# A+ Indexes: Highly Flexible Adjacency Lists in Graph Database Management Systems

by

Shahid Khaliq

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

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**

Adjacency lists are the most fundamental storage structure in existing graph database management systems (GDBMSs) to index input graphs. Adjacency lists are universally linked-list like per-vertex structures that allow access to a set of edges that are all adjacent to a vertex. In several systems, adjacency lists can also allow efficient access to subsets of a vertex's adjacent edges that satisfy a fixed set of predicates, such as those that have the same label, and support a fixed set of ordering criteria, such as sorting by the ID of destination vertices of the edges. This thesis describes a highly-flexible indexing sub-system for GDBMSs, which consists of two components. The primary component called *A+ indexes* store adjacency lists, which compared to existing adjacency lists, provide flexibility to users in three aspects: (1) in addition to per-vertex adjacency lists, users can define per-edge adjacency lists; (2) users can define adjacency lists for sets of edges that satisfy a wide range of predicates; and (3) provide flexible sorting criteria. Indexes in existing GDBMS, such as adjacency list, B+ tree, or hash indexes, index as elements the vertices or edges in the input graph. The second component of our indexing sub-system is secondary B+ tree and bitmap indexes that index aggregate properties of adjacency lists in A+ indexes. Therefore, our secondary indexes effectively index adjacency lists as elements. We have implemented our indexing sub-system on top of the Graphflow GDBMS. We describe our indexes, the modifications we had to do to Graphflow's optimizer, and our implementation. We provide extensive experiments demonstrating both the flexibility and efficiency of our indexes on a large suite of queries from several application domains.

**Acknowledgements**

First and foremost, I would like to thank my supervisor, Professor Semih Salihoglu, for all the guidance and support he has provided me over the past two years. Semih introduced me to the world of data systems and has been consistently present every step of the way in my learning process. He has taught me everything I know about good research work and academic writing. I am eternally grateful to him for always being available, one knock away, whenever I had trouble grasping a difficult concept or hit a road block in my work.

I want to thank my colleague Amine Mhebdi for all his work on the Graphflow project and for the nights we spent coding together in the lab. His valuable insights and feedback played a big part in bringing my thesis to fruition. Furthermore, I want to thank my friend Siddhartha Sahu for all the help and guidance he has provided me throughout my time at UWaterloo.

I also want to express my sincerest gratitude to my thesis readers, Ken Salem and Tamer Özsu, for taking out valuable time from their busy schedules to read my thesis and provide their valuable comments.

I would like to thank my family for their unconditional love and support. This thesis would not have been possible without my parents, Nasreen Khaliq and Abdul Khaliq, whose hard work and dedication has brought me to this point. I want to thank them for teaching me the importance of good education and for putting me through the best schools throughout my life. I also want to thank my siblings, Nadia Khaliq and Ikram Khaliq, for always being supportive and for motivating me to keep pushing forward.

Finally, I would like to thank my fiancée, Sehar Khan, for being the one biggest source of happiness in my life and for making the past couple of months so easy for me. Her constant love and affection distracted me from my lack of sleep and helped me give this thesis my best.

## Dedication

This thesis is dedicated to my parents and my fiancée.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The term *graph database management system* (GDBMS) in its contemporary usage refers to data management software such as Neo4j [34], JanusGraph [25], Tigergraph [43], and Graphflow [26, 33] that adopt the property graph data model [36]. In this model, application data is represented as a set of nodes, which primarily represent the entities in the application, and directed edges, which represent the connections among pairs of entities. Each node and edge has a unique ID and an arbitrary number of key value properties. GDBMSs have lately gained popularity among a very wide range of applications from fraud detection and risk assessment in financial sector to recommendations in e-commerce and social networks [40]. One reason GDBMSs appeal to users is that GDBMSs are broadly optimized to perform fast traversal-like operations on graphs, which leads to efficient query evaluation. This is done primarily, and universally, by indexing the edges in an input graph in some variant of an *adjacency list* structure [9]. These lists store sets of edges that are: (i) maintained and accessible (only) *per vertex*; (ii) pre-satisfy some predicates; and (iii) can be sorted according to some property of the edges. Different systems have different but fixed design decisions for the predicates and either fixed or a limited sorting criteria of the lists, optimizing each system to be efficient on a different but restricted set of queries. In this thesis, we describe a highly flexible adjacency list indexing sub-system that consists of two components. The primary component of our indexing sub-system consists of *A+ indexes* that are adjacency list indexes that can: (i) be maintained and accessible both *per vertex* and *per edge*; (ii) store edges that pre-satisfy a wide range of properties; and (iii) be sorted according to a wide range of criterion. A+ indexes allow users to optimize a GDBMS to be efficient for a much wider range of queries than existing GDBMSs. The second component of our indexing sub-system consists of secondary B+ tree and bitmap indexes that index aggregate properties of the adjacency lists on A+ indexes. We have

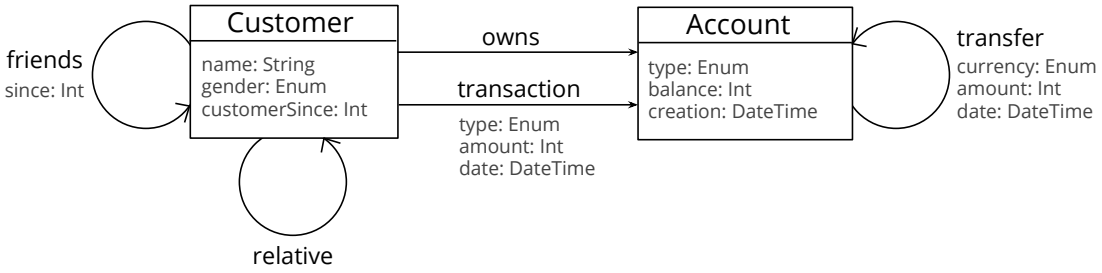



Figure 1.1: Financial Property Graph Schema

implemented our indexing sub-system on top of the Graphflow GDBMS [26].

In the rest of this introductory section, we first discuss the strengths and limitations of existing adjacency lists in GDBMSs in Section 1.1 and then give an overview of how A+ indexes address their limitations in Section 1.2. In Section 1.3, we describe our secondary indexes and finally in Section 1.4 give an outline of the thesis. Figure 1.1 shows the schema of the property graph that we use as our running example. The graph represents bank accounts and customers and the relationships between these entities. We assume vertices and edges have IDs stored, respectively, in `vID` and `eID` properties, and each edge also has `srcID` and `dstID` properties indicating, respectively, the vertex IDs of its source and destination. We note that in many GDBMSs, such as Neo4j, Graphflow, or JanusGraph, the `label` property of the edges, and the `label` property of the vertices are treated as a special property that the system and its query language treat as first-class citizens[1]. We use the openCypher [38] query language of Neo4j in our running examples.

## 1.1 Overview of Adjacency Lists in Existing Systems

Existing GDBMSs, as well as many other graph processing software [32, 41], index their input graphs in adjacency lists that are associated with vertices. That is, given the ID of a vertex $v$, there is a fast access path to the edges *adjacent to* $v$. Often systems maintain two lists for each $v$, a *forward* and a *backward* adjacency list, which are linked list-like data structures that store the set of edges that are, respectively, outgoing from or incoming to $v$. Adjacency lists are accessed by database operators primarily when matching different

[1]Some systems refer to edge or vertex `label` as `type` as well. We uniformly refer to them as `label` properties.

query edges in a query $Q$ (specificallly, the subgraph pattern of $Q$). GDBMSs evaluate a query $Q$ in one or more steps, where in each step an operator takes a partial match $t$ of $Q$ and accesses the edges adjacent to one or more of the vertices in $t$ and extends $t$ to a larger partial match of $Q$. For example, in Neo4j this operation is done by the EXPAND operator, in Graphflow by the EXTEND/INTERSECT (E/I) operator.

**Example 1.1.1** *Consider the following query that returns all 2-paths starting with a node representing a customer Alice:*

---

MATCH $(a_1)-[z_1]->(a_2)-[z_2]->(a_3)$ WHERE $a_1$.name = 'Alice'

---

Above, $a_i$ and and $z_j$ are variable names given to, respectively, the query vertices and query edges. In every GDBMS we are aware of, this query is evaluated in three steps: (1) scan the nodes and find a node, say with `vID` 1 (`v1`), with `name` Alice and match this to variable $a_1$; (2) use `v1` to access Alice's forward adjacency list (or lists) and extend $a_1$ to $a_1 \rightarrow a_2$ edges; and (3) use the matched $a_2$'s `vID`s to access their forward adjacency lists and extend $a_1 \rightarrow a_2$'s to $a_1 \rightarrow a_2 \rightarrow a_3$ paths.

Since the adjacency lists of existing GDBMSs are associated with vertices, a common predicate satisfied by the edges in an adjacency list is that they have the same `srcID`, for forward lists, or same `dstID`, for backward lists. Several systems go beyond this basic predicate by partitioning the adjacency lists into more refined lists that pre-satisfy other predicates. Neo4j, JanusGraph, and the previous version of Graphflow described in reference [33], partition the edges according to the special `label` property. Conceptually, for each vertex $v$ and label $l$, the predicate these refined lists pre-satisfy is that for each $e$, $e$.`label`=$l$. This allows efficient processing of subgraph queries when query edges are restricted to specific labels:

**Example 1.1.2** *Consider the following query that returns all transfers made from the accounts owned by Alice:*

---

MATCH $(a_1)-[z_1$:OWNS$]->(a_2)-[z_2$:TRANSFER$]->(a_3)$ WHERE $a_1$.name = 'Alice'

---

*The "$z_1$:`OWNS`" is syntactic sugar in Cypher for the $z_1$`.label=OWNS` predicate. A system with the refined adjacency lists described above can have an efficient plan for this query by finding the Alice node as before and directly accessing all of Alice's `OWNS` edges to detect the*

*account nodes owned by Alice, and finally directly accessing the* ***TRANSFER*** *edges of these accounts, where the* ***OWNS*** *and* ***TRANSFER*** *edges are accessible and maintained as separate lists. The advantage gained by the refined adjacency lists is that the query plan does not have to run the* `label=OWNS/TRANSFER` *predicate on any edge, as the system ensures that these predicates are pre-satisfied by the edges in the accessed adjacency lists.*

Several prototype graph algorithm implementations in literature use slightly more refined adjacency lists, e.g. where edges have the same edge `labels` as well as the same destination vertex `label` [7]. We are unaware of more refined adjacency lists in existing systems.

Although the adjacency list designs of existing system allow efficient processing of some queries, they are also restrictive in the following aspects.

### Restriction 1: Limited Predicates

The design decisions about what the predicates that edges in adjacency lists pre-satisfy is system-specific and fixed. The lists of existing systems will not be highly optimized beyond queries that access edges with the same edge label and possibly by vertex label, as the next example demonstrates:

**Example 1.1.3** *Consider querying the transactions from Alice's accounts made in US dollars currency:*

---

*MATCH* $(a_1)-[z_1{:}OWNS]->(a_2)-[z_2]->(a_3)$
*WHERE* $a_1.name$ = *'Alice' AND* $z_2.currency$=*USD*

---

*Here the query plans of existing systems will read all* ***TRANSFER*** *edges from the accounts owned by Alice and, for each edge, read its currency and verify whether or not it is USD.*

If queries accessing transactions in USD are important and frequent for an application, maintaining a direct access to those edges would be beneficial. Similarly one can consider applications benefiting from direct access to edges that satisfy numerous other predicates both on: (i) edge properties, e.g., transfers with `amount`>1000 USD; as well as (ii) properties on the destination nodes of the edges, e..g, transfers made to accounts with `balance`> 5000.

**Restriction 2: Per-vertex Adjacency List Limitation**

In some queries, extending the notion of adjacency from vertices to edges may be beneficial.

**Example 1.1.4** *Consider the following query, which is the core of a very important and popular class of queries in financial fraud detection*[2]*.*

---

*MATCH* $(a_1)$*-[*$z_1$*:TRANSFER]->*$(a_2)$*-[*$z_2$*:TRANSFER]->*$(a_3)$*-[*$z_2$*:TRANSFER]->*$(a_4)$
*WHERE* $z_1$*.eID=17 AND* $z_2$*.amount <* $z_1$*.amount AND* $z_3$*.amount <* $z_2$*.amount*
*AND* $z_2$*.date >* $z_1$*.date AND* $z_3$*.date >* $z_2$*.date*

---

*The query is looking for a three-step* money flow path *from a particular* `TRANSFER` *edge with* `eID` *17 (call* `e17`*) with the constraint that every additional transfer happens at a later date, and for a smaller amount. In this query the predicates compare properties of an edge on a path with the previous edge on the same path. Suppose a system has scanned and matched the first query edge* $z_1$ *to* `e17`*, which is say an edge from vertex* `v2` *to* `v3`*. Then existing systems have to read all of the* `TRANSFER` *edges from* `v3` *and filter those that have later* `date` *values than* `e17` *and also have the appropriate* `amount` *values.*

Instead, when the next query edge to match $z_2$ has predicates depending on the query edge $z_1$, these queries can be much faster evaluated if adjacency lists can be associated with edges: a system can directly access the *forward adjacency list of* `e17`, i.e., a list of edges whose `srcID` is the same as `e17`'s `dstID`, pre-satisfying the predicates required by the query and perform the extension.

**Restriction 3: Limited Edge Sorting Criteria**

Finally, some systems also maintain the adjacency lists in sorted order according to some criterion. For example, JanusGraph allows users to build *local indices* over the adjacency lists that sort the edges using an edge property, say `date`. Let `nbrID` refer to the `srcID` or `dstID` of edges in an adjacency lists that stores, respectively, the incoming or outgoing neighbors of a node. The previous version of Graphflow sorts the edges according to the `nbrID` of the edges. There are two well known benefits of sorting the edges: (1) sorting on a property allows quick search within an adjacency list, say to find all `TRANSFER` edges that were on a particular day; and (2) sorting allows fast intersections of adjacency lists on

[2]Private communication with two graph software providers to two banks.

the sorted property to perform matching of multiple query edges at the same time. Most notably, this has been demonstrated by several references [2, 5, 26, 33] studying the new *worst-case optimal* join algorithms for evaluating cyclic subgraph queries:

**Example 1.1.5** *Consider the following query that finds all friendship triangles of Alice:*

---

MATCH $(a_2)<-[z_1$:FRIENDS$]-(a_1)-[z_2$:FRIENDS$]->(a_3)$, $(a_2)-[z_3$:FRIENDS$]->(a_3)$
WHERE $a_1$.name = 'Alice'

---

*In the Graphflow system, where adjacency lists are sorted by* **`nbrID`***, this query is evaluated by a plan that scans each* **`FRIENDS`** *edge Alice→$a_2$ and intersects Alice and $a_2$'s* **`FRIENDS`** *adjacency lists, which are already sorted by the* **`nbrID`** *values, to match the c vertices.*

The benefits of this sorted-order become more prevalent when detecting larger cliques, say cliques of 4 or 5 friends, as the system can do 3- or 4-way intersections to match 3 or 4 query edges directly. The ordering criterion in existing systems however are either fixed or limited to values on the properties of the edges, limiting the set of queries, for which efficient query plans using pre-sorted lists can be generated.

