TOWARDS INTERACTIVE METHODS FOR GATHERING INSIGHTS FROM DATA

by

Jianqiao Zhu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 01/18/2019

The dissertation is approved by the following members of the Final Oral Committee:
Jignesh Patel, Professor, Computer Sciences, UW-Madison
Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison
Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison
Mark Craven, Professor, Biostatistics and Medical Informatics, UW-Madison

To my wife, daughter and parents.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Jignesh Patel. This dissertation would not have been possible without all his guidance, encouragement and help. Prof. Patel has a great personality that has influenced me a lot. He is energetic, optimistic, insightful and is always with great willpower. It was really my fortune to work with him and learn from him. Meanwhile, sincerely thanks Prof. Somesh Jha for his advising on knowledge and skills related to the programming language and compiler aspects.

I am grateful to University of Wisconsin-Madison and particularly the Computer Sciences department. I had a good life and study experience in UW-Madison. Our department has a great faculty with excellent courses and research resources. I would also like to thank our graduate coordinator Angela Thorp for her many helps throughout my graduate study. Also many thanks to all CS staff who have helped me a lot on various administrative matters.

I would like to thank many of my friends for their accompanying throughout the graduate life. Thank my colleagues Harshad Deshmukh, Navneet Potti, Saket Saurabh, Rathijit San, Zhiwei Fan, Zuyu Zhang and Dylan Bacon for the great collaboration on various projects and research work.

I would like to acknowledge the National Science Foundation and the DARPA Transparent Computing program as they have funded most of my graduate research. One chapter of this disseration is based on the system we have built for the TC program, and it is really great experience to work with people from the TC program and development such a novel and intersting system.

During the years in UW-Madison, I met the most important person in my life – Yuqi. Yuqi is also a PhD student studying in UW-Madison and it is really a gift of life that I have ever met her. Now we have married and had our dearest daughter.

Thanks my parents for their understanding and support for the many years of my graduate study. Thanks all my family members who have cared me a lot during these years before I could finally graduate from schools. Finally, a special thank to my uncle for taking me to his office often to have access to a computer when I was young. There I developed my interests on computer systems and programs, and my domain expertise started converging towards a degree of Computer Science since then.

Abstract

Big Data technologies are now widely adopted by enterprises to aid knowledge discovery and decision making. To support modern data applications, contemporary relational database systems (RDBMSs) have been augmented and even redesigned to better support large-scale analytical query processing. Typically, the end user is an analyst who interacts with the system in a "compose query – execute query – interpret output – compose new query based on insights from the output" loop. The key aspects of this paradigm is to allow the analyst to discover insights from the underlying data as efficiently and effectively as possible.

In this dissertation, we focus on a high-performance relational data platform, Quickstep, and propose two components to reduce query execution time for complex join queries and facilitate interactive analysis on provenance graph data, respectively, thus contributing to improving the productivity of the end users in each analytical scenario.

First, we introduce a novel query execution strategy called LIP for robust query processing. LIP collapses the space of left-deep query plans for star schema warehouses down to almost a single point near the optimal plan. In addition to this robustness benefit, it also significantly speeds up query execution in the left-deep subplan space. Besides the immediate application of LIP, we believe our work opens a novel approach to the notion of "robustness", one that is focused on query execution strategies possibly tailored to corresponding query plan (sub-)spaces.

Second, we build on top of Quickstep a new system called QuickGrail that supports efficient and effective querying on large provenance graphs. The QuickGrail system comes together with an expressive domain-specific query language that allows a human analyst to evaluate complex filter / lineage / path / pattern matching queries to yield possibly very large subgraphs as intermediate results, and do set operations such as union, intersection, subtraction on the subgraphs. The intermediate results can be efficiently concretized and used as inputs for subsequent iterations of exploratory analyses. We explain in detail the underlying implementations that support all the QuickGrail operations with high performance, robustness and scalability.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Big Data technologies are now widely adopted by enterprises to aid knowledge discovery and decision making. A common data platform for this task is a relational data platform, and this setting is the focus of the proposed thesis.

To support modern data applications, contemporary relational database systems (RDBMSs) have been augmented and even redesigned to better support *large-scale analytical query processing*. Unlike traditional transactional workloads, analytical query processing features complex and long-running queries on large (by size/volume) datasets. Typically, the end user is an analyst who interacts with the system in a "*compose query – execute query – interpret output – compose new query based on insights from the output*" loop. The key aspects of this paradigm is to allow the analyst to discover insights from the underlying data as efficiently and effectively as possible.

To work with a database system in this analytical data processing scenario, the *productivity* of the data analyst can be greatly improved if the system can be enhanced with features to accelerate the interactive loop. There are two avenues for efficiency improvements. The first is *system efficiency* and the second is *human efficiency*. Methods for system efficiency improvement target query processing and query optimization mechanisms to improve the speed with which queries are evaluated. Methods for human efficiency improvement target the expressive power and code reusability of the query language surface, as well as the consumption of the output of queries. This thesis aims to target specific problems along both these efficiency dimensions.

The larger backdrop of this thesis is the Wisconsin Quickstep project. Quickstep is a RDBMS that targets modern hardware settings that consists of multi-core, multi-socket, and large main-memory configurations. Such configurations are increasingly common, and Quickstep aims to deliver high-performance out-of-the-box in such settings. At its core, Quickstep implements a collection of relational algebraic operators, using efficient algorithms for each operation. This "kernel" can be used to run a variety of applications. These applications include traditional SQL analytics (a.k.a. Data

Warehousing), but can also include other analytical application classes. These other classes are graph analytics, many propositional machine learning algorithms, and relational learning. These applications can be supported as they can be mapped to an underlying relational algebra (using methods such as [18, 50, 23, 26]).

To set the stage for the core part of this thesis, the building block is a SQL optimization and execution framework that can be used both to improve the efficiency with which traditional SQL queries are executed and to server as a computational platform where a broader category of applications can be built on top of. Chapter 2 describes the Quickstep query optimizer and execution engine that have been built as part of this thesis.

We propose two additional components in this thesis to reduce query execution time for complex join queries and facilitate interactive analysis on provenance graph data, respectively, thus contributing to improving the productivity of the end users in each analytical scenario. Specific details are as follows.

First, we propose a novel *query evaluation strategy* named *Lookahead Information Passing* (LIP) to accelerate *join processing*. Complex analytical queries typically involve many join operations to combine data from multiple tables, where the *order* in which tables are joined often dramatically affects performance. It is standard technique for the query optimizer to use dynamic programming to search through the plan space, based on a *cost model*, to find the best join ordering. However, optimizers may at times miss the optimal plan, with sometimes disastrous impact on performance. Nevertheless, even with the optimal plan, the processing of joins in complex queries still often take a dominant amount of time due to the great number of *hash-table lookups* and *intermediate result table materialization* (in memory). LIP is a technique that can both improve the *time efficiency* and *space footprint* of a query plan that contains many hash joins. As its name suggests, LIP looks ahead for each operator in the query plan and aggressively applies semijoin optimization to push the *selectivities* from future joins down, via *LIP filters* – typically implemented as Bloom filters, to be applied as early as possible. Moreover, LIP effectively collapses the optimization space of certain classes of query plans, thus reduces the *query optimization overhead* and increases the *robustness* to cardinality estimation errors. This aspect of the thesis is described in Chapter 3.

In addition to LIP, we further introduce two query optimization techniques in Chapter 4 under an unifying theme of eliminating redundant computation and materialization using "*drop early, drop fast*" approaches. The first technique pushes down certain disjunctive predicates more aggressively than is common in traditional query processing engines. The second technique transforms certain joins into cache-efficient semi-joins using *exact filters*. An exact filter can be pushed down as a kind of LIP filter where the original join or semi-join gets completely eliminated. Both techniques improve Quickstep's performance on data warehousing workloads.

Second, we build a system named *QuickGrail* on top of Quickstep to support interactive exploration on large *provenance graphs*. The basic types of queries to a provenance graph are *filter*, *lineage* and *path* queries, and (as far as we know) many existing systems cannot even efficiently handle these basic queries on large graphs. The QuickGrail system comes together with a domain-specific query language that allows a human analyst to filter the graph to yield (possibly very large) *subgraphs* as intermediate results, and do subgraph manipulations such as graph *union*, *intersection*, *subtraction*, as well as finding paths among subgraphs. The queries are guaranteed to be fulfilled in a timely manner so that the analyst can interactively and iteratively dig through a provenance graph using typical problem-solving techniques such as *trial-and-error* and *divide-and-conquer*. Furthermore, QuickGrail also supports a class of *pattern matching* queries for extracting complex connection patterns. Though it is known that the general graph pattern problem is NP-hard, we alleviate the problem by allowing false positive results when a query is not *acyclic*. The processing time of a pattern matching query is *linear* to (each of) the graph size, query size and certain configurable parameters, so that the overall query processing time is bounded and predictable.

Chapter 2

Quickstep Query Optimizer and Execution Engine

In this chapter, we first briefly describe the Quickstep data model and query language, then introduce its query optimizer. Finally, we describe in detail Quickstep's execution engine and explains how it utilizes the high degree of parallelism offered by modern processors.

2.1 Data Model and Query Language

Quickstep uses the standard relational data model, and SQL as the query language. Currently, the system supports the following basic types: `INTEGER` (32-bit signed), `BIGINT/LONG` (64-bit signed), `REAL/FLOAT` (IEEE 754 *binary32* format), `DOUBLE PRECISION` (IEEE 754 *binary64* format), fixed-point `DECIMAL`, fixed-length `CHAR` strings, variable-length `VARCHAR` strings, `DATETIME` / `TIMESTAMP` (with microsecond resolution), date-time `INTERVAL`, and year-month `INTERVAL`.

2.2 System Overview

The architecture of Quickstep is shown in Figure 2.1. The system has a SQL *parser* that converts the input query into a syntax tree, which is then transformed by an optimizer into a physical plan. The *optimizer* first converts the syntax tree into a logical plan, and uses a rules-based approach [21] to transform the logical plan into an optimal physical plan. The query optimizer is described in Section 2.3.

