated by the Pattern Optimizer and Query Rewriter. In addition, R.qcurr records the current model state maintained by tracking requests in the current sequence. Field R.tracker is a VALUETRACKER object that maintains parameter values from the previous S requests, where S is the configured scope length parameter. Finally, R maintains information about results that have already been prefetched: pfCrsr is a cursor over the combined, rewritten query; pfActions is the list of ACTION objects that were associated

```
structure R
  M = request model  ▷ The request model built after training
  qcurr = M.$s_0$          ▷ The current state
  pfCrsr = NIL         ▷ A cursor over the rewritten query
  pfRow = NIL          ▷ The first row of pfCrsr containing all join-encoded results
  pfActions = [ ]      ▷ List of ACTION currently in use
  pfOffset = 0         ▷ Current offset in pfActions
  tracker = new VALUETRACKER      ▷ Maintain parameter values for $S$ preceding requests
end
▷ Process an OPEN($a$, InVals) request using R
function RUN-OPEN( R, $a$, InVals )
  ADDINPUT( R.tracker, InVals ) ▷ Add InVals to tracker
  ▷ The next state is given by following the $a$ edge from R.qcurr
  nextstate ← R.M.$\delta$( R.qcurr, a )

  ▷ Try to find an ACTION object in the list of already prefetched results matching $a$
  action ← NIL
  for i ← R.pfOffset...R.pfActions.length do
    if q = R.pfActions[i].resultquery  then
      action ← R.pfActions[i]
      R.pfOffset ← i+1
  if action $\neq$ NIL $\wedge$ CHECK-CORR-PREDICTIONS( action, InVals )  then
    ▷ If $a$ was prefetched with correctly predicted parameters, use prefetch
    crsr ← USE-PREFETCH( R, $a$, action )
  else if nextstate.actions $\neq \emptyset$
    ▷ Submit combined query that encodes $a$ and all prefetched queries
    R.pfActions ← nextstate.actions
    action ← R.pfActions[0]
    R.pfOffset ← 1
    crsr ←SUBMIT-PREFETCH( R, $a$, action, InVals )
  else
    ▷ Submit the original query to the DBMS unmodified
    CLEAR-PREFETCH(R)
    crsr ←SUBMIT-UNMODIFIED( R, $a$, action, InVals )
  R.qcurr ← nextstate
  return crsr
end
```

Figure 5.32: Pseudo-code for Scalpel batch request Prefetcher.

with the most recent fetch sent to the DBMS, and pfOffset is the index within pfActions indicating the Action object that is expected to match the next Open request.

Figure 5.32 shows how Scalpel uses R to maintain the current state qcurr, submit combined queries to the DBMS, and decode prefetched results. The Run-Open procedure is called by the Call Monitor component of Scalpel when the client application submits Open(a,InVals). First, Run-Open adds the input parameters InVals to the ValueTracker object stored in R.tracker (line 746). The ValueTracker is used to verify that predicted correlations actually hold and that prefetched results match what is requested. Next, Run-Open uses R.M.δ to find the next state that the model will move into after query a (line 748). The nextstate object holds bookkeeping information that may be used to execute this request.

Next, Run-Open looks in the list R.pfActions to see if there is an Action object in the list that matches query a. Even if such an Action object is found, the parameter values used to prefetch the associated results might not match the actual values supplied in the call Open(a, InVals). This would represent a failure of the training period in that a correlation that always held during training is not true at some point during the run-time, but we give correct results in this case by detecting the difference and avoiding the use of inappropriate prefetched values. The Check-Corr-Predictions function (not shown) compares actual parameter values InVals to the predicted correlation sources identified in the Action object.

If an appropriate Action object is found for a and the predicted parameter values match the actual values (InVals), then the needed results have already been prefetched and they are encoded in R.pfCrsr. The Use-Prefetch function (line 758) returns a result set for a that is implemented by decoding the results in pfCrsr using the rules described in the associated Action object. Function Use-Prefetch is shown in Figure 5.33.

If no Action object in R.pfActions matches the current request a, then no prefetched results are available. Function Run-Open will submit a demand fetch to the DBMS. If the nextstate object has a non-empty list of Action objects, the Submit-Prefetch function is called (line 764) to submit a combined query and decode the results for a. Function Submit-Prefetch is shown in Figure 5.33. If, on the other hand no prefetch actions have been specified in nextstate, the Submit-Unmodified function is called (line 768) to submit request a to the DBMS without modification.

After a cursor is obtained, either by decoding prefetched results, submitting and decoding a combined request, or submitting a to the DBMS unmodified, Run-Open changes the current state to nextstate (line 769).

Figure 5.33 outlines how Scalpel implements the Submit-Prefetch and Use-Prefetch functions. The Submit-Prefetch function opens a cursor R.pfCrsr over the rewritten, combined query. Then, it fetches the first row from this cursor, storing the result in R.pfRow. This will either be the only row (if all prefetched queries are at-most-one-row), or it will be the first

```
772 function SUBMIT-PREFETCH( R, $a$, action, InVals )
773   ▷ Submit the rewritten query
774   open R.pfCrsr cursor for action.submitquery:
775   R.pfRow ← fetch R.pfCrsr                ▷ Fetch values for join-encoded queries
776   if R.pfRow = NIL
777     ▷ Query $a$ returned no rows: we cannot use prefetch; return an empty cursor for $a$
778     CLEAR-PREFETCH()
779     return EMPTYCURSOR()
780   else
781     return USE-PREFETCH( R, $a$, action, InVals )
782 end
783
784 function USE-PREFETCH( R, $a$, action )
785   if action.acttype = BATCH-JOIN then
786     ▷ If the results for $a$ were null-supplied, return an empty cursor
787     if NULL-SUPPLIED( R.pfRow, action ) then return EMPTYCURSOR()
788     else return JOINCURSOR( R.pfRow, action )
789   else
790     ▷ Decode an outer-union encoded result set
791     return UNIONCURSOR( R.pfCrsr, action )
792 end
```

Figure 5.33: Pseudo-code for prefetching.

branch of `DT_UNION` with type field of -1. It may be the case that the query a returns no rows for the given input values. In this case (line 778), the prefetched results of the remaining queries are not available: they are eliminated by the join with empty a. In this case, SUBMIT-PREFETCH returns an empty cursor for a and clears the prefetched data structures by a call to CLEAR-PREFETCH (not shown). Otherwise, if the first fetch was successful, SUBMIT-PREFETCH calls USE-PREFETCH to interpret the results for a using the associated ACTION object.

Function USE-PREFETCH is called when we have a request a with the appropriate result set encoded by a previous demand fetch. USE-PREFETCH decodes this result according to the rules in the ACTION object. For an `acttype` of BATCH-JOIN, the result is encoded in the field `R.pfRow`. Either zero or one row is encoded for a, and this is determined using the NULL-SUPPLIED function, which uses non-nullable fields identified in the ACTION object to determine if the result was generated by the left outer lateral construct (and hence a is empty). If so, an empty result set is returned; otherwise, a single-row result set is returned based on the appropriate attributes of `R.pfRow`.

Alternatively, if a has an ACTION object of type BATCH-UNION, then the results for a are encoded as a branch of an outer union in `R.pfCrsr`. The USE-PREFETCH function returns a

UNIONCURSOR object that returns the rows of `R.pfCrsr` that match the `type` field specified in the ACTION object.

When the client application submits a FETCH(`crsr`) request, Scalpel's Call Monitor component calls RUN-FETCH (not shown). If the cursor is a result of a prefetched result set (either JOINCURSOR or UNIONCURSOR) the appropriate row is decoded from the prefetched results. Otherwise, the fetch is submitted unmodified. In any case, the row returned is added to the VALUETRACKER using the ADDOUTPUT function. This allows the `R.tracker` to be used to verify that prefetched queries guessed input parameters correctly.

5.5.1 Summary of Prefetcher

At run-time, structure `R` is used to maintain state that permits prefetching. This state includes M, the finite state model generated during the training period. The model consists of a set of states and transitions between them, and the transitions are optionally annotated with an `Action` object that provides a prefetch that Scalpel should perform when the associated request is submitted when in the originating state. When a prefetch request is submitted, the first row of the combined result is used to satisfy prefetched at-most-one-row queries, and the remaining queries are satisfied by interpreting the branches of the union.

When a request `Q` is submitted, it is compared to the list of prefetched results until a match is found or the end of list is reached. If it is not found, a demand fetch will be sent to the DBMS, along with any configured prefetch requests. If `Q` is found in the list `R.pfQ`, then the results for the query are available. The results are either interpreted from `R.pfRow` (if `Q` is at-most-one-row) or interpreted as the rows of the combined result with a `type` field matching the query offset in `R.pfQ`.

5.6 Experiments

We performed a variety of experiments that were designed to answer two general questions. First, how effective is semantic prefetching at reducing the execution time of query batches when they occur? Second, how significant is the training overhead that is introduced by the Scalpel system? The first question is addressed in Section 5.6.1. The second is addressed in Section 5.6.2.

5.6.1 Effectiveness of Semantic Prefetching

To study the effectiveness of Scalpel's prefetching, we consider a scenario in which Scalpel, during its training phase, has identified a rewrite rule that predicts that a query Q_0 will be followed by a batch of queries Q_1,Q_2,. . .,Q_L. Each query is a simple single-row selection from a table `T`, and each predicted query Q_i is correlated with $Q_{\mathrm{i}-1}$, its immediately predecessor in the query batch.

We wrote a simple driver program, shown in Figure 5.34, to generate a run-time query stream for Scalpel. The program generates the initial query, Q_0, followed by a prefix of the remainder of the batch, and then repeats this N times. The length of the prefix is controlled by a selectivity parameter P. When $P = 1$, each run-time batch is exactly as predicated in Scalpel's rewrite rule. When $P < 1$, some of the queries predicted in Scalpel's rewrite rule are not generated at run-time. The selectivity parameter simulates the effect of predicates in application code which may cause conditional execution of queries in a batch.

```
▷ N is the number of batches to generate.
▷ L is the batch length.
▷ P controls predicate selectivity.
procedure GenQueryBatches( N, L, P )
  for iteration $\leftarrow$ 1 to N do
    generate query $Q_0$
    for i $\leftarrow$ 1 to PL do
      generate query $Q_i$
end
```

Figure 5.34: Code for Generating Query Stream

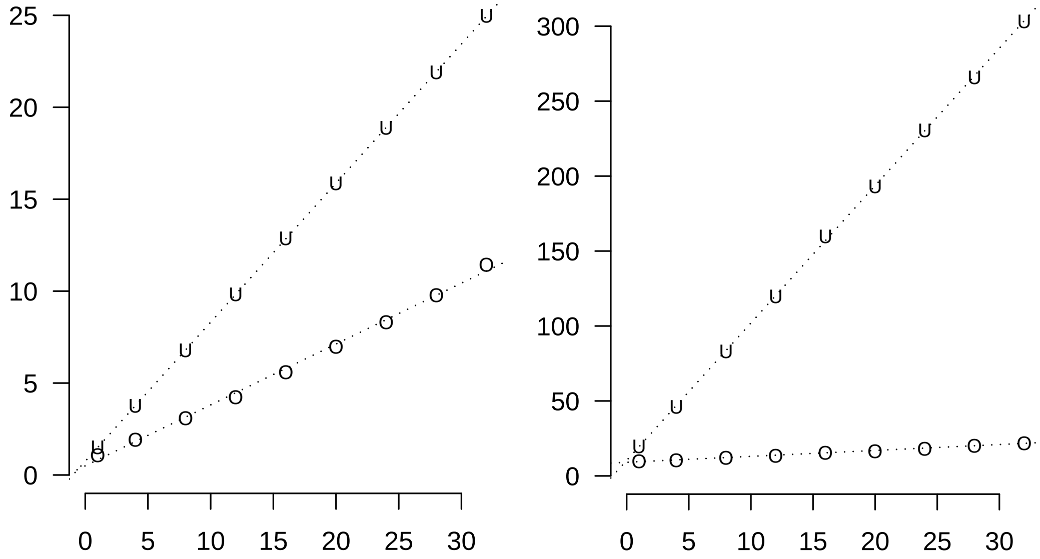
For each experiment, we choose values for N, L, and P and we execute the resulting query stream twice, once without Scalpel and once with it. In the former case, which we call *unoptimized*, each query is passed directly to the database server for execution. In the latter *optimized* case, queries are passed through Scalpel, which applies its rewrite rule to prefetch the results of each query batch.

We studied five different system configurations with varying network latency. These configurations are described in Table 3.5. These configurations used a total of four different client and server computers, the properties of which are described in Table 3.4

The driver program is implemented in Java using JDBC. All results are reported using Java 2 Standard Edition, version 1.5.0. A prototype version of Scalpel was used for the experiments. We experimented with three different commercial database systems behind Scalpel. License restrictions prevent us from identifying them. In each case, the table T used by the queries was populated with 4096 rows and was fully cached. The results for the three database systems were consistent, although with different constants, so we have presented the results for only one system. We use $N = 1024$ for all configurations except WAN, where only 256 iterations were used due to the very high latencies involved.

5.6.1.1 Batch Length

The benefit of prefetching depends on the number of queries that are successfully prefetched and on the latency of the communication. In this experiment, we fixed $P = 1$, varied the batch length L, and measured execution time on each of the five network configurations. Since $P = 1$, this experiment represents the ideal case in which Scalpel has accurately predicted the occurrence of a query batch that does not involve any application predicates.



(a) LAN1

(b) WiFi

Figure 5.35: Mean time (ms) per iteration for unoptimized (U) and optimized (O) strategies with varying batch length L, $P = 1$.

Figure 5.35 shows the results for two of the five system configurations. As the number of successful prefetches increases, the relative benefit increases. One of the database systems that we considered returned an error for a batch length of 45 due to the complexity of the generated prefetch query (which contains a join between 46 quantifiers). Prefetching provides savings even in the low-latency LCL configuration, although much higher gains are available as the interconnect latency grows (Section 3.6 summarizes the configuration LCL and the other configurations we use). Table 5.2 summarizes the results of a linear regression for each of the five configurations. The slope of the prefetch strategy remains relatively constant in the different configurations, while the original strategy incurs the per-request overhead for each additional query.

Setup	Unoptimized	Optimized
LCL	$0.37 + 0.43L$	$0.26 + 0.32L$
LAN1	$0.75 + 0.76L$	$0.50 + 0.33L$
LAN0.1	$1.06 + 1.08L$	$1.28 + 0.35L$
WiFi	$10.35 + 9.16L$	$9.11 + 0.39L$
WAN	$653.27 + 456.58L$	$485.03 + 0.21L$

Table 5.2: Mean time (ms) per iteration vs batch length (L).

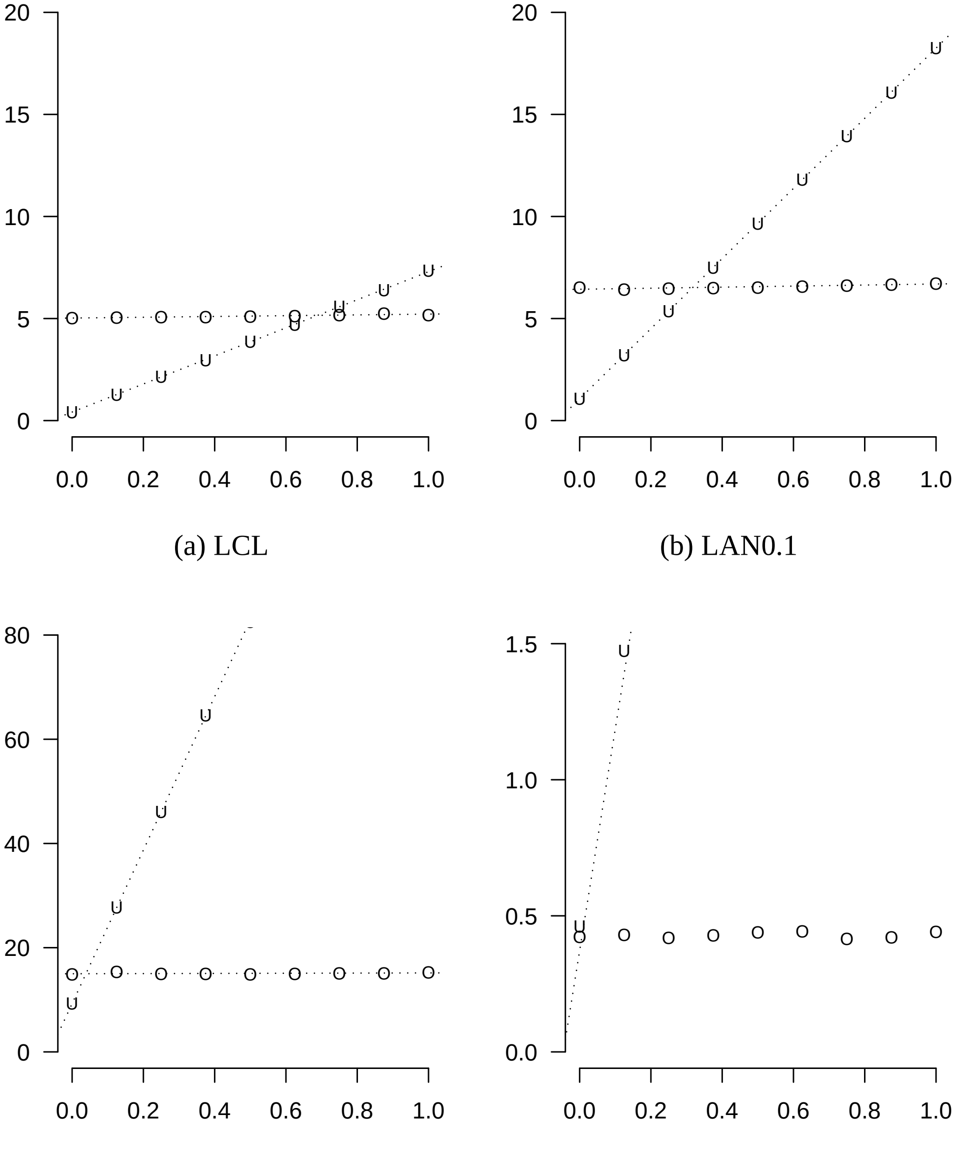
5.6.1.2 Useful Prefetches

For the next experiment, we fixed $L = 16$ and varied the selectivity in the range $0 \leq P \leq 1$. When P is small, only a small portion of the query batch prefetched by Scalpel is actually used by the application. Such overly aggressive prefetching can occur for several reasons. First, Scalpel may be unaware that some of the batch queries are conditional, i.e., there may be prediction error. Alternatively, Scalpel may be aware that part of the batch is conditional and yet decide, on a cost basis, that prefetching is still worthwhile.

Figure 5.36 shows the results of our measurements for four configurations. All points are the average of 1024 iterations. As expected, the unoptimized strategy has a strong dependence on the proportion P of queries that are actually submitted, while the optimized strategy has only a weak dependence on P. The prefetching strategy has a weak dependence on P because the costs associated with tracking the context as queries are opened, detecting if prefetched results are valid, and interpreting the prefetched results are small, but non-zero.

In general, as P increases, there is a threshold above which the optimized (prefetching) strategy becomes worthwhile. A comparison of the configurations in Figure 5.36 shows that this threshold depends on system parameters: in particular, it depends on network latency. In the low-latency LCL configuration, prefetching does not begin to pay off until $P > 0.75$. In the higher latency LAN0.1 configuration, prefetching pays off once $P > 0.4$. Even more aggressive prefetching is called for in higher latency configurations such as WiFi and WAN. These data demonstrate one of the advantages of Scalpel's run-time optimization strategy, since the amount of network latency may not be known at development time, or may vary among instances of the application program.

Table 5.3 summarizes the results of a linear regression for the experiment of Figure 5.36 with all configurations except WAN. For that configuration, the variations caused by sampling error exceed the effect of increasing P, and the slope derived from such an analysis is misleading.



(a) LCL

(b) LAN0.1

(c) WiFi

(d) WAN

Figure 5.36: Mean time per iteration for unoptimized (U) and optimized (O) strategies with $L = 16$ and varying P. All times in milliseconds except (d) in seconds.

However, we can see in Figure 5.36(d) that it is clearly much lower than the slope in the unoptimized case.

Setup	Unoptimized	Optimized
LCL	$0.42 + 6.98P$	$5.02 + 0.19P$
LAN1	$0.75 + 12.10P$	$5.26 + 0.29P$
LAN0.1	$1.07 + 17.18P$	$6.43 + 0.26P$
WiFi	$9.28 + 147.28P$	$15.00 + 0.16P$
WAN	$354.1 + 8227.9P$	omitted

Table 5.3: Mean time (ms) per iteration vs proportion of useful prefetches (P)

5.6.1.3 Breakdown of Costs

Prefetching can act to reduce latency, but it also affects the processing costs at both the client and the server. Prefetching can increase server costs by generating wasted work in cases when all of the prefetched results are not used by the application. Scalpel also introduces additional costs at the client for monitoring and rewriting the query request stream. On the other hand, Scalpel can reduce processing overhead at both the client and the server by reducing the number of OPEN requests that are submitted.

Figure 5.37 shows a breakdown of total query execution times for the case $L = 16$ and $P = 0.75$. In the LCL configuration, total elapsed time is reduced primarily by a reduction in client CPU costs (1.8ms to 0.6ms). Latency is relatively unaffected, but server processing costs are increased (from 3.7ms to 4.5ms). This additional server cost is related to the work needed for the 4 queries that are not fetched in the original strategy, and it is also related to the higher cost of executing the combined query compared to the original single-table queries.

In the other network configurations, the situation with server costs is reversed: the optimized strategies are slightly cheaper than the original. The additional costs that affect the LCL configuration are offset by the savings in interpreting and formatting messages. The unoptimized strategy uses 104 communication packets with 3.4KB, while the optimized strategy uses only 8 packets with 1.4KB. This reduction was not as significant in the efficient shared memory link used in the LCL configuration, but in the TCP/IP configurations it more than offsets the additional costs associated with the 4 wasted prefetched results. These setups also enjoy client-side savings similar to those of the LCL setup, and also exhibit a more marked improvement in latency.