**Example 1.1.6** *Consider the following query:*

---

*MATCH $(a_2)<-[z_1$:RELATIVE$]-(a_1)-[z_2$:FRIENDS$]->(a_3)$*
*WHERE $a_1$.name = 'Alice' AND $a_2$.customerSince = $a_3$.customerSince*

---

*If adjacency lists could be sorted by* **`customerSince`***, a plan can directly intersect the* **`FRIENDS`** *and* **`RELATIVE`** *adjacency lists of Alice to find those friends and relatives with the same value of the customerSince property.*

## 1.2 A+ Indexes

A+ indexes allow users to optimize a GDBMS to be efficient for a much wider range of queries than existing GDBMSs. Specifically, A+ indexes give users more flexibility in three dimensions to address the three limitations of existing adjacency lists we overviewed above:

1. Adjacency lists can be declared and maintained both per vertex and per edge. Conceptually, a per edge adjacency list is a set of edges that are all *adjacent* to a specific edge $e$, either to the source or destination of e and in their forward or backward directions, and is quickly accessible given the `eID` of an edge. Along with the predicate mechanism we describe momentarily, this allows users to define fast access paths to sets of edges that pre-satisfy predicates that depend on another *adjacent edge* in query, allowing the system to generate highly efficient plans for the queries similar to Example 1.1.4.
2. Unlike existing systems, users can put custom predicates for the edges in their adjacency lists to satisfy. For per-vertex lists , we extend the fixed predicates of existing systems, which only depend on the properties of the edges, to also depend on the properties of the neighbor nodes. Users can use comparisons other than equality and any number of conjunctions in their predicates. For edge properties, the properties can also depend on the properties of the edge for which the adjacency lists is defined. For example, these flexible predicates allow users to declare fast access paths for queries in Examples 1.1.3 and 1.1.4.
3. Finally, users can choose to maintain their adjacency lists sorted according to any property of the edges or the neighbor node's properties. This allows generating several very efficient plans that use worst-case optimal join-style multi-way intersections for not only cyclic but also acyclic queries, as in Example 1.1.6.

In relational terms, one can think of the graph-structured data GDBMSs store as consisting of a `Vertex(vID, label, prop1, prop2, ...)` and an `Edge(eID, srcID, dstID, label, prop1, prop2, ...)` table. Existing adjacency lists can effectively be thought of as *partitioned materialized views* over queries on these tables that are (1) are partitioned by the srcID or dstID columns; (2) satisfy some predicates; and (3) are sorted by some criterion. Our A+ indexes effectively generalize the types of view queries that adjacency lists can be constructed from and how they are partitioned. We make these connections more concrete in later chapters of these thesis.

## 1.3 Secondary Indices For Indexing Adjacency Lists

Adjacency lists are primarily used to speed up the database operators such as Neo4j's EXPAND and Graphflow's E/I operators. To speed up SCAN operators, say to match edges from the entire graph that satisfy a particular predicate, traditionally database systems use B+ tree indices to index vertices and edges. In the case of edges, a B+ tree index allows detecting, for example, all `TRANSFER` edges in the entire graph on a particular date or with a range of dates. However, often the number of edges in graphs are a lot larger

than the vertices, so the size of a B+ tree index on the edges can be very large. For example, a public Twitter who-follows-whom graph contains 35x more edges than vertices (1.46B vs 41.6M). Since adjacency lists are first-class citizens in any GDBMS, GDBMSs have the possibility to index aggregate properties on their adjacency lists. In our indexing sub-system, in addition to all vertices and edges, each adjacency list in an A+ index is treated as a unit of data that has an ID, `aID`, and is indexible. We support two secondary indices to index these adjacency lists:

- **Min/Max B+ trees on numeric properties:** Users can create B+ tree indexes that index the minimum or maximum of numeric properties of the edges in adjacency lists. For example, if the maximum `amount` of the `TRANSFER` edges in an adjancency list is indexed and a query plan is scanning for `TRANSFER` edges with more than 1000 USD, it can find and detect all adjacency lists whose maximum is more than 1000, avoiding the scan of many adjacency lists that are guaranteed to not contain any matching edges. Compared to regular B+ tree indices on all edges, these aggregate B+ trees are smaller in size but less selective as well, allowing users to tradeoff speed with index size.
- **Bitmaps on categorical properties:** Users can create bitmap indexes on a categorical property of edges in an adjacency list, allowing the system to quickly grab lists that contain edges with a particular value of that property. For example, users can generate a bit map index on the `currency` property, and a Scan operator can quickly detect and scan all adjacency lists that contain at least one `TRANSFER` in USD currency.

## 1.4 Thesis Outline

The outline of this thesis is as follows:

- Chapter 2 introduces A+ indexes and our secondary indices users can declare around A+ indexes. We also make the connection between A+ indexes and materialized views in relational databases.
- Chapter 3 describes the Graphflow systems' query plans that use our indices, how the optimizer generates these plans, and the changes we had to make to systems' operators.
- Chapter 4 describes the implementation of our indexing sub-system.
- Chapter 5 presents extensive experiments demonstrating the benefits users can get from using A+ indexes and our secondary indexes.
- Finally, Chapters 6 and 7 review related and future work, respectively.

# Chapter 2

# A+ Indexes

In this chapter, we describe A+ indexes and with ample examples demonstrate their flexibility. Section 2.1 describes a running example which we use throughout this chapter. Sections 2.2 and 2.3 describe per-vertex and per-edge A+ indexes and present several examples. In Section 2.4, we make a connection between our per-vertex and per-edge A+ indexes and materialized views and query-rewriting using materialized views in relational database management systems. Section 2.5 describes a common scenario where applications define multiple A+ indexes, one for each value of an edge or a vertex property. Section 2.6 describes our secondary indexes which can be created over the adjacency lists in an A+ index.

## 2.1 Running Example

Fig 2.1 shows an example instance of the financial property graph previously described in Fig 1.1. The dotted vertices represent `Customers`. Three of the `Customer` vertices are highlighted for emphasis and their `name` property values are shown on the vertices. There are 14 other `Customer` vertices, which represent the friends and relatives of the three highlighted `Customer` vertices. These are drawn as small circles and their `customerSince` properties are shown on them. The `friends` and `relative` edges, having labels of the form $f_i$ and $r_i$ respectively, are represented by arrows between `Customer` vertices.

Solid vertices represent `Account` vertices. The three highlighted `Customer` vertices have `owns` edges to the `Account` vertices. `Account` vertices have their `type` property shown on them. The `type` property can be `CHQ`, representing a chequing account, or `SAV`, representing

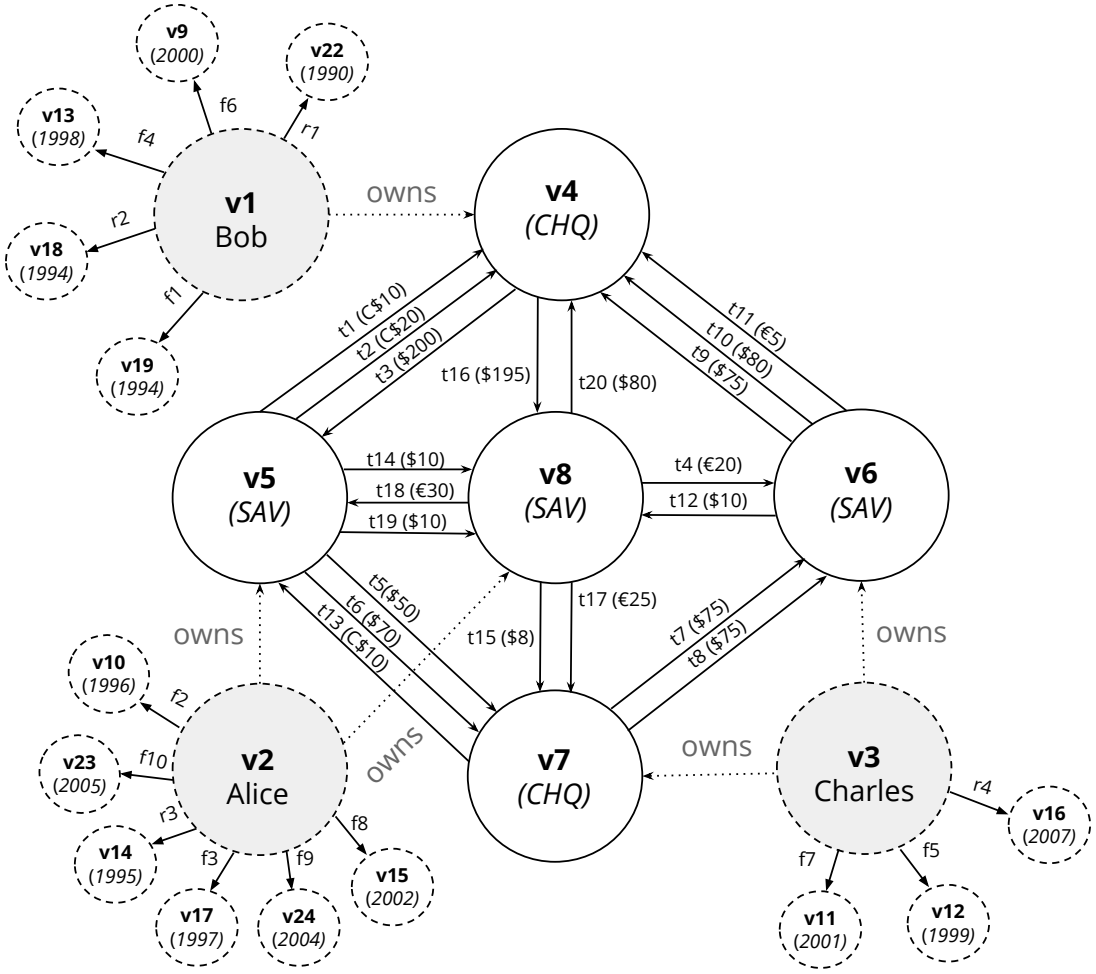



Figure 2.1: Financial Property Graph Example

a savings account. Each edge of type `transfer`, labeled $t_i$, is represented by an arrow between the `Account` vertices and has a `date` attribute which is omitted from the figure. Instead, we give each transfer edge an ID such that $t_i.date < t_j.date$ if and only if $i < j$. For demonstrative purposes we will only use $<$ comparisons on the `date` properties of `transfer` edges in the queries in our examples. The `amount` property of each `transfer` edge is shown in parenthesis next to its ID. `$`, `C$`, and € denote each `transfer` edge's currency as `USD`, `CAD` and `Euros`, respectively.

| | | | | | |
|---|---|---|---|---|---|
| **v5** → | [t2, t6, t1, t5] | **v1** → | [f1, f4, f6] | **v1** → | [r1, r2] |
| **v6** → | [t11, t9, t10] | **v2** → | [f2, f3, f8, f9, f10] | **v2** → | [r3] |
| **v8** → | [t17, t15, t20] | **v3** → | [f4, f5] | **v3** → | [r4] |

(a) CHQ to SAV Transfers (b) Friends (c) Relatives

Figure 2.2: Per-vertex A+ index examples. Each adjacency list stores, both, the edge IDs and the corresponding neighbor vertex IDs. For simplicity, we omit vertex IDs in these examples. We also ignore empty adjacency lists.

## 2.2 Per-Vertex A+ Indexes

Formally, a *per-vertex A+ index* is a triple of one of the following forms:

1. $(v_b\text{−}[e_{adj}]\text{→}v_{nbr}, \text{P}, \Phi)$
2. $(v_b\text{←}[e_{adj}]\text{−}v_{nbr}, \text{P}, \Phi)$

Above `(i)` $v_b$ represents the vertices that the adjacency lists will be bound to; `(ii)` $e_{adj}$ represents the edges that will be indexed in the adjacency lists; and `(iii)` $v_{nbr}$ represents the neighbors of $v_b$ along those edges. `P` is a set of arbitrary conjunctive predicates containing literals and comparisons on properties of $v_b$, $v_{nbr}$ and $e_{adj}$. Finally, $\Phi$ is a property of either $e_{adj}$ or $v_{nbr}$ on which the adjacency lists are sorted.

**Example 2.2.1** *Consider the following A+ index creation query, written in a syntax we added to Cypher, and corresponding formally to the index:* $(v_b\text{−}[e_{adj}]\text{→}v_{nbr}, \{v_b.type = \text{'SAV' AND } v_{nbr}.type = \text{'CHQ'}\}, \varnothing)$.

---

*CREATE A+ LIST INDEX* $(v_1$*:Account)*$-[e$*:transfer*$]\rightarrow(v_2$*:Account)*
*WHERE* $v_1$*.type = 'SAV' AND* $v_2$*.type = 'CHQ'*

---

*This query creates a per-vertex A+ index that, for each* ***SAV*** *account vertex, stores the outgoing transfer edges to* ***CHQ*** *accounts. Figure 2.2a shows the actual adjacency lists this query would generate on our running example. Note that the* $v_{nbr}$*.type = 'CHQ' predicate allows us to index edges adjacent to vertices based on a property of the neighbor vertices. This is a feature that does not exist in the adjacency list indexes of existing GDBMSs. This index allows an efficient access path for queries that contain paths traversing SAV account to CHQ account edges.*

**Example 2.2.2** *Consider now the following two A+ index creation queries.*

*CREATE A+ LIST INDEX* $(v_1$*:Customer)*$-[e$*:friend*$]\rightarrow(v_2$*:Customer)*
*SORT BY* $v_2$*.customerSince*

*CREATE A+ LIST INDEX* $(v_1$*:Customer)*$-[e$*:relative*$]\rightarrow(v_2$*:Customer)*
*SORT BY* $v_2$*.customerSince*

*These queries create two per-vertex A+ indexes that, for each customer vertex, store the outgoing friends and relative edges, respectively. Figures 2.2b and 2.2c show the adjacency lists generated by these queries on our running example. These adjacency lists are sorted by the* `customerSince` *property of* $v_{nbr}$*, which is yet another feature lacking in adjacency list indexes of existing GDBMSs. For queries with equality predicates on this property, these two indexes allow plans to do quick adjacency list intersections. Recall the query from Example 1.1.6, that asked for friends and relatives of Alice that have been customers since the same year. For this query, in the absence of these indexes, a query plan would first match the 'Alice' vertex. It would then expand the match to Alice's friends and relatives before it can evaluate the given equality predicate. If the above indexes are present, another query plan for this query can match the Alice vertex as before, then grab its friends and relative adjacency lists from the indexes and simply intersect them to get the results, which as we will demonstrate in our evaluations, is a more efficient plan.*

One important property of per-vertex A+ indexes is that each edge $e(src, dst)$ in the input graph can appear at most once in a per-vertex A+ index. This is because $e$ either does not satisfy the predicates P, in which case it will not appear in the A+ index, or $e$ satisfies $P$ and depending on whether the index is bound to $src$ or $dst$, will appear in only one adjacency list. As we will discuss in Chapter 3, this is why we are able to use per-vertex A+ indexes in scan operations that scan the edges of a graph, possibly with some predicates, as the first operation in plans. In contrast, we will not be able to use per-edge A+ indexes in scan operations.

## 2.3 Per-Edge A+ Indexes

Formally, a *per-edge A+ index* is a triple of one of the following forms:

1. $(v_1-[e_b]\rightarrow v_2-[e_{adj}]\rightarrow v_3$, P, $\Phi)$

2. $(v_1{-}[e_b]{\rightarrow}v_2{\leftarrow}[e_{adj}]{-}v_3,\ \mathrm{P},\ \Phi)$
3. $(v_3{-}[e_{adj}]{\rightarrow}v_1{-}[e_b]{\rightarrow}v_2,\ \mathrm{P},\ \Phi)$
4. $(v_3{\leftarrow}[e_{adj}]{-}v_1{-}[e_b]{\rightarrow}v_2,\ \mathrm{P},\ \Phi)$

Above `(i)` $e_b$ represents the edges that the adjacency lists will be bound to and `(ii)` $e_{adj}$ represents the edges that will be indexed in the adjacency lists. `P` is a set of arbitrary conjunctive predicates containing literals and comparisons on properties of $v_1$, $v_2$, $v_3$, $e_b$ or $e_{adj}$, with the restriction that there is at least one predicate between one of $v_1$ or $e_b$ and one of $e_{adj}$ or $v_3$, i.e., between $e_b$ and $e_{adj}$, $e_b$ and $v_3$, $v_1$ and $e_{adj}$, or $v_1$ and $v_3$. We will discuss this restriction momentarily after we give an example of a per-edge A+ index. Finally, $\Phi$ is a property of either $e_{adj}$ or $v_3$ on which the adjacency lists are sorted.