The *catalog manager* keeps track of the logical and physical schemas of the database. Currently, the catalogs keep simple statistics, including estimated table cardinalities, distinct values for each

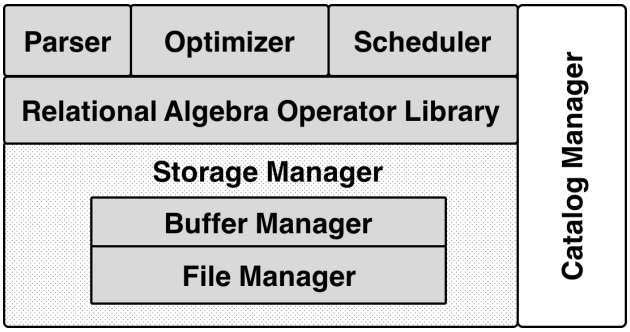


Figure 2.1: The Quickstep architecture.

attribute, as well as the minimum and maximum values for numerical attributes. More sophisticated statistics such as histograms are planned for addition in the future. The catalog manager can export the schema information for use in external systems such as a stand-alone optimizer like Orca [44] or Apache Calcite [13].

The execution plan, in the form of a directed acyclic graph (DAG) of relational operators, is created by the optimizer and sent to the *scheduler*. The current *relational operator library* contains the implementation of various relational operators including selection, projection, joins (equi-joins, semi-joins, anti-joins and outer-joins), aggregation, sorting, and top-k.

The *storage manager* organizes the data into large multi-MB blocks. Each block contains tuples from a single table, and is treated as a "mini-database." Different blocks, even within the same table, may have different physical organizations. The external view of each block is that of a bag of tuples, and query processing simply invokes set-oriented methods on each block. This design allows for the physical schema of each block to evolve independently. There are no global indices in Quickstep; instead indices are also self-contained within blocks. Different block formats are supported including column stores and row stores (see [14] for more details). The default layout is a column store.

2.3 Query Optimizer

The Quickstep query optimizer has a modular architecture. It is based on a sequence of rules that incrementally transform a parse tree into an efficient execution DAG of relational operators. This architecture is illustrated in Figure 2.2.

Similar to the approach outlined in [21], these rules operate on different intermediate representations: the *logical* and *physical query plan trees*. Each rule transforms the query plan tree by either annotating it or changing its structure. During query optimization, the rules are applied in a fixed sequence. The final execution DAG is passed to the scheduler for execution. The sequence of

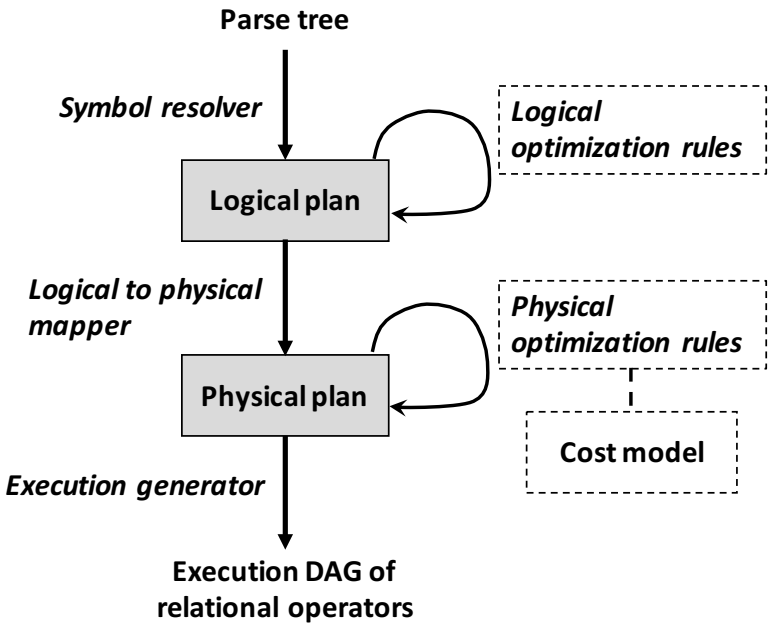


Figure 2.2: The Quickstep query optimizer architecture.

rules can be extended simply by adding a new transformation function in the appropriate position in the sequence.

All logical optimization rules are heuristic-based. Physical optimization rules can be either heuristic-based or cost-based. Typical heuristic-based optimizations such as *filter pushdown*, *projection collapsing*, and *projection pushdown* are implemented as one-pass tree traversals. Some complex optimizations such as *subquery unnesting*, *join order optimization*, and *LIP filter planning* require multiple traversals of the query plan tree and use auxiliary data structures within the rules.

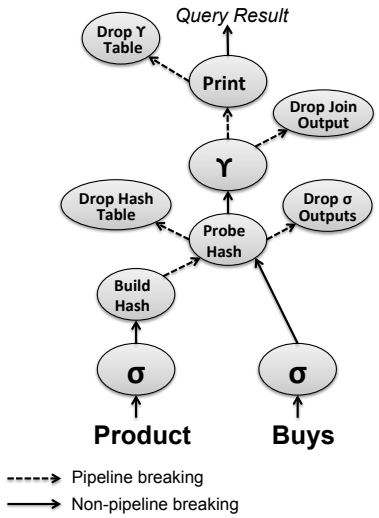
2.4 Query Execution

To fully utilize the high degree of parallelism offered by modern processors, Quickstep complements its block-based storage design with a work order-based scheduling model (cf. Section 2.4.2) to obtain high intra-query and intra-operator parallelism.

2.4.1 Threading Model

The Quickstep execution engine consists of a single *scheduler* thread and a pool of *workers*. The scheduler thread uses the query plan to generate and schedule work for the workers. When multiple queries are concurrently executing in the system, the scheduler is responsible for enforcing resource allocation policies across concurrent queries and controlling query admittance under high load.

```
SELECT SUM(sales)
FROM   Product P NATURAL JOIN Buys B
WHERE  B.buy_month = 'March'
AND    P.category = 'swim'
```



(a) (b)

Figure 2.3: Plan DAG for the sample query

The workers are responsible for executing the relational operation tasks that are scheduled. Each worker is a single thread that is pinned to a CPU core (possibly a virtual core), and there are as many workers as cores available to Quickstep. The workers are created when the Quickstep process starts, and are kept alive across query executions, minimizing query initialization costs. The workers are stateless; thus, the worker pool can *elastically* grow or shrink dynamically.

2.4.2 Work Order-based Scheduler

The Quickstep scheduler divides the work for the entire query into a series of *work orders*. In this subsection, we first describe the work order abstraction and provide a few example work order types. Next, we explain how the scheduler generates work orders for different relational operators in a query plan, including handling of pipelining and internal memory management during query execution.

The optimizer sends to the scheduler an execution query plan represented as a directed acyclic graph (DAG) in which each node is a relational operator. Figure 2.3 shows the DAG for the example query shown below. Note that the edges in the DAG are annotated with whether the producer operator is blocking or permits pipelining.

Work Order

A *work order* is a unit of intra-operator parallelism for a relational operator. Each relational operator in Quickstep describes its work in the form of a set of work orders, which contains references to its inputs and all its parameters. For example, a *selection work order* contains a reference to its input relation, a filtering predicate, and a projection list of attributes (or expressions) as well as a reference to a particular input block. A selection operator generates as many work orders as there are blocks in the input relation. Similarly, a *build hash work order* contains a reference to its input relation, the build key attribute, a hash table reference, and a reference to a single block of the input build relation to insert into the hash table.

Work Order Generation and Execution

The scheduler employs a simple DAG traversal algorithm to activate nodes in the DAG. An active node in the DAG can generate *schedulable* work orders, which can be fetched by the scheduler. In the example query, initially, only the Select operators (shown in Figure 2.3 using the symbol σ) are active. Operators such as the probe hash and the aggregation operations are initially inactive as their blocking dependencies have not finished execution. The scheduler begins executing this query by fetching work orders for the select operators. Later, other operators will become active as their dependencies are met, and the scheduler will fetch work orders from them.

The scheduler assigns these work orders to available workers, which then execute them. All output is written to temporary storage blocks. After executing a work order, the worker sends a completion message to the scheduler, which includes execution statistics that can be used to analyze the query execution behavior.

Implementation of Pipelining

In our example DAG (Figure 2.3), the edge from the Probe hash operator to the Aggregate operator allows for data pipelining. As described earlier, the output of each probe hash work order is written in some temporary blocks. Fully-filled output blocks of probe hash operators can be streamed to the aggregation operator (shown using the symbol γ in the figure). The aggregation operator can generate one work order for each streamed input block that it receives from the probe operator, thereby achieving pipelining.

The design of the Quickstep scheduler separates control flow from data flow. The control flow decisions are encapsulated in the work order scheduling policy. This policy can be tuned to achieve different objectives, such as aiming for high performance, staying with a certain level of concurrency/CPU resource consumption for a query, etc. In the current implementation, the scheduler

eagerly schedules work orders as soon as they are available.

Output Management

During query execution, intermediate results are written to temporary blocks. To minimize internal fragmentation and amortize block allocation overhead, workers reuse blocks belonging to the same output relation until they become full. To avoid memory pressure, these intermediate relations are dropped as soon as they have been completely consumed (see the Drop σ Outputs operator in the DAG). Hash tables are also freed similarly (see the Drop Hash Table operator). An interesting avenue for future work is to explore whether delaying these Drop operators can allow sub-query reuse across queries.

Chapter 3

Improving the Evaluation of Join Queries on Star Schema Databases

Query optimizers and query execution engines cooperate to deliver high performance on complex analytic queries. Typically, the optimizer searches through the plan space and send a selected plan to the execution engine. However, optimizers may at times miss the optimal plan, with sometimes disastrous impact on performance. Complementary to prior work on improving robustness in query optimization, in this chapter, we develop the notion of robustness of a query evaluation strategy with respect to a space of query plans. We also propose a novel query execution strategy called *Lookahead Information Passing (LIP)* that is robust with respect to the space of (fully pipeline-able) left-deep query plan trees for in-memory star schema data warehouses. LIP ensures that execution times for the best and the worst case plans are far closer than without LIP. In fact, any plan in that space is theoretically guaranteed to execute in near-optimal time. We empirically validate these claims using benchmark workloads as well as synthetic workloads that include skew and correlation. With LIP we make an initial foray into a novel way of thinking about robustness from the perspective of query evaluation, where we develop strategies (like LIP) that collapse plan sub-spaces in the overall global plan space. Such techniques can be used both to immunize against poor plan selection by an optimizer, as well as to increase the efficiency of plan space search in an optimizer (since entire sub-spaces collapse to have identical execution times).