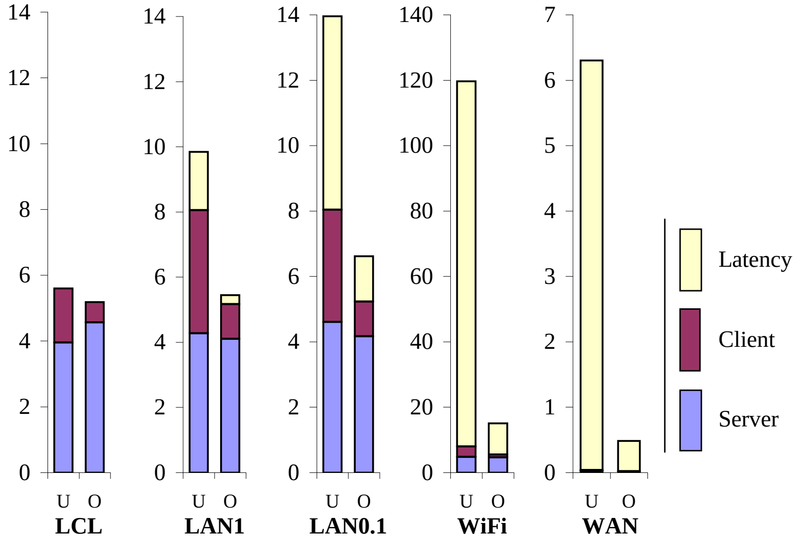
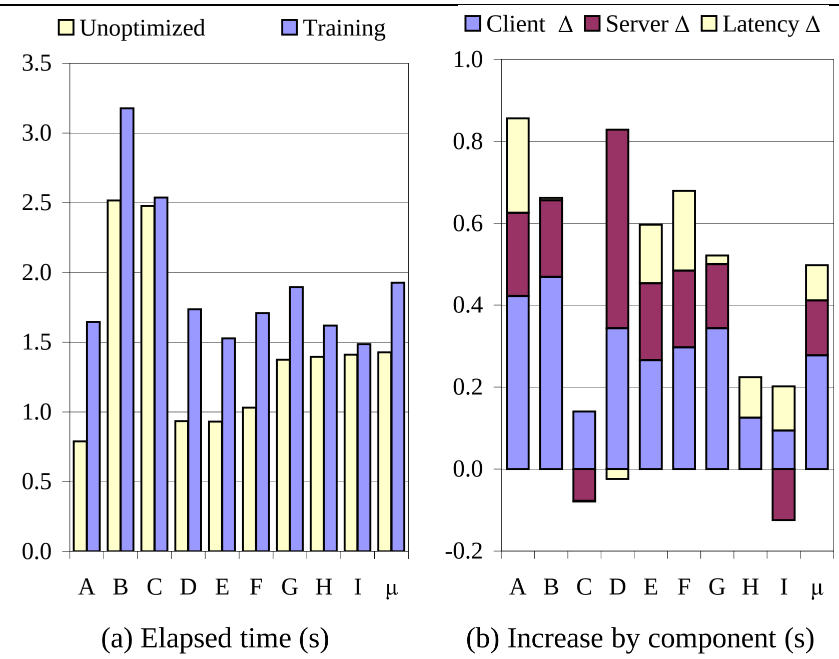


Figure 5.37: Breakdown of execution costs for unoptimized (U) and optimized (O) strategies, with $P = 0.75$ and $L = 16$. Times in ms except WAN in s.

5.6.2 Scalpel Training Overhead

We assessed the overhead of Scalpel's training algorithm using the dbunload case study (described in more detail in Section 8.2). Figure 5.38(a) shows the elapsed time for the unoptimized system (Scalpel not attached) and Scalpel in training mode for 9 databases (labelled A-I) in the LAN1 configuration, with μ showing the average of all 9 executions. On average, training increases run-time by 0.5 seconds, or 35%. Figure 5.38(b) shows the difference in execution time between the unoptimized and training configurations for client CPU, server CPU, and latency. In most of the tests, increases in client CPU cost accounted for most of the increase (on average, 0.28s). However, server costs also increased (average of 0.13s) as did the request latency (0.09s). The increase in client costs is expected due to the additional bookkeeping and optimization steps performed by Scalpel during the training period. Scalpel also issues additional requests to the DBMS to retrieve catalog information (in order to determine if a query will return at most one row). These additional requests are responsible for part of the increase of server and latency components. It is interesting to note that in two instances (C and I), the server costs were lower during the training period. This is due to variance in the server costs and not any action of Scalpel to reduce server costs during training.



(a) Elapsed time (s) (b) Increase by component (s)

Figure 5.38: Training overhead on dbunload case study.

5.7 Summary of Batch Request Patterns

Application programs that we have examined do not generate request streams that are uniformly random. Instead, some sequences of requests are more likely that others. By identifying common sequences, we can choose to prefetch requests that are likely to be submitted, reducing the costs associated with per-request overhead.

The Pattern Detector monitors a client application during a training period, building a trie-based data structure. This trie maintains frequency information that can be used to predict the likelihood of future requests. The trie also maintains sets of correlations that have held each time that a request is observed in a context.

The Pattern Optimizer uses the trie generated during training to select a list of queries to prefetch. It uses the Query Rewriter to generate a combined query that returns the results for the submitted query and any prefetched queries. The combined query is generated using lateral derived tables and unions. The trie generated during training may contain significant redundancy as a result of including every context encountered during training in the trie. The Pattern Optimizer generates M, a redundancy-removed finite state model for trace T that selects states based on the shortest contexts that provide distinct prefetch choices. The model M is annotated with

Action objects that contain the combined query, list of prefetched queries, and predicted correlation source for each input parameter of each prefetched request.

At run-time, Scalpel uses model M to track the current state and submit prefetches. As OPEN requests are intercepted by the Call Monitor, the Prefetcher uses the δ transition function defined by M to move to the next state. If prefetched results are available, the Prefetcher checks that the actual parameter values match the predicted values. If so, the prefetched results are decoded and per-request overhead is eliminated. If a prefetched result is not available, a demand fetch must be submitted. In this case, the Action object is consulted for a combined query that should be submitted instead of the current request.

Section 5.6 provided experimental results that demonstrated the effectiveness of rewriting batch request patterns. Figure 5.24 shows the high overhead associated with sequentially executing requests: over 97% of each of these requests is due to per-request overhead. Prefetching can save some, but not all of this overhead. While prefetching is useful in many cases, we must be careful to consider the probability that requests will be submitted, as shown in Section 5.6.

By using a training period, Scalpel identifies opportunities where prefetching is beneficial. Exploiting these opportunities significantly reduces the costs associated with per-request overhead.

6 Combining Nested and Batch Request Patterns

Chapter 3 described how Scalpel identifies nested patterns of requests during a training period. In these nested patterns, inner queries have parameters that are predicted to be equal to parameters of outer queries (or perhaps a constant). The inner queries are executed up to once per row of their parent query, although local predicates can prevent this. Based on observations during the training period, the Pattern Optimizer uses the Cost Model to select an execution strategy for each pattern of nested queries. The Query Rewriter generates combined queries that are issued instead of the original to retrieve encoded results sets that are predicted to be needed.

Similarly, Chapter 5 described how Scalpel builds a trie-based data structure during a training period to predict batches of requests that occur when a prefix of requests can be used to predict a likely future sequence (with each request in the sequence representing an OPEN, FETCH*, CLOSE sequence). The predicted future queries are also predicted to have actual parameter values that are predicted to be equal to input or output parameters of the preceding queries (or possibly a constant). The future requests are not submitted every time the prefix is observed; instead, Scalpel estimates this probability based on the relative frequency observed during a training period. After the training period is complete, the Pattern Optimizer uses the Cost Model to select a list of queries to prefetch for each prefix of requests. Finally, the Pattern Optimizer generates a redundancy-removed finite state model that contains a set of states with associated queries generated by the Query Rewriter. These alternate queries are issued instead of the original to retrieve encoded results sets that are predicted to be needed in the near future.

The optimizations performed by Chapter 3 and Chapter 5 are complementary, but were developed independently. In this chapter, we give a sketch of how we can simultaneously detect nested and sequential patterns of execution. Treating these two types of pattern in a combined way allows us to extend the set of prefetching opportunities that we can exploit.

6.1 Example Combining Nesting and Batches

Figure 6.1 provides an example of a program that generates both nested request patterns and batches of predictable sequences of requests. This example has been artificially constructed to demonstrate particular features of the combined approach. Figure 6.2 shows the context tree that is created by Scalpel after observing traces generated by F_4.

```
802 function $F_4$(p1)
803   open c1 cursor for g:
804     SELECT g1, g2 FROM G WHERE g3=:p1
805   while r1 $\leftarrow$ fetch c1 do
806     if r1.g1 $\neq$ 0 then
807       fetch row r2 from h:
808         SELECT h1 FROM H WHERE h2=:p1
809       fetch row r3 from i:
810         SELECT i1 FROM i WHERE i2=:r2.h1
811       fetch row r4 from j:
812         SELECT j1 FROM J WHERE j2=:r3.i1 AND j3=:r1.g2
813     else
814       fetch row r5 from i:
815         SELECT i1 FROM i WHERE i2=:r1.g1
816   close c1
817   fetch row r6 from k:
818     SELECT k1 FROM K WHERE k2=:p1
819 end
```

Figure 6.1: An example containing nesting and batches.

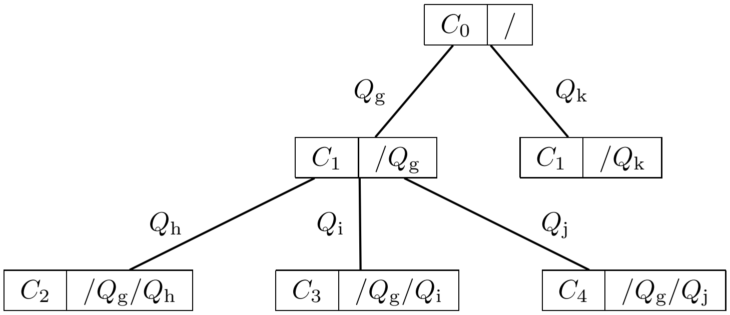
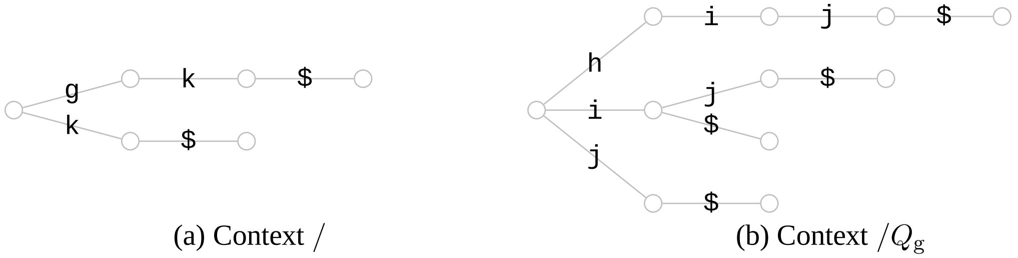


Figure 6.2: Context tree for Figure 6.1.

By itself, the nest-based optimizer might consider prefetching Q_h, but it is unable to prefetch Q_i or Q_j. The query Q_i is used in two distinct ways. The first (line 810) uses an input parameter that depends on the preceding query Q_h. The nesting prefetcher cannot generate a rewritten query that provides the desired result. The second use of Q_i (line 815) is an example that could be handled by the nesting prefetcher. However, the context tree does not provide a way to distinguish this case from the previous case. The correlations for this second use are not monitored separately, leading the nesting prefetcher to miss the correlation in this second case to the output parameter `r1.g1`.

The context tree of Figure 6.2 identifies one opportunity to prefetch a nested pattern. If we consider sequences of patterns, we can find more opportunities. Figure 6.3 shows two suffix tries that we would find after observing the sequence of requests. Figure 6.3(a) shows the trie considering only queries that are immediate children of the root context $/$, while Figure 6.3(b) shows the trie for the queries that are immediate children of context $/Q_{\mathrm{g}}$.



(a) Context $/$ (b) Context $/Q_{\mathrm{g}}$

Figure 6.3: Suffix trie for Figure 6.1; (a) shows the result at the root context $/$, (b) shows the result in context $/Q_{\mathrm{g}}$.

The suffix trie data structure allows us to identify additional prefetch opportunities. It identifies that query Q_{k} always follows Q_{g}, with parameters that can be predicted once we observe Q_{g}. Further, the suffix trie is able to distinguish the two different uses of Q_{i}. When Q_{h} is submitted, it is always followed by Q_{i} then Q_{j}. However, it is not able to prefetch Q_{j} because it depends on a parameter (`r1.g2`) that is drawn from an outer query, a source not considered by the batch prefetcher that we described.

The nesting and batch prefetchers are complementary, and we can achieve more savings if we combine the Pattern Detector for the two types of patterns. The combined Pattern Detector could provide improved structural patterns (distinguishing the two uses of Q_{i}), and also could find more feasible prefetch candidates by considering a broader set of possible correlation sources.

6.2 Combining Context/Suffix Trees

For nested patterns, we represented contexts in the tree as a sequence of queries separated by $/$. Context $/Q_{\mathrm{g}}/Q_{\mathrm{h}}$ is used to represent query Q_{h} opened while Q_{g} is currently open. For batch patterns, we used sequences of characters to represent a context, where we show only the character subscript of each query. In that notation, context hij represents query Q_{j} being submitted after Q_{h} then Q_{i}. We can combine these two notations as follows. For every query Q_a that is currently open, the context contains a substring $a/$. We use sequences of characters to represent requests preceding each of the open queries. With this combination, the context $/kg/hi$ means we have processed a sequence such as the one shown in Figure 6.4. Note that at step 6, the context would be $/kg/h/$ as Q_{h} is still open.

```
 1  $c_k \leftarrow \textsc{Open}(Q_{\mathrm{k}})$
 2  $\textsc{Fetch}(c_k)$
 3  $\textsc{Close}(c_k)$
 4  $c_g \leftarrow \textsc{Open}(Q_{\mathrm{g}})$
 5  $c_h \leftarrow \textsc{Open}(Q_{\mathrm{h}})$
 6  $\textsc{Fetch}(c_h)$
 7  $\textsc{Close}(c_h)$
 8  $c_i \leftarrow \textsc{Open}(Q_{\mathrm{i}})$
 9  $\textsc{Fetch}(c_i)$
10  $\textsc{Close}(c_i)$
```

Figure 6.4: Example trace generating context $/kg/hi$.

The trace in Figure 6.4 is consistent with the context $/kg/hi$. We can also consider generalizations that use fewer preceding queries to define the context. With this approach, the trace of Figure 6.4 is associated with the following contexts: $\{/kg/hi, /kg/i, /g/hi, /g/i\}$. These are generalizations of the form given by the suffix tries we used for batch request patterns. The difference is that the generalization is occurring at two levels: the outermost level, and the level nested inside an open query g.

We can combine nesting and batch detection by using a generalization of the context tree data structure. We build multiple suffix tries, each associated with a nesting level. The original context tree moved to a single child node when a query was opened. With this proposal, we would move to a set of children, with a different child associated with each of the nodes on the update path of the suffix trie from the longest suffix to the root.

We would maintain a set of active contexts. These are contexts that match the current request sequence. They may be nodes in the same suffix trie, for example $/g/hi$ and $/g/i$, or they may be nodes from separate tries rooted at different parent nodes, such as $/kg/h$ and $/g/h$. When we observe a request $\textsc{Open}(Q_{\mathrm{a}})$, we would push the current set C_1 of active contexts onto a stack and form a new set C_2. The new set $C_2 = \{wa/ \mid w \in C_1\}$ is formed by appending the characters $a/$ to each element w in set C_1. When we observe the associated $\textsc{Close}()$ request, we would pop the set of active contexts, returning to C_1. Then, we would follow the approach of Figure 5.6 to generate all following nodes based on a walk along the frontier.

Figure 6.5 shows the set of active contexts after processing a sequence of requests that might be generated by the code in Figure 6.1.

N	**Request**	**Active Contexts After Request**
1	$c_g \leftarrow \textsc{Open}(Q_\mathrm{g})$	$\{/g/\}$
2	$c_h \leftarrow \textsc{Open}(Q_\mathrm{h})$	$\{/g/h/\}$
3	$\textsc{Close}(c_h)$	$\{/g/h, /g/\}$
4	$c_i \leftarrow \textsc{Open}(Q_\mathrm{i})$	$\{/g/hi/, /g/i/\}$
5	$\textsc{Close}(c_i)$	$\{/g/hi, /g/i, /g/\}$
6	$c_j \leftarrow \textsc{Open}(Q_\mathrm{j})$	$\{/g/hij/, /g/ij/, /g/j/\}$
7	$\textsc{Close}(c_j)$	$\{/g/hij, /g/ij, /g/j, /g/\}$
8	$\textsc{Close}(c_g)$	$\{/g, /\}$
9	$c_k \leftarrow \textsc{Open}(Q_\mathrm{k})$	$\{/gk/, /k/\}$
10	$\textsc{Close}(c_k)$	$\{/gk, /k, /\}$
11	$c_g \leftarrow \textsc{Open}(Q_\mathrm{g})$	$\{/gkg/, /kg/, /g/\}$
12	$c_i \leftarrow \textsc{Open}(Q_\mathrm{i})$	$\{/gkg/i/, /kg/i/, /g/i/\}$
13	$\textsc{Close}(c_i)$	$\{/gkg/i, /gkg/, /kg/i, /kg/, /g/i, /g/\}$
14	$\textsc{Close}(c_g)$	$\{/gkg, /kg, /g, /\}$

Figure 6.5: Example trace generating context $/kg/h$.

In addition to the structure of the combined context tree/suffix trie, we must consider how to detect correlations in this combined scheme. In the nesting approach, we maintained a scope for each context. This scope provided the set of correlation sources that we considered might predict parameters of the nested queries. The batch detection used a dictionary over a sliding window of S previous requests. In the combined scheme, we could implement a dictionary that maintains the values of S previous requests at each level of nesting.

6.3 Current Implementation

The sketch in Section 6.2 described one way that we could integrate detection of nested and batch request patterns. The integration is quite tight, allowing correlation detection to work across nesting and sequence boundaries. Further, the information maintained during the training period permits rewrites that simultaneously combine nests and batches. However, the time and space requirements of the sketched approach are rather alarming. It may be possible that we can achieve much of the benefit of the complete integration in a reasonable space limit if we can exploit the significant redundancy present in the suffix tries. We leave this as an area of future consideration.

At present, we have implemented a relatively modest approach to integrating nesting and batch request pattern detection. Each of these is implemented separately, with its own correlation detection. The batch Pattern Detector operates only on queries opened at the top level of nesting. The Pattern Detector, Pattern Optimizer, Query Rewriter, and Prefetcher operate in parallel for ach type of pattern. If the nested request optimizations choose an alternate strategy for a query, that query is not considered by the batch request optimizer.

The results that we presented in Chapter 3 used only the nested request components; likewise, results in Chapter 5 used only batch request components. The results that we present for case studies in Chapter 8 use the combined approach described here.

Even this loose integration of nesting and batch detection is reasonably effective. The request patterns used for dbunload are moderately (Section 8.2), but it does not appear it would benefit much further from a more comprehensive integration of nesting and batch optimization. The SQL-Ledger system has an even simpler structure within its nested requests, with at most two distinct queries opened inside an outer query. The full power of integrating batch and nesting detection would not help with predicting what queries will be executed next. However, the SQL-Ledger case study does contain correlations between an inner query and a preceding query. At present, Scalpel's loose integration does not detect this correlation, and it is unable to prefetch the associated query.

6.4 Summary of Combining Nested and Batch Request Patterns

There are application request patterns where a pattern detector can make more effective decisions if it considers both nested and batch patterns. The combination can provide better structural predictions, for example by distinguishing special cases by considering both preceding queries and nesting. Further, this approach extends the set of possible correlations that we can consider. We can consider a parameter of request Q_{a} to be correlated to to requests that preceded Q_{a} at the same nesting level, to any enclosing request, and to any request preceding enclosing requests.

At present, our prototype implementation uses a loose integration of the optimizations for nested request patterns and batch request patterns. When the two types of optimizations conflict, the actions selected by the nested request optimizer are use in preference. Our experience with case studies suggests that the loose integration achieves most of the benefit that a full integration would give for the practical systems we examined. It seems that shared correlation prediction would be the biggest improvement for these systems.

7 Prefetch Consistency

The Scalpel system prefetches results before the client application submits the associated request. This can lead to a data consistency problem, where prefetched data do not contain updates that would have been observed if the data were not prefetched. There are two possibilities for these updates: either they are performed by the current transaction (described in Section 7.1), or they are performed by another transaction (Section 7.2).

7.1 Updates by the Same Transaction

When a transaction performs an update, it expects the results of that update to be reflected in the results of future queries. If Scalpel has prefetched the results of anticipated requests, these prefetched results will not reflect the modifications.

In principle, it is possible that Scalpel could alter the update statement sent to the server so that it returns enough information to modify Scalpel's prefetched results to properly reflect the changes. Alternatively, enough information could be returned to identify prefetched results that are now stale, and Scalpel could merely send demand fetches for these stale results instead. The altered update statement could perform a join between the updated rows and the combined queries already prefetched by Scalpel. The updated row set is identified by the update statement and any modifications performed by triggers executed in response to the update.

This approach expends significant effort in implementation complexity and more expensive updates in order to precisely determine which prefetched results are still valid. Another approach could be based on a static analysis of the prefetch queries used by Scalpel and update statements submitted by the same transaction. In some cases, the updates cannot possibly affect prefetched results. For example, an update could apply to tables not referenced by the prefetch queries. If, further, no triggers fired by the update could affect these results, then the update statement is not a *significant update* to the prefetched queries. If we statically analyze an update statement and find it could not possibly affect our prefetched results, we could retain them; otherwise, we could discard them and answer future queries that would have been satisfied by prefetched results with results from a demand fetch. In this case, the work of prefetching the discarded results is wasted, but no consistency problems are introduced.

At present, Scalpel implements a simple policy whereby it assumes that *any* update by a transaction is significant to prefetched results. Scalpel monitors all EXECUTE requests during training and at run-time. During training, Scalpel recognizes when an update could invalidate prefetched data, and it chooses to avoid the prefetch in that case (as the work might be wasted). If an update is encountered at run time that was not anticipated based on the training period, the prefetched results are not used to answer further OPEN requests. While this represents wasted work, it does not affect consistency.

This simple policy of our current prototype is safe, if possibly sub-optimal, for OPEN requests submitted after an update from the same transaction. However, there is another possible source of data inconsistency. If there is a cursor already open over prefetched results, we should be concerned that the rows fetched from the prefetch cursor match the semantics that were given without prefetching.

SQL/99 [13] defines *cursor sensitivity* as follows. If a change to SQL-data would have caused different results to be returned by a cursor had the cursor been opened after the change, then the change is said to be a *significant* to the cursor. If the effects of a change are observed by a cursor, the change is said to be a *visible change*. SQL/99 defines the following sensitivity values.

SENSITIVE If a cursor is declared as SENSITIVE, then all significant changes are visible.

INSENSITIVE If a cursor is declared as INSENSITIVE, then no significant changes are visible.

ASENSITIVE If a cursor is declared as ASENSITIVE, then the visibility of significant changes is implementation dependent.

The default sensitivity is ASENSITIVE, which allows implementations to provide the most efficient sensitivity.

In principle, Scalpel could use SENSITIVE cursors for its prefetched results. With the exception of the client hash join strategy, this would ensure that significant changes are visible, and that cursors that are open and decoding prefetched data would give SENSITIVE semantics. However, typical implementations of SENSITIVE cursors are inefficient, requiring a demand fetch to the server for each row. For now, Scalpel uses only ASENSITIVE sensitivity. If a cursor is opened with either SENSITIVE or INSENSITIVE sensitivity, Scalpel submits it to the server for processing (INSENSITIVE cursors could be supported by using INSENSITIVE for Scalpel's prefetch cursors, but that could require an INSENSITIVE cursor over the larger combined query results, something which we avoid in the current prototype).

In summary, Scalpel guarantees prefetch consistency with respect to monitored updates done by the same transaction. Prefetched results are not used for future OPEN requests after an EXECUTE as it might possibly have been significant to the prefetched results. Scalpel only uses prefetching for ASENSITIVE cursor types, which allows it to deliver results matching the requested semantic using prefetched data.

7.2 Updates by Other Transactions

Section 7.1 discussed prefetch consistency only with respect to modifications by the same transaction; in addition, we must consider the updates that may be performed concurrently by other transactions. While Scalpel can detect changes from the monitored connection, it cannot observe changes made by other connections connected to the same DBMS.

If prefetched results are not used across transaction boundaries, then prefetch consistency is provided by the ACID properties of the DBMS (at least in theory). If prefetched results are used across transaction boundaries, the ACID properties do not hold. We could consider approaches similar to those that we considered in Section 7.1; for example, we could send additional requests at the beginning of a new transaction to determine which prefetched results are (possibly) invalid, then either update them or invalidate them. For example, we could use WITH HOLD cursors [13] for Scalpel's prefetched results, which would allow us to hold the cursor open across transaction boundaries (for a single connection). Alternatively, we could follow an approach similar to that proposed by Guo et al. [87], allowing the client application to specify whether it is willing to use stale data prefetched from previous transactions.