**Example 2.3.1** *Consider the following A+ index creation command which formally corresponds to the index* $(v_1{-}[e_b]{\rightarrow}v_2{-}[e_{adj}]{\rightarrow}v_3,\ \{e_b.date < e_{adj}.date\ AND\ e_{adj}.amount < e_b.amount\},\ \varnothing)$.

---

*CREATE A+ LIST INDEX*
$(v_1{:}Account){-}[e_b{:}transfer]{\rightarrow}(v_2{:}Account){-}[e_{adj}{:}transfer]{\rightarrow}(v_3{:}Account)$
*WHERE* $e_b.date < e_{adj}.date$ *AND* $e_{adj}.amount < e_b.amount$

---

*This query creates a per-edge A+ index that, for each transfer edge, stores the outgoing edges from its destination vertex which represent later and smaller transfers, that is edges with a higher value of the* **`date`** *property but a lower value of the* **`amount`** *property. Figure 2.3 shows the actual adjacency lists this query would generate on our running example. This query allows us to index edges based on a property of an adjacent edge which, again, is a feature that does not exist in the adjacency list indexes of existing GDBMSs. This index would be highly useful for queries containing both of the predicates it pre-satisfies. If we take the financial fraud detection query discussed in Example 1.1.4 and execute it on our example graph in the absence of this index, a plan would need to evaluate both the predicates on all 2-edge transfer paths. In our example, even if all transfer edges are directly accessible using a per-vertex A+ index of the form* $(v_1{:}Account){-}[e_1{:}transfer]{\rightarrow}(v_2{:}Account)$*, the number of these 2-edge paths is 53. On the other hand, using the per-edge A+ index from this example allows a plan to directly access the 10 2-edge paths which satisfy the given predicates.*

| | | |
|---|---|---|
| **t3** | → | [t5, t6, t14, t19] |
| **t7** | → | [t11, t12] |
| **t8** | → | [t11, t12] |
| **t16** | → | [t17, t18] |

Figure 2.3: Per-edge A+ index. As before, each adjacency list stores, both, the edge IDs and the corresponding neighbor vertex IDs. For simplicity, we omit the vertex IDs and also ignore empty adjacency lists.

Observe in the above example that unlike per-vertex A+ indexes, an edge $e$ in the graph can appear in multiple adjacency lists in the index. For example, in Figure 2.3, the edges `t11` and `t12` appear both in the adjacency list for edge `t7` as well as `t8`. We emphasize two consequences of this. First, because edges may be duplicated in a per-edge A+ index, we do not use them in scan operations, as we might scan and output the same edge twice, leading to incorrect outputs. Second, recall the restriction for per-edge A+ indexes that at least one of the predicates in `P` has to be between one of $v_1$ or $e_b$ and one of $e_{adj}$ or $v_3$ variables. We impose this restriction because if all the predicates are localized to a single query edge $v_1$−[$e_b$]→$v_2$ or $v_2$−[$e_{adj}$]→$v_3$ in the definition, then we would redundantly generate duplicate adjacency lists, and defining instead per-vertex A+ indexes would avoid this redundancy. We give an example.

**Example 2.3.2** *Consider the following per-edge A+ index definition:*

---

*CREATE A+ LIST INDEX*
*($v_1$:Account)−[$e_b$:transfer]→($v_2$:Account)−[$e_{adj}$:transfer]→($v_3$:Account)*
*WHERE $e_{adj}$.amount < \$100*

---

*Now consider the checking account v7 in the input graph of our running example in Figure 2.1. For each of the four incoming edges of v7,* `t5`*,* `t6`*,* `t15`*, and* `t17`*, this per-edge A+ index would contain the same adjacency list that consists of all outgoing edges of v7:* {`t7`,`t8`,`t13`}*, because the predicate is localized only to a single edge. Instead, a user can define a per-vertex A+ index with the same predicate and bound it to v7 and achieve the same access path to the edges* {`t7`,`t8`,`t13`}*.*

## 2.4 Connection to Materialized Views

In relational terms, per-vertex and per-edge A+ indexes can be understood as *partitioned* materialized views as follows. There are multiple ways to represent a property graph as a set of relational tables. For our purposes, let us represent the vertices and edges in the input graph in two tables: a `Vertex(vID, label, prop1, prop2, ...)` and an `Edge(eID, srcID, dstID, label, prop1, prop2, ...)` table, where the `prop1`, `prop2`, etc. are the columns representing the different properties that appear on the vertices and edges of the graph. Then the edges stored in a per vertex A+ index, say ($v_b$−[$e_{adj}$]→$v_{nbr}$, P, Φ), can conceptually be seen as a materialized view of the following SQL query:

---

SELECT $v_b$.vID, $e_{adj}$.eID, $e_{adj}$.dstID
FROM Vertex $v_b$, Edge $e_{adj}$, Vertex $v_{nbr}$
WHERE $v_b$.vID = $e_{adj}$.srcID & $e_{adj}$.dstID = $v_{nbr}$.vID & P

---

The ($v_b$←[$e_{adj}$]-$v_{nbr}$, P, Φ) per-vertex A+ index definition would be similar with $v_b$ and $v_{nbr}$ swapped in the WHERE clause. The bounding of A+ index to $v_b$ effectively can be thought of as partitioning the result of this view by the $v_b$`.vID` and the ordering by Φ thought of as ordering these partitions by some property of $e_{adj}$ or $v_{nbr}$ (equivalently we could rewrite the view with an ORDER BY clause that first sorts by $v_b$.vID and then by Φ).

Similarly, the edges stored in a per-edge A+ index, say ($v_1$−[$e_b$]→$v_2$−[$e_{adj}$]→$v_3$, P, Φ), can conceptually be seen as a materialized view of the following SQL query:

---

SELECT $e_b$.eID, $e_{adj}$.eID, $e_{adj}$.dstID,
FROM Vertex $v_1$, Edge $e_b$, Vertex $v_2$, Edge $e_{adj}$, Vertex $v_2$
WHERE $v_1$.vID = $e_b$.srcID & $e_b$.dstID = $v_2$.vID = $e_{adj}$.srcID & $e_{adj}$.dstID = $v_3$.vID & P

---

The bounding to $e_b$ can be thought of as partitioning the result of this view by the $e_b$`.eID` and the ordering by Φ thought of as ordering these partitions by some property of $e_{adj}$ or $v_3$. Existing systems support adjacency lists that only fall under the first type of materialized views and that only support a fixed set of predicates `P` and the ordering criteria $\phi$. The flexibility of our A+ indexes is its ability to support two types of views and the arbitrary predicates and ordering users can use in defining these views.

Finally, as we will discuss in Chapter 3, our use of these A+ indexes in query optimization can be thought of as query rewriting using materialized views. Although our work is built on top of a native GDBMSs, we believe these connections can give insights into how similar A+ indexes can be developed on non-native GDBMSs that are developed on top of RDBMSs, such as Oracle' GDBMSs [39].

## 2.5 Defining Multiple A+ Indexes

The adjacency lists in the previous version of Graphflow and Neo4j that we described in Chapter 1 effectively give the system an efficient access path to each type of edge each vertex has. This is an example of an important and common scenario where an application may need a set of A+ indexes (per-vertex or per-edge), one for each value that a vertex or edge property can take in an equality predicate, that differ only in that predicate. Instead of defining these A+ indexes one by one, we have added syntactic sugar to our A+ index definition queries, where the user can define a single A+ index and use the asterisk (*) symbol in an equality predicate on a vertex or edge property to indicate that the system should generate one A+ index for each possible value of that property. We give an example:

**Example 2.5.1** *Consider the Cypher command below.*

---

*CREATE A+ LIST INDEX* $(v_1{:}Account){-}[e{:}transfer]{\rightarrow}(v_2{:}Account)$
*WHERE* $v_1.type = *$ *AND* $v_2.type = *$

---

*This query effectively creates four A+ indexes for each combination of values that* $v_b.type$ *and* $v_{nbr}.type$ *can take over [CHQ, SAV]. As we will discuss in Chapter 4, when there is single asterisk in such definitions, our implementation adopts an efficient CSR-like physical layout, which we refer to as* ***partitioned A+ indexes****, to store all of the expanded A+ indexes.*

## 2.6 Secondary Indexes on A+ Indexes

Database systems treat certain units of data as indexable elements in their indices based on properties of these elements. In RDBMSs, these indexible elements are tuples of tables. For example a B+ tree index in an RDBMS can index tuples, identified by some tuple

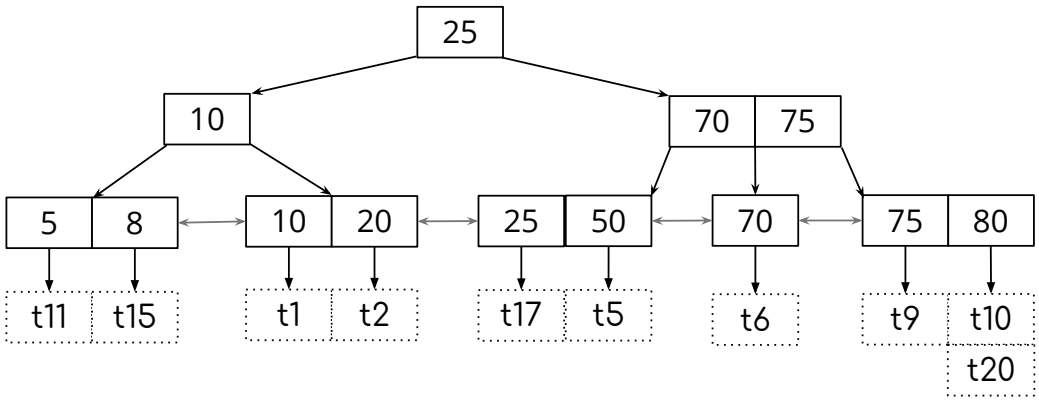
ID, based on a property of that tuple, allowing fast access for queries that have range or equality predicates on the property. Similarly, existing GDBMSs treat vertices and edges in graphs as indexable elements. For example, JanusGraph [25] supports hash map indexes to index both vertices and edges on a property.

One problem with indexes over edges is that often the number of edges in real-world graphs are very large, so these indexes can be quite large and require significant memory. The organization of the edges in GDBMSs in adjacency lists allows a GDBMS to treat adjaceny lists also as indexable elements, based on an *aggregate property* of the edges stored in these adjacency lists. As we show in this section, this can allow GDBMSs to support indexes over individual edges that are much smaller in size. As we will show however that these *secondary indexes* will also be less selective than indexes directly over individual edges. The purpose of these secondary indexes is primarily to avoid scanning adjacency lists that do not store any edges satisfying a given predicate in the query.

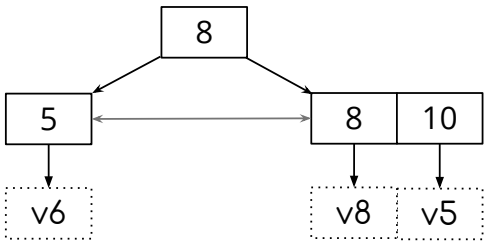
We have implemented two types of secondary indexes that index aggregate properties of adjacency lists. Each secondary index is over a per-vertex A+ index $L$ in the system and will index an aggregate property of each adjacency list in $L$. We do not support secondary indexes over per-edge A+ indexes because, recall that, per-edge A+ indexes can store each edge multiple times across different adjacency lists. Therefore, we cannot use them in scan operators.

### 2.6.1 Min/Max B+ Tree Indexes Over Numeric Edge Properties

These indexes are used to index the minimum or the maximum value of a numeric edge property over all the edges in an adjacency list, for every adjacency list in an A+ index. Figures 2.4a and 2.4b respectively show a full B+ tree index and a min value B+ tree index over the `amount` property of the transfer edges in the A+ index shown previously in Figure 2.2a. To evaluate the predicate $e.amount \leq 5$, a plan can use the full B+ tree index to instantly access `t11` in its scan operation. Another plan can use the smaller min value B+ tree index to narrow down the search to vertex `v6` by ignoring the vertices, whose adjacency lists are guaranteed to contain only edges with amounts greater than 5. The predicate then needs to be applied on only the three edges in the adjacency list of `v6` and the seven edges in the adjacency lists of `v5` and `v8` can be completely ignored. Hence, for selective range and equality predicates, secondary min/max B+ tree indexes can bring significant performance benefits while being smaller than full B+ tree indexes.




(a) A full B+ tree for every edge in the A+ index.




(b) A min value B+ tree for every adjacency list in the A+ index.

Figure 2.4: B+ Tree index examples. Each B+ tree has a fanout of 2 and the pointers between the leaf nodes are omitted from these examples.

### 2.6.2 BitMap Indexes over Categorical Edge Properties

A bitmap index is useful for indexing edge properties having a small domain of values. In min/max B+ tree secondary indexes, the indexed aggregate property of an adjacency list $l_i$ is either $\min_{e \in l_i}(e.property)$ or $\max_{e \in l_i}(e.property)$. In bitmap indexes, for each value, `VAL`, in the domain of a property, the aggregate property of the adjacency lists that is indexed can be thought of as the output of the following disjunctive query: $\vee_{e \in l_i}(e.property = VAL)$. For queries with equality edge predicates, a scan operator can use a map index to discard all adjacency lists which have no edges satisfying the predicate. As an example, Figure 2.1 shows a map index which exists for the `currency` edge property on the A+ index shown in Figure 2.2a. If a query contains the edge predicate $e.currency =$ '$CAD$', a scan operator will only access the adjacency list of `v5`, saving the cost of predicate evaluation on the six edges present in the adjacency lists of `v6` and `v8`. These map indexes can be extended to disjunctive queries of arbitrary predicates on the edges. For example, a system can keep a list of all adjacency lists of an A+ index, that have at least one edge whose `amount` >

| | v5 | v6 | v8 |
|---|---|---|---|
| **CAD:** | 1 | 0 | 0 |
| **USD:** | 1 | 1 | 1 |
| **EURO:** | 0 | 1 | 1 |

Table 2.1: Categorical Property Secondary Index

1000.

Although it is not the focus of this thesis, we have also implemented B+ tree indexes that index vertex and edge properties for the completeness of the secondary indexes component of Graphflow.

# Chapter 3

# Query Plans and DP Optimizer

This chapter describes, both the background on the Graphflow system and the modifications we had to make to the system for it to appropriately use our indexing subsystem. Section 3.1 describes the system's operators and for each operator, our modifications to it. Section 3.2 describes the new plan space for queries and a modified plan cost estimation technique that takes into account the usage of our new indexes. Section 3.3 describes the system's dynamic programming-based optimizer and our modifications to it.

## 3.1 Graphflow Operators

The version of the Graphflow system [33] on which we build this work supports only edge and vertex labeled subgraph queries. In this work, we consider a wider range of queries that can also contain predicates on the vertex and edge properties. So in this work, a query $Q(G_Q, P)$ consists of a labeled subgraph pattern $G_Q(V_Q, E_Q)$, and a set of conjunctive predicates $P$ on properties of the query vertices and edges in $G_Q$. The previous version of the system consisted of three operators: `Scan`, `Extend/Intersect (E/I)`, and `HashJoin`. We review the `Scan` and `E/I` operators and the modifications we made to them in this section. The `HashJoin` operator was used to join two sub-queries, each consisting of at least two edges. We removed the `HashJoin` operator for our study because we focus on indexes in this study and the availability of the indexes does not affect the use of `HashJoin` in query plans. `HashJoin` in reference [33] was primarily used to decompose a subgraph query into multiple components. To support arbitrary predicates, some of which may not already be pre-satisfied in A+ indexes available in the system, we also added a new `Filter` operator that is capable of evaluating predicates on vertex and edge properties.

We note that Graphflow's query execution follows the Volcano model [18], where operators pass each other tuples that are outputs of a sub-query of Q, i.e., a query that consist of a subset of the query vertices and edges in $G_Q$ and a subset of the predicates $P$. We next give an overview of Graphflow's operators and our modifications.