3.1 Introduction

Relational database management systems (RDBMSs) have a unique internal organization where query execution can be viewed as a composition of basic relational algebraic (RA) operations. This underlying framework allows RDBMSs to examine equivalent RA compositions during the query

optimization phase, examining and picking a plan that leads to efficient query execution. This aspect of being able to manipulate RA expressions is crucial to the RDBMSs' ability to execute complex queries efficiently even on large databases.

Query optimization, however, is a complex task. Work on query optimization is nearly as old as research in RDBMSs. Decades of research in this area have yielded a plethora of techniques for plan enumeration, cardinality and cost estimation, and dynamic query optimization. Despite these remarkable advancements, it is well known [29, 41] that query optimizers still falter in some cases, producing query plans that have disastrously worse performance than optimal.

Rather than directly improving the capability of query optimizers, in this work, we take an approach that is complementary to most prior work in query optimization. The question we seek to address is: *Can we develop query execution techniques that dramatically reduce the impact of a poor choice of a query plan?* Thus, the big picture view of our approach is to focus on developing efficient *query evaluation techniques* that increase the *robustness* of query plans, by mitigating as much as possible issues related to bad plan selection within a subspace of plans. To scope the work even further, in this chapter we focus primarily on increasing the robustness of plans to errors in join order selection.

Lookahead Information Passing (LIP), the query evaluation strategy that we propose in this work, is targeted at the common scenario of star schema data warehouses. In such workloads, a natural space of "good" plans for a query optimizer is that of fully pipeline-able left-deep join trees. For instance, consider Query 4.3 in the Star Schema Benchmark, shown in Figure 3.2. Figures 3.2a and 3.2b show two of the 24 possible left-deep query plans for this query, resulting from all permutations of the 4 dimension tables in the query.

Figure 3.1 illustrates how the use of LIP increases robustness in query processing. The execution times of the naive evaluation strategy (without using our proposed techniques), marked using

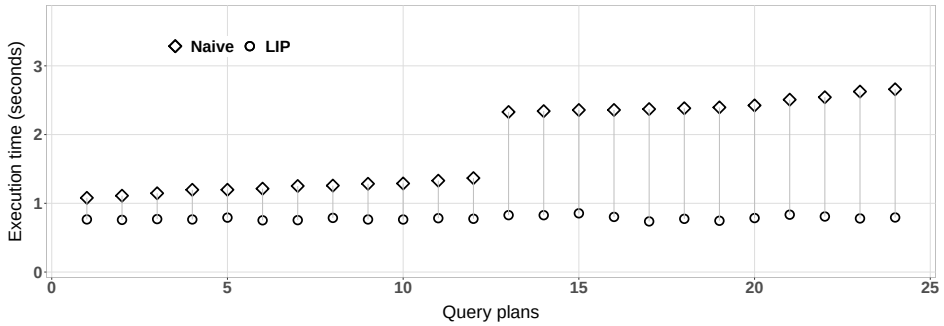


Figure 3.1: All 24 possible left-deep query plans for SSB Query 4.3 in increasing order of execution time.

```
SELECT d_year, s_city, p_brand1,
       SUM(lo_revenue - lo_supplycost) AS profit1
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_partkey = p_partkey
  AND lo_orderdate = d_datekey
  AND c_region = 'AMERICA'
  AND s_nation = 'UNITED␣STATES'
  AND (d_year = 1997 OR d_year = 1998)
  AND p_category = 'MFGR#14'
GROUP BY d_year, s_city, p_brand1
ORDER BY d_year, s_city, p_brand1;
```

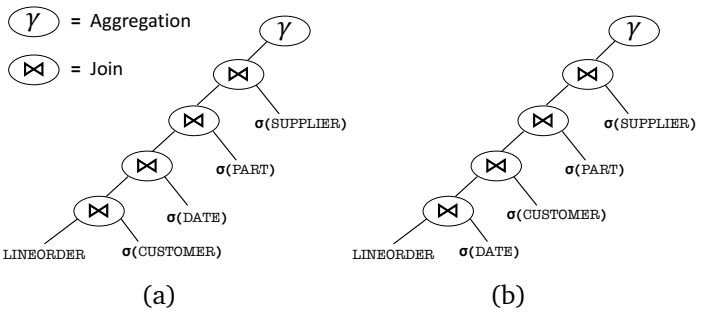


Figure 3.2: Query 4.3 from Star Schema Benchmark and two left-deep query plans for it.

diamonds, can be anywhere from 1.1s to 2.7s, depending on the join order selected. On the other hand, the execution times using LIP are all within 0.1s of each other (well within the variance of execution times for any particular query plan). Further, not only does LIP have negligible overhead, we actually see an improvement in performance for typical queries. For instance, in 8 of the 13 SSB queries, the query plan using the LIP technique had an execution time better than the best query plan without using LIP.

In essence, the LIP technique consists of two key components. First, we pass succinct filter data structures (such as Bloom filters) from the "outer" relation in all the joins to the inner one. We can thus approximately pre-filter the fact table (inner relation) before performing the join operation, greatly speeding it up. For the case of a hash join, for instance, this optimization reduces both the number of hash table probes required as well as the cost of materializing intermediate results in a vectorized query execution engine such as ours. Second, we use an adaptive reordering algorithm that dynamically converges to the optimal ordering in which the outer relation filters are applied.

Note that the filter structures can be built as part of the hash table build operator, and their application can be folded into the first hash table probe operator. Thus, a query optimizer does not need to alter the plan to make use of our technique. For this reason, LIP is an execution technique that is independent of the query optimization method.

The idea of passing a filter between the two sides of a join operation in the LIP technique bears resemblance to the well-known semi-join optimization. However, to the best of our knowledge, ours is the first work to aggressively use this optimization across multiple joins in a join tree. For this reason, the benefit of semi-join optimization for robust query processing has not been noted

in literature. It is also important to note that the adaptive reordering phase of the LIP technique is crucial for both reducing sensitivity to estimation errors as well as speeding up performance. In fact, without this reordering, there is only a marginal improvement in robustness, largely overweighed by the additional cost of building and applying these filters.

In this work, we introduce a novel approach to robust query processing: one that focuses on query execution techniques that are immune to poor choices made by the query optimizer. In this initial foray, we have limited the scope to star schema data warehouses and left-deep query plans. LIP is also applicable to left-deep join tree subplans within larger query plans, as we demonstrate (in Section 3.4.5) using an example query from the TPCH benchmark.

While we admittedly address a simple scenario in this initial work on robustness of query execution strategies, we are able to present elegant analytical results that we find at once insightful and intuitive. We believe that these insights will allow us to tackle more complex scenarios in future work. Finally, we hope that others in the community will join us in exploring this line of research into the important problem of robust query processing strategies.

We now summarize the contributions made in this chapter.

1. A formal definition for the robustness of a query evaluation strategy, along with a simple analytical model that allows us to derive closed-form results about performance and robustness in a plan space under different evaluation strategies.
2. A specific strategy, Lookahead Information Passing, that dramatically increases robustness to errors in join order selection, with little overhead (and often, improved performance), for the case of left-deep query plans in a star schema data warehouse.
3. Theoretical guarantees (based on the simple models) for the claims about robustness and near-optimality.
4. Some notes about implementation issues that had to be addressed in order to apply these techniques in a high-performance main memory analytics database system.
5. Empirical evaluation of LIP, including a new synthetic data and query generator that allows us to study the impact of skew and correlation on execution times.

3.2 Preliminaries

We begin this section by defining the star schema. We scope the analysis in this chapter to the plan space consisting of left-deep join trees for select-project-join (SPJ) queries in such a schema, operating in an in-memory setting. Then, we use a cost model to evaluate the performance of plans in this space, without using LIP. The results from this baseline analysis allow us to formally define the notion of robustness that we use in the rest of the chapter. Finally, we provide a brief

introduction to Bloom filters, the key data structure used in our implementation of LIP.

3.2.1 Star Schema and Left-deep Join Trees

Data warehouses used for decision support systems often follow the Kimball method [27] which results in a *star schema*. In this chapter, we focus on this important pattern.

A star schema consists of a *fact* table F, often containing information about events such as sales and shipments, as well as a set of N *dimension* tables $\{D_1, D_2, \ldots, D_N\}$, containing additional descriptive attributes such as details about customers or products. The dimension tables are typically orders of magnitude smaller than the fact table, and are related to it through *primary key - foreign key* constraints.

In developing the LIP technique and its theoretical guarantees, we limit our focus to the selection and primary key - foreign key join operations in SPJ queries, since this component often dominates the total query execution time. Such a query involving $n \leq N$ of these dimension tables is represented by the relational algebraic expression:

$$F \bowtie \sigma(D_1) \bowtie \sigma(D_2) \bowtie \ldots \bowtie \sigma(D_n) \tag{3.1}$$

All the notation used throughout this chapter is summarized in Table 3.3. Note that for notational convenience, we have dropped the selection predicate on F, though our results do not depend on this assumption. We also add that while our presentation in this chapter deals only with SPJ queries, our methods are applicable to SPJ subplans within larger queries as well, as we illustrate using an example from the TPCH benchmark in Section 3.4.5.

Given a query in this declarative form, it is up to an optimizer to determine the best execution plan to compute the results. Note that the fact table typically has a much larger cardinality than the dimension tables, and the only key constraints are between the fact and dimension tables. This observation implies that the optimal join order is highly likely to use the fact table as the "outer" relation in every join. For instance, in the case of a hash join, the fact table must be on the probe side.