At present, Scalpel does not use prefetched results across transaction boundaries. We have found applications that would benefit from this type of prefetching (for example the SQL-Ledger system, described in Section 8.3). The prefetching is useful when a high-level user operation is accomplished by more than one sub-transactions. However, the benefit in these cases results from batch request patterns, not from nested request patterns. Hence, we would save at most one instance of the per-request overhead U_0. It is possible a nested request pattern could be implemented across transaction boundaries (using a WITH HOLD specification for the outer cursor). However, we have not found instances of this in the systems we investigated. Therefore, it appears we do not lose much in practice by restricting ourselves to prefetching within transaction boundaries.

7.2.1 Weak Isolation

The ACID properties of DBMSs suggest that we need only prove consistency for a serial execution of individual transactions, as we have shown in Section 7.1. However, full ACID properties are not always guaranteed in practice. In some cases, client applications request a weak isolation level that does not provide serializable semantics. Gray et al. [84] defined a number of degrees of consistency that are less than serializable. By lowering consistency, applications can improve concurrent performance; presumably, semantic issues are controlled by additional application logic. The SQL standard [13] defined four levels of isolation, with the intention that these would provide an implementation-independent definition of degrees of consistency, without referring to a particular implementation, such as locking. Berenson et al. [18] pointed out that the definitions used in the SQL standard [13] are ambiguous; further, they fail to capture essential elements of the original isolation definitions used by Gray et al. [84]. Berenson et al. proposed a variant of the standard SQL definitions, but noted that these alternate definitions were essentially a disguised form of locking. Adya, Liskov and O'Neil [4, 5] provided a more generalized definition of isolation levels that can be used for optimistic protocols as well as for locking.

The anomalies related to relaxed concurrency are typically exacerbated by prefetching. Not only can the original anomalies occur (such as dirty reads, non-repeatable reads, and phantoms), but also new temporal anomalies can occur. A read request r_1 may observe the effects of an update from another transaction, while a later request r_2 that is satisfied from prefetched data may not observe the request. This situation only occurs when a demand fetch is sent to the DBMS after Scalpel has prefetched some results but before all of the prefetched results have been consumed. One way to avoid introducing new temporal anomalies would be to detect this situation. When any demand fetch is sent to the DBMS, Scalpel could discard any prefetched results.

While this approach has the theoretical benefit of avoiding introducing new temporal anomalies, it does so at the expense of limiting prefetch opportunities and wasting work. It seems likely that application developers that select relaxed isolation levels for efficiency would not prefer this approach. Instead, a better approach would be to allow application developers to control the recency of the data they receive. One approach is proposed by Guo et al. [87], whereby application developers explicitly encode currency and consistency requirements.

Another approach is that of the snapshot isolation level. With snapshot isolation, defined by Berenson et al., each transaction reads data that was committed as of the transaction start (additionally, the transactions own updates are reflected). Scalpel works very well in this environment, as shown in Section 7.1. Scalpel does not introduce any new temporal anomalies in this situation, and the anomalies permitted by snapshot isolation are not modified by Scalpel.

At present, Scalpel makes no attempt to avoid introducing new temporal anomalies at relaxed isolation levels. If a transaction executes with serializable isolation, Scalpel produces consistent and current data (with the cautions described in Section 7.1 used for reflecting the transaction's own updates). If the weaker snapshot isolation is used, Scalpel does not introduce new anomalies. Otherwise, Scalpel introduces temporal anomalies that are not present in the original system, and which are not described by the original isolation level selected by the transaction. An application developer must be cautious when using Scalpel in this environment; it may be that the introduced anomalies are acceptable in the quest for best possible performance, but it may be the case that these must either be controlled through application logic or use of serializable isolation. Finally, we note that the issues related to prefetching data before they are used are similar to the concerns of caching systems such as the MTCache described by Guo et al. [87]. In fact, prefetching and caching form very good complements; developments in caching semantics are useful in describing the effects of prefetching, and prefetching can be used by a cache to choose a victim to evict. It is an interesting topic for future study to consider how prefetching and caching can be integrated in a way that provides efficient results (avoiding the dangers of replication described by Gray et al. [83]), while providing semantics that match the application developer's needs.

8 Case Studies

We applied Scalpel to real-world systems to evaluate its effectiveness and how its features interact with practical concerns. In Section 1.3, we described a set of client applications that we have investigated to assess the possible benefits of Scalpel. We manually inspected these systems, and found opportunities for optimizations based on nested request patterns and also batch request patterns. We found batch opportunities in all of the systems we considered. While we did not find nested request patterns in all of these systems, it did appear in several of the systems we considered.

In this chapter, we present detailed results for two of these systems as case studies of using Scalpel in practical systems. Section 8.1 describes an issue that we found in both case studies—the problem of how transaction boundaries are handled during training and at run-time. Section 8.2 describes our investigation of the dbunload program. This program contains significant nesting that is optimized effectively using Scalpel's nesting rewrites. Batch request patterns also offer an improvement, although their role is limited in many configurations by the overwhelming number of requests associated with the nested patterns. Section 8.3 describes our work with the SQL-Ledger system. SQL-Ledger is a web-based double-entry accounting system. The SQL-Ledger system also provides several opportunities for optimizing nested request patterns.

8.1 Transaction Boundaries

One issue that we faced in both case studies is how Scalpel should deal with transaction boundaries. The dbunload program uses a single transaction to generate the text to re-create a schema. When we are training Scalpel, we would like to use a long trace that is typical of the usage patterns that we expect at run-time. In order to build a sufficiently long and representative trace with dbunload, we need a way to uses several transactions in a single training period. We accomplish this with Scalpel by using special out-of-band queries. When a transaction ends, we ensure that all open queries are closed. At present, Scalpel does not support `WITH HOLD` cursors, which can be held open across transaction boundaries. Scalpel also uses an out-of-band character in the suffix trie (or path compressed suffix trie) data structure, then moves the current update point of the trie to the root. This out-of-band character allows multiple sequences to be combined into one suffix trie. Gusfield [88] provides more details on the approach.

The out-of-band queries allow Scalpel to build a single suffix trie that summarizes multiple traces, and this allows dbunload to be executed in several different configurations in order to provide a realistic training period. A related concern is that Scalpel might try to prefetch results in one transaction then use them in another. As discussed in Chapter 7, such an approach leads to serialization anomalies. In particular, we found that the SQL-Ledger system tends to use relatively short transactions. A single user activity might lead to more than one transaction being issued, and Scalpel's batch request optimization could reduce latency by prefetching across session boundaries. At present, we do not permit such prefetches. The out-of-band queries in the suffix trie are used to prevent rewrites that cross transaction boundaries.

8.2 Case 1: dbunload

The first case we considered is the dbunload program. dbunload is provided with Adaptive Server Anywhere (ASA), a full-featured DBMS produced by iAnywhere Solutions, a Sybase company. The dbunload program generates a text-based 'unload' of a database, suitable for recreating both the schema and instance data. When unloading data, the majority of the time is spent unloading the instance data: this involves fetching row data from the DBMS, formatting it as text, and outputting it. In this usage pattern, the cost of outputting the database schema is usually a negligible portion of the elapsed time. The dbunload utility is also used extensively to unload only the schema for a database. For example, customers can use this approach to maintain revision control of the database schema. A variation on this mode outputs the schema for only one table or a small set of tables. Scalpel cannot improve the performance of the (coarse-grained) operations used to unload instance data, so we focused on schema unloading operations.

8.2.1 Characterization of the Program

The dbunload program works with database instances generated by several different versions of ASA, including variations on different upgrade paths that may have been applied to upgrade older database versions. These different versions have varying features, and some common features are implemented differently in different versions. For example, two database versions store column constraints in slightly different formats. The dbunload program issues version detection queries to determine the version level of the database instance to be unloaded. The results of the version detection are stored in global configuration variables.

The dbunload program is over 11 years old, and is implemented in C code. All queries are submitted using a cursor abstraction layer implemented in ESQL. This abstraction layer provides a simple OPEN,FETCH,CLOSE style interface. The OPEN interface takes as parameters a unique cursor name and a SQL string. Any query input parameters are encoded in the SQL text using string substitution. All fields returned from the cursor interface are returned as text strings.

The query for each cursor is constructed dynamically using string formatting commands. Query parameters are included in the query text using string substitution. Further to parameter substitution, queries are constructed based on dynamic configuration parameters that depend on the results of the version detection queries and invocation parameters of dbunload. An appropriate variant of each query is generated by textually substituting appropriate variants into the query to be submitted. For example, if dbunload is used for a database instance without support for named constraints, the value NULL is textually substituted in place of the field name that stores the constraint name. In order to handle some complex versioning issues, global parameters are used in some locations to generate subselects and additional joins in order to retrieve necessary information. In general, textual substitution is used to generate arbitrary query constructs.

In addition to substitution based on global configuration parameters, some queries are submitted by a function that is called in multiple contexts, each requiring a different query variant. For example, the code to unload a constraint for a column, table, foreign key, primary key, or declared unique constraint is shared in the `GetConstraints()` function. This function is called from four separate call sites within dbunload, and each call site generates a different variant of the query.

The code of dbunload is arranged into functions and modules following good C coding style. There are 46 functions that open cursors, with 35 opening only one, 7 opening 2, 1 opening 3, and 2 functions opening 5 cursors. In total, there are 63 different cursor open call sites within dbunload. The presence of predicates within these functions means that these cursors are not opened every time there is an opportunity to do so.

Functions that open cursors use a loop over the results if more than one row can be returned. The body of these loops generates the text needed to re-create the associated schema object. In some cases, the schema object has sub-structure nested within it. For example, a table schema object generates the text for its and in these cases, dbunload typically calls other functions that open cursors and output the text representation of the nested objects.

8.2.1.1 A Java Version of dbunload

The Scalpel prototype that we have implemented is written in Java with a JDBC interface. In order to assess the benefits of using our proposed optimizations, we implemented a Java version of the dbunload tool by transliterating the source from C/ESQL to Java/JDBC. The primary changes of the source code were the following:

1. Use Java string concatenation (+) for string literals instead of C's style (adjacent strings).

2. Use Java string comparisons instead of `strcmp`, etc.

3. Use exception handling instead of C-style return code checking.

4. Replace uses of C preprocessor by either a) manually substitute one variant of a macro or b) use Java `final static boolean` fields.

5. Rename cursor names that were ambiguous.

6. Replace uses of the ESQL cursor interface with calls to a Java cursor abstraction layer based on JDBC.

The Java implementation of dbunload is about 4400 lines of code, and it submits the same queries as the original C version. Due to the different implementation, performance results cannot be directly applied to improvements that could be made to the C version, but they should be considered indicative of what is possible.

Figures 8.1 and 8.1 show simplified pseudo-code for the top level requests issued by the dbunload tool. In this simplified pseudo-code, we use the notation `Q(name[,parms...)` to represent a query named `name` being opened with an optional list of parameters. In cases that the query returns a single row or simple list of strings, we use it notationally as a function call (for example, on line 820). In other cases, the query may return more than one row and the dbunload program processes the results using a loop. We show this as `Q(name[,parms...)` {`...`} (for example, line 858).

The main-line for the program is `Do-Unload` (line 888). First, it calls `Get-DB-Version` (line 822) to determine characteristics of the database instance being considered. This version information reflects whether the database contains various catalog tables and columns that were added over the years in different versions of ASA. The results are stored in global variables (`V1 ...V27`). Next, the main-line calls `Load-Exclude-Tables` (line 851) to load a list of system-defined schema objects that should not be output by dbunload. The names and types of these excluded schema objects are stored in a table (`EXCLUDEDOBJECT`) that was added in a version of ASA. The `Load-Exclude-Tables` procedure first issues a query to see if this database instance has the `EXCLUDEDOBJECT` table; if so, it issues a second query to see if there are any rows in the table. If both of these conditions are true, then the procedure loads in-memory arrays of objects to be excluded using two distinct queries (`C_EXCLUDE` and `C_EXCLUDEB`) parameterized with the type of object to be stored in the appropriate array.

In combination, `Get-DB-Version` and `Load-Exclude-Tables` submit up to 34 OPEN requests using 8 distinct queries. The presence of local predicates means that not all of these queries are always executed. However, the predicates evaluate to false only for older versions of databases. If we consider a customer that does a regular schema unload of a (current) database instance, then all of these queries will be executed in sequential order. If Scalpel could predict this case, it could avoid the costs associated with fine-grained access.

After initializing the version global variables and loading arrays of schema objects, the `Do-Unload` mainline issues queries to retrieve the name of the `DBO` owner for the database (a

```
procedure Table-Exists(own,tbl) return Q( C_TABLEEXISTS,own,tbl ) end
procedure Col-Exists(tbl,col) return Q( C_COLEXISTS,tbl,col ) end
procedure Get-DB-Version()
  V1 ← Table-Exists( "SYS", "SYSGROUP" )
  V2 ← Col-Exists( "SYSCOLUMN", "scale" )
  V3 ← Col-Exists( "SYSCOLUMN", "column_type" )
  V4 ← Col-Exists( "SYSTABLEPERM", "referenceauth" )
  V5 ← Table-Exists( "SYS", "SYSINFO" )
  V6 ← Table-Exists( "SYS", "SYSATTRIBUTE" )
  V7 ← Table-Exists( "SYS", "SYSPROCEDURE" )
  V8 ← Table-Exists( "SYS", "SYSSYNC" )
  V9 ← Col-Exists( "SYSINDEX", "hash_limit" )
  V10 ← Table-Exists( "SYS", "SYSREMOTETYPE" )
  V11 ← Table-Exists( "SYS", "SYSREMOTEOPTIONTYPE" )
  V12 ← Table-Exists( "dbo", "ml_property" )
  if V10  then V13 ← PubExists()
  V14 ← Col-Exists( "SYSCOLUMN", "check" )
  V15 ← Q( C_GETDBPROPERTY, "NamedConstraints" )
  V16 ← Table-Exists( "SYS", "SYSUSERTYPE" )
  V17 ← TabCheck( encrypted pwd )
  V18 ← Col-Exists( "SYSCOLPERM", "privilege_type" )
  V19 ← Col-Exists( "SYSARTICLE", "query" )
  V20 ← Table-Exists( "SYS", "SYSJAVACLASS" )
  if V20  then V21 ← Q( C_JAVAEXISTS )
  V22 ← Table-Exists( "rs_systabgroup", "rs_lastcommit" )
  if V19  then V23 ←Q( C_GETDBPROPERTY, "FileVersion" )
  V24 ← Table-Exists( "SYS", "SYSCAPABILITY" )
  V25 ← Table-Exists( "SYS", "SYSEVENT" )
  V26 ← Table-Exists( "SYS", "SYSWEBSERVICE" )
  V27 ← Col-Exists( "SYSTABLE", "source" )
end
procedure Load-Exclude-Tables()
  if Q( C_USEEXCOBJTBLA ) and Q(C_USEEXCOBJTBLB) then
    Q( C_EXCLUDE,'P' ) ; Q( C_EXCLUDE,'V' )
    Q( C_EXCLUDEB,'U', 'E' ) ; Q( C_EXCLUDE,'E' )
    Q( C_EXCLUDE,'T' )
end
procedure Do-Users()
  Q( C_DBSPACES ) {...} ; Q( C_USERS,V10,V1 ) {...}
end
procedure Do-Logins() Q( C_SYSLOGIN ) {...} end
procedure Do-Servers()
  Q( C_SERVERS ) {...} ; Q( C_CAPABILITIES ) {...} ; Q( C_EXTLOGINS ) {...}
end
```

Figure 8.1: Pseudo-code for dbunload top-level requests (*continued.*)

```
procedure Do-User-Classes() Q( C.C_USERCLASSES ) {...} end
procedure Do-User-Tables() Q( C_TABLES,V10,V24,V9,V6 ) {...} end
procedure Do-User-Views() Q( C_VIEWS,V1,V27 ) {...} end
procedure Do-Events() Q( C_EVENTS,V27 ) {...} end
procedure Do-Services() Q( C_SERVICES ) {...} end
procedure Do-Unload-Publications()
  if V8 then Q( C_PUBLICATIONSA ) {...} else Q( C_PUBLICATIONSB ) {...}
end
procedure Do-Mobilink() Q( C_MOBILINK ) {...} end
procedure Do-Unload-Remote-Options()
  Q( C_REMOTE_OPTIONS ) {...}
end
procedure Do-Remote()
  Q( C_REMOTEMSGT ) {...} ; Q( C_REMOTEPUB ) {...}
  Q( C_REMOTESUBUID ) {...} ; Do-Unload-Publications();
  Q( C_REMOTEE ) {...}
  if V11 then Do-Unload-Remote-Options()
end
procedure Do-DBSync()
  ldt ← Col-Exists("SYSSYNC","last_download_time" )
  Q( C_REMOTEF,ldt ) {...}
end
procedure Do-Options() Q( C.C_OPTIONS ) {...} end

procedure Do-Unload()
  Get-DB-Version()
  Load-Exclude-Tables()
  G.dbo-name ← Q( C_DBO_NAME1 )
  if not G.dba-name ← Q( C_DBO_NAME2 ) then G.dba-name ← Q( C_DBO_NAME3 )
  G.charset ← Q( C_GETDBPROPERTY,"charset" )
  if G.dbinfo ← Q( C_GETDBINFO ) then
    Do-Users() ; Do-Logins()
  if V24 then Do-Servers()
  if V21 and not G.remove-java then Do-UserClasses()
  Do-User-Tables()
  if G.unload-schema then Do-User-Views()
  if V25 and G.unload-schema then Do-Events()
  if V26 and G.unload-schema then Do-Services()
  if V12 then Do-MobiLink()
  if V10 and G.unload-schema then
    if not G.dbsync-db then Do-Remote()
    if V8 then Do-DBSync()
  if G.unload-schema then Do-Options()
end
```

Figure 8.2: Pseudo-code for dbunload top-level requests.

system-defined user that owns system-created schema objects) and a canonical version of the active DBA user id. These retrieved values are stored in a global object, G. Retrieving the user ids takes either 2 or 3 OPEN requests. Next, Do-Unload retrieves options used to create the database instance using the C_DBINFO query. This call fails if the SYSINFO table is not present (it was not created in very early versions of ASA that are no longer supported).

At this point, dbunload has issued up to 39 OPEN requests using 13 distinct queries to gather configuration information that will be used to control what gets output. Next, Do-Unload calls 13 procedures (Do-Users to Do-Options) to output statements to re-create schema objects. Predicates based on global configuration values (G) and version information retrieved by earlier queries (V1...V27) control which of these routines are actually invoked. Within each output routine, one or more queries are opened and their result sets are processed in a loop. For example, Do-Users opens two cursors and processes their rows. Within the body of each loop, text is output to generate the appropriate statements. Further, additional (nested) queries may be submitted to gather more information about related objects. These nested queries are discussed in Section 8.2.1.2.

In total, dbunload issues up to 61 top-level OPEN requests with 35 distinct queries. The sequence of requests is predictable, although predicates within dbunload prevent some possible queries from being submitted, either because they are not appropriate for the version of the database instance (predicates depending on results of earlier version-detection queries) or because of input parameters the prevent the associated schema objects from being unloaded. The sequence of top-level queries is fixed, and the actual parameter values used to open queries can be predicted based on the position in the sequence and the results of earlier requests. Scalpel can rewrite these queries using predictions based on observed traces. While such a rewriting is beneficial, it has a fixed benefit regardless of the size of the database instance being unloaded. The number of sequential requests that can be prefetched does not depend on the number of rows of any of the tables, but this is not the case when we look at the nested requests that are opened within the body of the processing loops.

8.2.1.2 Nested Queries

The code in Figures 8.1 and 8.2 omitted nested queries that are submitted from within the processing loop of an outer query. Figure 8.3 shows the context tree of nested requests discovered by Scalpel. Five of the main output routines in dbunload open nested queries to generate text for nested objects. These nested queries are typically executed once for each row of the outer query, although local predicates may prevent them from being executed in some cases. If Scalpel can predict nested queries that will be executed, it can save a number OPEN requests that varies with the size of the outer result sets. For large schemas, this type of prediction will have a much greater

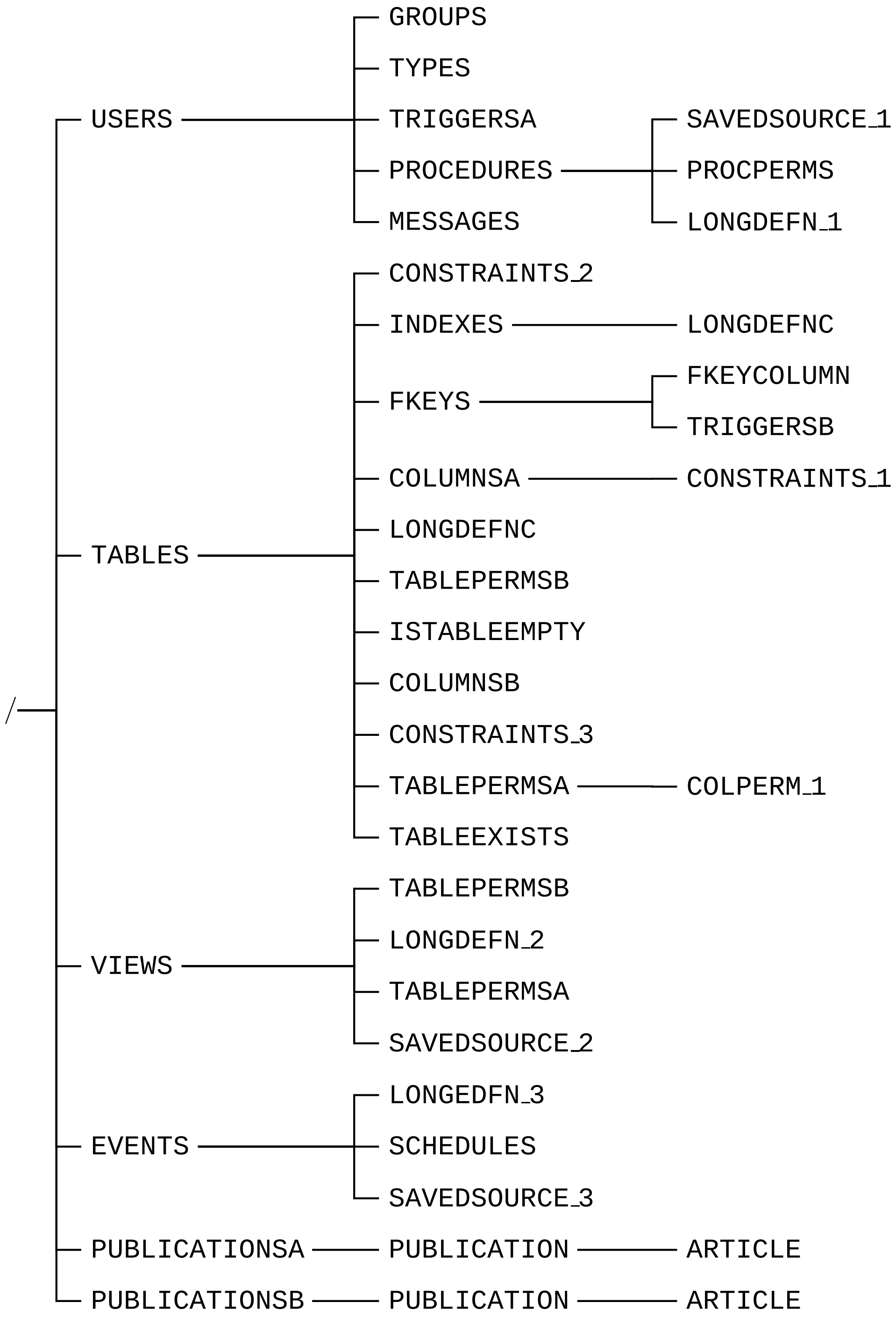


Figure 8.3: dbunload context tree identified by Scalpel. Child-less top-level contexts are omitted.

```
function Do-Users()
  if V10 then
    remoteauthfield ← 'UP.remotedbauth'
  else
    remoteauthfield ← ''N''
  if V1 then
    remarksfield ← ', UP.remarks'
    usergroupfield ← ', UP.user_group'
  else
    remarksfield ← ''
    usergroupfield ← ''
  open C_USERS cursor for USERS:
    SELECT user_id, user_name, password, resourceauth, dbaauth,
      scheduleauth, $remoteauthfield $remarksfield $usergroupfield
    FROM SYS.SYSUSERPERM UP
    ORDER BY user_id
  while r1 ← fetch C_USERS do
    if not Special-User( r1.user_name ) then
      ▷ Write statements to create the user
      Do-User-Perms( r1.user_id, r1.user_name, ...)
      if V7 then
        if not G.exclude-procedures then
          Do-Procedures( r1.user_id, r1.user_name, ...)
        if not G.exclude-triggers then
          Do-Triggers( r1.user_id, r1.user_name, ...)
      if V16 then
        Do-User-Messages( r1.user_id, r1.user_name, ...)
        Do-User-Types( r1.user_id, r1.user_name, ...)
  close C_USERS
end
function Do-User-Perms(user_id, user_name, ...)
  ▷ Write statements to grant authorities to user (RESOURCE, DBA, etc.)
  if V1 then
    open C_GROUPS cursor for GROUPS:
      SELECT group_id.user_name
      FROM SYS.SYSUSERPERM GID, SYS.SYSGROUP G, SYS.SYSUSERPERM U
      WHERE GID.user_id = G.group_id
      AND U.user_id = G.group_member
      AND G.group_member = $user_id AND ...
    while r2 ← fetch C_GROUPS do
      ▷ Write statements to grant membership in the current group.
    close C_GROUPS
end
```

Figure 8.4: Pseudo-code for dbunload to output database users. Identifiers starting with $ represent text substitution within a query.

impact than the batch prediction. For example, Doppelhammer et al. [57] described an SAP R/3 installation with 10,055 tables. Such a setup would take a significant length of time to unload, especially in a high-latency configuration.