### 3.1.1 Scan

Graphflow contains two `Scan` operators:

#### ScanVertex

The `ScanVertex` operator matches a single query vertex from $G_Q$ and passes the ID of the vertex to the next operator in the plan. Previously, this operator needed to loop over every vertex in the graph. Our modifications allow the operator to now use a B+ tree index on a vertex property to evaluate a predicate, if required, and only pass the IDs of the satisfying vertices to the next operator.

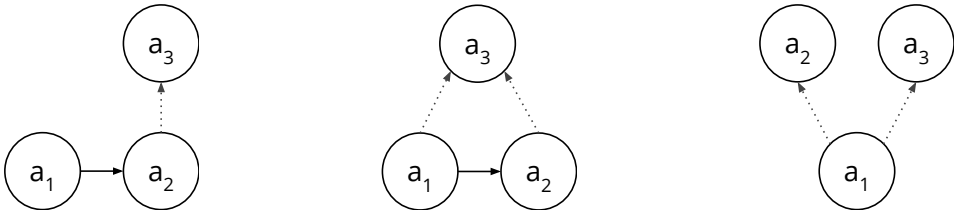
#### ScanEdge

The `ScanEdge` operator matches a single query edge from $G_Q$ and passes the edge's ID and the IDs of its source and destination vertices to the next operator in the plan. Previously, the operator did this by iterating over each edge in every default forward adjacency list in the system. We have made the following modifications:

1. **A+ index usage:** The operator can now be configured to scan over the adjacency lists in any A+ index in the system. If the query edge has predicates on it and an A+ index $A$ pre-satisfies all or a subset of these predicates (but no extra predicates), a `ScanEdge` operator can use $A$. Furthermore, if the adjacency lists in the A+ index are sorted on some edge or neighbor vertex property, say $e.amount$, an additional predicate on the sort order property, say $e.amount > 500$ may be provided. To evaluate the predicate, the operator does a binary search over each adjacency list and scans only the edges satisfying the additional predicate.
2. **Regular B+ tree index on edges usage:** The operator can be configured with an edge predicate, say $e.amount > 500$, and a regular B+ tree index that indexes edges on the edge property in the predicate, say $e.amount$. Using the B+ tree index, the operator directly accesses only those edges which satisfy the predicate.

3. **Secondary index usage:** The operator can be configured with: (i) an A+ index $A$; (ii) one of the secondary min/max B+ tree or bitmap indexes $B$ on the given A+ index; and (iii) a predicate. The operator uses the secondary index $B$ and the predicate to filter out any adjacency lists from the A+ index which are guaranteed to not contain any satisfying edges, as described in Sections 2.6.1 and 2.6.2. For example if $B$ is max B+ tree index on `amount` property of edges and the predicate is $e.amount > 500$, the operator avoids scanning the adjacency lists whose maximum `amount` value is at most 500 and only scans the edges in the remaining adjacency lists and applies the predicate to those edges.

### 3.1.2 Extend/Intersect (E/I)




(a) Extend (b) Intersect (c) Multi-extend

Figure 3.1: E/I Operations

The `E/I` operator in the previous version of Graphflow was used for extending a partial match of a query $Q$ by one query vertex. The operator took as input a tuple $t$ of (k-1)-matches, i.e., a match of a sub-query $Q_{k-1}$ that consists of $k-1$ query vertices, and extended $t$ to one or more k-matches. The `E/I` operator was configured with one or more *adjacency list descriptors* (ALDs), one for each query edge connecting $Q_{k-1}$ to the $k$'th query vertex being extended to. Each ALD was a ($i$, $dir$, $t_e$) tuple, where $i$ represents the index of a vertex in $t$, $dir$ was either `forward` or `backwards` and $t_e$ was the label of the edge the ALD represented. If a single ALD was provided, i.e., there is a single query edge between $Q_{k-1}$ and the $k$'th query vertex, a simple extend operation was done as shown in Figure 3.1a. If multiple ALDs were provided, a fast multi-way intersection operation was done as shown in Figure 3.1b. This required each adjacency list to be sorted by the neighbor vertex IDs. We have made the following modifications:

1. **A+ index usage:** The `E/I` operator can now use any A+ index (per-vertex or per-edge) in the system. Each ALD has an additional `indexID` field for the ID of the A+ index to be used to match the edge the ALD represents. If the `indexID` is the ID

of per-edge A+ index, then the $i$ field in the ALD will refer to a matched edge in the $k-1$-match $t$. If there are multiple ALDs provided, then all of the specified adjacency lists have to be sorted by the same property and the operator will intersect the specified adjacency lists on that property. Unlike the previous version of Graphflow, the intersection can now extend to either: (i) 1 new query vertex, so generate a $k$-match, which happens if all of the given ALDs are sorted by neighbor vertex ID; or (ii) to multiple query vertices at the same time, if the sort predicate is on a different property. For example, recall the query in Example 1.1.6, where we searched for Alice's friends and relative pair who have the same `customerSince` values. Then after matching the Alice vertex, an `E/I` operator can intersect the friends and relative adjacency lists that are sorted by `customerSince` values and extend a single query vertex to three query vertices as in Figure 3.1c.

2. **Predicate evaluation:** If the operation is a simple extend, as shown in Figure 3.1a, and the adjacency lists in the A+ index being used are sorted by a property of the edge or vertex the operator is matching, a predicate on the property can also be provided along with the ALD. The predicate is evaluated by the operator using binary search and the operator only extends to the matches satisfying the predicate.

### 3.1.3 Filter

The new `Filter` operator always follows one of the above matching operators and is configured with a set of predicate on the properties of query vertices and edges which have already been matched by the previous operators in the plan. The operator applies these predicates in order of decreasing selectivity and passes on the satisfying tuples to the next operator in the plan.

## 3.2 Plan Space and Cost Estimation

Our plan space consists of linear plans that start with `Scans` and consist of combinations of `E/I` and `Filter` operators that match the entire query and evaluate the predicates. Each `E/I` operator needs to correctly be configured, i.e., have the right ALDs, to extend a sub-query by one or more query vertices, and the union of the `E/I` operators need to have matched the entire labeled subgraph part of the query. The union of all sets of predicates evaluated in all operators in a query plan is equal to $P$, which is the set of all predicates given in the query. We next give an example, that we will also use in describing the costs of plans.

**Example 3.2.1** *Consider the following query:*

*MATCH* $(a_2)<-[z_1:RELATIVE]-(a_1)-[z_2:FRIENDS]->(a_3)$
*WHERE* $a_1.name$ = *'Alice' AND* $a_2.customerSince = a_3.customerSince$ *AND* $a_3.gender$ = *'Male'*

*This is a variant of the query to find friends and relatives of Alice that have been customers since the same year, where we also require that the friends have their* `gender` *values be* `Male`*. Figures 3.2a, 3.2b, and 3.2c show three example plans for this query that are in our plan space.*

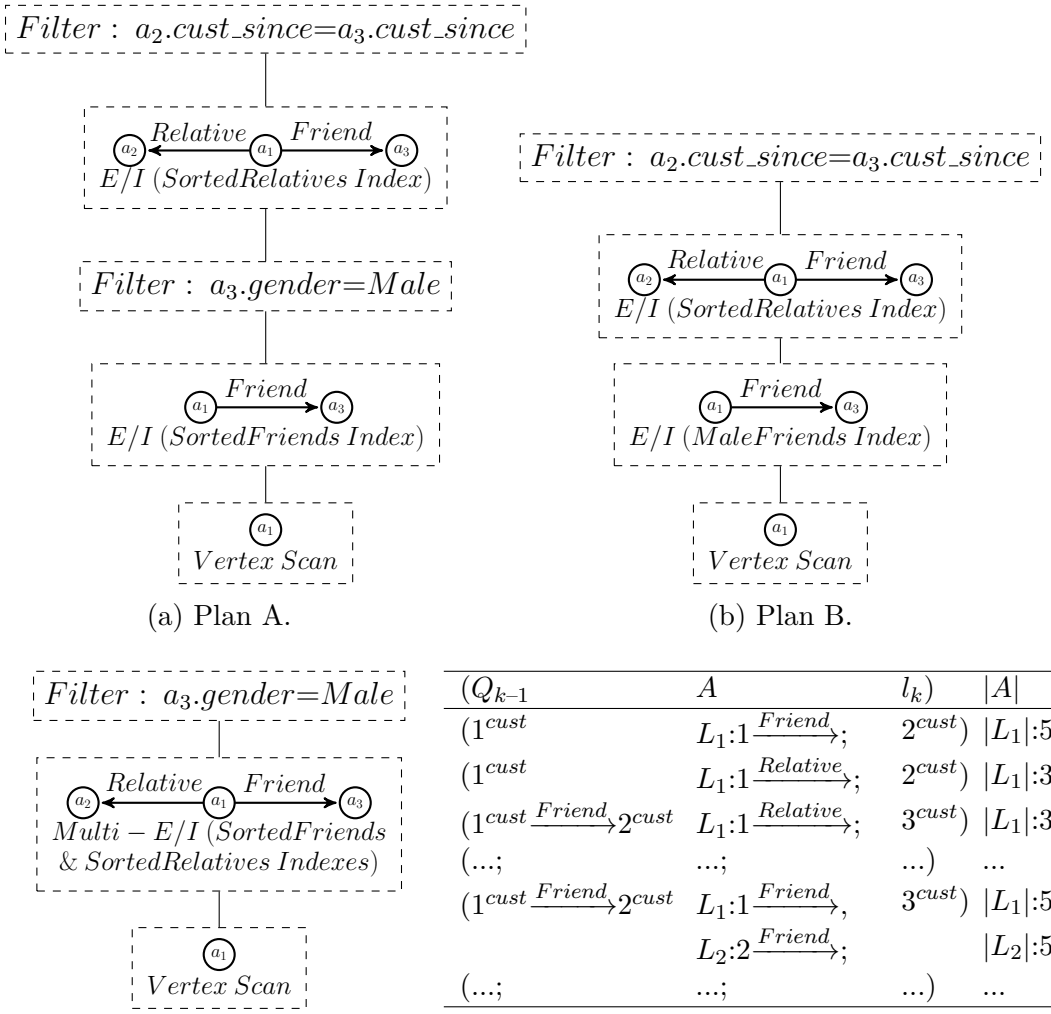
### 3.2.1 Cost Metric

In our work, we did not change Graphflow's default cost metric, which as we explain and demonstrate is good enough to pick efficient plans in the presence of A+ indexes. Graphflow's cost metric is called *intersection cost* [33] (`i-cost`), which is defined as the sum of the lengths of the adjacency lists that will be accessed by the `E/I` operators of a plan (and some default values for the `ScanVertex` and `ScanEdge` operators, which we explain momentarily). Formally, let $Q_{i1}, ..., Q_{im}$ be the $m$ sub-queries that a plan $P_A$ evaluates during its execution, where the original query $Q$ is $Q_{im}$. These are the sub-queries that are evaluated in the `E/I` operators. And let $A_{i1}...A_{im}$ be the adjacency list descriptors of the `E/I` operators. Then `i-cost` of $P_A$ is defined as:

$$\sum_{Q_{ij} \in Q_{i1}...Q_{im}} \sum_{t \in Q_{ij}} \sum_{(i,dir) \in A_{i,j}} |t[i].dir| \tag{3.1}$$

Here, the second summation loops over each tuple in sub-query $Q_{ij}$, so essentially loops as many times as the actual cardinality of $Q_{ij}$, so for each intermediate tuple $t$ that the plan generates throughout its computation, `i-cost` effectively is the sum of the lengths of the adjacency lists that are accessed in extending $t$ to its partial matches in $Q_{i(j+1)}$. This is the formal definition of `i-cost`. Given this definition, the system estimates the cardinalities of different sub-queries and the sizes of the adjacency lists that will be accessed by the `E/I` operators. Given these estimates, we first describe the estimated cost of each operator and then discuss in Section 3.2.2 how these costs are estimated.

- Cost of `ScanVertex` and `ScanEdge` are respectively, the estimated number of vertices and edges that will be scanned by `ScanVertex` and `ScanEdge`.




(a) Plan A. (b) Plan B.

(c) Plan C.

(d) Subgraph Catalogue.

| $(Q_{k-1}$ | $A$ | $l_k)$ | $\lvert A\rvert$ | $\mu(Q_k)$ |
|---|---|---|---|---|
| $(1^{cust}$ | $L_1$:1$\xrightarrow{Friend}$; | $2^{cust})$ | $\lvert L_1\rvert$:5 | 5 |
| $(1^{cust}$ | $L_1$:1$\xrightarrow{Relative}$; | $2^{cust})$ | $\lvert L_1\rvert$:3 | 3 |
| $(1^{cust} \xrightarrow{Friend} 2^{cust}$ | $L_1$:1$\xrightarrow{Relative}$; | $3^{cust})$ | $\lvert L_1\rvert$:3 | 3 |
| (...; | ...; | ...) | ... | ... |
| $(1^{cust} \xrightarrow{Friend} 2^{cust}$ | $L_1$:1$\xrightarrow{Friend}$, | $3^{cust})$ | $\lvert L_1\rvert$:5 | 2 |
| | $L_2$:2$\xrightarrow{Friend}$; | | $\lvert L_2\rvert$:5 | |
| (...; | ...; | ...) | ... | ... |

Figure 3.2: Example of three different plans evaluating the same query and a subgraph catalogue.

- Cost of `E/I` operator $o$ is given by the following formula. Suppose that $o$ takes as input partial matches of sub-query $Q_{ij}$ and extends it by one or more query vertices to $Q_{i(j+1)}$) accessing one more adjacency lists coming from A+ indexes $I : A_1, ..., A_k$. Suppose that the edges in A+ index $A_t$ pre-satisfy zero or more predicates $P_t$ that

are not vertex or edge labels. Then the formula we use to estimate the cost of $o$ is:

$$\text{cost}(o) = \mu(Q_{ij}) * \sum_{A_t \in I} \left( |A_t| \prod_{p_t \in P_t} \mu(p_t) \right)$$

Here, $\mu(Q_{ij})$ is the estimated cardinality the matches of $Q_{ij}$, $|A_t|$ is the estimated size of the adjacency list, specified by $A_t$ if we ignore the non-label predicates, and $\mu(p_t)$ is the estimated selectivity of the pre-satisfied predicate $p_t$ in the adjacency list specified by $A_t$. The formula inside the parentheses is an estimate for the size of one adjacency list that $o$ will access for one input tuple $t$. We discuss why this estimate is broken down into an estimate of the adjacency list size by only considering the label predicates and then selectivities of non-label predicates momentarily. The outside summation, sums the estimated sizes of all of the adjacency lists that will be accessed by $o$ for one input tuple $t$, and the multiplication with $\mu(Q_{ij})$ estimates the size of the adjacency lists that $o$ will access across all of its input tuples.

- Cost of `Filter` operators are ignored. This will be justified when we explain how the cost and cardinalities are estimated momentarily.
- Cost of a plan $P_A$ is simply the sum of the costs of each operator in the plan.

### 3.2.2 Cost and Cardinality Estimation

#### Cardinality and Adjacency List Size Estimations

For labeled queries, we estimate the cardinality of each $Q$ as well as the *average lengths* of the adjacency lists that will be accessed when extending each $t$ using the previous Graphflow's default estimators. Graphflow does these estimations by using a data structure called the *subgraph catalogue* [33]. Briefly, the subgraph catalogue is a graph pattern-based technique to estimate the cardinality of labeled subgraph queries. An example subgraph catalogue is given in Figure 3.2d. Each entry in the catalogue has a key, which is a triple $(Q_{k-1},\ A,\ l_k)$ that consists of of a partial labeled query $Q_{k-1}$ and a list $A$ of additional labeled query edges one can append to $Q_{k-1}$ to extend it to one more query-vertex with label $l_k$. The value in the catalogue for this key is a pair $(|A|,\ \mu)$, where $|A|$ is the estimated average sizes of each of the adjacency lists that would accessed to match each query edge in $A$, and $\mu$ is the average cardinality of the $Q_k$ that would be formed. For example, the fourth entry in Figure 3.2d effectively estimates the `i-cost` and cardinality of extending

a $1^{Customer} \xrightarrow{Friend} 2^{Customer}$ edge, where 1 and 2 are canonical labels used to indicate the source and destination vertex, with two more `Friends` query edges, one from the source and one from the destination, to form a triangle. The entry says that on average, for each edge matching $1^{Customer} \xrightarrow{Friend} 2^{Customer}$, the size of the adjacency list from 1 and 2 both will be of length 5, and on average two triangles will form. The information in the catalogue is used to estimate both the cardinality of each labeled sub-query as well as the sizes of the adjacency lists that the `E/I` operators will use in extending partial matches by one or more query vertex. We refer the reader for the exact formula used to estimate the cardinalities of subgraphs to reference [33].