We can visualize such a plan as a left-deep tree of joins. Such left-deep trees allow for full pipelining of the query results and are generally the optimal plan shape for queries of this type. We further scope the chapter to only examine query plans with this shape.

Example 1 *The Star Schema Benchmark (SSB) [39] is a widely used variation of the TPC-H benchmark, and is a star schema data warehouse. The benchmark database consists of a fact table* `LINEORDER` *and four dimension tables, with foreign key constraints between the* `LINEORDER` *table and each of the dimension tables. There are 13 queries in the benchmark, split into 4 groups, and each query is of the*

form select-project-join-aggregate. For instance, Figure 3.1 shows the Query 4.3, which involves joins between all the tables in the database. As noted above, there can be a number of left-deep plans for a specific query. Two such plans as shown in Figures 3.2a and 3.2b for Query 4.3.

Category	**Notation**	**Meaning**	**Remarks**
Star Schema	F	fact table	see expression in Equation 3.1
	$D_1, ..., D_N$	dimension tables	
	n	number of joins in the query plan	
Bloom Filter	r	bit array size: number of bits per inserted object	configured by query optimizer
	k	number of hash functions	
	ϵ	false positive rate	see Equation 3.12
Query Plans	$1, 2, ..., n$	join order $D_1, D_2..., D_n$	represents any arbitrary join order
	$1', 2', ..., n'$	optimal join order (its reverse is the worst join order)	see Section 3.2.2
	$P_{12...n}$	query plan in the naive evaluation strategy without LIP	
	P_b, P_w	best and worst naive query plans	
	$B_{12...n}$	query plan using LIP, but no adaptive reordering	see Section 3.3.2
	B_b, B_w	best and worst query plans with LIP, no adaptive reordering	
Cost Model	1	cost per tuple for predicate evaluation and hash table insertion or probe	see Section 3.2.2
	β	relative cost of Bloom filter insertion or probe	
	$\sigma_1, ..., \sigma_n$	selection predicate and join selectivities	
	$\sigma_{\text{min}}, \sigma_{\text{max}}$	minimum and maximum selectivities among σ_i	
	BuildCost(·)	cost of selection, and building hash table and bloom filter	
	HashTableProbeCost(.)	cost of probing hash tables in LIP strategies	
	T(·)	total cost of query plan	
	BloomProbeCost(·)	cost of probing Bloom filters	see Section 3.3.2

Table 3.3: Summary of notation used in Sections 3.2 and 3.3

While structurally similar, these plans can have widely varying execution times, depending on the selectivities of the predicates appearing in the query. For example, the predicate on `CUSTOMER` *table in SSB Query 4.3 has a selectivity of* 20%*, whereas that on* `DATE` *is* 28%*. Simply scheduling the join between* `LINEORDER` *and* `CUSTOMER` *before that with* `DATE` *(as in Figure 3.2a rather than Figure 3.2b) would result in about* 6% *fewer hash table probes in total.*

Each permutation of the dimension tables, called a *join order*, results in a distinct left-deep query plan. Thus there are exponentially many (in fact, $n!$) different query plans under consideration by the optimizer. Given a permutation $\pi : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., n\}$, we can use the notation $P_{\pi(1)\pi(2)...\pi(n)}$ to refer to the query plan corresponding to the join order $D_{\pi(1)}, D_{\pi(2)}, ..., D_{\pi(n)}$. However, without loss of generality, we will use the plan $P_{12...n}$ in our analysis.

A query optimizer must estimate the selectivities of the predicates in order to pick the best join order. In real world data, unlike in benchmark databases, it has been observed that there is a lot of skew in the data distribution within a column, and that column values are often correlated with each other. Accurate selectivity estimation under such a scenario remains a challenge despite years of research.

3.2.2 Modeling Performance Without LIP

The primary focus of this chapter is the effect of join order selection on the performance of star schema key join queries. To obtain a baseline, we first model the performance of query plans that do not use the LIP technique. We refer to this as the *naive evaluation strategy*.

For the analysis below, we focus on the star schema SPJ query shown in Equation 3.1. Without loss of generality, we pick the plan $P_{12...n}$ as an arbitrary left-deep hash join tree for this query. In the theoretical results below, we have assumed that all data in the tables is independent and uniformly distributed. Under this assumption, the selectivity of the join between F and D_i is equal to the selectivity of the predicates on D_i. Our experiments in Section 3.4.3 show that this assumption does not affect the efficacy of LIP in handling real-world data with skew and correlation.

Cost Model

Let us use a simple model to derive the costs for the different operations in the execution of the query plan $P_{12...n}$. In this model, we assign unit cost each per tuple to the operations of checking whether it satisfies a selection predicate, inserting it into a hash table, and probing whether it is contained in the hash table.

As an example, consider a query with only one dimension table D containing 100 tuples. If 10 tuples pass the selection predicate and are inserted into a hash table, the total cost of these operations in our model is 100 (for selection) $+10$ (for insertion) $= 110$ units. If this hash table is probed using a fact table containing 1000 tuples, the probe cost is 1000 units and the total cost for the naive evaluation strategy is 1110 units.

For brevity and clarity of exposition, we have refrained from generalizing this per-tuple cost model to incorporate the number of attributes in the tuple or the size of the hash table. We also

omit the cost of materializing intermediate join results, instead assuming that it is folded into the unit probe cost. These assumptions are most justified for columnar query execution engines that apply late materialization.

Hash Table Build Phase

In the naive evaluation strategy, we first apply the selection predicates on the dimension tables and then build hash tables on the results. Denoting the selectivity of the predicate on D_i by σ_i, the build cost is:

$$\text{BuildCost}(P_{12...n}) = \sum_{i=1}^{n}(1+\sigma_i)|D_i| \tag{3.2}$$

Note that this cost is the same for all join orders.

Hash Table Probe Phase

Next, we probe each of the hash tables in the join order specified by the plan. Let us assume the selectivity of the join predicate between F and D_i is also σ_i. Then, the cost of probing the hash table on D_1 using F is $|F|$, and the result has a cardinality $\sigma_1|F|$. The subsequent probes have costs $\sigma_1|F|$, $\sigma_1\sigma_2|F|$ and so on. Letting $\sigma_0 = 1$, the hash table probe cost in our model can be written as:

$$\text{HashTableProbeCost}(P_{12...n}) = \sum_{i=1}^{n}\sigma_0\sigma_1...\sigma_{i-1}|F| \tag{3.3}$$

Bounds on Cost of Any Plan

The total cost $T(P_{12...n})$ of this plan is the sum of the build and the probe cost terms.

$$\begin{aligned} T(P_{12...n}) &= \text{BuildCost}(P_{12...n}) + \text{HashTableProbeCost}(P_{12...n}) \\ &= \sum_{i=1}^{n}(1+\sigma_i)|D_i| + \sum_{i=1}^{n}\sigma_0\sigma_1...\sigma_{i-1}|F| \end{aligned} \tag{3.4}$$

It is intuitively clear (and can be proved by induction on n) that the HashTableProbeCost above is minimized when the probes are done in ascending order of selectivities, and maximized when using descending order. These sorted join orders therefore also have the minimal (or maximal) total costs, since the BuildCost is independent of the join order. Let us denote the best (minimal cost) join order by the sequence $1', 2', ..., n'$ and the corresponding plan $P_{1'2'...n'}$ by P_b. Then, the worst (maximal cost) join order is the reverse sequence $n', ..., 2', 1'$, and we denote $P_{n'...2'1'}$ as P_w. Let us also use $\sigma_{\text{min}} = \sigma_{1'}$ and $\sigma_{\text{max}} = \sigma_{n'}$ to denote the minimum and maximum selectivities.

$$\sigma_{\text{min}} = \sigma_{1'} \leq \sigma_{2'} \leq ... \leq \sigma_{n'} = \sigma_{\text{max}} \tag{3.5}$$

Replacing each of the selectivity factors σ_i for $i > 0$ in the summation in Equation 3.3 above with σ_{max} gives us an upper bound for the cost of any plan.

$$\begin{aligned}\text{HashTableProbeCost}(P_{12\ldots n}) &\leq \sum_{j=0}^{n-1} \sigma_{\text{max}}^j |F| \\ &= \frac{1-\sigma_{\text{max}}^n}{1-\sigma_{\text{max}}}|F| \end{aligned} \tag{3.6}$$

Similarly, we can get a lower bound for the cost any plan by substituting σ_{min}.

$$\begin{aligned}\text{HashTableProbeCost}(P_{12\ldots n}) &\geq \sum_{j=0}^{n-1} \sigma_{\text{min}}^j |F| \\ &= \frac{1-\sigma_{\text{min}}^n}{1-\sigma_{\text{min}}}|F| \end{aligned} \tag{3.7}$$

Note that the bounds in Equations 3.6 and 3.7 depend on the lowest and highest selectivities, and they apply to *any* plan (i.e., any join order).

Robustness: Cost Difference Between Plans

Recall that the BuildCost is the same for all join orders. Thus, the difference in total cost between any two join orders is just the difference between their HashTableProbeCosts. This cost difference between the best and the worst plans is:

$$T(P_w) - T(P_b) = \sum_{i=1}^{n-1} \left(\sigma_{n'}\ldots\sigma_{(n-i+1)'} - \sigma_{1'}\ldots\sigma_{i'}\right)|F| \tag{3.8}$$

In Section 3.2.3 below, we will formally define a notion of robustness that depends on this cost difference. Note that this equation has $n-1$ difference terms in the summation. We will now find a lower bound for this expression in terms of $\sigma_{\text{max}} - \sigma_{\text{min}}$. The i^{th} difference term above is the difference between the terms $\sigma_{n'}\ldots\sigma_{(n-i+1)'}$ and $\sigma_{1'}\ldots\sigma_{i'}$, each a product of i factors. The larger term $\sigma_{n'}\ldots\sigma_{(n-i+1)'}$ is the product of the largest i factors, i.e., the last i selectivities in the sequence $1', 2', \ldots, n'$. The smaller term $\sigma_{1'}\ldots\sigma_{i'}$ is the product of the smallest i factors, i.e., the first i selectivities in the sequence $1', 2', \ldots, n'$.