Figure 8.4 shows an abstraction of a portion of the `Do-Users` procedure. This procedure outputs SQL statements to re-create all of the users in a database, along with the procedures, triggers, messages, and user-defined data types defined by each user.

The code supports a number of customizations, shown in Figure 8.4 as `if` statements using global variables: both version variables (V1...V27) and global parameters (G) are used. For example, the `G.exclude-procedures` predicate (line 929) controls whether procedures will be output at all, based on user-configuration options. This predicate is similar in effect to the P_0 predicate used in Section 3.6 because it does not depend on values from an outer query. The check on line 925 is used to see if a user is a 'special' userid created by ASA (these do not need to be re-created). In contrast to the `G.exclude-procedures` predicate, this predicate depends on outer row values and it is therefore similar to the P_1 predicate used in Section 3.6.

The `Do-Users` procedure represents only a small portion of the overall dbunload program. Figure 8.3 shows the queries of dbunload that are involved in a nesting relationship (an additional 32 queries are opened at the top level and they are not displayed).

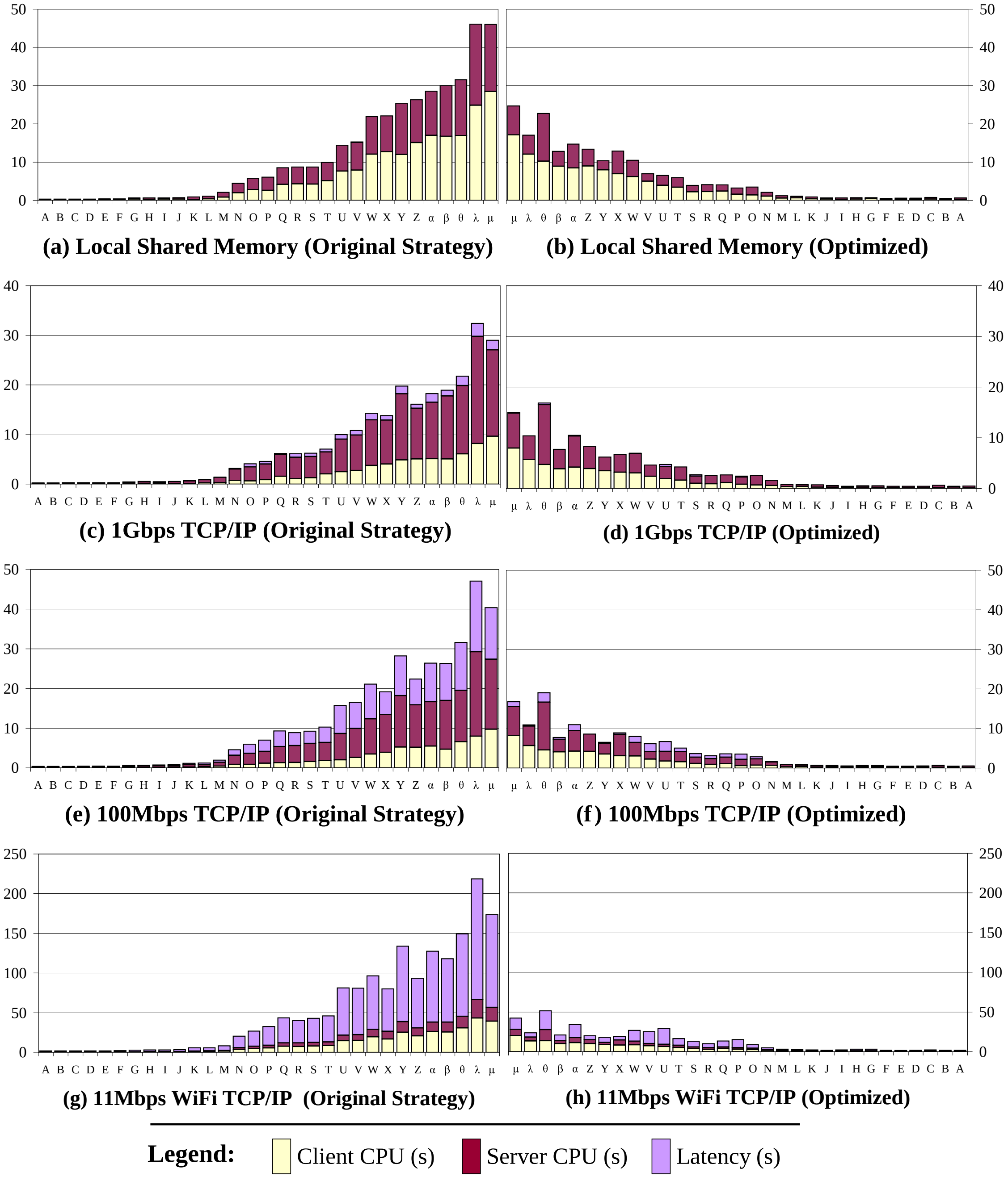
The dbunload program can be run in either a complete mode where everything is output, or a very selective mode where specific types of objects are excluded or specific named objects are included or excluded selectively.

8.2.2 Evaluating Scalpel Using dbunload

In order to evaluate the effectiveness of Scalpel, we performed a series of measurements of dbunload when using Scalpel on a variety of network configurations. We used 31 customer databases to evaluate dbunload performance. Figure 8.5 shows the run-time with and without Scalpel's optimizations for these 31 databases on a variety of network configurations. These databases are labelled with A,B,...,Z,$\alpha, \beta, \theta\, \lambda, \mu$ in order of increasing cost on the LCL configuration. In these tests, all of Scalpel's optimizations were permitted for nested request patterns, and we show the time to unload the entire schema. The results show that Scalpel provides significant savings, even in low latency configurations such as LCL (local shared memory).

8.2.3 Batch Prefetching with dbunload

The results in Figure 8.5 show the improvement Scalpel provides when optimizing only nested request patterns. We also considered the benefits provided only by batch prefetching. We ran dbunload in selective mode, outputting the schema for a single table in order to establish how much



(a) Local Shared Memory (Original Strategy)

(b) Local Shared Memory (Optimized)

(c) 1Gbps TCP/IP (Original Strategy)

(d) 1Gbps TCP/IP (Optimized)

(e) 100Mbps TCP/IP (Original Strategy)

(f) 100Mbps TCP/IP (Optimized)

(g) 11Mbps WiFi TCP/IP (Original Strategy)

(h) 11Mbps WiFi TCP/IP (Optimized)

Figure 8.5: Running time (s) of dbunload on different network configurations. Original times are shown on the left in increasing order by cost, and times with nested request pattern optimizations are shown on the right in the opposite order.

of an improvement is possible for the portions of the application that Scalpel's batch optimizations might apply to.

	U	O	Δ	Δ%
# OPEN calls	39	14	25	64%
Elapsed (ms)	997	336	661	66%
Client cost (ms)	109	78	31	29%
Server cost (ms)	47	63	-16	-33%
Packets to DBMS	190	69	121	63%
Packets to client	202	73	129	64%

Table 8.1: Benefits of batch pattern optimization for dbunload. Columns show unoptimized results (U), optimized results (O), absolute difference ($\Delta = U - O$), and relative difference (Δ%=Δ/U).

At the end of the training period, Scalpel stored information for 15 distinct queries and 46 contexts with prefetch size ranging from 0 to 19, average 4.7. Of these, 18 had no prefetch queries associated. Of the 28 contexts with associated prefetches, the average number of prefetched queries was 8.1. For a typical execution of dbunload, we observed 43 queries and built an atomic suffix trie with 818 nodes; the path compressed suffix trie contained only 50 nodes.

Table 8.1 shows the benefits of using the optimizations we have described with dbunload run in selective mode on a WiFi configuration. The table shows the unoptimized results (U) for the system run without Scalpel attached, the optimized results (O), the difference $\Delta = U - O$ and the percent difference $\Delta\% = \Delta/U$.

Scalpel's optimizations are able to eliminate 25 of the OPEN requests that are sent to the DBMS, which includes eliminating the per-request overhead for each of these. As a consequence, total elapsed time was reduced by 66%, and the client processing cost was also reduced by 29%. Interestingly, the server processing costs increased in the prefetching case. As all of the prefetches in this test were useful, this extra cost must be related to optimizing and executing the complex combined query.

8.2.4 Summary of dbunload

The results for dbunload are promising. When using all prefetching algorithms, Scalpel provides significant savings, even in low-latency configurations. For example, the run-time for database μ

is reduced from 45s to 25s in the LCL configuration, and from 40s to 17s in the LAN0.1 configuration. Scalpel acts to extend the set of systems for which dbunload provides excellent results. Without Scalpel's optimizations, λ runs in 220s on a WiFi configuration; the optimizations allow it to run in 24s, which is faster than the 46s for λ in the LCL configuration.

While dbunload provides many opportunities for optimizing nested request patterns, it also contains sequences of requests that can be predicted using Scalpel's batch optimizer. When run in selective mode to unload the schema for a single table, Scalpel provides an improvement of 661ms, or 66% in running time. This absolute improvement is present when running in the full mode, but it is overshadowed in relative terms by the nested request patterns.

8.3 Case 2: SQL-Ledger Case Study

We investigated SQL-Ledger as a second case study. SQL-Ledger is a web-based double-entry accounting system written in Perl. The system is configured with a DBMS storing persistent accounting entries, a web server that executes Perl scripts to implement business logic, and a web browser that presents the user interface.

Figure 8.6 gives an overview of the structure of the SQL-Ledger system. A web browser presents a user interface, and a web server executes the business logic using the common gateway interface (CGI). The SQL-Ledger business logic communicates with a DBMS using the Perl DBI library and a vendor-specific DBD module. We use 'V' as a placeholder in Figure 8.6 to represent a specific DBMS implementation.

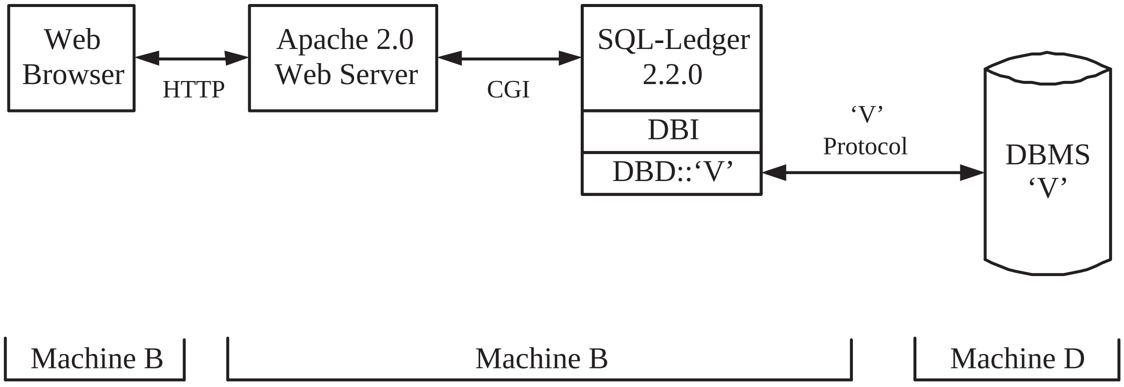


Figure 8.6: SQL-Ledger system structure. 'V' is used as a placeholder for a specific DBMS vendor implementation.

In general, the web browser, business logic, and DBMS can be placed on separate machines. We simulated a web user and the business logic on machine B, and used a commercial DBMS

table	Rows (SF1)	Rows (SF10)	Description
customer	10	100	Customer name, address, and shipping information
parts	10	100	Part name, price, inventory, and description
makemodel	30	300	Name of a specific make and model of a part
partsgroup	2	20	Name of logical group of parts
customertax	20	200	Types of tax charged for each customer
partstax	20	200	Types of tax charged for each part
ar	100	10000	Invoices
invoice	1486	145020	Invoice line items
acc_trans	4558	445060	Ledger entries for line items, tax, and total

Table 8.2: Initial table sizes for scale factors SF1 and SF10.

on machine D communicating using a 100Mbps LAN (configuration LAN0.1). The LAN0.1 and other configurations are described in Section 3.6.

The web browser presents a menu of links to activities that a user can perform, such as adding a transaction. Each of these activities may require multiple steps. For example, when adding an invoice, a separate step is used for each invoice line. For each step, the user presses an Update button that submits a partially completed form to the web server. The business logic scripts parse the partially completed form, issue database requests to retrieve additional information, and format a new form to be returned to the user. The last step of a user activity consists of using a Post button to apply the requested changes to the database server.

Some of the business logic scripts that are executed during a user activity issue nested database requests that might benefit from our proposed optimizations.

We populated the database with synthetic data and simulated the activities of a user working with the system. We focused only on the accounts-receivable activities. Table 8.2 shows the tables that we populated with synthetic data and how many rows we used when generating the data. We used two scale factors, SF1 and SF10, to simulate the system being used with different configurations. Scale factor SF1 represents SQL-Ledger being used in a small company, while SF10 represents the needs of a medium sized company. For each invoice in the initial population, we generated an average of 14.5 line items selected using a uniform distribution of $[10, 20)$. The line item records are stored in table `invoice`, and each line item requires 3 records in table `acc_trans` due to the double-entry accounting requirements. In SF10, we use 100 times the initial invoices of SF1 to represent a company that not only has 10 times the customers but also has been in business for 10 times as long.

We simulated a user performing accounts receivable activities with this synthetic data using a remote browser emulator (RBE) implemented in Perl. Table 8.3 shows the activities that we simulated based on the description in the SQL-Ledger manual [170]. For each activity, we show the proportion of simulated sessions that perform the activity. We also show the steps that may be performed by our emulated user during the activity and the average number of times the step is performed during a simulated activity (Freq). For each step, we show the number of database requests submitted for SF1 and SF10 in both the original setup and with Scalpel's optimizations.

At SF1, the response times are quite reasonable. All individual step times are sub-second except for two report activities (Tax Collected and A/R Aging). Both of these reports execute in under 2 seconds. When moving to SF10, many operations remain quite quick; however, some operations get significantly slower when moving to the higher scale factor. This slow-down may be due to nested queries of the type optimized by Scalpel (leading to the higher rate of requests shown in Table 8.3). We used the Scalpel prototype to assess the benefits of our proposed optimizations in this system.

8.3.1 Configuring the System for Measurement

When we configured the SQL-Ledger system for analysis, we noted some interesting details about the system and how it reacted with Scalpel's optimizations. This section describes our measurement setup and items that we discovered.

8.3.1.1 Perl to JDBC Bridge

The SQL-Ledger system is implemented in Perl using DBI, but our Scalpel prototype is implemented in Java supporting JDBC. In order to use our prototype with this case study, we used a DBI-JDBC bridge (DBD::JDBC 0.64). Database requests from the Perl business logic are sent via TCP/IP to a Java server process that executes appropriate JDBC calls. This is an atypical configuration for SQL Ledger, and it introduces additional latency. The elapsed time for simulated user sessions was up to 12 times slower for the measured activities when compared to a direct DBI connection. This latency is higher for user activities with many small requests of the type optimized by Scalpel. In order to provide a fair comparison, we stored a trace of JDBC requests made by the Java server and replayed this trace either directly to the vendor's JDBC driver or using the Scalpel prototype.

We used the following sequence of measurements:

1. Populate the SQL-Ledger database at SF1 or SF10.

		# Database Requests			
		Original		Optimized	
Activity	**Freq**	**SF1**	**SF10**	**SF1**	**SF10**
A. Add Sales Invoice (35% of sessions)		206.6	1429.9	102.9	118.5
0. Navigate to start screen	1	19.0	19.0	19.0	19.0
1. Choose Customer, Invoice #, and Order #.	1	7.1	8.0	7.1	8.0
2. Choose account, currency, and exchange rate.	1	1.0	1.0	1.0	1.0
3. Add one line item	10	138.0	1351.4	34.3	40.0
4. Select a part from a list.	7.5	0.0	0.0	0.0	0.0
5. Print a packing list.	1/3	0.4	0.4	0.4	0.4
6. Print invoice.	1/3	0.3	0.4	0.3	0.4
7. Post new invoice.	1	40.8	49.7	40.8	49.7
B. Add Cash Receipt (35% of sessions)		31.3	46.4	28.0	28.7
0. Navigate to start screen.	1	4.0	4.0	4.0	4.0
1. Choose customer.	1	11.8	26.2	8.5	8.5
2. Choose currency and exchange rate	1/2	0.5	0.5	0.5	0.5
3. Apply money to outstanding invoices.	1	1.0	1.0	1.0	1.0
4. Post the new receipt	1	14.0	14.7	14.0	14.7
C. Transaction History (10% of sessions)		58.3	85.0	49.3	71.5
0. Navigate to start screen.	1	2.0	2.0	2.0	2.0
1. Submit search criteria.	1	1.0	1.0	1.0	1.0
2. Navigate to a random invoice.	3	55.3	82.0	46.3	68.5
D. Tax Collected (10% of sessions)		3.2	3.1	3.2	3.1
0. Navigate to start screen.	1	2.0	2.0	2.0	2.0
1. Submit search criteria.	1	1.2	1.1	1.2	1.1
E. A/R Aging (10% of sessions)		9.2	63.4	4.0	4.0
0. Navigate to start screen.	1	2.0	2.0	2.0	2.0
1. Submit search criteria.	1	7.2	61.4	2.0	2.0

Table 8.3: Simulated user activities.

2. Make a backup copy of the database.
3. Record a trace of JDBC operations performed while simulating 500 user activities in the proportions and step frequency given in Table 8.3.
4. Restore the database from backup
5. Replay the stored trace using the vendor's JDBC driver. Optionally configure Scalpel to intercept requests for training or run-time optimization.

Steps (4) and (5) above can be repeated to test a variety of configurations.

Replaying a stored trace provides flexibility in measuring different configurations. However, it requires that the results returned match the order they were seen when recording the trace. If the order does not match, then the parameter values used from the trace file will not match what the client application would have submitted. We avoid this complication by considering only the client hash join (H) and outer join (J) strategies for rewriting nested request patterns. These two strategies do not affect the order of rows from outer queries so they can be used with replayed workloads.

8.3.1.2 Primary Keys

Scalpel relies on finding a candidate key for queries to be optimized. Some of the SQL-Ledger tables did not have an explicit key defined. We defined primary keys for the `ar`, `customer`, `defaults`, `exchangerate`, and `makemodel` tables based on attributes that appeared to be a candidate key. Key creation is an important step to provide the most opportunities for effective rewrites.

8.3.2 Parameters as Literals

The SQL-Ledger system generates most of its queries with parameter values embedded as literals within the query text, although a few queries do contain parameter markers in the query text with actual parameter values passed explicitly to the open request. We configured Scalpel to scan all queries that do not contain parameter markers. Any constant literals within the query are replaced with parameter markers to generate a query template with separate actual parameter values. This extra scanning adds to the overhead of Scalpel as all submitted queries must be scanned before determining whether the query requires special action.

When detecting the possible source for a parameter, we extended the algorithm of Figure 3.7 to also consider parameters that have the same value for every execution. These parameters result from Scalpel's inability to distinguish between substitution parameters in a dynamically generated query and constant literals that are the same for every invocation.

8.3.3 Query Variants

The SQL-Ledger system generates queries dynamically based on input from the user. For example, the `WHERE` clause of a query may be built to include predicates for only the search fields supplied by a user. Further, the `ORDER BY` clause may be built to match user-specified sorting requirements. In some cases, additional joins and subselect expressions are included based on configuration parameters or user input.

The number of distinct queries generated dynamically can be fairly high. For example, the `retrieve_item` function called during the Add Sales Invoice activity generates a query based on 4 boolean conditionals, leading to up to 16 variants being presented at run time.

These query variants appear to Scalpel as distinct queries. Variants do not pose a problem for correctness, but they require a longer training period in order to learn patterns for all of the distinct variants that can be submitted. If Scalpel could recognize that variants result from the same dynamically generated query, this training time could be reduced and also the storage requirements for the context forest could be reduced.

8.3.4 Complex Combined Queries

The optimizations we have described generate more complex queries from simpler original queries. This additional complexity raises the possibility that the DBMS query optimizer will choose a plan that is worse than the original strategy while providing cost estimates to the contrary. We found this to be the case for the A/R Aging activity. In this activity, a report is generated with one line for each customer with outstanding amounts owing. An outer query is used to iterate over these customers, and an inner query is submitted for each row. This inner query contains a 4 branch `UNION`; each branch contains a join and a subselect. Scalpel uses the client hash join (H) strategy to optimize this nested query.

We found that the optimized strategy took over 12s on the SF10 database while the original (nested) strategy took only a little over 8s. The problem was that the DBMS was selecting a suboptimal plan that was worse than the original nested strategy. The SQL-Ledger system does not explicitly create database statistics for the tables it uses. After creating statistics for all pertinent tables, we found that the optimized strategy executed in a little over 2s.

This problem raises two interesting points. First, the optimization decisions made by Scalpel rely on estimates from the DBMS. If these estimates are in error, then poor optimization choices will result in longer execution times. To some extent, such situations can be improved by combining information that Scalpel obtains during the training period with the DBMS estimates. Second, the complex queries generated by Scalpel's rewrites may not be handled well with existing DBMS technology. This appears to be especially true when using the `LATERAL` keyword or variants thereof. This problem could be avoided by using a product-specific interface to represent the combined queries instead of expressing the rewritten query in SQL. Such an interface could simply combine query plans for the outer and inner query using a nested loops join in the DBMS. This approach would eliminate the communication costs associated with the nested strategy in the client, but it would not permit any improvements beyond communication costs. The approach of directly specifying the execution plan follows a suggestion of Chaudhuri and Weikum [37]: it reduces the uncertainty in execution costs at the expense of missing out on possible opportunities to choose a better strategy.