When there are additional non-label predicates in the query, we estimate the cardinality of queries and sizes of the adjacency lists by assuming independence and multiplying the estimates we get from the catalogue with each non-label predicate's selectivity. Our current implementation supports numeric valued properties. The system uses equi-width histograms [24] that are kept on each numeric property, which are used to estimate the selectivities of non-label predicates. This explains why in `E/I` operators' costs we break the estimate of the adjacency list sizes into an adjacency list estimate size with only label predicates, which is done using the catalogue, and then multiplying this estimate with the selectivities of the non-label predicates on the query edge.

### Final Cost Estimates of Operators

- **`ScanVertex` and `ScanEdge`:** If `ScanVertex` reads labeled vertices, the system keeps track of the number of vertices with different labels and those numbers are used. Similarly `ScanEdge` operator's cost is estimated as the number of edges that will be scanned and the system keeps track of the number of edges with different labels. Recall that if the A+ index `ScanEdge` is configured with is sorted on a property and there is a predicate, the operator also evaluates this query with binary search. In this case, we multiply the estimate of the cost of the ScanEdge with the selectivity of the predicate. We ignore the cost of the binary search as we noticed that incorporating that into our estimates does not make visible difference in the quality of the plans we pick. If `ScanEdge` uses a secondary index, we estimate the number of adjacency lists that will be scanned by operator. For B+ tree secondary indexes this is done by maintaining a histogram of the min or max values of the indexed values. For bitmap indexes, we keep the exact selectivities directly for each categorical value.

- **`E/I`:** We use the cost formula in Equation 3.2.1 using the estimation technique described above. The only additional detail is that similar to `ScanEdge` operator, the

`E/I` operators that use only a single adjacency list that is sorted on some property can be configured with an additional predicate. In this case, we use Equation 3.2.1 to estimate the cost of the operator ignoring this extra predicate (so only considering the label and non-label predicates satisfied by the A+ index that the operator will use) and then multiply this estimate with the selectivity of the additional predicate.

We end this section by giving an example of cost estimation, which will also justify why we ignore the costs of `Filter` operators.

**Example 3.2.2** *Consider the query in Example 3.2.1 above and consider the catalogue information in Figure 3.2d. We also assume the existence of the vertex-bound A+ indexes in the system from Example 2.2.2, which would contain the* ***friend*** *and* ***relative*** *edges of vertices sorted by their* ***customerSince*** *properties. We will refer to these A+ indexes as* ***SortedFriends*** *and* ***SortedRelatives****. We also assume there is one more A+ index in the system, which we refer to as* ***MaleFriends*** *and is defined as:*

---

*CREATE A+ LIST INDEX* $(v_1$*:Customer)*$-[e$*:friend*$]\rightarrow(v_2$*:Customer)*
*WHERE* $v_2$*.gender = 'Male'*

---

*Let us calculate the costs of the three plans shown in Figures 3.2a, 3.2b, and 3.2c on a graph that we assume contains* 10 *vertices.*

- *Plan A: The* ***ScanVertex*** *will have an* ***i-cost*** *of 10 and output 10 vertices, the first* ***E/I*** *according to the first entry in the catalogue will have an adjacency list of size 5 on average, so incurring a cost of 10*5=50 and we also estimate, looking at the $\mu$ field of the first catalogue entry that this operator will output 10*5 = 50 partial matches. Then, we ignore the cost of the following* ***Filter****, but assuming 50% of the customers are male, we estimate that the cardinality of its output will be 25. The next* ***E/I*** *operator will take 25 tuples and looking at the third catalogue entry will access adjacency list of size 3 on average, son incurring an* ***i-cost*** *of 75, so the total* ***i-cost*** *will be 10 + 50 + 75 = 135.*

- *Plan B: Has 10* ***i-cost*** *for* ***ScanVertex*** *as before but uses a more selective* ***MaleFriends*** *A+ index. So the* ***i-cost*** *of the first* ***E/I*** *operator is 10*5*0.5 = 25. The second* ***E/I*** *operator is the same as Plan A's second* ***E/I*** *operator, i.e., that take the same number of tuples and use exactly the same A+ index for its extension, so the* ***i-cost*** *of Plan B would be 10 + 25 + 75 = 110. We emphasize that because Plan B used a*

*more selective A+ index than Plan A, that already pre-satisfied a predicate that Plan A had to explicitly evaluate through a filter, Plan B's second* `E/I` *operator has less cost that Plan B's. This justifies why we ignore the costs of filters. Instead of adding extra costs to Plan A for having to evaluate an additional predicate, we reduce the cost of Plan B for* not having to evaluate the predicate.

- *Plan C: Has 10* `i-cost` *for* `ScanVertex` *as before. The next* `E/I` *operator now performs an intersection by accessing two adjacency lists from two A+ indexes, which according to the first two entries of the catalogue will have sizes of 5 and 3, respectively. So its* `i-cost` *will be 10*8, so the total cost of the plan will be 10 + 80 = 90.*

## 3.3 DP Cost Based Optimizer

In the absence of a `HashJoin` operator, Graphflow has a dynamic programming cost based optimizer which takes as input a query $Q(G_Q(V_Q, E_Q), P)$ and starts by first enumerating all possible single vertex (`Scan Vertex`) and two vertex (`Scan Edge`) plans. Next, for $k = 2..|V_Q|$, the optimizer computes and stores the lowest cost plan[1] for each sub-query $Q_k$, by considering an extension from every $Q_{k-1}$ which contains one fewer query vertex than $Q_k$. As we describe momentarily, we will relax this to look for extensions to $Q_k$ from smaller sub-queries $Q_1...Q_{k-2}$ to as well. The output of the optimizer is the lowest estimated cost plan for $Q$. Next, we describe our modifications to the optimizer. For reference, Algorithm 1 shows the pseudocode for our modified optimizer. The optimizer consults the `IndexStore`, which stores all information about the A+ and secondary indexes the system maintains, e.g., the predicates that each A+ index satisfies.

### 3.3.1 Modifications to the Optimizer

#### Predicate Evaluation

Following every matching operator, the modified optimizer checks if any predicates can be evaluated. All `Filter` operators are pushed as far down as possible and predicates are set to be evaluated in order of decreasing selectivity.

[1]We have removed the *sub-query map* because the effects of intersection caching are irrelevant to our study of indexes in this work.

**Algorithm 1** Modified DP Optimization Algorithm

**Input:** $Q(V_Q, E_Q, P_Q)$

QPMap = $\emptyset$
enumerateAllVertexAndEdgeScans($Q$, QPMap);
**for** $k = 2, ..., |V_Q|$ **do**
    **for** $V_k \subseteq V$ s.t. $|V_k|$ = k **do**
        $Q_k(V_k, E_k, P_k) = \Pi_{V_k} Q$;
        *// Extends and intersections*
        **for** $v_j \in V_k$ let $Q_{k-1}(V_{k-1}, E_{k-1}, P_{k-1}) = \Pi_{V_k - v_j} Q_k$ **do**
            $P_{new} = P_k$ - $P_{k-1}$;
            **for** iCombination ∈ getAllIndexCombinations($Q_{k-1}$, $Q_k$) **do**
                $P_r$ = getUnsatisfiedPredicates($P_{new}$, iCombination)
                plan = QPMap($Q_{k-1}$).extend($Q_k$, iCombination).filter($P_r$);
                **if** cost(plan) < cost(QPMap($Q_k$)) **then**
                    QPMap($Q_k$) = plan;
        *// Multi-extends*
        **for** $j = 1, ...,$ i - 2 **do**
            **for** $V_j \subseteq V_i$ s.t. $|V_j|$ = j let $Q_j = \Pi_{V_j} Q_i$ **do**
                **if** $Q_j$ in QPMap **then**
                    **for** $V_e \in$ getDisjointVertexSets($Q_j$, $Q_k$) **do**
                        $Q_i = \Pi_{V_j + V_e} Q_k$; $P_{new} = P_k$ - $P_i$;
                        **for** iCombination ∈ getAllIndexCombinations($Q_i$, $Q_k$) **do**
                            plan = QPMap($Q_i$).extend($Q_k$, iCombination).filter($P_r$);
                            **if** cost(plan) < cost(QPMap($Q_k$)) **then**
                                QPMap($Q_k$) = plan;
**return** QPMap(Q);

### `E/I` Enumeration

As in the previous version of Graphflow, the optimizer searches for extensions from each $Q_{k-1}$ to $Q_k$ using one more A+ indexes. Since these sub-queries can have non-label predicates, the optimizer searches the `IndexStore` for all A+ indexes, per-vertex or per-edge, that could extend $Q_{k-1}$ to $Q_k$ and the lists of additional label or non-label predicates these possible extensions could satisfy. The details of this search is omitted and is done in the `getAllIndexCombinations` call on line 9. This is where we make a connection to query rewriting using materialized views. If we think about A+ indexes as materialized views, the optimizer effectively searches for possible query rewrites, where different A+ index extensions effectively with a single join satisfy different predicates in the query. For each possible extension, the optimizer keeps track of the additional predicates that the possible extensions would not satisfy and adds a `Filter` operator to the plans with the appropriate predicates on line 10. Then using the cost estimation techniques we described, the optimizer in lines 12 and 13 pick the lowest estimated cost of extending $Q_{k-1}$ to $Q_k$. In addition , in lines 15-23, the optimizer also looks for extending all smaller sub-queries $Q_1, ..., Q_{k-2}$ to $Qk$ directly.

# Chapter 4

# Implementation

This chapter describes the implementation of our A+ and secondary indexes. Section 4.1 describes the structure of Graphflow's adjacency lists in memory. Section 4.2 describes how A+ indexes are implemented how they are stored in the index store. And finally, Section 4.3 describes our implementation of secondary B+ tree and map indexes.

## 4.1 Adjacency Lists

Every adjacency list in Graphflow is a unit of contiguous data in memory, holding information about the incoming or outgoing edges for either a vertex or an edge. Every adjacency list consists of primitive arrays containing edge IDs and the corresponding neighbor vertex IDs.

### 4.1.1 Partitioned Adjacency Lists

Recall from Section 2.5 that for equality predicates, users may use the asterisk (*) notation to create multiple A+ indexes, one for each value in the domain of a categorical property. If a single such predicate exists in the index creation query, instead of creating multiple A+ indexes for every value of the property, we create a single A+ index which holds *partitioned* adjacency lists. A partitioned adjacency list is a CSR-like structure which, apart from holding arrays for edge and neighbor vertex IDs, contains two additional arrays. One of these arrays contains all the unique values of the property which are present in the adjacency list on the stored vertices or edges. The second array contains a pointer for each

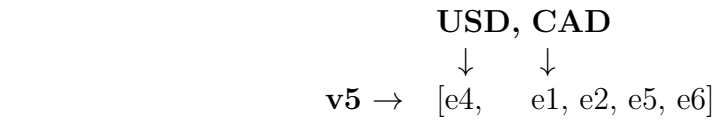



Figure 4.1: Partitioned adjacency list example for e.currency = *. Edge e4 has USD currency while edges e1, e2, e5 and e6 have CAD.

of these values to *groups* in the adjacency list which consist of vertices or edges, all having the same specific value of the property. Figure 4.1 shows an example of such a partitioned adjacency list.

## 4.2 A+ Indexes

Every A+ index defined in Graphflow is assigned a unique name and consists of a contiguous array of adjacency lists. Per-vertex A+ indexes take a vertex ID and return an adjacency list using a map in which each global vertex ID is mapped to a local array index in the adjacency lists array. The array index is a local vertex ID defined per label where for a given vertex label with $m$ vertices, the local vertex IDs assigned are from 0 to $m$-1. The mapping for global to local vertex IDs is stored in a single object shared between all indexes and with the vertex property columnar store. Each per-edge A+ index contains its own mapping from global edge IDs to array indices in the adjacency lists array. The mapping in all indexes compacts the size of the array for a smaller memory footprint.

Index creation queries are treated as regular queries in Graphflow. Query plans for these queries have an additional special `SINK` operator appended to the end which is responsible for aggregating the result of the query into adjacency lists and encapsulating them into an index structure. Fixed pre-defined query execution plans exist in the system for the creation of new indexes.

### 4.2.1 Index Store

Every A+ index in Graphflow is assigned a unique name and stored in the index store data structure. The default indexes in the system cannot be dropped and no filtering predicates can be defined on them.

The index store uses a linear index matching technique to look for stored indexes of a particular form. It is necessary to look for exact matches to safeguard against creat-

ing duplicate indexes in the system. The query optimizer, in addition to exact matches, requires partial index matches as well because indexes having predicates which partially match the predicates in a query can be used for query execution. For every edge extension, the query optimizer gives the index store an `index descriptor`, which stores the label of the query edge, the labels of the source and destination vertices and the predicates on the edge and the vertices. The index store first does a map lookup for all indexes matching the `(SourceVertexLabel, EdgeLabel, DestinationVertexLabel)` triple in the descriptor. Then, it goes through each index match one at a time and compares the predicates the index pre-satisfies with the predicates provided in the descriptor. If an index has more restrictive predicates than those in the query, the index is discarded. A list of matching indexes is returned to the optimizer.

### 4.2.2 Default Index Configuration

The default index configuration in Graphflow consists of two A+ indexes existing for every `(SourceVertexLabel, EdgeLabel, DestinationVertexLabel)` combination in the graph. One index is for incoming edges and the other one for outgoing edges. Each index has a sorting on the IDs of neighbor vertices.

Database administrators have the option to provide a `storage configuration file` to update these default indexes. An update to a default index may involve changing the sort on it from neighbor ID to vertex or edge property. These operations give DBAs freedom to mold the data layout to suit the schema of their datasets and the needs of their workloads. There may be a small memory cost associated with these update operations because some types of adjacency lists require more memory than others. Our experiments in Chapter 5 show that, with a small cost to memory, these configuration changes can bring significant performance improvement to query runtime.

## 4.3 Secondary Index Implementation in Graphflow

Secondary B+ tree and map indexes must be defined on top of existing A+ indexes. The index store keeps track of every A+ index's auxiliary indexes. Whenever a plan is using a specific A+ index for the initial `Edge Scan` operator and an indexed property predicate is present in the query, an auxiliary index is selected and the operator is given an iterator over this auxiliary index. This allows the operator to discard adjacency lists which have no edges that satisfy the predicate. The following `Filter` operator in the plan therefore has less work to do.

### B+ Tree Indexes

We use a standard implementation of B+ tree written from scratch. The keys in the B+ tree are always `int` property values. When the index is created on a partitioned A+ index, the values are of the primitive `long` datatype where the first 32 bits represent the global ID of the vertex and the last 32 represent the index of the partition in the adjacency list. If the index is created for a regular A+ index, the values are of datatype `int` and represent the global vertex ID. In both cases, an iterator over the A+ index uses its global vertex ID to local array index map to grab the adjacency list corresponding to the global vertex ID and gives it to the operator.

### Bitmap Indexes

A bitmap index is a collection of bitmaps, in which for each value in the domain of the indexed property, a bitmap is created, with the number of bits equal to the number of adjacency lists in the A+ index. Each index, therefore, holds as many bitmaps as the values in the domain of the indexed property. For a bitmap corresponding to any given property value, a `True` bit represents the existence of an edge in the corresponding adjacency list with the same value of the property.