To obtain an upper bound, we begin by replacing all the $i-1$ factors apart from the first factor $\sigma_{n'}$ in the larger term with the corresponding $i-1$ smaller factors from the other term. Then, we replace these $i-1$ factors by the smallest selectivity $\sigma_{1'}$.

$$\begin{aligned}\sigma_{n'}\sigma_{(n-1)'}\ldots\sigma_{(n-i+1)'} - \sigma_{1'}\sigma_{2'}\ldots\sigma_{i'} &\geq (\sigma_{n'} - \sigma_{1'})\sigma_{2'}\ldots\sigma_{i'} \\ &\geq (\sigma_{n'} - \sigma_{1'})\sigma_{1'}^{i-1}\end{aligned}$$

Plugging the above bound into Equation 3.8, we get the following lower bound:

$$T(P_w) - T(P_b) \geq \sum_{i=1}^{n-1} \sigma_{1'}^{i-1}(\sigma_{n'} - \sigma_{1'})|F| \\ = \frac{1 - \sigma_{\text{min}}^{n-1}}{1 - \sigma_{\text{min}}}(\sigma_{\text{max}} - \sigma_{\text{min}})|F| \tag{3.9}$$

3.2.3 Robustness

We will see that, in general, the difference in execution cost between the worst and the best plans grows linearly with the size of the fact table. Further, it also grows with the spread of selectivities of the predicates used in the query, i.e., $\sigma_{\text{max}} - \sigma_{\text{min}}$, because intuitively, errors in join order selection are more disastrous to performance when the selectivities are more different from each other. In fact, if we assume that the selectivities (and hence cardinalities) are estimated with some error tolerance δ, i.e., if each estimate $\hat{\sigma_i}$ is within δ of actual σ_i, then we expect the worst plan to be worse than the best in proportion to δ.

Our cost model does not incorporate the second order effects such as the size of the hash table on probe cost and the impact of number of output attributes on materialization cost. In the special case when the predicate selectivities are all similar, $\sigma_{\text{max}} - \sigma_{\text{min}}$ is negligibly small or even 0, and these second order effects dominate the cost difference between plans. Addressing this shortcoming of our model is left out of scope of this chapter. Consequently, we assume that $\sigma_{\text{max}} - \sigma_{\text{min}}$ is non-zero in the definitions below.

To make this notion more concrete, let us formally define robustness such that we can use it to compare different query evaluation strategies.

Definition 1 Θ*-Robustness: An evaluation strategy* $\mathcal{E}$ *is said to be* Θ*-robust with respect to a plan space* $\mathcal{P}$ *if the maximum deviation in performance of any plan in* $\mathcal{P}$ *(including the worst plan* $\mathcal{E}_w$*) from the best one* $\mathcal{E}_b$*, normalized by the fact table cardinality and spread of selectivities in a query, is at most* Θ*.*

$$\frac{T(\mathcal{E}_w) - T(\mathcal{E}_b)}{(\sigma_{max} - \sigma_{min})|F|} \leq \Theta, \qquad \sigma_{max} \neq \sigma_{min} \tag{3.10}$$

Definition 2 θ*-Fragility: An evaluation strategy* $\mathcal{E}$ *is said to be* θ*-fragile with respect to a plan space* $\mathcal{P}$ *if the maximum deviation in performance of any plan in* $\mathcal{P}$ *(including the worst plan* $\mathcal{E}_w$*) from the best one* $\mathcal{E}_b$*, normalized by the fact table cardinality and spread of selectivities in a query, is at least* θ*.*

$$\theta \leq \frac{T(\mathcal{E}_w) - T(\mathcal{E}_b)}{(\sigma_{max} - \sigma_{min})|F|}, \qquad \sigma_{max} \neq \sigma_{min} \tag{3.11}$$

By Equation 3.9 above, the naive (non-LIP) evaluation strategy is θ-fragile with respect to the space of left-deep hash join trees, for $\theta = \frac{1-\sigma_{\text{min}}^{n-1}}{1-\sigma_{\text{min}}}$.

For robustness, we want to pick an evaluation strategy which guarantees that mistakes made by the query optimizer are not too expensive. But a θ-fragile strategy such as the naive strategy necessarily has a large spread in performance between the best and worst plans, particularly when θ is large. Thus, such a strategy adds fragility to plan selection. On the other hand, a Θ-robust strategy guarantees that even the worst plan in the plan space is not much more expensive than the optimal, particularly when Θ is small. In Section 3.3, we will develop such a Θ-robust strategy that makes plan selection more robust.

3.2.4 Bloom Filter

A Bloom filter [10] is a probabilistic data structure that succinctly represents a set. It is used to maintain approximate information about membership in the set. When probed to test for membership of an object in the set, if the Bloom filter returns `false`, then the object is guaranteed not to be a members of the set. It is also guaranteed to return `true` for all members of the set. But the filter may also wrongly return `true` for objects that are not members. The probability of such *false positives*, denoted ϵ, can be fixed by appropriately configuring the filter.

A Bloom filter consists of a configurable number of bits in a bit array, as well as a configurable number k of hash functions. These k hash functions can be thought of as mapping each object into k bit positions in the bit array. To insert an object into the filter, we set the k bits at the positions indicated by the k hash functions. To test whether an object is in the filter, we check whether *all* of the corresponding k bit positions are set. If any of these k bit positions is unset, then we can be sure that the object is not in the set. But it is possible that all these k bits are set even though the object had never been inserted, resulting in a *false positive*.

Optimal Configuration: If the Bloom filter bit array size is configured to be r bits per object inserted into the filter, then the rate of such false positives is [34]:

$$\epsilon \simeq \left(1 - e^{-k/r}\right)^k \tag{3.12}$$

Based on Equation 3.12, one can derive the theoretically optimal configuration of the Bloom filter for a given target false positive rate. For instance, to get a false positive rate of 1%, we can use a Bloom filter with about 9.6 bits per object (regardless of the size of the object) and 6 hash functions.

3.3 Lookahead Information Passing

In this section, we first introduce *Lookahead Information Passing* (LIP), and then show the benefits of the LIP evaluation strategy for robustness as well as performance.

The key insight behind the LIP strategy is that in the space of left-deep join trees for star schema queries, a suboptimal plan schedules less selective joins before selective ones. Such a plan incurs additional cost relative to the optimal one due to extra hash table probes for tuples that are filtered out in the later joins.

Thus, we can mitigate this cost by forwarding information about later join predicates to earlier ones in the plan. Such a *lookahead filter* can be forwarded from the build tables involved in downstream joins to the probe table, where they can be applied prior to performing the hash table probe. The resulting hash table probes now involve far fewer tuples.

The LIP strategy is summarized in the algorithm below.

1. For each dimension table in the join tree, we build both a hash table as well as a succinct filter data structure like a Bloom filter on the selection result.
2. We then simultaneously probe all these filters using the selection result of the fact table, maintaining hit/miss statistics.
3. We adaptively reorder the filters during the probe, using the estimated selectivity. For a good choice of filter configuration, the result of this multiway filter probe is roughly equal to the final output result, albeit possibly with a few false positives. We describe this algorithm in more detail in Section 3.3.1.
4. Subsequently, we probe the hash tables and eliminate the false positives as well as collect build-side attributes required for further processing.

Thus, using succinct filter data structures (such as a Bloom filter), we can greatly reduce the hash table probe cost (which is the dominating cost term) in such multi-join queries. In fact, as we show in Section 3.3.3, there are even fewer hash table probes than in the optimal plan using naive evaluation. However, we bear the additional cost of building and probing the LIP filter itself. On balance, we still see a speedup since the LIP data structures (Bloom filters in this context) are more space-efficient than hash tables, and are more likely to fit in the processor caches, minimizing probe costs.

The small size of such filters also allows us to dynamically reorder their probes based on their observed selectivities. This adaptive reordering ensures that the number of probes to the filters is close to the number of hash table probes required in the optimal join order, regardless of the join order picked by the optimizer. In fact, nearly all join orders exhibit roughly the same execution time, and that time is roughly equal to the optimal execution time (or better).

We present an analytical cost model in Section 3.3.2 to support the claims made above. In Section 3.4, we corroborate these findings using empirical results as well.

For convenience, the LIP technique is described here using Bloom filters and hash joins, though neither choice is essential for the technique. Further, we limit our focus to the in-memory setting, where there is enough memory to hold the hash tables and all the selected tuples for each dimension table in the query in memory. Such settings are not uncommon in high-performance analytic environments. (We note that the overall framework can be expanded to cover the case when intermediate data has to be spilled to disk.The LIP technique is likely to be even more beneficial in this scenario, given that it dramatically reduces the size of intermediate relations. This is a direction for future work.)

3.3.1 Adaptive Reordering of Lookahead Filters

Motivation

In the LIP strategy, the lookahead filters from the dimension tables are probed by the fact table tuples prior t1o probing the join hash table, thereby greatly reducing the number of hash table probes. However, this gain comes at the cost of additional probes into the lookahead filters. Even though these probes are less expensive than hash table probes, they can still become a significant component of the overall execution time if their ordering is poorly chosen. For instance, if we always apply the lookahead filters in the same order as the join order, then a plan with a bad join order will cause a large number of tuples to be used for probing the low-selectivity filters, only to have them be dropped in the following high-selectivity filter probes. Ultimately, such a scenario leads to low robustness.

We now summarize the algorithm we use to mitigate this issue, followed by some implementation notes. Then, we use an analytical model to prove that the algorithm has fast convergence. Experimental results supporting the need for adaptiveness are presented in Section 3.4.4.

Algorithm Summary

Algorithm 1 shows how we dynamically adapt the lookahead filter probe order to mitigate the additional cost due to a bad fixed ordering. We maintain hit/miss statistics for all probes into each of the lookahead filters. Periodically, these statistics are used to estimate the observed selectivity of the underlying filters, called the *miss rate* here. Sorting the lookahead filters by their miss rates ensures that subsequent probes occur on the most selective filter first, then the next most selective filter, and so on. Since the convergence is usually very fast, the number of lookahead filter probes performed is therefore roughly the same as the number of hash table probes in the optimal (minimal

cost) join order, regardless of the join order selected by the query optimizer. Thus, this adaptive reordering is a crucial contributor to the robustness and the near-optimality properties of LIP.