8.3.5 Nested Query Patterns

After configuring the system as described, we used Scalpel's training mode on a SF1 database to identify nested request patterns that present opportunities for optimization. We configured Scalpel to consider only nested request patterns, ignoring for now the optimizations that can be achieved by recognizing batch request patterns.

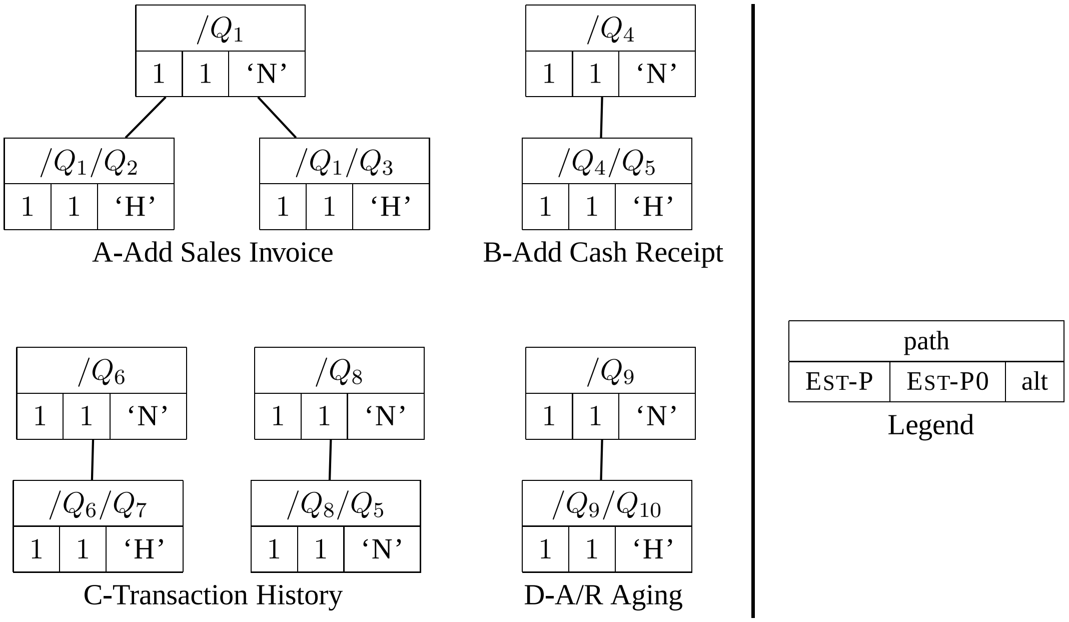


Figure 8.7: Nested request patterns found in SQL-Ledger. Note that $/Q_8/Q_5$ is not a feasible child of $/Q_8$ because Scalpel did not predict the source of all correlation values.

Figure 8.7 shows the non-trivial context sub-trees detected by the Pattern Detector. Due to query variants, the full context tree is actually somewhat larger. There were three variants of Q_1 discovered, and two variants of Q_9. These resulted in additional sub-trees; we have collapsed these for a simpler presentation.

Figure 8.7 also shows the measured predicate selectivity and the execution strategy selected by Scalpel. Only Q_5 can be used in a join strategy as the other queries could return more than one row.

Note that query Q_5 appears in two contexts. Query Q_5 is submitted by the function `get_exchangerate` (described in Figure 1.3). The `get_exchangerate` function is called from both `get_openinvoices` in the Cash Receipt activity (giving context $/Q_4/Q_5$) and by the function `create_links` in the Transaction History activity (giving context $/Q_8/Q_5$. In the Cash Receipt activity, Scalpel discovers the correlation between the `currency` and `transaction_date` parameters and attributes of Q_4, the outer query. In the Transaction History activity, Scalpel discovers that the `transaction_date` is correlated to an attributes of Q_8. However, the `currency` attribute does not match any attribute of Q_8. Because of this, $/Q_8/Q_5$ is *not* a feasible prefetch candidate.

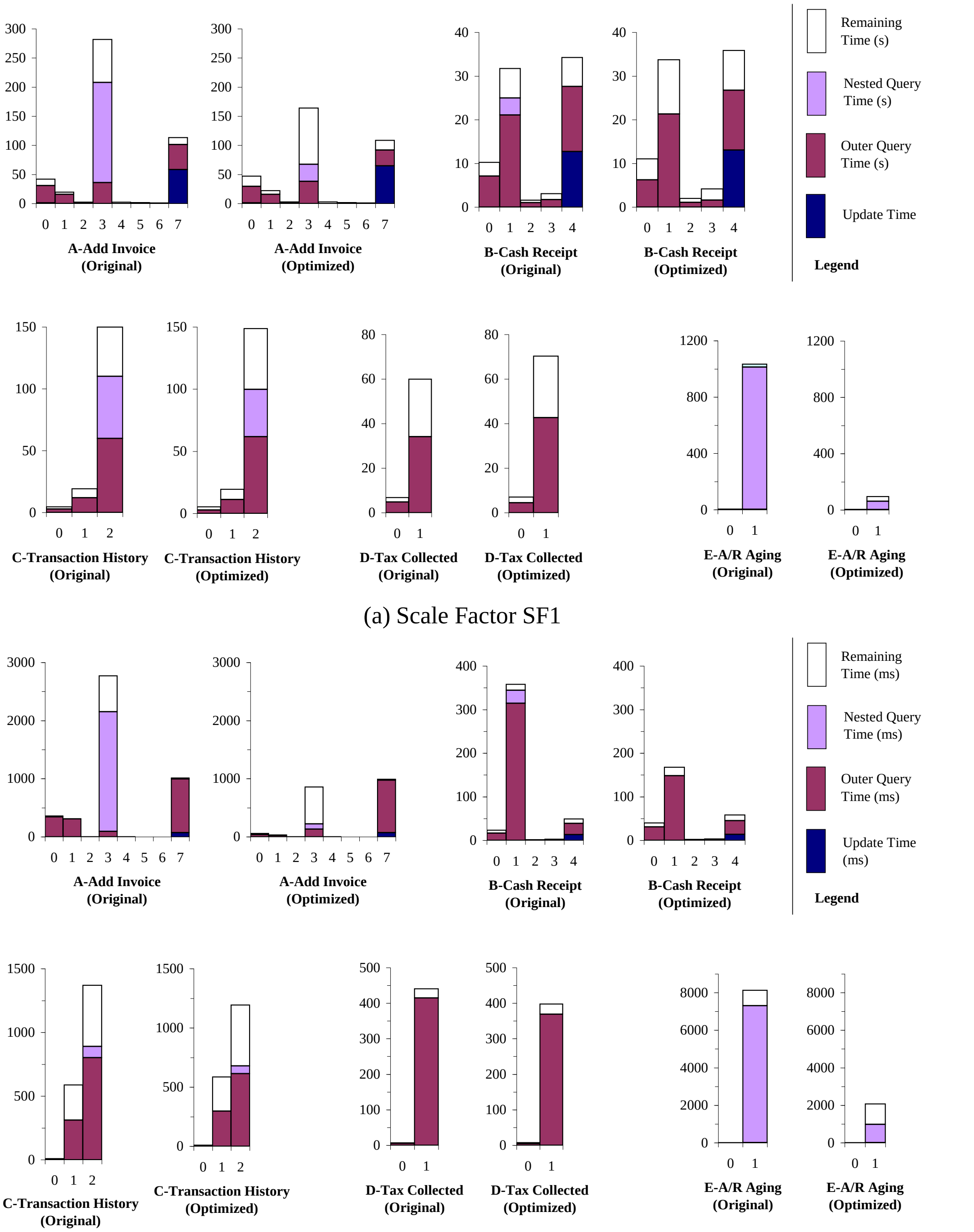
Although the value of `currency` cannot be predicted by looking at attributes of containing queries, the value could be predicted in two ways. First, the value of `currency` is constant within a particular execution of Q_8. Scalpel could recognize that the parameter is loop invariant. Partitioned strategies could exploit a loop-invariant attribute correlation because they have the parameter values for the first submission of the inner query when the rewritten combined query is submitted. Unified strategies could not use this correlation source. Second, the `currency` attribute could be predicted by considering queries that sequentially precede Q_8. At present, the correlation detection of nested request patterns and batch request patterns is not integrated (as described in Chapter 6). This combination offers promise for extending the set of possible candidates that can be detected.

8.3.6 Performance Results

Table 8.4 summarizes the performance results for a SF1 and SF10 database when executing a stored trace of 500 user activities using the optimizations selected for the nested query patterns shown in Figure 8.7. The tests were performed using configuration LAN0.1 (Section 3.6). For the tests with nested request patterns, we used a single SF1 trace for both training and timing, and we used a separate SF10 trace for timing with the model built by training on the SF1 trace.

For SF1, the savings of latency are relatively modest. However, significant savings are achieved in the number of queries submitted (a reduction of nearly half). This saving in queries has a corresponding decrease in the associated network costs.

The training mode did not introduce a significant amount of latency, although it did increase client processing costs due to the requirements of finding nesting patterns and identifying parameter correlations. The network and server costs are slightly higher during training due to the need to retrieve catalog information from the DBMS.



(a) Scale Factor SF1

(b) Scale Factor SF10

Figure 8.8: SQL-Ledger elapsed database time (ms) for original and optimized requests. Each bar represents a different step (Table 8.3).

	SF1			**SF10**	
	Original	**Optimized**	**Training**	**Original**	**Optimized**
Total time (s)	171.5	97.5	172.4	1,439.2	629.8
Server CPU (s)	29.3	23.9	38.6	578.8	283.7
Client CPU (s)	48.6	51.9	60.1	373.9	319.6
Queries Submitted	39,909	20,572	39,909	285,913	24,982
Top-Level	16,130	161,30	16,130	18,791	18,791
Nested	23,779	4,442	23,779	267,122	6,191
Update Requests	5,263	5,263	5,263	6,818	6,818
Network Packets	163,223	86,969	179,181	1,200,287	160,583
Client → Server	83,195	44,900	91,383	610,739	93,466
Server→ Client	80,028	42,069	87,798	589,548	67,117
Network Bytes (MB)	57.2	40.7	61.9	457.6	199.8
Client → Server	22.2	15.9	23.5	137.2	22.1
Server→ Client	35.1	24.8	38.4	320.5	177.7

Table 8.4: Costs of 500 user activities. Optimized times include only nested request optimizations. Scalpel was trained and timed on a single SF1 trace, and timed only on a SF10 trace using the same model built from the SF1 training period.

When we consider the SF10 database, the optimizations have a much more dramatic effect. The elapsed time is reduced by over 800s. Further, this savings in latency is associated with significant reductions in server and client processing costs, along with lower communication costs.

Figure 8.8 shows the elapsed time for each of the steps performed by our simulated user. The contributions of updates, top-level queries, and nested queries are shown separately. The remaining time consists of Scalpel costs to decode the combined result (for example, looking up values in a hash table) as well as overhead in our trace replay process.

All simulated activities except D-Tax Collected contain some nested requests. However, only two activities benefit to a great extent from our rewrites: A-Add Invoice and E-A/R Aging. These two activities improve significantly in the SF1 database and dramatically in the SF10 database.

8.3.7 Preliminary Result for Batch Request Patterns

The DBMS (B) we used with SQL-Ledger does not support NULL values in the SELECT list as we currently generate them for outer union rewrites. The other two DBMS products that we tested were not such strict disciplinarians, but they are not supported by the SQL-Ledger system. DBMS B does support our optimizations if an appropriate CAST is used to provide a data type for each NULL value. As we have not extended Scalpel to track column data types, we present preliminary results showing only the outer-join based rewrite. As this rewrite is only available for at-most-one queries, this restricts the class of prefetches Scalpel considers.

		Optimized	
	Unoptimized	**(Join Only)**	**(Union+Join)**
Queries submitted to DBMS	38,525	37,233	n/a
Peak # nodes	n/a	23,525	23,525
# FSM states	n/a	27	38
States with any prefetch	n/a	7	28
Queries prefetched	0	1,416	n/a
Useful	0	1,292	n/a
Wasted	0	124	n/a
Total latency	80.3	69.0	n/a

Table 8.5: Preliminary results for batch request optimizations in SQL-Ledger.

Table 8.5 shows preliminary results when using Scalpel's batch request optimizations in the LAN1 configuration. The last column shows results including both outer union and outer join rewrites; only training results are shown as the outer union strategies cannot be executed. The middle two columns give results with no optimizations and with only the outer join optimizations, respectively.

The second row shows the peak number of nodes in the suffix trie. An atomic suffix trie was used; however, it does not show the $O(n^2)$ model size that is possible (this could be up to $O(10^9)$ for this many requests). The modest growth is due the moderately short transactions used by SQL-Ledger. Each transaction is a separate trace, terminated with an out-of-band character ($) in the trie. For this reason, the atomic suffix trie behaves reasonably in this setting.

After removing redundancy, we find a smaller FSM, with 27 states when using outer join only (J) and 40 states with outer join and outer union (UJ). Of these, only 7 (J) and 28 (UJ) states have an associated list of queries to prefetch: the remainder of the states are used to track context.

Table 8.6 shows the distribution of batch length for the optimization with only outer join (J) and with both outer union and outer join (UJ). The J strategy is only able to prefetch 2 requests at a time before encountering a query that might return more than one row. The outer union ap-

Batch Length	Join Only	Union+Join
0	20	11
1	0	0
2	7	15
3	0	4
4	0	8

Table 8.6: Distribution of batch prefetch lengths.

proach does not have this restriction, and when we include it as well, we find prefetch lengths up to 4 queries.

At run-time, the join-only approach prefetched results for 1,416 queries. Of these prefetches, 1,292 were ultimately used, and the other 124 were wasted effort. Overall, the prefetching using only outer joins saved about 14% of the latency associated with this test. If we consider also outer union rewrites, it is likely we would save even more of this latency. Further, the savings would also increase in higher latency configurations, such as WiFi, WAN, or even LAN0.1.

8.3.8 Summary of Case Studies

In summary, our examination of dbunload and the SQL-Ledger system demonstrates that opportunities for our proposed optimizations do appear moderately often in at least some systems. While opportunities for nested request rewrites are relatively rare, they can have substantial benefit when they are optimized. There are significantly more opportunities for our batch-based rewrites, and these also reduce overall latency.

Scalpel can identify these optimization opportunities using a training period that does not excessively degrade system performance, and it is able to automatically rewrite submitted requests to take advantage of the selected optimizations. Some caution must be used when applying the Scalpel system in order to get the most benefit. For example, in the SQL-Ledger study we needed to add primary keys and ensure that database statistics were created before the full benefit of the optimizations was achieved.

9 Related Work

In this chapter, we position our work within the field of related research. There has been a significant amount of work on prefetching techniques for many different problem domains. Section 9.1 provides an overview of related work in this area, with particular reference to approaches similar to those we propose. In particular, prefetching techniques based on a probabilistic model of past requests are closest to the results we have presented. Section 9.2 discusses the theoretical basis for this type of prediction.

Once we have predicted a likely sequence of future requests, we would like to process the sequence efficiently. We have described how Scalpel generates combined queries. For nested request patterns, the combined query encodes the result of a join that we have detected in the clients request stream; for batch request patterns, the combined query encodes the results for a predicted sequence of queries. Section 9.4 discusses other work related to efficiently processing a known sequence of queries.

9.1 Prefetching

The idea of prefetching has been applied in many different contexts in the field of computing. Prefetching has long been supported for device I/O in operating systems [66, 152]. Smith [172] provided an early bibliography of work related to prefetching and caching. Somewhat more recently, prefetching has been used for prefetching data from main memory into a processor cache. Smith [174] provided an early survey of this field, and Vanderwiel and Lilja [182] provide a survey of recent results. An interesting use of prefetching is the idea of *hoarding*, where files are copied to a mobile device in anticipation of their being needed while disconnected from the high-speed network. Several approaches to hoarding have been proposed based on predictive techniques related to those we use [120, 121, 176, 177, 208].

The problems associated with fine-grained access are well known, and there are a number of practical solutions that can improve performance by avoiding fine-grained access. In fact, the problems of latency are becoming relatively more important; as Patterson [143] notes, prefetching is one way to address the pervasive problem that improvements in latency lag significantly behind improvements in bandwidth. It appears that this gap is likely to widen over time due to fundamental physical limitations in our system implementations.

A prefetching mechanism must *identify* future requests that are likely to be submitted. In Section 9.1.1, we describe related work on prefetching that uses the physical layout of data to identify the data to be prefetched. As we describe, this approach is simple to implement and exploits the large and growing disparity between random and sequential access. However, it does not help if the client access pattern does not match the physical layout. Section 9.1.1 also describes various approaches that are used to re-order a clients access to better match the physical layout.

When a client's reference pattern is not sequential and re-ordering is not feasible, a prefetching system may choose to use observations of the client's previous reference behaviour to predict future reference patterns. Section 9.1.2 describes related work that monitors the sequence of selected data items to predict likely future items. This type of prediction scheme does not exploit efficient sequential access, so it is not as useful for prefetching blocks from a disk. Instead, the approach is particularly useful when the granularity of the predicted item is large (for example, entire files) or when the data stored can be efficiently accessed in random order (for example, when fetching data from another workstations memory)

When using patterns of fetched data items, a prefetching approach needs space that is proportional to the number of data items. Further, this approach does not offer suggestions when novel sequences of data items are reference. Section 9.1.3 describes *semantic prefetching*. With semantic prefetching, the server understands something about the data that is stored. For example, it might know that a fetched object contains references to other objects. The server can use this understanding to make prefetching decisions that use the *intension* instead of the *extension*. In this way, models may be smaller, and prefetching decisions can be made when novel data *items* are referenced, provided that the data types are known.

Finally, Section 9.1.4 summarizes related prefetching approaches that have been previously considered.

9.1.1 Prefetching Based on Physical Layout

Computer systems have long supported sequential prefetching for I/O, particularly from stream oriented devices. Feiertag and Organick [66] describe the I/O support in the MULTICS system, while Ritchie and Thompson [152] describe how the the Unix system was designed (influenced by MULTICS) to default to sequential access. This default sequential access was used on the premise that many files would be processed sequentially. With this access pattern, it is a very good idea for the operating system and I/O components to prefetch data sequentially beyond the last value requested by the client. If the client is following a sequential access pattern, then the prefetched data will be used, reducing exposed latency. The most appealing aspect of sequential prefetching is its simplicity and low cost. It is simple for the server to implement, and it does not require complicated analysis or bookkeeping. Further, if an application is scanning an entire file, it is very likely to do so in sequential order, so the prefetching is likely to produce many hits. Finally, the most compelling advantage of sequential prefetching is that it does not cost very much to read a few more pages sequentially ahead, so the cost of satisfying the prefetch request is low. Gray [82] suggests that current trends indicate that sequential access will soon be 500 times faster than random access. Even if we have a low hit rate for prefetched data, the increase in cost due to prefetching will not be substantial, and the savings can be considerable.

Smith [173] investigated using sequential prefetching to improve the performance of database applications using an IMS database. He found that sequential prefetching was very effective in some cases, although it was important to detect sequential scans as opposed to random scans. This detection was used to choose a variable number of blocks to fetch. Smith [175] also used sequential prefetching when proposing a disk cache; however, in that case, Smith used only a single block of read-ahead.

The actual reference behaviour of database systems has been examined by several researchers [36, 55, 94, 99, 101, 187, 207]. Many database workloads do exhibit sequential access to data pages. For example Hsu, Smith and Young [94] found that the TPC/D benchmark and several real-world workloads exhibit access that is significantly aided by sequential prefetching. However, clearly not all workloads are sequential. Kearns et al. [102] found that almost any reference behaviour may be found in the workload of a database system; however, they found that reference behaviour can be predicted and exploited.

Kotz and Ellis [109, 110] considered prefetching techniques within a single file when the consumer is a multi-processor system executing. For example, the client could be executing a scientific or database workload. They considered 4 strategies, each based on recognizing when a portion of the reference behaviour is sequential. In this way, their proposed system can exploit sequential prefetching when it is useful, and avoiding the wasted work in regions of the reference trace that do not exhibit sufficient locality of reference to make sequential prefetching worthwhile.

While sequential prefetching is beneficial when the client is reading sequentially (for example, processing an entire data set), there are also cases where a client demonstrates significant spatial locality without using a sequential pattern. Prefetching can be achieved in this case by fetching data a page at a time, returning more data than requested from a region spatially near the requested item. This approach shares the simplicity and low-cost benefits of sequential prefetching, although it also only helpful if the client application does exhibit locality within the prefetched page size.

The *page server* architecture of OODBMSs is an example of this type heuristic to prefetch objects into a client's cache. When an application accesses an object that is not resident on the client, a demand fetch is sent to the server for the appropriate page. The returned page contains the requested object along with the other objects grouped nearby. Again, the incremental cost to the server for fetching and transmitting the entire page instead of only one object is minimal (it may even be cheaper to send the entire page rather than worrying about interpreting the contents and copying out the single object). Here, the hope is that the application will be able to use some of the other objects from the page, saving further demand fetches to the server.

Liskov et al. [129] extended the idea of page servers to fetch an arbitrary sized set of objects in the neighborhood of the requested object in the THOR project. Run-time monitoring was used to decide the size of the neighborhood that would be prefetched. By using a variable-sized window, THOR was able to achieve the benefits of large, multi-page sequential prefetch, and also to reduce to no prefetching in situations where the hit rate was too low to benefit the application.

Page servers are effective if the client application has an access pattern that exhibits locality within a page: in that case, other objects in the page are also likely to be used. Further, in many cases it is more efficient to ship entire pages than individual objects. However, the application reference pattern may be such that only one or a small number of the objects on a page are needed. In this case, significant resources are wasted. Voruganti, Özsu and Unrau [184, 185] propose a hybrid architecture, where pages or objects can be fetched from the server. This approach achieves the benefits of a page server when the access pattern makes that effective, and avoids the downside of a page server when the clients access pattern does not match the page layout.

Shao et al. [169] presented Clotho, which can be viewed as similar to the hybrid architecture of Voruganti, Özsu and Unrau [184]. Clotho is a system that allows the layout of data on disk to differ from the organization in memory. This flexibility allows different levels in the memory hierarchy to provide efficient prefetching.

The approach of prefetching based on physical layout is simple to implement, and the cost to the server for implementing the prefetching is low since prefetched data items are ‘nearby’ to demand-fetched items. However, this approaches only produce good hit rates if the access pattern mirrors the physical layout of the data. An application may exhibit logical locality of reference that does not correspond to a physical local exploitable by a prefetching system. In a file system, this can occur if a file that is accessed in logically sequential order becomes physically fragmented. In other cases, an algorithm may visit pages in an order that is logically relevant, such as following disk pointers, but not physically sequential. In such cases, we may be able to re-order the data so that the physical layout is useful for prefetching the access patterns submitted by the client.

Akyürek and Salem [8, 9] suggested that we can convert locality of logical reference patterns into physical locality by rearranging disk blocks for efficient access. Akyürek and Salem used various first-order statistics to select an appropriate placement of disk blocks.

Yin and Flanagan [202] use a similar approach to re-order the pages needed to start up a program. When a program loads, it may read in a number of program and configuration files. These files are typically read in an order defined by the program control flow, which typically does not match the physical layout of files on disk. Yin and Flanagan proposed five algorithms to choose a re-ordering of disk blocks based on observed request sequences.