# Chapter 5

# Evaluation

In this chapter, we evaluate the performance of A+ and auxiliary indexes. Our experiments are demonstrative and aim to answer the following questions:

1. **A+ index performance:** How is the performance of A+ indexes compared to different types of default storage schemes found in existing GDBMSs? And what is the trade-off between performance benefits and memory usage?
2. **Optimizer goodness:** How good are the plans our optimizer picks for various A+ index configurations including workload specific ones?
3. **Auxiliary Indexes performance:** How efficient are auxiliary B+ tree and Bitmap indexes? What is the exact memory performance trade-off they provide compared to secondary indexes that index all edges.

Our preliminary results comparing against other systems, such as Neo4j showed that our performance is faster on a broad range of queries. Such baseline comparisons are common in the database community but we think there is little to learn from them because there are many fundamental differences between Graphflow and existing GDBMSs, so attributing them to A+ indexes is difficult. For completeness of this work, in future work, we plan to conduct several comparative evaluations.

## 5.1 Hardware Setup

For all our experiments, we use a single machine that has two Intel E5-2670 @2.6GHzCPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical cores. We only use one logical core. We set the maximum size of the JVM heap to 500 GB and keep JVM's default minimum size.

## 5.2 Queries & Datasets

In our evaluation, we focus on queries whose subgraph structure is acyclic[1] for two reasons. First, the edge label partitioned adjacency lists that are sorted by neighbour IDs, which were used in the previous version of Graphflow and described in reference [33], are good for queries whose structures are cyclic. So, we instead focus on sets of acyclic queries which require different adjacency list structures. Second there is evidence that overwhelming majority of graph queries are acyclic in practice. For example, reference [10], which studies query logs that consist of over 180 million SPARQL queries over four different knowledge graphs and reports that 99.95% of the submitted queries were acyclic.

We rely on two datasets to build query workloads for two different applications:

- **Application 1: Social Network Analysis**:
  We use the LDBC SnB[2] data generator to create synthetic social network graphs. Figure 5.1 shows the schema of the LDBC SnB graph. The data generator allows choosing a scaling factor to control for the size of the generated graph.
- **Application 2: Fraud Detection**:
  We are unaware of an existing public financial dataset, so we repurpose the LiveJournal dataset from SNAP [30]. LiveJournal is a Russian social networking service. The dataset contains 4.8M nodes and 68.9M edges. We modify the dataset such that each node is an ACCOUNT vertex and each edge represents a TRANSFER between two accounts. Nodes have a numerical property BALANCE in the range 0 to 50,000 and edges have three properties: (1) Date: a unix time stamp (5 years ago to present); (2) Amount: in the range 1 to 1,000; and (3) Currency: USD, CAD, or Euro.

[1] We note however that the relational versions of some of our queries over a Vertex and an Edge table are cyclic when we consider their *join hypergraphs*. In a join hypergraph, each node is an attribute and each hyperedge is a relational table, so in our case either the Vertex or Edge table.

[2] LDBC = Linked Data Benchmark Council & SnB = Social Network Benchmark

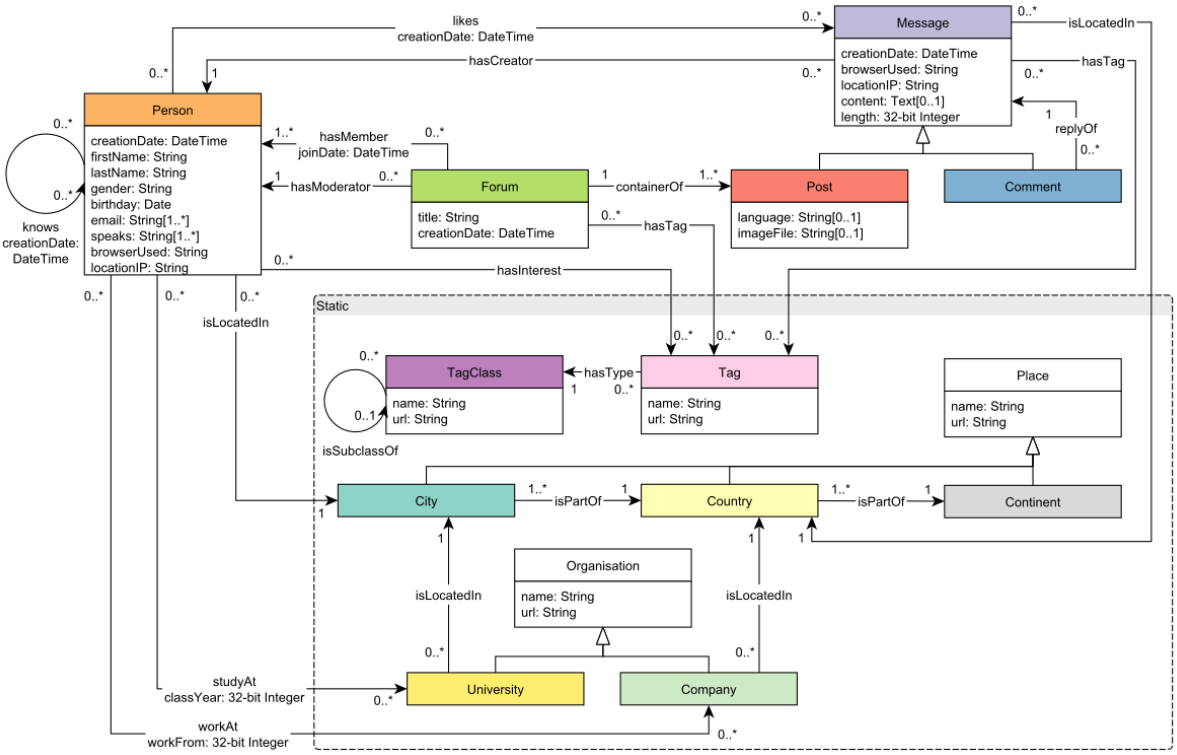



Figure 5.1: LDBC SnB Schema
(Obtained from LDBC SnB specification document v0.32).

## 5.3 A+ Index Performance

A+ indexes provide performance benefits by helping with label and predicate filtering in a query. We demonstrate the flexibility of our A+ indexes and the performance benefits as well as memory and speed-up trade-offs. In this section, we compare different index configurations on the two aforementioned applications. Reasoning about the workload, in our case demonstrative queries selected per application, we create a workload specific index configuration, which we compare with baseline index configurations. We overview the index configurations below.

### 5.3.1 Baseline Index Configurations

There three baseline index configurations we compare against:

- `Simple`: The configuration is made of two indexes, the forward and backward adjacency lists sorted by neighbour vertex ID.
- `Neo4J`: The configuration builds on top of the `Simple` configuration by adding the predicate e.label=$\star$. This is effectively Neo4j's [35] default adjacency list configuration.
- `Graphflow`: Two indexes exists, forward and backward adjacency lists, for each `(SourceVertexLabel, EdgeLabel, DestinationVertexLabel)` triple in the system. Each adjacency is sorted by neighbour vertex ID with no predicates applied. As described in Chapter 4, this is the default index configuration for Graphflow.

### 5.3.2 Workload-specific Index Configuration

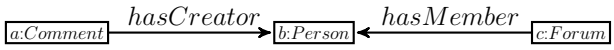
The workload specific index configuration depends on the application we will be testing. We will have two workload-specific configurations, one for each application. The configuration in this case updates the `Graphflow` index configuration and is expected to represent the indexes a DBA would naturally create.

### 5.3.3 Comparison per Application

For each application, we present the set of demonstrative queries and then the *plan spectrum charts* i.e. the set of all plans generated with their run-time. The plan spectrum charts contain the plan our query optimizer picks which is shown by an $\times$ marker. Since the optimizer relies on dynamic programming and for each subquery picks the best plan, we introduce the notion of a *finalist* plan. For a query Q with n vertices, a finalist plan is a plan which does not get pruned for any subquery of Q and is chosen or pruned only when picking the best plan for Q. The plan spectrums contain the finalist plans as filled colored circles and pruned plans as circles with a hollow middle.

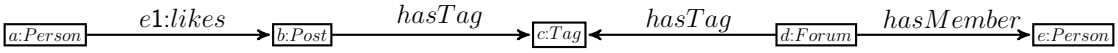
#### Social Network Analysis

The goal of this experiment is to test the performance benefits brought by our per-vertex A+ indexes. We focus on comparisons where the memory required for each baseline configuration and workload-specific configuration is very similar. Specifically we will keep the number of edges indexed in any configuration similar, although there will be minor memory consumption differences between in each configuration due to several implementation



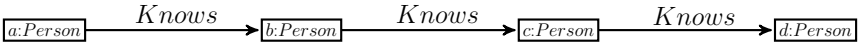

$WHERE$ b.gender=$Male$, c.creationDate>=$P$-50, a.length>=$P$-50

(a) Q1.



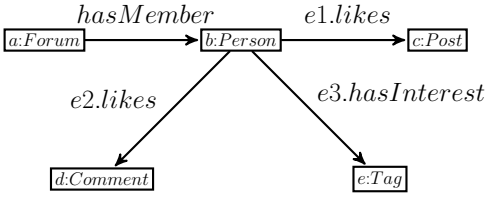

$WHERE$ e.gender=$Male$, a.browserUsed=$Chrome$, d.creationDate>$P$-75, e1.date<$P$-50

(b) Q2.



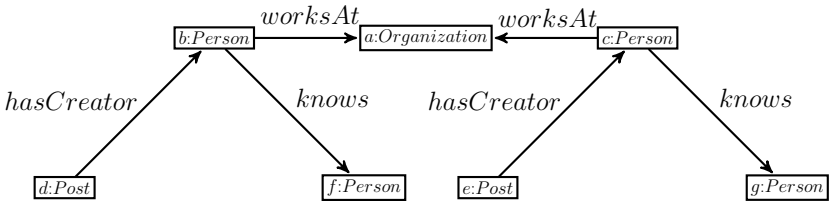

$WHERE$ a.birthday>b.birthday, c.birthday<d.birthday

(c) Q3.




$WHERE$ b.gender=$Male$, e1.date>$P$-90, e2.date>$P$-90, e3.date>$P$-90

(d) Q4.




$WHERE$ b.gender=$Male$, c.gender=$Female$, f.birthday=g.birthday, d.creationDate= e.creationDate

(e) Q5.

Figure 5.2: Social Network Application Queries.

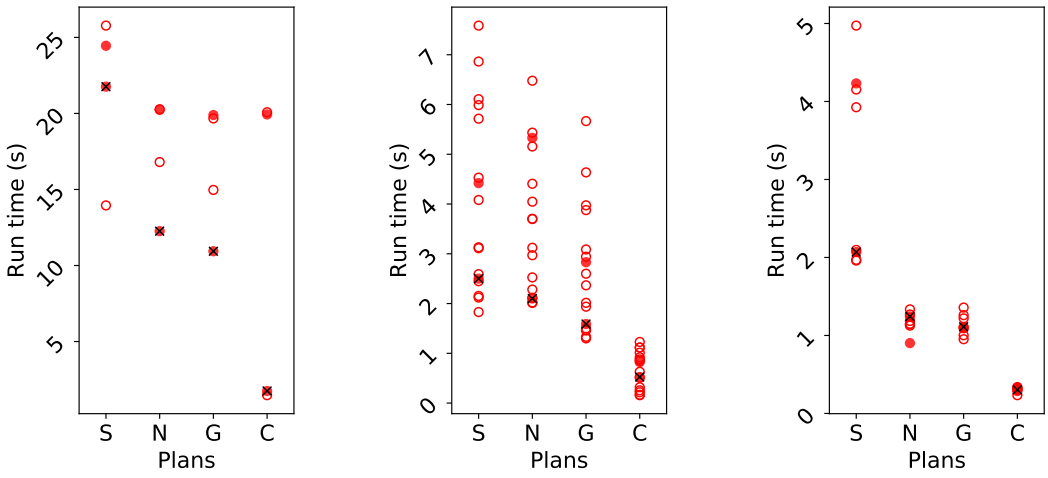
details that we will explain. Our goal is to demonstrate that with minimal additional memory costs, users can configure their adjacency lists to specific workloads to gain significant performance benefits.

Figure 5.2 shows 5 demonstrative queries we consider for a social network analysis application. We use the `P-X` notation in predicates to represent a value of the predicate property at the Xth percentile. For example, the predicate $e1.date <$ `P-51` has a selectivity of 50% because half the values of date property lie below the 51th percentile.

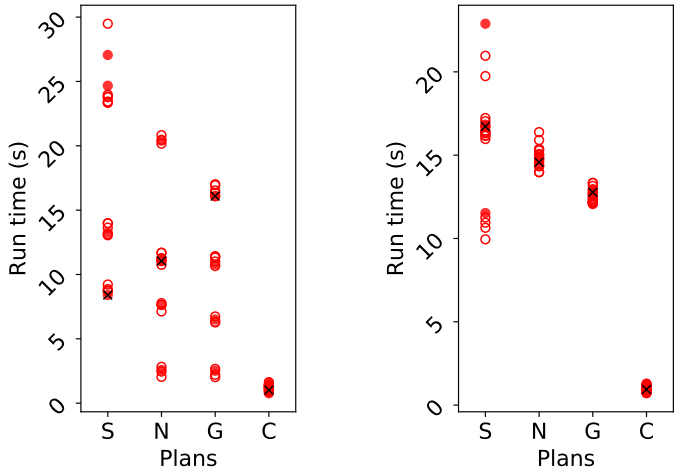
The workload-specific configuration for this application updates some of the A+ indexes in the default `Graphflow` configuration in two possible ways: (i) by adding a sorting criteria; and/or (ii) by "partitioning" the A+ index with an equality match-all (*) predicate on a vertex or edge property. Recall that the default `Graphflow` configuration has one A+ index for each `(SourceVertexLabel, EdgeLabel, DestinationVertexLabel)` combination. The updates we do to the indexes in the default configuration are shown in Table 5.1. Any index in the default `Graphflow` configuration that is not shown in the table is also part of the workload-specific configuration as is. Consider the indexes 7 and 10 in Table 5.1 that add gender = * predicate to two default indexes. Q1, Q2, Q4, and Q5, which all contain an equality predicate on gender, will gain speed-ups from these two indexes. For example, in the presence of index 10, the `Person` vertices with a specific value of the `gender` property can be quickly accessed when extending forward from a `Forum` vertex or, in the presence of index 7, backwards from an `Organisation` vertex.

Figure 5.3 shows the plan spectrum charts for this application. First, we expect the `Simple` configuration to have worse plans than the other configurations because the plans in every other configuration avoid the execution of label predicates whereas the plans in the `Simple` configuration require explicitly executing these predicates in extra `Filter` operations. Second, we expect the workload-specific configuration to shift the plan space of `Graphflow` configuration's plan space as the indexes allow for more efficient plans.

The plans spectrums of these queries are shown in Figure 5.3. For all queries except Q5, the spectrum of plans under the `Simple` configuration is worse than other plans. Our understanding of why there are plans for Q5 under the `Simple` configuration that are more efficient than `Neo4j` and `Graphflow` configuration is not complete. We suspect this is due to a CPU caching effect that is happening in those plans under the `Simple` configuration but at the time of writing, we could not fully investigate the issue. More importantly, we emphasize that for all queries the plans under the workload-specific configurations are significantly faster than `Graphflow` configuration as well as the others. The spectrums demonstrate the robustness of A+ indexes as even the worst plans under the workload-specific configuration are broadly better than the best plan under other configurations.




(a) SnB Q1 (b) SnB Q2 (c) SnB Q3




(d) SnB Q4 (e) SnB Q5

Figure 5.3: Social network application plan spectrum charts. `S` represents the `Simple` configuration, `N` represents the `Neo4j` configuration, `G` represents the default `Graphflow` configuration and `C` represents the `Workflow Specific Configuration`.