Algorithm 1: Filtering with adaptive reordering

Input: *filters* – an array of m lookahead filters
tuples – an array of n tuples
Output: indices of tuples that pass filtering

results $\leftarrow \emptyset$
foreach f in *filters* **do**
  count[f] $\leftarrow$ 0
  miss[f] $\leftarrow$ 0
batch_size $\leftarrow$ 64
$n \leftarrow$ |*tuples*|
loc $\leftarrow$ 0
while loc $< n$ **do**
  probe_batch $\leftarrow$ an array of tuple indices from *loc* to *min*(*loc* + *batch_size*, n) $- 1$
  foreach f in *filters* (in order) **do**
    result_batch $\leftarrow \emptyset$
    foreach i in *probe_batch* **do**
      if f contains *tuples*[i] **then**
        result_batch $\leftarrow$ *result_batch* $\cup \{i\}$
    count[f] $\leftarrow$ *count*[f] + |*probe_batch*|
    miss[f] $\leftarrow$ *miss*[f] + |*probe_batch*| - |*result_batch*|
    probe_batch $\leftarrow$ *result_batch*
  Sort *filters* in ascending order of $\frac{miss[f]}{count[f]}$
  results $\leftarrow$ *results* $\cup$ *probe_batch*
  loc $\leftarrow$ *loc* + *batch_size*
  batch_size $\leftarrow$ *batch_size* $\times$ 2
return results

Critical Implementation Aspects

We have implemented the LIP technique in the Quickstep RDBMS, whose storage subsystem horizontally partitions each table into small blocks of a few megabytes each. The algorithm above is run for each such block in the fact table. We begin the probe by creating a small batch $probe\_batch$ of a few hundred tuples. We then probe the first lookahead filter f using the batch of tuples, keeping

statistics about the number of probes, and hits/misses in an auxiliary data structure. The tuples whose keys are found to be hits in f (including false positives) are written into a $result\_batch$, which is then used to probe the next filter in bulk. After all the filters have been probed using the first batch, we sort the filters in ascending order of their miss rates, computed from the hit/miss statistics in the auxiliary data structures. This new ordering is used for the next cycle of batched probes.

The batching of tuples in the algorithm is necessary for efficiency because the aggregate size of all lookahead (Bloom) filters is often larger than the processor cache size. The probes are most cache-efficient when we allow one filter at a time to warm up the cache and become cache-resident by consecutively probing it with tuples in a batch. Note another implementation detail: to avoid the cost of copying tuples between $probe\_batch$ and $result\_batch$, we only use a single $batch$ data structure containing tuple references. Our execution engine performs the lookahead filter probes as part of the hash join probe operator for the bottom-most join in the query plan. After all the filters have been probed using a batch, the resulting tuples are used to probe the bottom-most hash table. The remaining hash tables are only probed after an entire output block is produced.

While large batch sizes benefit from cache residence of the Bloom filters, they slow down the convergence rate of the adaptive algorithm, since reordering is only done between batches. To balance the two effects, we adapt the batch sizes by iteratively doubling it at the end of every cycle, along with the adaptive reordering. Both the batch size and the lookahead filter order are reset after completing all the probes for a given storage block of the fact table.

Convergence Rates

To examine the convergence procedure in our model, we assume that join predicates are independent and that the tuples in a block are randomly distributed. The independence assumption ensures that the observed miss rate of a given filter does not depend on which filters were probed prior to it. The random distribution assumption ensures that, within a block, observed miss rates for the tuples at the beginning of a block are roughly the same as those for tuples anywhere else in the block. Note that this assumption does not require the tuples in the entire table to be randomly distributed: for instance, the adaptive algorithm can gracefully deal with partitioned tables or biases in the order of insertion of tuples into the fact table.

Under the above assumptions, according to the law of large numbers, the observed miss rate for the i^{th} filter converges to its selectivity, say γ_i. Note that in the case of a Bloom filter, this γ_i includes both the selectivity of the predicate on the corresponding dimension table, as well as a (configurable) false positive rate. Consider the probability that, after probing using N_i tuples, the observed miss rate $\hat{\gamma}_i$ is off from γ_i by more than a factor δ. Modeling the observed miss rate as a

Binomial random variable with mean γ_i, using Chebyshev's inequality we can derive that:

$$\Pr\left(1-\delta < \frac{\hat{\gamma_i}}{\gamma_i} < 1+\delta\right) \geq \frac{1-\gamma_i}{N_i\gamma_i\delta^2}$$

We see that the observed miss rate for a filter converges to its true selectivity at a rate proportional of the number of tuples used to probe the filter.

Note that this number of probes for the i^{th} filter within a batch depends on the selectivities of the prior filters, in that highly selective prior filters $1, 2, ..., (i-1)$ may result in too few tuples being used to probe the i^{th} filter. We have not had found this to be an issue in practice. Instead, for typical selectivities in the range 5% to 25%, we have found that the ordering of the filters converges to the optimal in just 3-4 adapter cycles. For instance, if the true selectivity $\gamma = 0.10$, then after examining $3,800$ tuples, the estimation error δ is less than 5% with 95% probability.

3.3.2 Robustness Through LIP

Next, we derive analytical results for the performance and robustness of the LIP evaluation strategy, similar to the results derived in Section 3.2.2 for the naive evaluation strategy.

Corresponding to the plan $P_{12...n}$ in the naive evaluation strategy, consider the equivalent plan $B_{12...n}$ that uses LIP to whittle down the fact table before probing the hash tables in the order defined by the subscript sequence.

Cost Model

Let us model the per-tuple cost of a Bloom filter insertion or probe by a factor β relative to the unit cost for per-tuple hash table operations.

As an example, consider a query with only one dimension table D containing 100 tuples. Suppose 10 tuples pass the selection predicate and get inserted into the hash table and Bloom filter. The total cost of these operations in our model is 100 (for selection) + 10 (for hash table insertion) + 10 β (for Bloom filter insertion) = $110 + 10\beta$ units. If the Bloom filter is probed using a fact table containing 1000 tuples, the probe cost is 1000β units. Assuming a false positive rate of 10% for the filter, the probe results in 200 selected tuples, rather than the ideal 100 (extra 10% of 1000 tuples). These 200 tuples are then used to probe the hash table at a cost of 200 units. Thus, the total cost of this plan using the LIP technique is $310 + 1010\beta$ units.

Hash Table and Bloom Filter Build Phase

In the LIP evaluation strategy, we first apply the selection predicates on the dimension tables and build both a hash table and a Bloom filter on each result. The total cost for these operations is

independent of the selected join order and is given by:

$$\text{BuildCost}(B_{12...n}) = \sum_{i=1}^{n}(1 + \sigma_i + \beta\sigma_i)|D_i| \tag{3.13}$$

Implementation note: In a multithreaded execution environment, each Bloom filter must be constructed in parallel by multiple threads. It is clear that simply making every writer thread acquire an exclusive mutex lock before writing to a Bloom filter causes a huge drop in scalability of query processing. In fact, this Bloom filter build phase can easily become the bottleneck for the entire query. Instead, we use a parallel Bloom filter construction algorithm that uses the fact that insertions into the Bloom filter are commutative and associative. Each thread, while scanning the input dimension table, constructs its own thread-local copy of the Bloom filter. All these local filters have the same configuration, set by the query optimizer. Finally, all the thread-local filters for a particular table are unioned together using the bitwise-OR operation on the bit array.

Bloom Filter Probe Phase

Recall from Section 3.2.4 that as a probabilistic data structure, the Bloom filter has a certain false positive rate, say ϵ, that we can appropriately configure. Consider the Bloom filter built on the selection result of the dimension table D_i. If we probe this filter using some N tuples from F, we would expect to obtain hits for not only the $\sigma_i N$ tuples truly passing the predicate, but also up to ϵN false positives. Of course, only at most N tuples can be hits, notwithstanding the false positive rate. Thus the selectivity of this Bloom filter is $\max(1, \sigma_i + \epsilon)$. For simplicity, we will assume that $\sigma_i + \epsilon < 1$ hereafter.

The next step in the strategy is to probe all the Bloom filters, along with adaptive reordering of the filters. As we have shown in Section 3.3.1, the adaptive reordering algorithm converges quickly to the optimal ordering of Bloom filters. This ordering is the same as the increasing order of selectivities, which is also the optimal ordering we have seen before for hash table probes. As before, we denote this ordering by the sequence $1', 2', ..., n'$. We ignore the negligibly small overhead of the probes and adaptive algorithm until convergence. Thus the BloomProbeCost is:

$$\text{BloomProbeCost}(B_{12...n}) = \left[1 + \sum_{i=1}^{n-1}(\sigma_{1'} + \epsilon)...(\sigma_{i'} + \epsilon)\right]\beta|F| \tag{3.14}$$

Note that this cost has the optimal selectivity order $1', 2', ..., n'$ on the right hand side, but is independent of the selected join order $1, 2, ..., n$. We note in passing that instead of probing the Bloom filters adaptively, if the join order were to be used for probing, then this cost term would no longer be independent of the join order. In fact, in that case, this cost term would actually dominate the overall query execution cost, greatly impacting robustness.