Rearranging data items is also possible for page server OODBMS products. This tuning is difficult to achieve in general, and it is impossible for cases where two applications with different access patterns access the same data. There is no single optimal arrangement in this case. The problem is similar to selection of clustered indexes. There are no perfect solutions, but on-line reorganization can help to dynamically adjust the physical layout of data in order to improve prefetch performance. If the hit rate for prefetched data is low, system performance is likely to be worse than a system without prefetching. Even though the incremental cost to the server is not high, useless prefetches still waste network bandwidth and pollute the clients cache.

It is not possible to organize one copy of data to match distinct access patterns. However, the abstraction provide by DBMS products provides another possibility. We can have multiple copies of all or part of a data set, and these can be organized in different ways. A materialized view is one example of such an arrangement, and we can view index-only retrieval as another such approach.

Gerlhof and Kemper [78] discussed practical details of implementing prefetching in a page-based OODBMS. They used a cost model that measured the expected speedup due to a prefetching strategy, and considered the cost of any administrative overhead and additional bandwidth required for prefetching. They found that performance can be substantially improved using prefetching, but the effectiveness is very dependent on the accuracy of the prediction of future requests, either through user hints or a predictor module. In order to improve the effectiveness, they implemented *prefetch support relations* [79]. These are relations stored in the server that contain the precomputed set of pages that are needed for a given operation with a particular set of parameter values.

In summary, prefetching based on physical layout is an obvious improvement. If data are frequently accessed sequentially, it is a good idea to fetch extra data items before they are requested. With this approach, the data store can use a simple heuristic that indicates what items to prefetch. The implementation is simple, and there is no need for complicated and memory intensive bookkeeping procedures. The most important benefit, however, is the low cost of fetching additional data items that are physically near the requested item. Fetching such nearby items can often be done without head movement of a disk, while a seek would be required if the items were demand fetched. This cost difference is an important reason for the popularity of sequential prefetching. Access patterns that are not physically sequential are often converted into sequential patterns through rearrangement of the data or the fetch order so that this cost difference can be exploited.

9.1.2 Prefetching Based on Request Patterns

In order to avoid the problems associated with prefetching based on physical layout, Palmer and Zdonik [142] proposed Fido, a cache that learns to fetch. The Fido system is based on training an *estimating prophet* to recognize patterns of object references in order to predict future references. The patterns were recognized using nearest-neighbour associative memory. The approach of Palmer and Zdonik trained the associative memory offline on access traces, and used the associative memory to predict future requests based on access patterns which approximately match a pattern in the training trace.

Grimsrud, Archibald and Nelson [86] proposed using order-1 predictors to prefetch from a disk. A large table (called the adaptive table) contains a tuple for each cluster on the disk. The tuple gives a prediction of the next cluster to be accessed, along with a weight that is used to express the confidence the system has in the prediction. These next cluster predictions can be used for prefetching; in conjunction, a disk rearrangement algorithm could re-order the disk to give efficient sequential access matching the logical order of access.

Krishnan and Vitter [116, 117, 183] used an approach like that of Palmer and Zdonik [142] in order to predict future object references based on observed references. Their work was based on a Lempel-Ziv compression algorithm, which was used to detect patterns in a reference trace that matched a pattern observed during training. Krishnan and Vitter [117] showed that this Lempel-Ziv compression technique provides the theoretically best expected performance under arbitrarily complex workloads. In practice, the Lempel-Ziv technique has some limitations; Curewitz, Krishnan and Vitter [48] explored practical issues related to using compression for prefetching, proposing a PPM-style compression algorithm as a practical method for prediction.

Griffioen and Appleton [85] build a probability graph where the nodes are files and arcs represent a file being accessed after the previous one. This probability graph generates first-order predictions based on files accesses. This allows their system to prefetch a file that is predicted to be used in the near future based.

Lei and Duchamp [126] used a similar approach to Griffioen and Appleton [85]. They monitored the files used by each program to build an *access tree* that encapsulates the programs file reference behaviour. Several access trees are stored for each program. When the current access tree matches a previous tree (above a pre-set threshold), the stored tree is used to prefetch all of the files that are anticipated to be needed for the program (up to a pre-set limit, `PREFETCH_CAPACITY` with default 15). The approach of Lei and Duchamp is interesting in that they are able to better exploit recency effects to give more benefit to more recent access trees. Probabilistic approaches such as the compression-based predictors tend to treat all reference patterns equally, without an inherent mechanism to age stale model.

Acharya, Franklin and Zdonik [3] used a simple prefetching heuristic to perform prefetching from a broadcast disk. Their approach was based on the estimated probability of a page being accessed in the future and an estimate of the time that would elapse before the page is again available Their probability estimates were based on zero-order estimates, namely the relative frequency of access for pages in the cache.

A number of other researchers have followed the approach of Curewitz, Krishnan and Vitter [48] in using a PPM-style predictor. Bartels et al. [14] implemented a PPM-style predictor in a Unix kernel to prefetch from a network of memory servers. Bartels et al. [14] found that the choice of the order m of the PPM model is important. They showed the unintuitive result (discussed in Section 5.2.1.2) that a longer context order may make worse predictions.

Madhyastha and Reed [130, 131] used neural networks and a hidden Markov model (HMM) to predict future client behaviour. The class of hidden Markov models is strictly stronger than either the order-k or even variable order Markov models that we employ (although order-k models can approximate hidden Markov models arbitrarily well with increasing k). There is a concern that the models generated by Madhyastha and Reed may grow quite rapidly; while they noted at most a linear growth in practice, they also employed pruning techniques to prevent excessive model growth. Madhyastha and Reed state that typically a file is accessed with only one type of pattern; in this case, they train a HMM with one state per disk block. Alternatively, if multiple access patterns are used for a single file, Madhyastha and Reed suggest building a composite HMM that combines the results of separate HMMs built for the distinct access patterns.

White and Skadron [192] used a prefetching approach based on a target cache technique used for indirect branch prediction in processor implementations. For each open file for a process, the operating system maintains a fixed-size structure. This structure maintains the addresses of the last n fetches submitted by the application, using this value to access the target cache. This can be seen to be an approximation of an order-m Markov model, but the implementation is designed to work well with a small amount of memory and time needed at each step.

Deshpande [56] also found that (as discussed in Section 5.2.1.2) it is difficult to choose a specific k. Instead, they simultaneously build separate order-k models for each possible k. Then, they prune this global model using a support threshold Φ. If a state has not been observed at least Φ times, it is pruned. Further, they use confidence intervals to detect states that do not provide a significant difference in predicted probability; these states are also pruned. The confidence intervals are built based on a configured confidence level α and the Wald approximation (Appendix A).

Drapeau, Roncancio and Guerrero [58] proposed using data mining techniques to find association rules for prefetching in WWW meta-searchers. An off-line process identifies frequent item sets to build a prediction model using earlier requests to predict future requests.

Predicting future object references based on patterns of object references can give very precise answers as to which objects should be prefetched if a pattern is correctly matched. However, all approaches that predict future object references based on past patterns of object references introduce a number of practical problems. First, the storage requirement is proportional to the number of objects referenced. If we limit the amount of storage available, then we reduce the effectiveness of the prediction. If we maintain this pattern information on the client machine, then the time and space problems may not be acceptable. Second, patterns based on object references do not generalize to semantically similar patterns that operate over different objects.

Wang et al. [186] explored how idle workstations can be used to make prefetching based on a PPM-style compressor effective. The idle workstations offload the work of maintaining the PPM model, allowing a higher order model to be used.

Kroeger and Long [118, 119] also used a PPM model to predict future accesses. However, the prediction is done at the level of files, not data blocks. In this way, the size of the model is significantly smaller. For example, their approach could recognize that certain files are accessed after a specific `Makefile` is opened by program `make`.

Yeh, Long and Brandt [201] also considered file-based prefetching. They considered the benefit of augmenting an order-1 model with the program and user that caused a file to be opened. In this way, they were able to achieve a 20% improvement over an unaugmented order-1 model. Further, they considered prefetching more than one file at a time (referred to as ‘deep’ prefetching in Section 5.3.1.2). In their configuration, they found that prefetching 2 files was most effective.

The prediction quality of predictors based on training is a concern. Amer et al. [10] suggest reducing the number of prefetches in order to increase the number of useful prefetches. Brandt [23] proposed a way to use a prediction based on a combination of multiple predictors. Brandt included a null predictor so that prefetches would not be issued if they are not useful.

Kraiss and Weikum [114] used a continuous-time Markov-Chain model to assess which documents to prefetch from a large off-line media library. They found that the precision afforded by a Markov-Chain model was significantly better than simpler first-order statistics. However, they found that the models grew to a few megabytes, with predictions taking on the order of milliseconds. Therefore, they cautioned about using such an approach directly for prefetching disk or memory pages.

In summary, existing work on prefetching based on request patterns considers patterns that have occurred in the individual items fetched. A request model is generated based on these individual items and used to make prefetching decisions. Since these prefetching approaches build a model on the individual data items previously fetched, they need storage proportional to the number of distinct objects in order to provide comprehensive predictions (although skew in the frequency of object access may reduce this space substantially). For example, consider a disk-based linked list where each page has a link to the next page. If the client traverses this list, the predictor needs storage proportional to the number of nodes in order to make the best possible predictions. If new nodes are inserted, the prefetching will fail until the model is updated with the new relationships.

9.1.3 Semantic Prefetching

The prefetching approaches based on on physical layout and request patterns does not consider the semantic meaning of the objects which the application is requesting. In fact, these objects contain a great deal of meta-information which could be used to predict future requests. In the disk-based linked-list example we described in the previous section, the server can predict the next node to be accessed if it understands the semantics of the data items. Further, this prediction does not rely on an expensive training period and large model that becomes easily obsolete.

Section 9.1.3.1 describes approaches based on static analysis of object attributes and relationships. These approaches use heuristics that provide prefetching hints without necessarily considering past application history. Section 9.1.3.3 describes how application hints have been used to implement prefetching. These approaches rely on deriving application-specific hints that guide prefetching and caching decisions. Section 9.1.3.2 describes proposed hardware-based value predictors that are used for speculative execution in explicitly multi-threaded processors. Section 9.1.3.4 outlines approaches based on models of request behaviour, where the model is built using the pattern of object types accessed instead of the pattern of individual objects.

9.1.3.1 Static Analysis of Attributes

Ammar [11] considered prefetching in a teletext system. The system presented a hierarchical tree of pages to users. Ammar proposed that the client terminal could decode the links in a page, estimate the probability of each link being taken, and use these to prefetch pages before the user requested them. In this setup, the client terminal was not able to prefetch future pages until a prior demand fetch had completed. The proposal of Ammar [11] anticipated prefetching approaches that have been proposed and implemented for the world-wide web (WWW). Davison [50] provides a summary of prediction approaches used for web request patterns.

Keller, Graefe and Maier [103] implemented an assembly operator which loaded the attributes of demand-fetched objects in a breadth-first search fashion. While this approach might be more expensive to implement than the approach based on the physical layout, Keller, Graefe and Maier described new execution techniques used to reduce the cost of assembling the objects. In order to prevent massive blowup due to breadth-first search expansion, the assembly operator relied on developers specifying predicates using UDFs that encode which objects should be assembled.

Day [51] suggests using objects as the unit of transfer from the server to the client. In Day's approach, a breadth-first traversal of object attributes is used to find a prefetch group[2], limited by a constant number of objects. He also considered alternative strategies based on clustering (a simulation of paging approaches), depth-first search, and application hints. Further, Day's approach prefers to send objects already in the server cache and likely not in the client cache. Later, THOR was changed [129] to use a sub-segment prefetching approach that sends a group of objects located on the same disk segment as the requested object. This approach allows the client to specify the size of the prefetch group. This approach was chosen because the object-graph approach [51] generated small prefetch groups, particularly near the edges of the object graph.

Kossmann, Haas and Ursu [108] observed that a common style of access for client applications is the 'fetch then manipulate' pattern. Applications fetch objects using a declarative query, then manipulate their attributes and methods. A naïve implementation would require a demand fetch for objects returned by the query. Instead, the authors proposed allowing the query to return attributes that were not requested by the application. The benefit of returning extra attributes is that they can eliminate demand fetches. However, this approach may cost more to execute, for example if the results are sorted then there is additional data to be included in the sort. The authors propose that this cost and benefit should be included during the join enumeration in order to find the most effective cache loading plan. The approach suggested could provide one level of prefetch, but did not consider fetching further objects or attributes.

The idea of prefetching state reachable from a demand-fetched object is more likely to find relevant items to prefetch than an approach based solely on physical organization. However, this comes at the price of increasing the cost of prefetching, since the objects referenced in the breadth-first search are likely not 'close' to the requested object. Further, the approaches in this section did not consider how likely it is that various attributes would be used or relationships followed. This could lead to a significant amount of wasted prefetching.

9.1.3.2 Value Prediction for Speculative Execution

Several modern processors are able to speculatively execute instruction in anticipation of their being needed. For example, instructions may be fetched and decoded in anticipation of a particular branch being taken. In some cases, this speculative execution is not directly possible because values produced by the executing program are not yet available. For example, a load instruction may refer to a memory location that is not yet computed. To increase the amount of instruction-level

[2] Day suggests *pre-sending* would be a better term, since the server determines the data to send. However, he uses the more traditional *prefetching* term.

parallelism (ILP), a number of researchers [72, 181] have investigated techniques that use hardware predictors that give a predicted value for each needed quantity. For example, a quantity may be predicted to be equal to the last value that was observed. This prediction can provide good accuracy when there is locality of reference. In some cases, the needed quantity is predicted to be the previous value combined a fixed delta. This prediction applies when an application uses strided access to data. The value prediction used for hardware speculative execution is necessarily simple because the prediction is performed in hardware for a large number of quantities.

9.1.3.3 Application Hints

One way to achieve accurate prefetching is to use hints from the client application. These hints may be either a guess of future pages that might be accessed (leading to *speculative prefetching*) or an accurate list of pages that will be used (leading to *informed prefetching*). Patterson et al. [144] described the TIP system, an implementation of informed prefetching and caching that uses hints from client applications to decide how to prefetch and cache. The TIP system use a cost-based model to dynamically allocate buffers between the competing needs of caching and prefetching.

The approach of manually inserted hints may make for effective prefetching decisions, but it does require a substantial burden from the system developers. Mowry [137] proposed using compiler modifications to automatically insert prefetch hints. Using the SUIF research compiler, Mowry was able to achieve improvements of up to a factor of 2 by automatically inserting prefetch instructions.

Cao et al. [29] noted that, even with informed prefetching, the decision of what prefetches to form is non-trivial. They allow applications to provide prefetching and caching hints, then employ an integrated caching and prefetching strategy which they show to be near optimal.

Chang and Gibson [35] used idle cycles when a workstation is waiting for an I/O to complete in order to speculatively execute past an I/O stall. They used this speculative execution to predict future reference behaviour. By providing these predictions to the TIP system [144], they achieved speculative prefetching.

Specific algorithms can also be tuned to provide hints of future references. Chen, Gibbons and Mowry [39] showed how prefetch instruction can be inserted to speed index scanning routines. Chen et al. [40] extended this by giving a fractal data layout that improves the latency from disk to memory and from memory to processor cache, while Chen et al. [38] described how to improve hash join algorithms by prefetching data from main memory into a processor cache. This reduces the significant amount of time that the CPU spends in cache stalls.

A recurring theme in database and operating systems research has been the idea of designing algorithms and data structures that exploit sequential access. One of the strengths of the relational model proposed by Codd [45] is that it permits an efficient access method to be selected based on the current query, rather than based on a single common usage pattern. Chamberlin et al. [34] described how System R used a cost-based model to select an access method for each relation, exploiting sequential access where costs warrant. More generally, Graefe [81] provides an overview of several algorithms in the database field that are designed to maximize sequential access to disks. In part, these algorithms are faster due to the inherently cheaper sequential access; in part, this speedup is due to the benefit they achieve from effective prefetching.

9.1.3.4 Type Reference Patterns

As noted above, models based on sequences of object references do not generalize to semantically identical patterns over different objects. The heuristic approaches described in Section 9.1.3.1 consider the data items being fetched, and use this to drive heuristics predicting future requests. This approach, however, does not adapt to application-specific request patterns that don't match the selected heuristic. The prefetch hints described in Section 9.1.3.3 allow this type of adaptation, but at the cost of application complexity. Another solution is based on a model of client reference behaviour when considering the object types being fetched instead of considering individual object identifiers. This approach learns patterns at the intension level rather than the extension level. Typically, such models are smaller and more generalizable than corresponding models based on object identifiers.

Knafla[106, 107] considered the probability of an application navigating each of its relationships to other objects using a Markov-Chain model. This statistical model gives estimates of the probability of accessing subsequent pages. If the probability of accessing a page is high enough, it is prefetched to the client.

Bernstein, Pal and Shutt [19] suggested that the *context* in which an object was fetched is an important factor to consider when deciding which related objects should be prefetched. For example, consider a query that returns a list of objects $a_1, a_2, \ldots, a_k$. If we next see a request to fetch the x attribute of object a_1, we may reasonably assume that we will soon fetch the x attribute for a_2 and the rest of the a_i in the list. This is much more likely than if a_1 and a_2 were fetched in different contexts, for example from different queries. The solution proposed by Bernstein, Pal and Shutt attempts to discover operations that should be applied to multiple objects, and uses this as the basis for prefetching.

Han, Moon and Whang [89, 90] note that the approach of Bernstein, Pal and Shutt [19] may perform poorly if an application uses a depth-first traversal scheme. In that case, objects prefetched near the top of the access tree may be evicted before traversal returns to use them. Han, Moon and Whang propose PrefetchGuide, an extension to the context-based prefetching of Bernstein, Pal and Shutt. The PrefetchGuide data structure also considers patterns in the types of objects referenced, tracking collections of objects returned by queries and by attribute references. In this way, it is able to detect iterative and recursive patterns. The iterative patterns are similar to the nested request patterns that we optimize (Chapter 3). The PrefetchGuide structure is built every time that a top-level query is used, and discarded when that query is closed. PrefetchGuide prefetches individual objects as it observes the client application submitting prefetches.

Bowman and Salem [22] described a semantic prefetching approach based on a recognition of nested request patterns of the type we describe in Chapter 3. As described in Chapter 3, this approach recognizes nested patterns that are equivalent to a distributed join. In contrast to the work of Han, Moon and Whang, this approach replaces all of the fetches for a nested pattern with a single join query, exploiting the relational processing power of the server DBMS.

Yao and An [198–200] propose a system called SQL-Relay, which consider sequences of queries in an OLTP or OLAP environment. They replace all literal constants to form a query template, and represent a client application by a probabilistic model they call *user access patterns*. This model contains states and edges annotated with queries and probabilities. States are identified with query templates, thereby forming an order-1 model. Further, the user access graph predicts values that will be used for parameters; these may be predicted to be a constant, input parameters of previous requests, or result attributes of previous requests. Yao and An [198, 199] show how user access patterns can be used to guide prefetching. They considered three types of query rewrites for prefetching: sequential, which prefetches v when u is submitted; union, which generates a union query $u \uplus v$ to prefetch v when u is submitted; and, probe-remainder, which submits a modified u' that retrieves results for u and some of the results needed for the predicted v query. For the probe-remainder approach, and additional ‘remainder’ query v' is required to retrieve the rest of the results for v. Both the union and probe-remainder approaches are only used when queries u and v fetch from the same relations. Further, a query v that depends on result attributes of u cannot be prefetched until the results for u are returned (recall that Scalpel accomplishes this with a join-based rewriting, as described in Section 5.4.2). Yao, An and Huang [200] also show how to use data mining techniques including n-gram modeling and sequence alignment to identify database user sessions boundaries.

Bilgin et al. [20] also considered prefetching the results of anticipated future requests. Bilgin et al. described a procedure that, if provided with a probabilistic model of a client’s data access graph, chooses a set of read-ahead queries that minimize the expected running time of the application. A dynamic programming algorithm is used to select the optimal strategy. The work Bilgin et al. [20] present so far considers only the optimization problem, relying on external modules to generate the combined queries and decode the results from the prefetched queries. In their work, they also considered prefetching ‘deep’ as Scalpel does.

9.1.4 Summary of Prefetching

Prefetching mechanisms are widely used today to limit the latency associated with demand fetches. There are several approaches to predicting the future requirements of applications, ranging from simple patterns (such as sequential fetching) to more complicated approaches that monitor application behaviour to train a predictive oracle. Alternatively, heuristics have been used to generate statistical models of application behaviour in order to predict future behaviour.

The existing prefetching mechanisms can work effectively for particular workloads. However, they do not exploit an important observation: we may be able to encode fetches using a relational operation such as join. If so, then we can exploit this realization to predict the future access pattern of the application, or even *rewrite* the requests using relational equivalences to use a more efficient join-based strategy.

9.2 Theoretical Underpinnings of Model-Based Prediction

Prefetching is based on a prediction of future items that are likely to be needed. This prediction is often based on heuristics (such as sequential prefetching), but it has also been implemented based on probabilistic models of the client based on experience. This modeling is an example of machine learning, characterized by Laird and Saul [122] as discrete sequence prediction.

Shannon [168] provided seminal results on probabilistic characterizations of sequences. His work was revolutionary to the broad field of information theory [15, 33, 49, 60, 61, 63, 69, 97, 122, 123, 127, 128, 133, 134, 148–151, 159, 188–190, 193, 210–214]. This field is concerned with the theoretical modeling of sequences of symbols. As such, it provides a theoretical underpinning for predictions based on a prior sequence, such as that used by Scalpel for batch request patterns.

Data compression has been used as the basis for several prefetching schemes. The reason for this is due to the fact that a good data compressor will form a good predictor of future symbols. Research on practical compression schemes therefore provides an excellent basis for prefetching algorithms that are able to operate efficiently in space and time and produce good predictions. This approach leverages the large body of excellent work on model based compression [17, 25–28, 41–43, 46, 71, 135, 138–140, 150, 178, 188, 189, 193, 195, 196, 204, 214].

In particular, the PPM algorithm introduced by Cleary and Witten [43] is the closest to our work. Moffat [135] provided a careful implementation of PPM giving an efficient implementation that produced quite good compression. The PPM algorithm relies on a bounded length suffix trie, choosing to use prediction contexts that are long enough to match special cases while avoiding the zero-frequency problem. Cleary, Teahan and Witten [41, 42] extended the PPM algorithm to PPM*, which stores unbounded length contexts. Larsson [124] showed how the algorithm of Ukkonen [180] can be extended so that a sliding window of text is used. Bunton [28] showed how path compressed suffix tries can be extended to maintain counts used for probability estimation: unfortunately, this introduced a worst case $O(n^2)$ time component.

The models designed for information theory and data compression are concerned with providing precise estimates of the probability distribution of the next symbol given a preceding sequence. The precision of this estimate is measured by entropy, and even small errors lead to inefficiencies in the compression or coding results. In contrast, Scalpel does not need a precise probability estimate; instead, it merely needs to know if prefetching a future query is significantly cheaper (at the α level). Further, data compression techniques for the zero-frequency problem form an important area of research, for example resulting in several variants of the PPM algorithm that implement different definitions of *escape probabilities* to handle novel characters in a state. This issue does not affect Scalpel to the same extent as it can choose merely to avoid prefetching in such a situation.

The field of language inference also deals with forming models of strings [1, 2, 7, 12, 31, 32, 53, 59, 62, 74, 91, 92, 95, 125, 138–140, 146, 153–157, 171, 179, 203–206]. In general, the idea of language inference is to build a model of a language from a set of examples. We are particularly interested in the recognition of stochastic grammars, which give predictions of likely future symbols. This recognition is commonly accomplished by building a representation of the sequence provided during training, then using state merging to combine states where the behaviour is sufficiently ‘similar’.