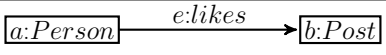
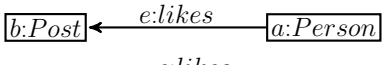
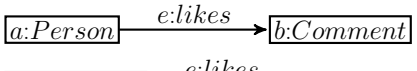
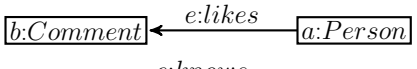
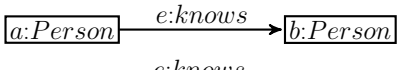
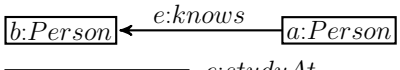
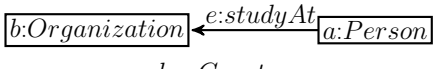
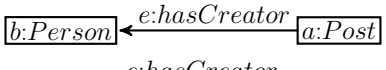
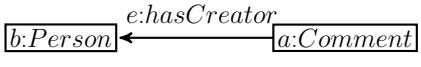
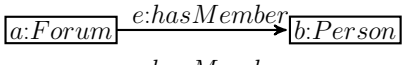
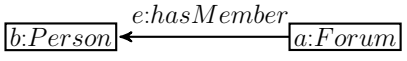
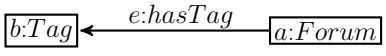
| Num. | Index | Sorting Property | Predicate |
|---|---|---|---|
| 1 |  | e.date | - |
| 2 |  | e.date | a.browserUsed = * |
| 3 |  | e.date | - |
| 4 |  | e.date | - |
| 5 |  | b.birthYear | - |
| 6 |  | a.birthYear | - |
| 7 |  | - | a.gender = * |
| 8 |  | a.creationDate | - |
| 9 |  | a.length | - |
| 10 |  | e.date | b.gender = * |
| 11 |  | e.date | - |
| 12 |  | a.creationDate | - |

Table 5.1: Social network application workload specific configuration.

This indicates that even if the optimizer makes mistakes, it will still able to pick highly efficient plans for this workload.

Table 5.2 summarizes the memory usage, best plan, and the picked plan run-time for each configuration on the 5 queries. We compare the memory usage of each configuration relative to the memory usage of the `Simple` configuration. The `Neo4j` configuration is significantly more expensive because every adjacency list stored is partitioned by edge labels and needs to store the labels of the edges it is storing, as well as pointers to their corresponding partitions. The indexes in the `Graphflow` configuration are not partitioned but multiple adjacency lists exist for each vertex in the different default indexes. This makes it more expensive than the `Simple` configuration but for the `SNB` dataset, it is cheaper than `Neo4j` and brings slight performance benefits. The workload specific configuration is slightly more expensive than `Neo4j` and `Graphflow` because it updates the `Graphflow` configuration and partitions some of the indexes.

| | | Simple | Neo4j | G.flow | Specific |
|---|---|---|---|---|---|
| | **Mem (MBs)** | 513 (1x) | 706 (1.38x) | 686 (1.33x) | 784 (1.53x) |
| **Q1** | BEST | 13.9 | 12.2 | 10.9 | 1.5 |
| | SELECTED | 21.7 | 12.2 | 10.9 | 1.7 |
| **Q2** | BEST | 1.8 | 2.0 | 1.3 | 0.2 |
| | SELECTED | 2.5 | 2.1 | 1.6 | 0.5 |
| **Q3** | BEST | 1.9 | 0.9 | 0.9 | 0.2 |
| | SELECTED | 2.0 | 1.2 | 1.1 | 0.3 |
| **Q4** | BEST | 8.4 | 2.0 | 2.0 | 0.8 |
| | SELECTED | 8.4 | 11.0 | 16.0 | 1.0 |
| **Q5** | BEST | 9.9 | 13.9 | 12.0 | 0.7 |
| | SELECTED | 16.7 | 14.5 | 12.7 | 0.9 |

Table 5.2: Social network application memory usage and run-time (secs) trade-off.

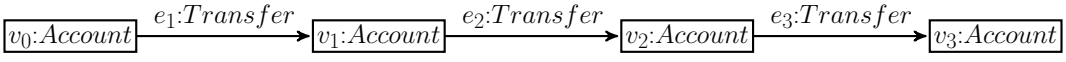
### Fraud Detection Queries

The goal of this experiment is to demonstrate when per-edge A+ indexes are preferable over per-vertex A+ indexes and to test the performance benefits they bring. Figure 5.4 shows 4 demonstrative queries we consider for a fraud detection application. These queries are all variants of money flow queries motivated by a real fraud detection application at a large e-commerce company's payment system, whose name we omit due to privacy reasons[3].

Table 5.3 lists the indexes for `VB` and `EB`, the two workload specific configurations we create for this application. Both configurations build upon the `Graphflow` configuration, where `VB`, the per-vertex A+ index configuration adds the first two indexes and `EB`, the per-edge A+ index configuration only adds the last index. Unlike the workload specific configuration for the Social Network Analysis application, we do not modify the indexes in the `Graphflow` configuration for this experiment. These default indexes are also not dropped because the indexes in our workload specific configurations all have filtering predicates and doing so will render the configurations useless for queries not present in our workload.

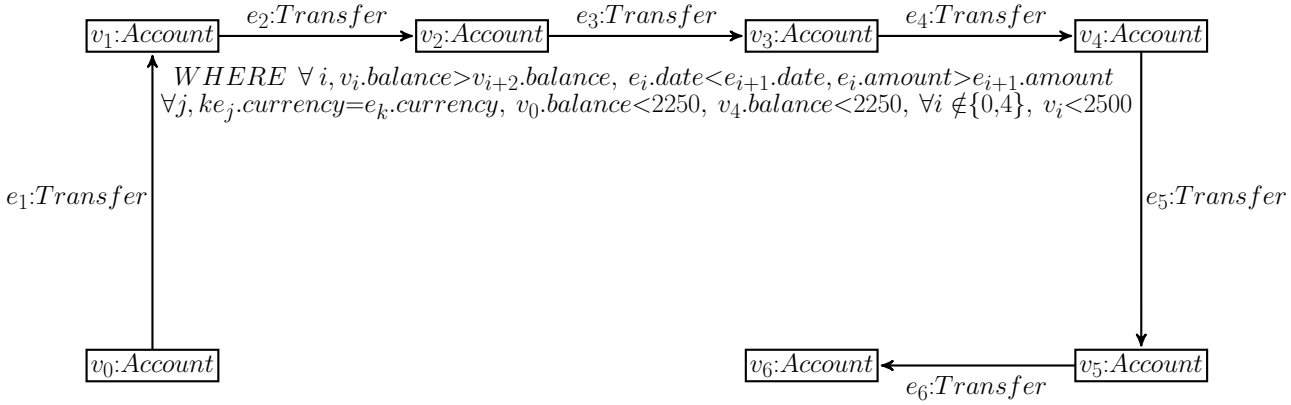
Figure 5.5 shows the plan spectrum charts for these queries. Since the queries for this experiment are expensive and are being executed on a large graph, it is prohibitive to run every possible plan. We therefore only run the top ten plans, which the optimizer thinks are cheapest, for each query. As before, among the three baseline configurations,

[3]Private communication with employees of the company.

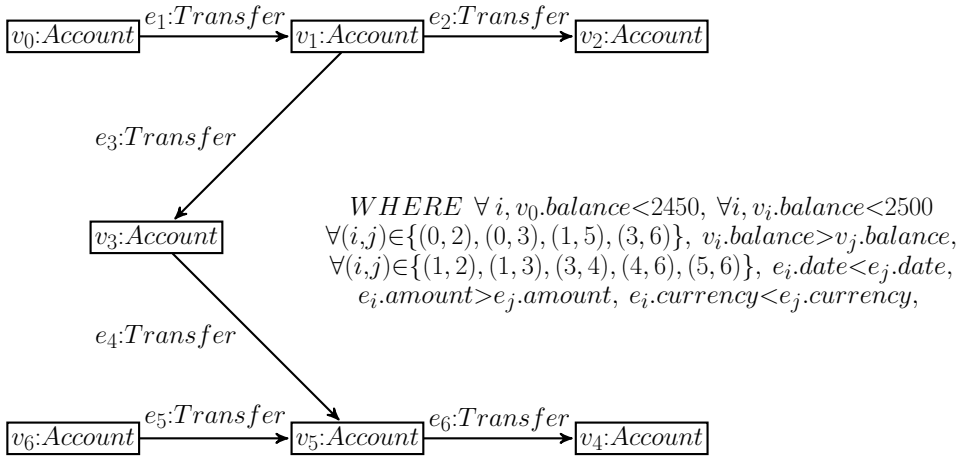



$WHERE\ \forall\, i, v_i.balance{>}v_{i+2}.balance,\ e_i.date{<}e_{i+1}.date,\ e_i.amount{>}e_{i+1}.amount,$
$\forall j,k, e_j.currency{=}e_k.currency,\ v_0.balance{<}2450,\ \forall\, l \neq 0, v_l.balance{<}2500$

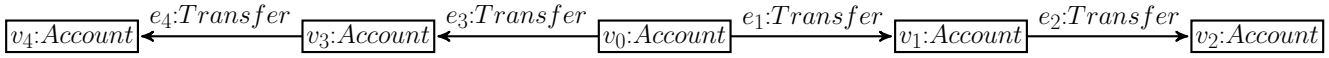
(a) Query 1.




(b) Query 2.




(c) Query 3.




$WHERE\ \forall\, i, v_i.balance{>}v_{i+2}.balance,\ v_2.balance{=}v_4.balance,\ v_0.balance{<}2450,\ \forall j \neq 0 : v_j.balance{<}2500$
$\forall\, i, j{\in}\{(1{,}2),(3{,}4)\}\ e_i.date{<}e_j.date,\ e_i.amount{>}e_j.amount,\ e_i.currency{=}e_j.currency$

(d) Query 4.

Figure 5.4: Fraud Detection Application Queries.

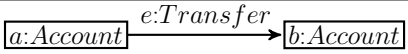
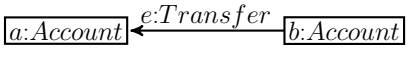
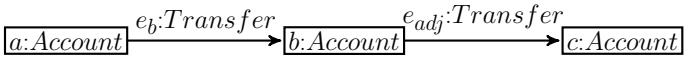
| Index | Sorting | Predicates |
|---|---|---|
|  | e.date | a.bal < 2500 |
|  | e.date | b.bal < 2500 |
|  | c.bal | c.bal < a.bal,<br>$e_{adj}$.date > $e_b$.date ,<br>$e_{adj}$.amt < $e_b$.amt,<br>$e_{adj}$.curr = $e_b$.curr |

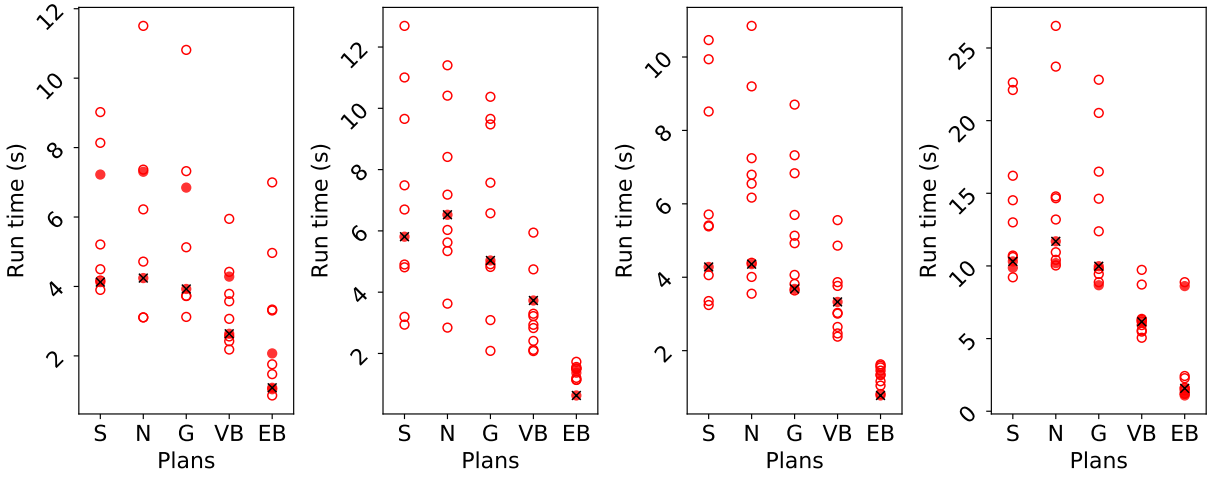
Table 5.3: Fraud detection application workload specific configurations. The first two indexes are used for a per-vertex configuration and the last is used for a per-edge configuration.

| | | Simple | Neo4j | G.flow | PVertex | PEdge |
|---|---|---|---|---|---|---|
| | **Mem (GBs)** | 1.74 (1x) | 2.0 (1.15x) | 1.74 (1x) | 1.84 (1.05x) | 1.77 (1.02x) |
| **Q1** | Best | 3.9 | 3.1 | 3.1 | 2.1 | 0.9 |
| | Selected | 4.1 | 4.2 | 3.9 | 2.6 | 1.1 |
| **Q2** | Best | 2.9 | 2.8 | 2.0 | 2.1 | 0.6 |
| | Selected | 5.8 | 6.5 | 5.0 | 3.7 | 0.6 |
| **Q3** | Best | 3.2 | 3.5 | 3.6 | 2.3 | 0.7 |
| | Selected | 4.2 | 4.3 | 3.6 | 3.3 | 0.7 |
| **Q4** | Best | 9.2 | 10.0 | 8.6 | 5.0 | 0.9 |
| | Selected | 10.3 | 11.6 | 9.9 | 6.2 | 0.9 |

Table 5.4: Fraud detection application memory usage and run-time (secs) trade-off.

the spectrums in Figure 5.5 show that the `Graphflow` configuration slightly outperforms the other two. From the spectrums, we see that the per-edge A+ index configuration, `EB`, outperforms the per-vertex A+ index configuration, `VB`. This is because the queries in our workload contain a lot of predicates between adjacent edges which are pre-satisfied by index in the per-edge A+ index configuration.

Table 5.4 summarizes the memory usage, best plan, and the picked plan run-time for each configuration on the 4 queries for this experiment. Recall from Table 5.2 that, for the social network analysis application, every configuration was significantly more expensive compared to the `Simple` configuration. This is not the case for this experiment because all the nodes in the dataset are of a single label, `Account`, and all the edge are of label `Transfer`. For the `Neo4j` configuration, this means that each adjacency list will effectively




(a) Fraud Q1 (b) Fraud Q2 (c) Fraud Q3 (d) Fraud Q4

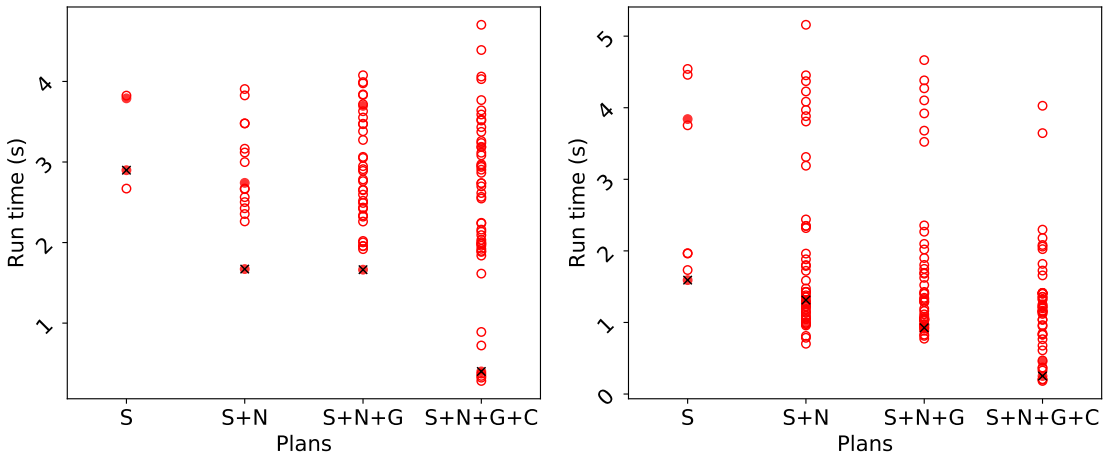
Figure 5.5: Fraud detection application plan spectrum charts. `S` represents the `Simple` configuration, `N` represents the `Neo4j` configuration, `G` represents the default `Graphflow` configuration, `VB` represents the per-vertex workflow specific configuration and `EB` represents the per-edge workload specific configuration.

have only one partition. For the `Graphflow` configuration, this means that the indexes are identical to the indexes in the `Simple` configuration. Both our workload specific configurations for this experiment use an insignificant amount of additional memory because the predicate pre-satisfied by the indexes these configurations add are very selective. The per-edge A+ index configuration in this experiment is shown to bring a magnitude of performance improvement, using 12% less memory than the `Neo4j` configuration.