Hash Table Probe Phase

Regardless of the order of the Bloom filter probes above, the probes result in a final relation of size $(\sigma_1 + \epsilon)...(\sigma_n + \epsilon)|F|$, which we use to probe the n hash tables built earlier. After probing the first hash table at a cost of $(\sigma_1 + \epsilon)(\sigma_2 + \epsilon)...(\sigma_n + \epsilon)|F|$ units, we eliminate the false positives obtained from the corresponding first Bloom filter, so that the resulting cardinality is $\sigma_1(\sigma_2 + \epsilon)...(\sigma_n + \epsilon)|F|$. Continuing this line of reasoning (and setting $\sigma_0 = 1$ for convenience), we get:

$$\text{HashTableProbeCost}(B_{12...n}) = \sum_{i=1}^{n} \sigma_0\sigma_1...\sigma_{i-1}(\sigma_i + \epsilon)...(\sigma_n + \epsilon)|F| \tag{3.15}$$

If we ignore the terms that are $O(\epsilon^2)$ or higher powers of ϵ, we can simplify the above equation to:

$$\text{HashTableProbeCost}(B_{12...n}) \simeq \sigma_1\sigma_2...\sigma_n|F| \sum_{i=1}^{n} 1 + \epsilon\left(\frac{1}{\sigma_i} + \frac{1}{\sigma_{i+1}} + ... + \frac{1}{\sigma_n}\right) \tag{3.16}$$

This HashTableProbeCost is minimized when the probes are done in ascending order of selectivity $1', 2', ...n'$ and maximized when using the descending (reverse) order. Thus, we see that replacing each of the terms in the inner summation of the above equation with $\sigma_{n'} = \sigma_{\text{max}}$ gives us a lower bound for the HashTableProbeCost of *any* plan with LIP.

$$\begin{aligned}
&\text{HashTableProbeCost}(B_{12...n}) \\
&\geq \sigma_1\sigma_2...\sigma_n|F| \sum_{i=1}^{n} 1 + \epsilon\left(\frac{1}{\sigma_{n'}} + \frac{1}{\sigma_{n'}} + ... + \frac{1}{\sigma_{n'}}\right) \\
&= \sigma_1\sigma_2...\sigma_n|F| \sum_{i=1}^{n} 1 + \epsilon\frac{n-i+1}{\sigma_{n'}} \\
&= \sigma_1\sigma_2...\sigma_n|F| \left[n + \frac{\epsilon}{\sigma_{\text{max}}}\frac{n(n+1)}{2}\right]
\end{aligned} \tag{3.17}$$

Substitution with $\sigma_{1'} = \sigma_{\text{min}}$ in the inner terms gives us the following upper bound for the cost of *any* plan in this strategy.

$$\text{HashTableProbeCost}(B_{12...n}) \leq \sigma_1\sigma_2...\sigma_n|F| \left[n + \frac{\epsilon}{\sigma_{\text{min}}}\frac{n(n+1)}{2}\right] \tag{3.18}$$

Robustness: Cost Difference Between Plans

The total cost $T(B_{12...n})$ for any plan is the sum of the three cost terms above. Since the first two terms are both independent of the join order, the difference between the total costs for any two plans is only due to the difference in the HashTableProbeCost terms. We can use the above upper

bound for the best query plan B_b and the lower bound for the worst query plan B_w to bound the difference between the HashTableProbeCosts of any two plans in this strategy.

$$\mathrm{T}(B_w) - \mathrm{T}(B_b) \leq \frac{1}{2}\sigma_1\sigma_2...\sigma_n\epsilon n(n+1)\left[\frac{1}{\sigma_{\text{min}}} - \frac{1}{\sigma_{\text{max}}}\right]|F| \tag{3.19}$$

Key Result: From Equation 3.19, it is clear that LIP with adaptive reordering is a Θ-robust evaluation strategy, for

$$\Theta = \frac{1}{2}\frac{\sigma_1\sigma_2...\sigma_n}{\sigma_{\text{min}}\sigma_{\text{max}}}\epsilon n(n+1) \tag{3.20}$$

3.3.3 Insights from the Analytical Model

LIP Makes Plans More Robust

Using Equation 3.9 in Section 3.2.3, we showed that the naive evaluation strategy without LIP is θ-fragile with respect to the space of left-deep hash join trees, for $\theta = \frac{1-\sigma_{\text{min}}^{n-1}}{1-\sigma_{\text{min}}}$. On the other hand, using Equation 3.19 above, we showed that LIP with adaptive reordering is Θ-robust with respect to the same plan space, for Θ given by Equation 3.20.

Usually, the selectivities σ_i are fairly small, lying in the range of 5% to 30%. In such a scenario, the product of $n-2$ selectivities in Θ bound for LIP strategy is likely to be much smaller than the factor quadratic in the number of joins n. Further, the false positive error rate ϵ can be made arbitrarily small by appropriately configuring the Bloom filter. In fact, it can even be made 0 by using an exact LIP data structure (such as a bitmap). To illustrate this point, let us take an example query with $n = 6$ joins with selectivities 5%, 10%, ..., 30%. Then, P_w and P_b have a cost difference of at least $0.21|F|$ units, whereas B_w and B_b have a cost difference of at most $0.002|F|$.

From this discussion, it is clear that *LIP theoretically guarantees robustness, whereas the naive evaluation strategy is likely to make plan selection much more fragile.*

LIP Makes Plans Nearly Optimal

The above discussion shows that the LIP technique dramatically improves the robustness of query plans by ensuring that the difference in execution cost between the worst and best plans is very small. We now consider how a LIP query plan compares to the optimal query plan using naive evaluation. Using similar methods as above, we can get a lower bound for the total cost of the optimal naive query plan P_b, as well as an upper bound for the worst LIP query plan B_w.

$$T(P_b) \geq \sum_{i=1}^{n}(1+\sigma_i)|D_i| + \frac{1-\sigma_{\text{min}}^n}{1-\sigma_{\text{min}}}|F| \tag{3.21}$$

$$T(B_w) \leq \sum_{i=1}^{n}(1+\sigma_i+\beta\sigma_i)|D_i| + \frac{1-(\sigma_{\text{min}}+\epsilon)^n}{1-(\sigma_{\text{min}}+\epsilon)}\beta|F| + \sigma_1\sigma_2...\sigma_n\left[n + \frac{\epsilon}{\sigma_{1'}}\frac{n(n+1)}{2}\right]|F| \tag{3.22}$$

In the above bounds, we see that the BuildCosts differ by $\sum_{i=1}^{n}\beta\sigma_i|D_i|$. Since the dimension tables are typically much smaller than the fact table, particularly after application of a selection predicate, this cost difference is usually a small fraction of the optimal cost.

As noted before, in the BloomProbeCost(B_w), the second term (ϵ) can be made very small, so that this term is roughly β times HashTableProbeCost(P_b). This is because as long as the false positive rate ϵ is small, we make roughly the same number of probes into the Bloom filter in any query plan using LIP with adaptive reordering as we make into the hash tables in the optimal naive query plan P_b. However, due to its small size, the Bloom filter is likely to be cache resident and hence make the probes much faster, i.e., $\beta \ll 1$. But, there is a tradeoff here as making ϵ smaller by increasing the Bloom filter size or number of hash functions also makes β larger (i.e., probes become more expensive).

Finally, as we have discussed before, the product of selectivities in the upper bound for HashTableProbeCost(B_w) is typically small enough to render the term a negligibly small fraction of the total cost of the optimal plan, despite the quadratic dependence on the number of joins n.

Summing the three terms, we see that $T(B_w)$ is typically at least as small as $T(P_b)$, and is often better. In other words, *not only is an LIP query plan guaranteed to run in approximately the time for the optimal LIP query plan, it is also nearly always as good as (and often better than) the optimal naive query plan.* Our empirical evaluation in Section 3.4 also confirms these theoretical insights.

3.4 Evaluation

All the experiments presented in this section were performed using the Quickstep database engine, which is an in-memory relational DBMS. The experiments were run on a machine with 160GB of main memory and dual socket Intel Xeon E5-2660 v3 processors, with 10 physical cores per socket (i.e. 40 virtual cores with hyperthreading). Hyperthreading was turned on.

We used three datasets for the experiments: a) Star Schema Benchmark (SSB) [39] at scale factor 100 (i.e., database size roughly 100 GB), b) a synthetic dataset to stress different data parameters, and c) TPC-H at scale factor 100. For the experiments below, unless stated otherwise, we use the SSB dataset.

In all the experiments, the buffer pool was large enough to contain the entire working set in memory. All reported query execution times are averages of 5 successive runs. Since we are interested in how robust our techniques make the query execution times, we enhanced the query optimizer to produce all possible join orders for each query. All joins were performed using hash joins, where the fact table was used to probe the hash tables built using the result of predicate evaluation on the dimension tables.

All experiments involving LIP used Bloom filters configured to have 1 identity hash function and size 8 bits per tuple estimated in the dimension table (after selection). This configuration choice is explained in Section 3.4.1 below.

3.4.1 Choice of Bloom Filter Configuration

While LIP can use various filter data structures, in this paper, we focused on Bloom filters. The configuration of a Bloom filter is defined by its size (number of bits in the bit array) and the number and choice of hash functions used in building or probing it. These parameter choices affect not only its false positive rate (denoted ϵ in this paper) but also the computational cost of operations on it (denoted β). The false positive rate can be obtained from the configuration using theoretical closed-form results [34]. However, in this paper, we are interested in the overall execution cost for a query, which depends on both these factors. Therefore, we use empirical analysis to optimize the choice of Bloom filter configuration for this in-memory database context.

Figure 3.4 shows the optimal execution time (i.e., for the best join order) among all 24 possible join orders for SSB Query 4.3, for different Bloom filter configurations. (Results for other queries are similar, and in the interest of space, we only present results for Query 4.3.) The four curves in the figure show the execution times when the number of hash functions used for building and probing the Bloom filter is varied from 1 through 4. The x-axis for the plots is the size of the Bloom filter, as a ratio of the estimated cardinality of the selection result for the dimension table it is based

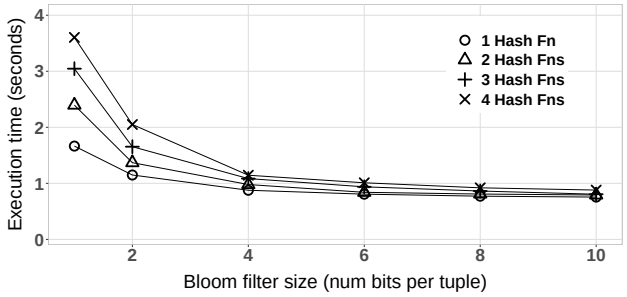


Figure 3.4: Comparison of optimal execution times for Query 4.3 using different Bloom filter configurations

on, i.e., the number of bits in the Bloom filter per tuple used to build the corresponding hash table.