Carrasco and Oncina [30–32] described an algorithm call ALERGIA (and variants thereof) based on a building a prefix tree structure from a training sequence, then using state merging based on similarity measures. A configuration parameter, α, is used to control how different two nodes must be to avoid merging. In the worst case, the algorithm runs in $O(n^3)$ time.

Young-Lai and Tompa [203, 206] observed that the ALERGIA algorithm has particular difficulty with nodes that have been observed few times. They noted that the merging criteria used in ALERGIA is not well founded, and added another configuration parameter β to control the type-II error during merging.

Our confidence-level approach is related to the setting of α and β above. When Scalpel finds there is not sufficient information to make a prefetching decision, however, it is able to consider a more generalized context or (safely) decide to make no prefetching decision. The consequence of type-II errors are therefore lessened in Scalpel’s environment.

9.3 Suffix Tries

The suffix trie data structure is a trie [73] built for all of the suffixes of a string. The suffix trie for a string contains significant redundancy. Path compression is a general approach for reducing the size of a trie. It was first introduced by Morrison [136] as the *Patricia tree* data structure, and this term is often used in the literature to describe path compressed tries in general. Path compression can be used for suffix tries, and the resulting structure is called a suffix tree (although we feel that path compressed suffix trie is a more reasonable name). It is perhaps surprising that a suffix tree can be built for a string in linear time; McCreight [132] attributes this discovery to Weiner, and McCreight improves on the algorithm presented by Weiner, saving about 25% in space. The algorithms of Weiner and McCreight are very important in giving efficient construction algorithm for an important structure that is pervasively useful in the string processing field Gusfield [88]. However, these algorithms were considered to be overly complex, and they were not widely used. Many typical uses of suffix tries consisted of using atomic suffix trees, or asymptotically more expensive construction algorithms.

Ukkonen [180] presented a novel implementation of a linear algorithm to build a suffix tree that is significantly simpler than the two previous approaches. The key observation used by Ukkonen is the use of *open edges*. We refer to this as the ∞ trick in Section 5.2.3. This trick is fundamental to the simplification provided by Ukkonen. In fact, Giegerich and Kurtz [80] conjecture that if Weiner had seen this trick, he would have implemented the simpler algorithm in 1973.

Bunton [28], demonstrated how path compressed suffix tries can be used to predict the probability of future characters in the PPM* algorithm, and de Rooij [54] presented similar results. In both cases, the maintenance of count fields on-line during construction of the suffix tree leads to a worst-case $O(n^2)$ time complexity. We avoid this problem in our approach by using end-of-trace markers ($). In this way, the number of leaves below a node provides the needed count.

9.4 Processing Sequences of Queries

If we know a sequence of queries that are to likely to be submitted, we can consider a variety of ways to efficiently produce the needed results. For batch request patterns, Scalpel generates a rewritten query using outer join and outer union constructs. Section 9.4.1 describes related work that can produce efficient results for this type of sequence of requests.

Scalpel also considers nested patterns of requests. When Scalpel detects what it believes to be a distributed join implemented in the client application, it generates a combined query using outer joins, outer unions, client merge join, and client hash join strategies. Other researchers have considered the problem of how to efficiently process a sequence of queries that involves nesting.

9.4.1 Batch Request Patterns

Sellis [160] presented early results that considered optimizing a known sequence of queries together, instead of one query at a time. In this way, the optimizer is able to choose locally suboptimal access plans that, through sharing, give a globally optimal plan. Sellis and Ghosh [161] showed that this optimization problem is, in general, NP-hard. However, they suggested heuristics that in general give results that significantly improve on the naive approach. The idea of multi-query optimization has been hampered to an extent because in current systems, the DBMS does not have a good idea of future requests that will be executed. The query sequence detection of Scalpel therefore provides a nice complement to multi-query optimization.

In some cases, multi-query optimization can be used because of the special structure of requests generated by the client. Kraft et al. [111–113] found that some OLAP tools intentionally generate a sequence of queries to answer a single user request. In part, this sequence is designed to limit the complexity of individual requests to avoid overloading the DBMS optimizer. Kraft et al. suggest coarse grained optimization, which uses heuristic rewrites to optimize this (known) sequence of queries.

9.4.2 Nested Request Patterns

A number of researchers have studied how to effectively execute queries that contain various forms of nesting [52, 64, 65, 77, 98, 104, 105, 162]. The approaches developed in that work are effective at choosing efficient evaluation plans for the correlated combined queries that we generate. However, the techniques are not directly applicable to the problem we consider because the nesting appears in the application, not the queries.

Florescu et al. [70, 197] translate web-pages defined in a declarative language including queries. In this approach, the authors are able to detect the source of binding parameters when a nested query is invoked. A single nested query may be used in more than one context; for some contexts, simplifications can be used to improve the performance of the nested query. A nested query can be simplified if the analysis can detect that a tuple from the outer query will appear in the results of the inner query. The authors term this rewrite *query simplification under preconditions*. For example, a nested query may include a $1:1$ join that retrieves attributes already available from a prior query. In this case, we can eliminate the join, using a single-table query to retrieve the new values. If there are multiple contexts of execution for the inner query, we may be able to perform the optimization in only some of these contexts. Further, the authors propose modifying queries in order to reuse their results in future queries. They identify a *conservative* approach which does no extra work but does include additional attributes not present in the original query, and an *optimistic* approach which additionally performs outer joins which load results needed by subsequent operations. The primary focus of this work was to avoid re-evaluating common expressions in order to improve the performance of generating dynamic web content. The issues of fine-grained access is not specifically addressed.

Shanmugasundaram et al. [166] and Fernández, Morishima and Suciu [67] studied efficient mechanisms to generate nested XML results from relational data sources, and this was studied further by Krishnamurthy [115] This work is similar to our work on combining nested queries, although it differs in that the structure of the nesting is known from the XML query (we infer this structure during a training period). Further, the combined result set is used to encode an XML result, while we decode the combined result set to generate the original nested relational results.

Shanmugasundaram et al. [163–167] considered efficient ways to translate XML queries into queries over a relational data model. In addition to stored procedure and correlated CLOB approaches (which do not appear to apply well to our problem), they considered outer union and outer join strategies. They called the outer join strategies *redundant relations* due to the processing and data redundancy introduced when unrelated children are combined with an outer query, and they did not consider it further after initial results found it to perform poorly.

Fernández, Tan and Suciu [68] introduced SilkRoute, a system that maps relational data to XML views. Initially, SilkRoute used a scheme similar to our client merge join. Individual queries were ordered and merged at the client. Fernández, Morishima and Suciu [67] extended this approach to also consider the outer union approach suggested by Shanmugasundaram et al. [165]. They also rehabilitated the outer join approach, using it when the inner query returns at most one row (therefore not introducing redundancy). They proposed optimization based on a view-tree structure to decided on which strategy would be used for each portion of the nested sequence. Our work extends the *view tree reduction* of Fernández, Morishima and Suciu, a heuristic that combines all at-most-one-row queries with their outer query using joins. In our work, we choose the queries to join together on the basis of a cost model that accounts for the effects of local predicates that can appear in client application.

The combined queries generated by Shanmugasundaram et al. and Fernández, Morishima and Suciu [67] moved correlated predicates from inner queries to the ON-condition of an outer join (an example is shown if Figure 3.14). In contrast, we use a LATERAL derived table construct. The LATERAL derived table is more general, allowing the inner query to use correlations in any location.

Other researchers have considered how query results can be fetched before the query is actually requested. Sapia [158] proposes the PROMISE system, which uses a model of user behaviour to predict future OLAP queries that might be submitted. Prediction is made based on a prediction profile that abstracts details of queries to detect higher level patterns. The prediction profile can either be defined by domain experts, or possibly from an analysis of query logs (although this process is not described). The predictions are used to aid in caching decisions and to prefetch data into a cache.

The research on non-first-normal-form (NF^2) query languages in general [98] and XML in particular is pertinent when we consider retrieving the encoded, nested result set that is used by our approach. Instead of encoding the nested result in an outer union or outer join, the result could be directly expressed in a nested relational language. However, this research does not directly help our particular problem due to the presence of local predicates. If these predicates are highly selective, then it is better to submit nested queries than a single decorrelated query.

10 Conclusions and Future Work

Latency is increasingly becoming a significant factor in database applications. Communication latency lags significantly behind advances in individual process components, increasing the relative importance of latency. Further, adaptation of existing database applications to new high-latency environments such as wireless access and WANs leads to higher absolute latency.

Latency is increasingly becoming an important factor for database applications, and this problem is exacerbated by fine grained access. We have studied a number of applications, and we present results for two of these in Chapter 8. All of the applications we studied had some sequences of related queries, which we call batch request patterns. For typical configurations, many of these queries are quite cheap with respect to the per-request latency U_0, even for low-latency local shared memory configurations. In addition to the prevalent batch request patterns that we observed, we also found that there are examples of nested request patterns in existing database applications. These nested patterns do not occur as frequently, but they can account for significant latency due to the number of inner requests that are submitted.

While it is possible in some cases to rewrite applications manually to avoid generating these fine-grained access patterns, such rewrites are complicated by a number of factors. One of these is the fact that a nested approach is in fact optimal for some configurations of the application program (for example, see Figure 3.30). In the cases where such optimal configurations are expected to occur in the majority of client deployments, it is prudent to choose the nested implementation, which is in any case easier to implement. Further, good software engineering practices may argue against the removal of nesting, as such removal might lead to the destruction of important properties such as code encapsulation.

Instead of manual tuning, we have presented a system, Scalpel, which automatically detects fine-grained nesting and batch request patterns. Scalpel uses a deployment-time training period that monitors a request stream to automatically detect predictable patterns of queries. Further, Scalpel maintains correlation information that is needed to predict the values that will be used for future requests.

After the training period, a cost-based optimizer is used to choose a prefetch strategy. Scalpel leverages the query processing capabilities of the DBMS to generate a prefetch request that fetches results for an original query and also for a list of predicted future queries. The relational processing power is able to prefetch the results for requests that depend on the results of previous requests, a capability that is not available to prefetching systems that do not consider request semantics.

10.1 Contributions

We have presented techniques for recognizing nested request patterns. Even without prefetching, this recognition may prove useful to application developers, enabling them to identify areas for improving their application. Nested request patterns are recognized not only by the nested structure of requests, but also by correlations between input parameters of the inner query and attributes of the outer query.

We have developed four approaches to efficiently combine the nested request patterns that we detect, based on the following: outer joins, outer unions, client hash joins, and client merge joins. Our combined queries are implemented using the `LATERAL` construct (and its obvious extension `LEFT OUTER LATERAL`). In this way, the combined queries we generate closely match the nesting structure that was implicitly present in the client application. In contrast with other approaches that are based on clever tricks such as moving correlated predicates into the ON condition of an outer join, our approach is general in that it supports correlation anywhere within the inner query. While our approach does generate complex nested queries, this complexity merely exposes the original complexity that was previously hidden in the application code. In some cases, the DBMS may be able to select a more efficient execution strategy for this request.

We showed how suffix tries can be used to find batch request patterns. Techniques such as suffix tries have been used for prefetching in the past, but we extend these with the ability to track correlations between input parameters and attributes of preceding requests.

We extended our suffix trie detection to path compressed suffix tries. In this way, we provided a linear time algorithm that maintains a set of probabilistic predictions of future requests, combined with a set of predicates that have always been true in the past. This algorithm is useful for efficiently detecting batch request patterns. It may also prove useful in other situations where we wish to efficiently learn a set of predicates that have always been true given a variable-length conditioning context.

We presented two techniques that can be used to prefetch the results of anticipated future queries, based on outer joins and outer unions respectively. These techniques generate combined queries that are efficiently executed by the DBMS and easily decoded to provide the original result sets.

Finally, we have presented experimental and case study evidence that the proposed techniques are practical and useful for existing database applications.

10.2 Future Study

The work that we have presented has identified a number of areas that deserve further study.

Our original study of applications also identified data structure request patterns. In this pattern, an 'outer' query is opened and its results are stored in an application data structure such as a linked list; then, the 'outer' query is closed and an 'inner' request is submitted for the rows in the data structure. The order of execution of the inner query does not necessarily match the original order the rows were fetched in. These patterns are something of a combination of nesting and batch request patterns. It remains an open question whether these patterns can be efficiently detected. When found, they can be executed using techniques similar to those used for nested request patterns.

Another topic for future reflection is our choice of using an explicit training period. In some respects, it would be better to have a dynamic implementation that continually adapts to changing configuration parameters. We chose to explicitly separate the training period for simplicity of presentation, but we might consider combining the training and run-time phase. In such a combined situation, we would be much more concerned about training costs. In particular, suffix tries used for batch pattern detection would need to be implemented efficiently in a smaller amount of memory.

The idea of integrating Scalpel into a system that provides semantic caching is appealing. Scalpel could be used not only for prefetching anticipated requests, but also for providing suggestions to a cache manager of the expected utility of each cached item. Such an integration would also provide a stronger basis for the issues of data consistency that are faced by Scalpel. At present, Scalpel provides results that are correct and consistent provided that a full ACID DBMS is used (or at least snapshot isolation). However, this consistency is achieved at the expense of foregoing prefetching opportunities where it is not provably safe. For example, Scalpel does not currently prefetch across transaction boundaries, although we have identified applications where that would be useful.

Finally, one topic that should be studied more thoroughly in the future is how a database application should be divided between a client process and DBMS. The relational model provides only slight guidance with this: the limitations of SQL pose certain limits on what can be executed in the server process. There is nothing, however, limiting what operations are performed in the client application. As we have seen, current client applications implement the equivalent of distributed joins and unions of results. It is clear that a client application can implement all of the relational operations, merely using a DBMS as a table store. We can consider automated tools that detect specific logical operations that the application performs, such as our recognition of nested and batch request patterns. Alternatively, we could consider 'cutting up' an application so that portions with fine-grained access to the data stored in the DBMS are executed in the server process, while the remained execute in the client process. In fact, this concept of cutting up an application was the source of the name of our system: Scalpel.

A Confidence Intervals

It is perhaps surprising to non-statisticians that there is active study on the topic of forming a confidence interval for a binomial parameter p. The approach described in most introductory textbooks is based on the asymptotic normality given by the central limit theorem. Let X be the number of times a test is true out of n trials. Then, $\hat{p} = X/n$ is an estimate of p. The interval CI_S defined in Equation A.1 is a $100(1-\alpha)$% confidence interval for p, where z_c is the $(1-c)$th quantile of the standard normal distribution.

$$CI_S = \hat{p} \pm z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})} \tag{A.1}$$

This definition of a confidence interval is one of the oldest; for example, Agresti and Coull [6] attribute its use to Laplace in 1812. The interval CI_S is called the *Wald confidence interval* for p because it is based on inverting the Wald hypothesis test for p. The interval CI_S is the set of values p_0 having P value exceeding significance level α in testing the null hypothesis $H_0 : p = p_0$ against the alternate hypothesis $H_a : p \neq p_0$ using the following test statistic:

$$z = \frac{\hat{p} - p_0}{\sqrt{(\hat{p}(1-\hat{p})/n}} \tag{A.2}$$

The points p_0 in the interval are those for which we cannot reject the null hypothesis (at the α level of significance).

The confidence interval CI_S is simple to compute and easy to motivate, which is why it is traditionally used in introductory texts. However, as noted by Brown, Cai and DasGupta [24], the standard interval tends to generate an interval that provides coverage probability rather lower than the nominal significance level; this is partly due to the approximation of the central limit theorem being somewhat weak with low n, but it is also a result of the discrete nature of the measured quantities. Better introductory books do provide some guidance that the Wald interval should only be used in some circumstances. For example, some suggest only using it if $n > 30$, others also caution that np and $n(1-p)$ should not be close to zero. In fact, Brown, Cai and DasGupta [24] show that none of these cautions adequately captures the erratic coverage properties of the standard interval. Even with large n and p relatively far from the end-points, the standard confidence interval can give results that fall seriously short of the stated significance level.

When guidelines for using the standard interval are not met, advanced textbooks refer the reader to an interval defined by Clopper and Pearson [44]. This interval is typically referred to as the ‘exact’ interval, and it has often been considered the gold standard of confidence intervals for binomial parameters. The Clopper-Pearson interval has endpoints that are solutions to the following:

$$\sum_{k=X}^{n} \binom{n}{k} p_0^k (1-p_0)^{n-k} = \frac{\alpha}{2} \tag{A.3}$$

$$\sum_{k=0}^{X} \binom{n}{k} p_0^k (1-p_0)^{n-k} = \frac{\alpha}{2} \tag{A.4}$$

(except that the lower bound is 0 when $X = 0$ and the upper bound is 1 when $X = n$). This interval is formed by inverting the equal-tailed binomial hypothesis test of H_0. The Clopper-Pearson interval is guaranteed to have coverage of *at least* $1 - \alpha$, which is why it is often considered to be better than the standard interval.

This assertion that the Clopper-Pearson interval is better than the standard is based on the assumption that it is better to be conservative, having an interval definition that never falls below the nominal coverage of the parameter α. However, this ‘exact’ interval achieves this guarantee by consistently generating intervals that are wider than necessary, giving a coverage larger than the nominal level. If the definition of a gold standard is that it always give at least the nominal coverage, we could as easily choose the unit interval $[0, 1]$.

This complaint has led to recent calls for using a confidence interval that generally provide coverage close to the nominal significance level. Agresti and Coull [6] suggest using an interval that they believe was first proposed by Wilson [194]. This interval is based on the score test for parameter p instead of the Wald test; the score test uses the log likelihood at the null hypothesis level of the parameter p rather than the maximum likelihood estimate $\hat{p}$ used by the Wald tests.

$$CI_W = \frac{X + z^2/2}{n + z^2} \pm \frac{z\sqrt{n}}{n + z^2} \sqrt{\hat{p}(1 - \hat{p} + \frac{z^2}{4n}} \tag{A.5}$$

The Wilson interval is shown in Equation A.5, where we let $z = z_{\alpha/2}$. It corrects two problems with the standard interval. First, the standard interval is centered about the ‘wrong’ point, $\hat{p}$. We can see that the Wilson interval is centered about the following weighted average:

$$\hat{p}\left(\frac{n}{n + z^2}\right) + \frac{1}{2}\left(\frac{z^2}{n + z^2}\right) \tag{A.6}$$

The mid-point falls between $\hat{p}$ and $1/2$, with the movement toward $1/2$ diminishing as n increases. This choice is a better mid-point for the confidence interval due to the skewed nature of the binomial distribution. The second problem that the Wilson interval corrects is that the standard interval is in fact too wide in general. The low coverage is caused by the wrong center. For these reasons, the Wilson interval appears certainly superior to the standard Wald interval. It can be considered an improvement on the Clopper-Pearson interval if we are willing to accept some cases where coverage falls below the nominal level.

The Wilson interval does have shortcomings of its own. First, it is relatively complex to motivate and present. More importantly, there is a small region $[0, r)$ where the coverage of the Wilson interval drops seriously below the nominal level. A solution to both of these problems is provided by Agresti and Coull [6]. Agresti and Coull make the following suggestion. Let $\tilde{n} = n + z^2$ and $\tilde{X} = X + z^2/2$. Let $\tilde{p} = \tilde{X}/\tilde{n}$. Equation A.7 shows the confidence interval suggested by Agresti and Coull.

$$CI_{AC} = \tilde{p} \pm z\sqrt{\frac{\tilde{p}(1-\tilde{p})}{n}} \tag{A.7}$$

In the case that we are choosing a 95% confidence interval, $z = z_{\alpha/2} = 1.96 \approx 2$. This gives $\tilde{X} = X + 2$ and $\tilde{n} = n + 4$, leading Agresti and Coull to call it the “add two successes and two failures” approach. The Agresti-Coull interval has the familiar form of the standard Wald interval, making it simpler to present. Further, it improves on the Wilson interval in that it avoids the region of seriously low coverage experienced by the Wilson interval. However, this improvement comes at the cost of being slightly conservative, with intervals that are slightly wider than the Wilson interval. We choose to use the Agresti-Coull interval due to its simplicity and the improvements in regions of p near 0 and 1.

In summary, the Clopper-Pearson confidence interval has traditionally been considered to be the gold standard (usually called the ‘exact’ interval) because it always provides intervals with coverage that is *at least* the nominal level, and usually more. This interval has not been traditionally popular because of the difficulty of computing the solutions to the endpoint equations. Instead, the Wald interval is typically suggested by statistics text books (with varying levels of guidance as to when it is appropriate). The Wald interval is based on a hypothesis test that uses the normal approximation with the sample standard deviation. This approximation has disturbing tendencies to produce erratic intervals that can fall alarmingly below the nominal coverage level, even with reasonably large n and p relatively far from 0 and 1. The problem is that the Wald interval is centered about the ‘wrong’ point, namely the maximum likelihood estimate $\hat{p} = X/n$. The Wilson interval corrects this centering problem by using the score hypothesis test instead of the Wald test. The Wilson interval is superior in all ways to the Wald interval, and it is superior to the Clopper Pearson test if we believe it is better to be generally close to the nominal coverage while occasionally falling below this nominal level. We use the Agresti-Coull interval, which is centered about the same point as the Wilson interval, but using a width that is generally slightly wider. This definition avoids small regions near 0 and 1 that cause the Wilson interval to fall below the nominal level. Further, this definition allows for a simple presentation and motivation.

Bibliography

[1] Naoki Abe and Manfred K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. In *Proceedings of the third annual workshop on Computational learning theory*, pages 52–66. Morgan Kaufmann Publishers Inc., 1990.

[2] Naoki Abe and Manfred K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9(2-3):205–260, 1992.

[3] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Prefetching from broadcast disks. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 276–285. IEEE Computer Society, 1996.

[4] Atul Adya. *Weak Consistency: a Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, March 1999.

[5] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering*, March 2000.

[6] Alan Agresti and Brent A. Coull. Approximate is better than exact for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, May 1998.

[7] Ahonen Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. PhD thesis, University of Helsinki, Finland, 1996.

[8] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transaction on Computing Systems*, 13(2):89–121, 1995.

[9] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement under Unix. *Software–Practice and Experience*, 27(1):1–23, 1997.

[10] Ahmed Amer, Darrell D.E. Long, Jehan-François Pâris, and Randal C. Burns. File access prediction with adjustable accuracy. In *Proceedings of the 2002 International Performance, Computing and Communication Conference (IPCCC)*. IEEE, 2002.

[11] Mostafa H. Ammar. Response time in a teletext system–an individual user's perspective. *IEEE Transactions on Communications*, COM-35(11), November 1987.

[12] Dana Angluin and Miklós Csürös. Learning Markov chains with variable memory length from noisy output. In *Proceedings of the tenth annual conference on Computational learning theory*, pages 298–308. ACM Press, 1997.

[13] ANSI. *Information Systems Database Language SQL*, September 1999. ISO/IEC 9075-1:1999.

[14] Gretta E. Bartels, Anna R. Karlin, Darrell Anderson, Jeffrey S. Chase, Henry Levy, and Geoffrey Voelker. Potentials and limitations of fault-based Markov prefetching for virtual memory pages. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 206–207. ACM Press, 1999.

[15] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order Markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[16] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[17] Timothy C. Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4), December 1989.

[18] Hal Berenson, Philip A. Bernstein, Jim N. Gray, Jim Melton, Patrick O'Neil, and Elizabeth J. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM Press, May 1995.

[19] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch for implementing objects on relations. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 327–338. Morgan Kaufmann, 1999.

[20] A. Soydan Bilgin, Rada Y. Chirkova, Timo J. Salo, and Munindar P. Singh. Deriving efficient SQL sequences via read-aheads. In *Data Warehousing and Knowledge Discovery: 6th International Conference, DaWaK*, volume 3181/2004, pages 299–308. Springer-Verlag, September 2004.

[21] Ivan T. Bowman. Architecture recovery for object-oriented systems. Master's thesis, University of Waterloo, 1999.