## 5.4 Optimizer Goodness

From the prior Section 5.3, each of the spectrums contained a $\times$ marker which showed the plan picked by the optimizer. Furthermore, the spectrums contained two types of plans, the finalists and the pruned ones. When many different indexes can be used for each extension during query planning, two questions arise: (1) How far is the optimizer's selection from the best plan? (2) Does the optimizer prune the correct plans, i.e. plans with less selective indexes?

To answer these questions, instead of testing index configurations one at a time, we



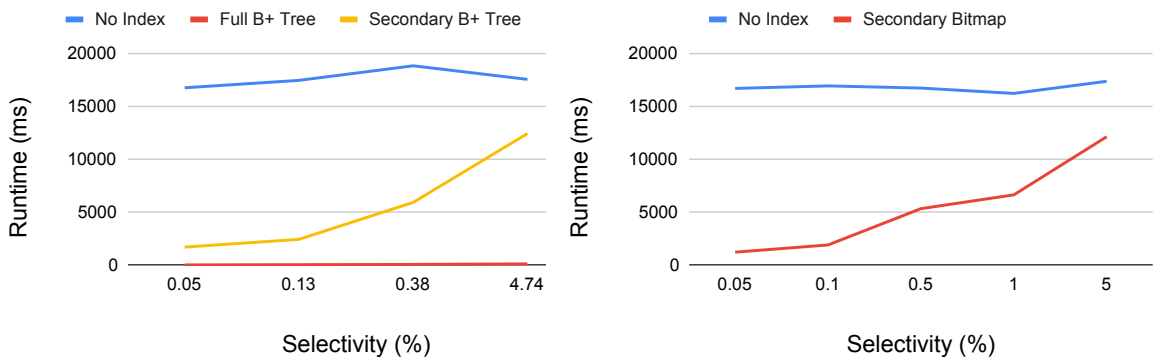

(a) SNB Q1 (b) SNB Q3

Figure 5.6: Optimizer Goodness Spectrum Charts.

append them to each other in this experiment. The `Neo4j` configuration, for example, now also contains all the indexes from the previous `Simple` configuration. The final workload specific configuration contains all the indexes from every other configuration. We run this experiment using `Q1` and `Q3` from Figure 5.2 on 0.3 scale of the SNB graph. We limit the number of generated plans to 50, so if the optimizer generates more, we rely on sampling. Figure 5.6 shows the spectrums for both these queries. It can be seen that the optimizer still picks good plans in the presence of many different indexes.

## 5.5 Auxiliary Indexes Performance

The goal of the experiments in this section is to demonstrate that our secondary indexes can help reduce runtime of queries with highly selective predicates. For the experiments in the previous section, the property values of the nodes and edges that we put on LiveJournal were uniformally distributed. For these experiments, we make the `amount` property on edges normally distributed with a mean of 50 and a standard deviation of 15. This roughly gives us a range of values between 0 and 100. Next, instead of the three uniformally distributed values of the `currency` property on these edges as before, we put 9 values with a predetermined selectivity of each. All our queries in this experiment have a single edge and a single edge predicate.




(a) B+ Tree Index

(b) Bitmap Index

Figure 5.7: Secondary index experiments.

### 5.5.1 B+ Tree Indexes

Figure 5.7a shows our results for single edge queries with the predicates `e.amount` $<$ `x`, where `x` $\in \{1, 5, 10, 25\}$. Both the B+ tree indexes were created on the default backward adjacency lists index of the graph. The full B+ tree index was indexing every edge while the secondary B+ tree index was indexing the `min` aggregate of every adjacency list in the index. The size of the full B+ tree index was ~592 MBs while the secondary B+ index was only using ~16 MBs (98% smaller) of storage. It is apparent from the graph in Figure 5.7a that a secondary B+ tree index on an A+ index can be quite useful for highly selective predicates.

### 5.5.2 Bitmap Indexes

Figure 5.7b shows our results for single edge queries with the predicates `e.currency` $=$ `x`, where `x` is one 5 different values with pre-determined selectivitities. The bitmap index is created on the default backward adjacency lists index of the graph. The size of generated bitmap index is 3.65 MBs. From the graph in Figure 5.7b, it is apparent that such an index can also be quite useful for highly selective predicates. Full map indexes, which allow O(1) access to edges satisfying an equality predicate, are not a part of this work but we expect them to be faster and more expensive in terms of storage than our bitmap indexes.

# Chapter 6

# Related Work

We begin this chapter by reviewing related work in existing adjacency list indexes in RDF systems and GDBMSs in Section 6.1. In Section 6.2, we review other indexes for queries other than the subgraph queries and predicates that we consider in this thesis. Finally, in Section 6.3, we review work in using materialized views in query optimization.

## 6.1 Indexes in RDF and Graph Systems

Previous work has studied indexing techniques for RDF systems which generally store data in the form of tables. Reference [1] outlines a technique for vertically partitioning RDF data by creating a table for each property in the data. Hexastore [44] takes this approach further by creating indexes for each RDF element, instead of just for properties. RDF-3X [37] creates B+ tree indexes on a giant triples table. SAP Hana [23] uses adjacency lists as indexes on top of relational tables to speed up graph queries. These approaches focus on tabular storage models, which is common in RDF systems. In contrast, our work focuses on native adjacency list storage which is commonly a linked-list like structured in native GDBMSs.

For systems storing the edges in adjacency list structures, a common technique is to store them in a *compressed sparse row* (CSR) format [9]. CSR is a compression technique for matrices in which non-zero values for every row of the matrix are stored in partitions in a contiguous array and pointers are used to access the values for any specific row. To our best knowledge, no existing GDBMS uses CSR storage but many existing graph processing systems, such as Green-Marl and Ligra adapt this type storage as it is known to give good

cache locality and decreases the size of the adjacency lists stored. Some of prior work study different aspects of CSR storage format. For example, reference [42] studies CSR-like partitioning techniques for large arrays containing incoming and outgoing edges, based on source and destination vertices. Reference [13] uses a similar technique in addition to re-pairing for graph compression in memory. Finally, [8] outlines a CSR-like structure for compressed contiguous adjacency lists (*adjacency table*). Our work does not focus on optimizing the physical layout of the edges in memory; instead we focus on providing fine grained control over which edges are stored in adjacency lists.

## 6.2 Other Indexes in Graph Databases

In this thesis, we introduced A+ indexes, which allow flexible adjacency list indexes for GDBMS, and demonstrated how A+ indexes can speed up query evaluation for subgraph queries with predicates. We next review prior work that have introduced advanced indexes to speed up other classes of queries: (i) complex subgraph queries; (ii) shortest paths queries; and (iii) regular path queries.

- Complex subgraph queries: Many approaches rely on indexes to speed up complex subgraph queries. GADDI [47] records the distance of each 2-label pair in the graph. Indexes in [48, 50] use a signature representing neighborhoods within a given distance for each vertex. Certain techniques index all of the triangles in the graph. Finally certain techniques rely on light weight indexes as part of the algorithms, these technique build a small index per query which provides faster access to query vertex candidates and are different from techniques indexing the whole graph over time. Our A+ indexes technique aims to provide speedups to a workload set and be maintained while these light weight indexes in the case of DP-iso [22] and CECI [6] for example provide speedups over the query for that particular run-time instance.

- Shortest paths queries: Evaluating shortest paths require indexing for speedups as well. Reference [14] uses a 2-hop cover, which is a collection of shortest paths such that any two vertices $v_i$ and $v_j$, the shortest path between them is a concatenation of two paths from the collection. Reference [45] explores orbit adjacency instead of vertex adjacency and local symmetry to obtain compact breadth-first-search trees to speedup the queries. Another type of work requires the top-k shortest path indexes for applications such as link prediction. Finally, reference [3] needs to find top-k shortest path and relies on the pruned landmark labelling technique [4], which keeps track of distance label for vertices to build an index and obtain the top-k shortest

paths from it quickly. All of these indexes are highly specialized for the shortest path queries meanwhile A+ indexes are general and can be used with classic shortest path algorithms.

- Regular path queries (RPQs): RPQs check the existence of a path between two nodes. There has been a lot of regular path queries using path indexes for examples in the context of XML [31] but we do not go into them in detail here.

## 6.3 Using Materialized Views in Query Optimization

Reference [20] outlines two types of approaches to using views in query optimization, based directly on the style of the optimizers in which these approaches are integrated into: (i) System-R style; and (ii) transformational approaches. Because Graphflow's optimizer is bottom-up System-R style optimizer, our approach falls under the former group. Briefly, in this approach, as plans for sub-queries are being considered, the optimizer checks if there are any views that overlaps with the sub-query, partially or fully, and can be used to generate an efficient plan. Our approach falls under this approach, where our optimizer effectively asks the `IndexStore` for matches of sub-query being enumerated in an A+ index and if presence of A+ Indexes, uses. As described in Chapter 2, Section 2.4, for our edge-bound A+ indexes, we perform a non-trivial query rewrite, which, to our best knowledge, cannot be done in existing RDBMS and GDBMSs. The transformational approaches perform the view selection as a logical transformation rule, as done for example by reference [16] in a prior version of Oracle DBMS, as well as several other work, such as references [15, 17]. We refer the reader to reference [20], for a more extensive overview of the work in this literature.

The question of finding overlaps between views and sub-queries is formally known as the query containment problem, which is the problem of determining if the outputs of one query $Q_1$ is contained in the output of $Q_2$ under all possible database instances. An extensive literature review of this problem is beyond the scope of this thesis but many aspect of this problem, such as containment of conjunctive queries [12], recursive queries [11], or queries with predicates that contain arithmetic comparisons [49] have been extensively studied in prior literature. Our current work limits A+ indexes to only contain conjunctive predicates, so our optimizer when selecting A+ indexes to use in query plans effectively performs query containment checks for conjunctive queries, as well as arithmetic containment. For example, our optimizer can select an A+ index with a predicate `v1.age` $>$ `500` to evaluate a query with predicate `v1.age` $>$ `1000`.

Finally, our queries in relational terms are select-project-join queries, where the joins are inner joins. Several prior work use materialized views with outer joins [29] and aggregations [19, 46], which are queries we have not studied in the context of graph databases.

# Chapter 7

# Conclusions and Future Work

In this thesis, we described a new indexing sub-system that consists primarily of A+ indexes, which are indexes that allow users to define both per-vertex and per-edge adjacency lists that can satisfy a wide range of predicates on the edges that are stored. In addition to A+ indexes, our indexing subsystem consists of secondary B+ tree and bit map indexes that index aggregate properties of the adjacency lists in A+ indexes. We implemented our indexing sub-system on top of the Graphflow GDBMS. We described the modifications we had to do to our optimizer to use A+ indexes during query evaluation and presented performance evaluations of our indexing sub-system.

Noticeably missing from this work is the maintenance of the indexes in our work in the presence of updates to the graph. The codebase on which our work is based currently does not support updates. Users ingest data from csv files, which are turned into an in-memory representation. Users can create and drop indexes on the base graph but cannot update the graph. One main direction of future work is how to efficiently support updates. We see two main research questions here. First, in the presence of multiple indexes, which are effectively materialized views, what kind of multi-query optimizations can be applied to maintain all of the indexes together. Second, if the indexes are sorted, what kind of physical implementation would allow their maintenance efficiently.

Another future work direction is to apply these techniques to non-native GDBMSs that directly use an underlying RDBMS. We believe it is very interesting and important to understand the modifications that are required to the underlying RDBMS to support A+ indexes, and the performance benefits these may bring to non-native GDBMSs.

# References


[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *TODS*, 42(4), 2017.

[3] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2–8. AAAI Press, 2015.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 349–360, New York, NY, USA, 2013. ACM.

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *PVLDB*, 11(6), 2018.

[6] Bibek Bhattarai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1447–1462, New York, NY, USA, 2019. ACM.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.

[8] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors, *ALENEX/ANALC*, pages 49–61. SIAM, 2004.

[9] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[10] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large sparql query logs. *Proc. VLDB Endow.*, 11(2):149–161, October 2017.

[11] Surajit Chaudhuri and Moshe Y. Vardi. On the Equivalence of Recursive and Nonrecursive Datalog Programs. In *PODS*, 1992.

[12] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *International Conference on Database Theory*, 1997],.

[13] Francisco Claude and Gonzalo Navarro. *Extended Compact Web Graph Representations*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[15] Daniela Florescu, Louiqa Raschid, and Patrick Valduriez. Answering Queries Using OQL View Expressions. In *In Workshop on Materialized Views, in cooperation with ACM SIGMOD*, 1996.

[16] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr, James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized Views in Oracle.

[17] Goldstein, Jonathan and Larson, Per-Åke. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *SIGMOD Record*, 30(2), May 2001.

[18] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), February 1994.

[19] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-Query Processing in Data Warehousing Environments. In *VLDB*, 1995.

[20] Alon Y. Halevy. Answering queries using views: A survey. 2001.

[21] Alon Y. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10(4), December 2001.

[22] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1429–1446, New York, NY, USA, 2019. ACM.

[23] Matthias Hauck, Marcus Paradies, Holger Fröning, Wolfgang Lehner, and Hannes Rauhe. Highspeed graph processing exploiting main-memory column stores. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 503–514, Cham, 2015. Springer International Publishing.

[24] Yannis E. Ioannidis. The history of histograms (abridged). In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 19–30, 2003.

[25] Janus Graph. https://janusgraph.org.

[26] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1695–1698, New York, NY, USA, 2017. ACM.

[27] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable Subgraph Enumeration in MapReduce. In *VLDB*, 2015.

[28] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable Distributed Subgraph Enumeration. In *VLDB*, 2016.

[29] Per-Åke Larson and Jingren Zhou. View Matching for Outer-join Views. In *International Conference on Very Large Data Bases*, 2005.

[30] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[31] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[32] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[33] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *CoRR*, abs/1903.02076, 2019.

[34] Neo4j. https://neo4j.com.

[35] Neo4j splitting relationship chains by label and direction. https://github.com/neo4j/neo4j/commit/366d30928d1b1590eba5daef92116ecc15aa36b1.

[36] Neo4j Property Graph Model. https://neo4j.com/developer/graph-database.

[37] Thomas Neumann and Gerhard Weikum. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, August 2008.

[38] openCypher. https://www.opencypher.org.

[39] Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html.

[40] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *VLDB*, 11(4), December 2017.

[41] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[42] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[43] Janus Graph. https://www.tigergraph.com/.

[44] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, August 2008.

[45] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 493–504, New York, NY, USA, 2009. ACM.

[46] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering Complex SQL Queries Using Automatic Summary Tables. In *SIGMOD*, 2000.

[47] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 192–203, New York, NY, USA, 2009. ACM.

[48] Shijie Zhang, Jiong Yang, and Wei Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *Proc. VLDB Endow.*, 3(1-2):1185–1194, September 2010.

[49] Xubo Zhang and Z. Meral Ozsoyoglu. On Efficient Reasoning with Implication Constraints. In *Deductive and Object-Oriented Databases*, 1993.

[50] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, September 2010.