[22] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 179–190, 2004.

[23] Karl S. Brandt. Using multiple experts to perform file prediction. Master's thesis, University of California Santa Cruz, June 2004.

[24] Lawrence D. Brown, T. Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statistical Science*, 16(2):101–133, 2001.

[25] Suzanne Bunton. A characterization of the dynamic Markov compression FSM with finite conditioning contexts. In *Data Compression Conference (DCC '95)*, March 1994.

[26] Suzanne Bunton. *On-Line Stochastic Processes in Data Compression*. PhD thesis, University of Washington, 1996.

[27] Suzanne Bunton. An executable taxonomy of on-line modeling algorithms. Technical Report UW-CSE-97-02-05, University of Washington, 1997.

[28] Suzanne Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3), 1997.

[29] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 188–197. ACM Press, 1995.

[30] Rafael C. Carrasco. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2):207–216, 2002.

[31] Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Conference on Grammar Inference*, 1994.

[32] Rafael C. Carrasco and Jose Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO (Theoretical Informatics and Applications)*, 33(1):1–20, 1999.

[33] Nicolo Cesa-Bianchi and Gabor Lugosi. On sequential prediction of individual sequences relative to a set of experts. In *COLT' 98: Proceedings of the eleventh annual conference on Computational learning theory*, pages 1–11. ACM Press, 1998.

[34] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim N. Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *CACM*, 24(10):632–646, 1981.

[35] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[36] Surajit Chaudhuri, Prasanna Ganesan, and Vivek Narasayya. Primitives for workload summarization and implications for SQL. In *Proceedings of the 29th VLDB Conference*, 2003.

[37] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *Proceedings of the 26th International Conf. on Very Large Databases*, pages 1–10, Cairo, Egypt, September 2000.

[38] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. In *Proceedings of the IEEE International Conference on Data Engineering*, March 2004.

[39] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, May 2001.

[40] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM Press, 2002.

[41] John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3), 1997.

[42] John G. Cleary, W. J. Teahan, and Ian H. Witten. Unbounded length contexts for PPM. In *Data Compression Conference*, pages 52–61, 1995.

[43] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4), April 1984.

[44] C.J. Clopper and E.S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, December 1934.

[45] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[46] Gordon V. Cormack and R. Nigel Horspool. Data compression using dynamic Markov modelling. *Computer Journal*, 30(6):541–550, 1987.

[47] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[48] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 257–266. ACM Press, 1993.

[49] A.S. Davis. Markov chains as random input automata. *The American Mathematical Monthly*, 68:264–267, March 1961.

[50] Brian D. Davison. Learning web request patterns. In A. Poulovassilis and M. Levene, editors, *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*, pages 435–460. Springer, 2004.

[51] Mark Stuart Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, MIT, 1995.

[52] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 197–208. Morgan Kaufmann, 1987.

[53] Colin de la Higuera and Franck Thollard. Identification in the limit with probability one of stochastic deterministic finite automata. In *ICGI-2000 : 5th International Colloquium on Grammatical Inference*, Lisbon, September 2000.

[54] Steven de Rooij. Methods of statistical data compression. Master's thesis, University of Amsterdam / Institute for Logic, Language, and Computation, September 2003.

[55] Peter J. Denning. The working set model for program behavior. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[56] Amol Deshpande. Selective Markov models for predicting web-page accesses. *ACM Trans. Inter. Tech.*, 4(2):163–184, 2004.

[57] Jochen Doppelhammer, Thomas Hoppler, Alfons Kemper, and Donald Kossmann. Database performance in the real world: TPC-D and SAP/R-3. In *Proc. ACM SIGMOD Conference*, pages 123–134, 1997.

[58] Stephane Drapeau, Claudia Roncancio, and Edgard Benitez Guerrero. Generating association rules for prefetching. In *ICDCS Workshop of Knowledge Discovery and Data Mining in the World-Wide Web*, pages F15–F22, 2000.

[59] Pierre Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 25–37. Springer-Verlag, 1994.

[60] Yariv Ephraim and Neri Merhav. Hidden Markov processes. *IEEE Transactions on Information Theory*, 48(6):1518–1569, June 2002.

[61] Eleazar Eskin. *Sparse Sequence Modeling with Applications to Computational Biology and Intrusion Detection*. PhD thesis, Columbia University, 2002.

[62] Yann Esposito, Aurelien Lemay, Francois Denis, and Pierre Dupont. Learning probabilistic residual finite state automata. In Pieter W. Adriaans, Henning Fernau, and Menno van Zaanen, editors, *Grammatical Inference: Algorithms and Applications, 6th International Colloquium: ICGI 2002, Amsterdam, The Netherlands, September 23-25, 2002, Proceedings*, volume 2484 of *Lecture Notes in Computer Science*. Springer, 2002.

[63] Meir Feder, Neri Merhav, and Michael Gutman. Universal prediction of individual sequences. *IEEE Transactions on Information Theory*, 38:1258–1270, July 1992.

[64] Leonidas Fegaras. Query unnesting in object-oriented databases. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 49–60. ACM Press, 1998.

[65] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.

[66] R. J. Feiertag and E. I. Organick. The multics input/output system. In *SOSP '71: Proceedings of the third ACM symposium on Operating systems principles*, pages 35–41. ACM Press, 1971.

[67] Mary F. Fernández, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. ACM SIGMOD Conference*, 2001.

[68] Mary F. Fernández, Wang-Chiew Tan, and Dan Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1–6):723–745, 2000.

[69] Lorenzo Finesso. *Consistent Estimation of the Order for Markov and Hidden Markov Chains*. PhD thesis, University of Maryland, 1991.

[70] Daniela Florescu, Alon Y. Levy, Dan Suciu, and Khaled Yagoub. Optimization of run-time management of data intensive web-sites. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 627–638. Morgan Kaufmann, 1999.

[71] Pasi Franti and Timo Hatakka. Context model automata for text compression. *The Computer Journal*, 41(7):474–485, 1998.

[72] Gabbay Freddy and Avi Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems*, 16(3):234–270, 1998.

[73] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1968.

[74] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 315–324. ACM Press, 1993.

[75] César Galindo-Legaria. Parameterized queries and nesting equivalencies. Technical Report MSR-TR-2000-31, Microsoft Corporation, April 2000.

[76] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 571–581, New York, NY, USA, 2001. ACM Press.

[77] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of ACM SIGMOD*, 1987.

[78] Carsten Andreas Gerlhof and Alfons Kemper. A multi-threaded architecture for prefetching in object bases. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 1994.

[79] Carsten Andreas Gerlhof and Alfons Kemper. Prefetch support relations in object bases. In Malcolm P. Atkinson, David Maier, and Véronique Benzaken, editors, *Persistent Object Systems, Proceedings of the Sixth International Workshop on Persistent Object Systems, Tarascon, Provence, France, 5-9 September 1994*, Workshops in Computing, pages 115–126. Springer and British Computer Society, 1994.

[80] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[81] Goetz Graefe. Query evaluation techniques in large databases. *ACM Computing Surveys*, 25(2), June 1993.

[82] Jim N. Gray. Interview: A conversation with Jim Gray. *Queue*, 1(4):8–17, 2003.

[83] Jim N. Gray, Pat Helland, Patrick O'Neil, and Dennis Sasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, June 1996.

[84] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. *Granularity of locks and degrees of consistency in a shared data base*, pages 175–193. Morgan Kaufmann Publishers Inc., 1998. Reprinted from Modeling in Data Base Management Systems. Amsterdam: Elsevier North-Holland, 1976.

[85] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX 1994 Technical Conference*, pages 197–208. USENIX Association, January 1994.

[86] Knut Stener Grimsrud, James K. Archibald, and Brent E. Nelson. Multiple prefetch adaptive disk caching. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):88–103, 1993.

[87] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: How to say "good enough" in SQL. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM Press, 2004.

[88] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[89] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs. *Information Sciences*, 152(1):47–61, 2003.

[90] Wook-Shin Han, Yang-Sae Moon, Kyu-Young Whang, and Il-Yeol Song. Prefetching based on type-level access pattern in object-relational DBMSs. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. IEEE Computer Society, 2001.

[91] Philip Hingston. Inference of regular languages using model simplicity. In *Proceedings of the 24th Australasian conference on Computer science*, pages 69–76. IEEE Computer Society, 2001.

[92] Philip Hingston. Using finite state automata for sequence mining. In *Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 105–110. Australian Computer Society, Inc., 2002.

[93] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[94] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level. *ACM Transactions on Database Systems*, 26(1):96–143, 2001.

[95] Jianying Hu, William Turin, and Michael K. Brown. Language modeling with stochastic automata. In *1996 International Conference on Speech and Language Processing*, October 1996.

[96] David Hume. *An Enquiry Concerning Human Understanding*. 1748.

[97] Phillipe Jacquet, Wojciech Szpankowski, and Izydor Apostol. A universal predictor based on pattern matching. *IEEE Transactions on Information Theory*, 48(6), June 2002.

[98] G. Jaeschke and Hans-Jörg Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California*, pages 124–138. ACM, 1982.

[99] Wei Jin, Xiabai Sun, and Jeffrey S. Chase. FastSlim: Prefetch-safe trace reduction for I/O system simulation. *ACM Transactions on Modeling and Computer Simulation*, 11(2):125–160, 2001.

[100] Immanuel Kant. *The Critique of Pure Reason*. 1781.

[101] John P. Kearns and Samuel DeFazio. Diversity in database reference behavior. In *SIGMETRICS ’89: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 11–19. ACM Press, 1989.

[102] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 273–282. ACM Press, 1994.

[103] Tom Keller, Goetz Graefe, and David Maier. Efficient assembly of complex objects. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 148–157. ACM Press, 1991.

[104] Werner Kiessling. On semantic reefs and efficient processing of correlation queries with aggregates. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 241–249, Stockholm, Sweden, August 1985.

[105] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 9(3), 1982.

[106] Nils Knafla. Analysing object relationships to predict page access for prefetching. In Ronald Morrison, Mick J. Jordan, and Malcolm P. Atkinson, editors, *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California, 1998*, pages 160–170. Morgan-Kaufmann, 1998.

[107] Nils Knafla. *Prefetching Techniques for Client/Server, Object-Oriented Database Systems*. PhD thesis, Division of Informatics, University of Edinburgh, 1999.

[108] Donald Kossmann, Laura M. Haas, and Ioana Ursu. Loading a cache with query results. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 351–362. Morgan Kaufmann, 1999.

[109] David F. Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, 1991.

[110] David F. Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Distributed and Parallel Databases*, 1(1):33–51, 1993.

[111] Tobias Kraft. *Rewrite-Strategien für generierte Anfragesequenzen im Online Analytical Processing*. Diploma thesis, Universität Stuttgart, 2002.

[112] Tobias Kraft and Holger Schwarz. Chicago: A test and evaluation environment for coarse-grained optimization. In *Proceedings of the 30th VLDB Conference*, pages 1345–1348, August 2004.

[113] Tobias Kraft, Holger Schwarz, Ralf Rantzau, and Bernhard Mitschang. Coarse-grained optimization–techniques for rewriting SQL statement sequences. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*. Morgan Kaufmann, September 2003.

[114] Achim Kraiss and Gerhard Weikum. Integrated document caching and prefetching in storage hierarchies based on markov-chain predictions. *VLDB Journal*, 7(3):141–162, 1998.

[115] Rajasekar Krishnamurthy. *XML-to-SQL Query Translation*. PhD thesis, University of Wisconsin–Madison, 2004.

[116] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis, Brown University, 1995.

[117] P. Krishnan and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, 1998. An extended abstract appears in *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, Vriginia, January 1994, pagees 392-401.

[118] Thomas M. Kroeger. Predicting file system actions from reference patterns. Master's thesis, University of California Santa Cruz, December 1996.

[119] Thomas M. Kroeger and Darrell D.E. Long. Design and implementation of predictive file prefetching algorithm-usenix01. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[120] Geoffrey Houston Kuenning. *Seer–Predictive File Hoarding for Disconnected Mobile Operations*. PhD thesis, University of California, Los Angeles, 1997.

[121] Geoffrey Houston Kuenning, Wilkie Ma, Peter Reiher, and Gerald J. Popek. Simplifying automated hoarding methods. In *MSWiM '02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 15–21. ACM Press, 2002.

[122] Philip Laird and Ronald Saul. Discrete sequence prediction and its applications. *Machine Learning*, 15(1):43–68, 1994.

[123] Glen G. Langdon, Jr. A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Transactions on Information Theory*, IT-29, March 1983.

[124] N. Jesper Larsson. Extended application of suffix trees to data compression. In J. A. Storer and M. Cohn, editors, *ProceedingsData Compression Conference*, pages 190–199, Snowbird, UT, 1996. IEEE Computer Society Press.

[125] Eric Lehman. *Approximation Algorithms for Grammar-Based Data Compression*. PhD thesis, Massachusetts Institute of Technology, February 2002.

[126] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *USENIX Conference Proceedings*, January 1997.

[127] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(11):75–81, 1976.

[128] Ronny Lempel and Shlomo Moran. Optimizing result prefetching in web search engines with segmented indices. *ACM Trans. Inter. Tech.*, 4(1):31–59, 2004.

[129] Barbara Liskov, Atul Adya, Miguel Castro, and Quinton Zondervan. Safe and efficient sharing of persistent objects in Thor. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 318–329. ACM Press, 1996.

[130] Tara M. Madhyastha. *Automatic Classification of Input/Output Access Patterns*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[131] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. In *IOPADS '97: Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 57–67. ACM Press, 1997.

[132] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1975.

[133] Neri Merhav and Meir Feder. Universal sequential learning and decision from individual data sequences. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 413–427. ACM Press, 1992.

[134] Neri Merhav and Meir Feder. Universal prediction. *IEEE Transactions on Information Theory*, 44(6):2124–2147, October 1998.

[135] Alistair Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communication*, 38(11):1917–1921, November 1990.

[136] Donald R. Morrison. PATRICIA–practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[137] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[138] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, 1996.

[139] Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[140] Craig G. Nevill-Manning and Ian H. Witten. Inferring lexical and grammatical structure from sequences. In *Compression and Complexity of Sequences*, pages 265–274, June 1997.

[141] Stefan Nilsson and Matti Tikkanen. Implementing a dynamic compressed trie. In *2nd Workshop on Algorithm Engineering (WAE '98)*, 1998.

[142] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 255–264. Morgan Kaufmann, 1991.

[143] David A. Patterson. Latency lags bandwidth. *Communications of the ACM*, 47(10):71–75, 2004.

[144] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95. ACM Press, 1995.

[145] Glenn Norman Paulley. *Exploiting Functional Dependence in Query Optimization*. PhD thesis, University of Waterloo, 2000.

[146] Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 421–432. ACM Press, 1989.

[147] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *The VLDB Journal*, 13(2):162–176, 2004.

[148] Jorma Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, IT-29(5), September 1983.

[149] Jorma Rissanen. Universal coding, information, prediction, and estimation. *IEEE Transactions on Information Theory*, IT-30(4):629–636, July 1984.

[150] Jorma Rissanen. Complexity of strings in the class of Markov sources. *IEEE Transactions on Information Theory*, 32(4):526–532, 1986.

[151] Jorma Rissanen and Glen G. Langdon, Jr. Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1), January 1981.

[152] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[153] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420. ACM Press, 1989.

[154] Dana Ron. *Automata Learning and its Applications*. PhD thesis, Hebrew University, 1995.

[155] Dana Ron, Yoram Singer, and Naftali Tishby. Learning probabilistic automata with variable memory length. In *Proceedings of the seventh annual conference on Computational learning theory*, pages 35–46. ACM Press, 1994.

[156] Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 31–40. ACM Press, 1995.

[157] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2–3), 1996.

[158] Carsten Sapia. PROMISE: Predicting query behaviour to enable predictive caching for OLAP systems. In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery (DAWAK 2000)*, September 2000.

[159] Stefan Schrödl and Stefan Edelkamp. Inferring flow of control in program synthesis by example. Technical Report 121, Univerity of Freiburg, 1999.

[160] Timos K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[161] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Dta Engineering*, 2(2), 1990.

[162] Praveen Seshadri, Hamid Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the 12th International Conference on Data Engineering*, pages 450–459, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[163] Jayavel Shanmugasundaram. *Bridging Relational Technology and XML*. PhD thesis, University of Wisconsin–Madison, 2001.

[164] Jayavel Shanmugasundaram. Querying XML views of relational data. In *Proceedings of the 27th VLDB Conference*, 2001.

[165] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 65–76. Morgan Kaufmann, 2000.

[166] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2–3):133–154, 2001.

[167] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[168] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423,623–656, 1948.

[169] Minglong Shao, Jiri Schindler, Steven W. Schlosser, Anastassia Ailamaki, and Gregory R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *Proceedings of the 30th International Conference on Very Large Data Bases*, August 2004.

[170] Dieter Simader. *SQL Ledger Accounting: User Guide and Reference Manual for Version 2.2*. DWS Systems Inc., March 2004.

[171] Yoram Singer. Adaptive mixtures of probabilistic transducers. *Neural Computation*, 9(9):1711–1733, 1997.

[172] Alan Jay Smith. Bibliography on paging and related topics. *SIGOPS Operating Systems Review*, 12(4):39–56, 1978.

[173] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, 1978.

[174] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[175] Alan Jay Smith. Disk cache–miss ratio analysis and design considerations. *ACM Trans. Comput. Syst.*, 3(3):161–203, 1985.

[176] Carl Downing Tait. *A File System for Mobile Computing*. PhD thesis, Columbia University, 1993.

[177] Carl Downing Tait, Hui Lei, Swarup Acharya, and Henry Chang. Intelligent file hoarding for mobile computers. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 119–125. ACM Press, 1995.

[178] W. J. Teahan. *Modelling English Text*. PhD thesis, University of Waikato, May 1998.

[179] Franck Thollard, Pierre Dupont, and Colin de la Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 975–982. Morgan Kauffman, 2000.

[180] Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[181] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[182] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[183] Jeffrey Scott Vitter. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.

[184] Kaladhar Voruganti, M. Tamer Özsu, and Ronald C. Unrau. An adaptive hybrid server architecture for client caching ODBMSs. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proc. Int'l Conf. on VLDB*, pages 150–161, Edinburgh, Scotland, UK, 7–10 September 1999. Morgan Kaufmann.

[185] Kaladhar Voruganti, M. Tamer Özsu, and Ronald C. Unrau. An adaptive data-shipping architecture for client caching data management systems. *Distributed and Parallel Databases*, 15(2):137–177, 2004.

[186] Jasmine Y.Q. Wang, Joon Suan Ong, Yvonne Coady, and Michael J. Feeley. Using idle workstations to implement predictive prefetching. In *HPDC '00: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*,

page 87. IEEE Computer Society, 2000. See also University of British Columbia Technical Report TR-00-06.

[187] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the spatio-temporal behavior of real traffic data. *Performance Evaluation*, 49:147–163, 2002.

[188] Marcelo J. Weinberger, Abraham Lempel, and Jacob Ziv. A sequential algorithm for the universal coding of finite memory sources. *IEEE Transactions on Information Theory*, 38(3):1002–1014, May 1992.

[189] Marcelo J. Weinberger and Gadiel Seroussi. Sequential prediction and ranking in universal context modeling and data compression. Technical Report HPL-94-111 (R.1), HP Computer System Laboratory, January 1997.

[190] P.J. Weinberger. A universal finite memory source. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.

[191] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[192] Brian S. White and Kevin Skadron. Path-based target prediction for file system prefetchings. Technical Report CS-2000-06, Department of Computer Science, University of Virginia, February 2000.

[193] F.M.J. Willems, Y.M. Shtarkov, and T.J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.

[194] Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

[195] Ian H. Witten and Timothy C. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, July 1991.

[196] Aaron D. Wyner and Jacob Ziv. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. *IEEE Transactions on Information Theory*, IT-35(6), November 1989.

[197] Khaled Yagoub, Daniela Florescu, Valérie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 188–199. Morgan Kaufmann, 2000.

[198] Qingsong Yao and Aijun An. Using user access patterns for semantic query caching. In *14th International Conference on Database and Expert Systems Applications*, Prague, Czech Republic, September 2003.

[199] Qingsong Yao and Aijun An. Characterizing database user's access patterns. In *15th International on Database and Expert Systems Applications, (DEXA 2004)*, pages 528–538, 2004.

[200] Qingsong Yao, Aijun An, and Xiangi Huang. Finding and analyzing database user sessions. In *The 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005)*, April 2005.

[201] Tsozen Yeh, Darrell D.E. Long, and Scott A. Brandt. Increasing predictive accuracy by prefetching multiple program and user specific files. In *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, 2002.

[202] Nianlong Yin and J. Kelly Flanagan. Reducing application load time by rearranging disk data. Master's thesis, Brigham Young University, 1998.

[203] Matthew Young-Lai. Application of a stochastic grammatical inference method to text structure. Master's thesis, University of Waterloo, 1996. See also Technical Report CS-96-36.

[204] Matthew Young-Lai. Adding state merging to the DMC data compression algorithm. *Information Processing Letters*, 70(5):223–228, June 1999.

[205] Matthew Young-Lai. *Text Structure Recognition using a Region Algebra*. PhD thesis, University of Waterloo, 2001.

[206] Matthew Young-Lai and Frank Wm Tompa. Stochastic grammatical inference of text database structure. *Machine Learning*, 40(2):111–137, 2000.

[207] Philip S. Yu, Ming-Syan Chen, Hans-Ulrich Heiss, and Sukho Lee. On workload characterization of relational database environments. *IEEE Transactions on Software Engineering*, 18(4), April 1992.

[208] Jinsuo Zhang, Abdelsalam (Sumi) Helal, and Joachim Hammer. UbiData: Ubiquitous mobile file service. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 893–900. ACM Press, 2003.

[209] Qiang Zhu. *Estimating Local Cost Parameters for Global Query Optimization in a Multidatabase System*. PhD thesis, University of Waterloo, 1995.

[210] Jacob Ziv. Coding of sources with unknown statistics–Part I: Probability of encoding error. *IEEE Transactions on Information Theory*, IT-18(13):384–389, 1972.

[211] Jacob Ziv. Coding of sources with unknown statistics–Part II: Distortion relative to a fidelity criterion. *IEEE Transactions on Information Theory*, IT-18(13):389–394, 1972.

[212] Jacob Ziv. An efficient universal prediction algorithm for unknown sources with limited training data. *IEEE Transactions on Information Theory*, 48(6), June 2002.

[213] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.

[214] Jacob Ziv and Abraham Lempel. Compression of individual sequence via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

Author Index

Index

Italicized page numbers indicate the location of definitions.

Colophon

This document was prepared with the LaTeX 2ε document preparation system, based on TeX version 3.141592 in the MiKTeX 2.4 distribution. Illustrations were created with METAPOST, the PSTricks package v0.2l, Visio 10.0, the statistics package R 1.9.0, Excel 9.0 and the *dot* graph layout program from AT&T Bell Laboratories.

The typographic style of this document is based on the `thesis.cls` file originally developed by Glenn Paulley and modified slightly for this document. Dave Mason and Glenn Paulley implemented the `code.sty` used to format code samples.

Sample code was composed in XML and translated to LaTeX 2ε or the Lua programming language (for testing) using XSL. The document dependencies were managed with `make`, with the `awk` scripting language used for additional help.

The bibliography for this document was prepared using BibTeX in combination with a citation database system developed by the author using Lua and Sybase Adaptive Server Anywhere 9.0.2. This system automatically generates the Author Index. All references cited in this paper were stored in electronic form and read on-line. The ACM digital library and the University of Waterloo library electronic journal collections thereby allowed several trees to be saved.

The text for this document was composed using Watcom vi 11.0 using the ‘crufty’ colour scheme.