It is clear that the best performance is obtained using just a single hash function, and about 8-10 bits per tuple. Further, the performance of the query is not sensitive to the size of the Bloom filter near this optimal size. This latter observation justifies the use of our technique even when the estimated cardinality of the table may be erroneous.

The fact that a single hash function gives the best performance at all Bloom filter sizes may seem surprising at first, particularly in light of the fact that theoretical results suggest that about 3 hash functions are required to get a low false positive rate. A deeper analysis using code profiling and CPU performance counters indicated that the main factor in the running time using LIP is the cache miss rate for the Bloom filter probing. If we use k hash functions, we make k lookups into the Bloom filter bit array for every positive tuple (those in the set), and 2-3 probes for the negative ones (those not in the set). Each lookup is a random reference which, for some of the larger dimension tables (i.e., larger bit array), is likely to be a cache miss. Choosing just a single hash function significantly reduces the cache miss rate.

Another factor that determines performance of the LIP approach is the choice of the hash functions. In our experiments with various different hash function families, we found that the identity hash function yields the best performance for this dataset. The results in Figure 3.4 are for the first hash function being an identity function, and the others being variants of Knuth's multiplicative hash functions [28]. More complex hash functions, while more "ideal" in a theoretical sense, were found to be too computationally inefficient in the case of this dataset.

3.4.2 Robustness to Join Order Selection

For each query in the SSB workload, we enumerated all possible join orders and ran them with and without LIP. Figure 3.5 shows the execution times for the first three queries (group 1) which have only one join. For the other queries, Figure 3.6 compares running times for all 24 join orders. In Figure 3.6, the execution times of query plans without using LIP is shown using diamonds, on the left in each subfigure, and they are connected to the execution times of query plans when using LIP

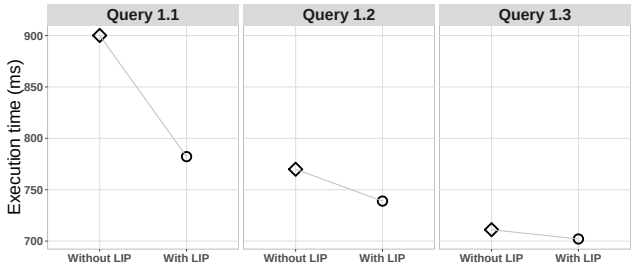


Figure 3.5: Execution times for SSB queries in group 1.

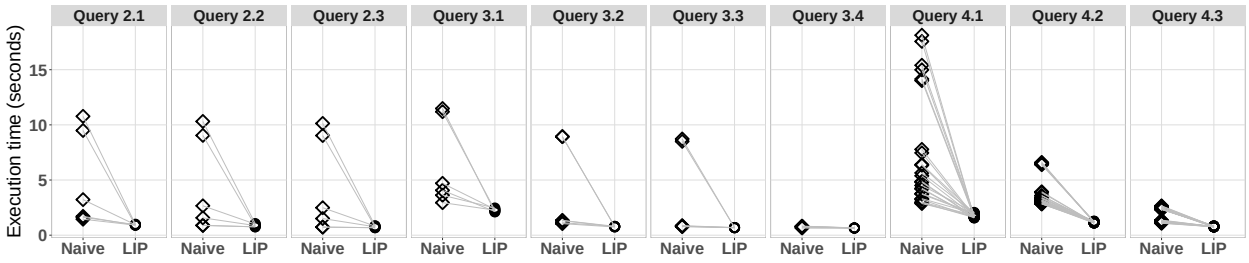


Figure 3.6: Execution times for the Star Schema Benchmark queries for groups 2–4, with and without LIP.

(the latter is shown using circles).

In the naive strategy, we see that for all queries where there are multiple possible join orders (i.e., excepting the queries in group 1), there are several query plans that are far worse than the optimal one. For instance, in case of Query 4.1, the worst plan (18.1s) is more than $6\times$ worse than the best plan (2.9s), and 10 of the 24 possible join orders have a running time at least double the optimal. On the other hand, the LIP strategy is much more robust to the join order. For the same query 4.1, all the 24 query plans have times within 5% of each other.

It is also remarkable that for most queries, the execution time of even the worst query plan using LIP is smaller than (or within the experimental error bounds) of the best query plan without LIP. In fact, comparing the total execution time for the entire benchmark, even choosing the worst query plan for every query along with LIP is better (17.4s) than choosing the optimal query plan for every query without LIP (17.6s), and far better than the worst plan for every query (90.9s).

3.4.3 Handling Skew and Correlation

In addition to the experiments using SSB, we also use a synthetic workload to stress-test our implementation using a large number of tables of widely-varying sizes, containing data with skew and correlation.

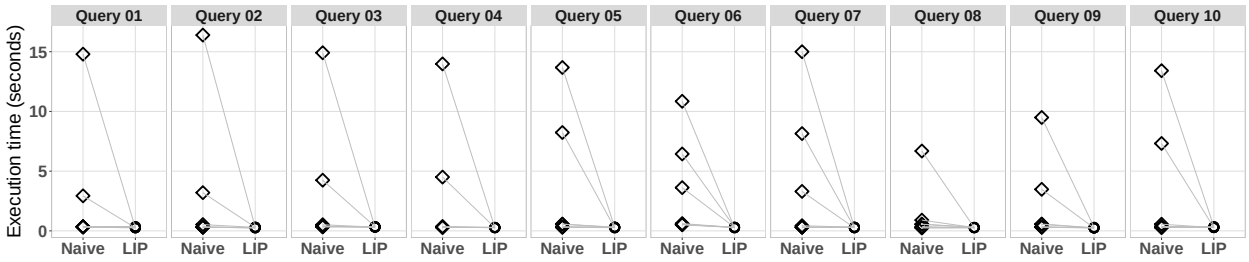


Figure 3.7: Execution times for sampled join orders of some queries in the synthetic workload.

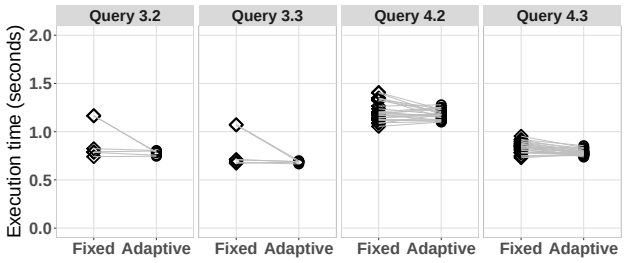


Figure 3.8: Execution times for fixed and adaptive lookahead filter probe ordering.

The synthetic data generator creates a star schema database consisting of a fact table and n dimension tables (we set $n = 12$). Each dimension table has three columns, one integer primary key and two character columns with 25 distinct values each. They vary in size (4 each have sizes in ratio 1:10:100). The fact table has a primary key and all 12 foreign keys and is 10x larger than the largest dimension table. The total database size is about 10 GB.

Each value in the character columns consists of a prefix and a suffix. The prefix is always picked uniformly, whereas the suffix is picked uniformly in half the tables and, in the rest, using an inverse exponential distribution (as in [41]). Further, in half the dimension tables, the two character column values are correlated, sharing the same prefix with probability 0.80. In the rest of the tables, these columns are independent. The foreign keys in the fact table are independent and uniformly distributed. The 12 dimension tables are obtained using all possible combinations of size, distribution and correlation above.

We ran 20 synthetically generated queries on this dataset, each joining the fact table with a random subset of the dimension tables using an appropriate randomly generated equality predicate. On average, each query contains six dimension tables.

For each query, we randomly sampled 26 different join orders from the plan space. Each join order was executed with and without LIP. Figure 3.7 shows the execution times for some of these queries (we omit the rest due to space constraints).

For every query, the best plan in the sample set had an execution time between 200 and 400ms. Using LIP improved the best time for each query by about 13% on average. Only three of the queries were negatively impacted, with a 4% slowdown at worst. More importantly from the perspective of robustness, the average difference in execution times between the worst and best plans went down from 14 seconds without LIP to just 150 ms with LIP.

3.4.4 Importance of Adaptiveness

In this experiment, we ran each SSB query using all possible join orders, with the order of LIP filter probing either fixed or chosen adaptively using Algorithm 1.

Figure 3.8 shows the resulting execution times for some selected queries. Overall, adaptive execution imposes little overhead (at most 8%). However, in some queries, such as queries 3.2 and 3.3, adaptiveness is crucial for maintaining robustness. For instance, for a particular join order for query 3.2, when we changed the bloom filter probes to be adaptive rather than the same fixed ordering as the join order, the slowdown from the optimal ordering dropped from 57.6% to just 6.3%. These results support our claim in Section 3.3.2 that adaptive reordering is a critical part of the robustness-enhancing property of the LIP technique.

3.4.5 Applying LIP to Subplans

Throughout this paper, we have limited our focus to left-deep join trees for star schemas. However, the LIP technique is more generally applicable to subplans with this pattern in larger query plans. To demonstrate this wider applicability, we picked Query 8 from the TPCH benchmark (scale factor 100) and applied LIP to star join subplans within each of the 675 query plans enumerated for this query by our optimizer.

Figure 3.9a shows one of the subplans following this pattern. While we have used `LINEITEM` as the "fact" table, the "dimension" tables in this pattern are themselves subplans having the same primary key as the `ORDERS`, `PART` and `SUPPLIER` tables. Note that while `ORDERS` is usually considered a fact table in the TPCH schema, for the purpose of LIP application, it can be considered as a dimension table due to the primary key - foreign key constraint between it and the `LINEITEM` table.

Figure 3.9b illustrates the gain in robustness using a box plot. With the naive execution strategy, the running times for this query varied from 2.1 s to 58 s, whereas LIP reduced this spread to between 1.3 s and 7.4 s. In addition to this $9\times$ reduction in the difference between running times of plans, every query plan also saw an improvement in performance, with speedups varying from $1.20\times$ to $18\times$ (geometric mean of speedups was 4.0). In fact, 25% of the query plans ran faster with LIP than the optimal plan with naive evaluation.

3.5 Related Work

We organize the related work in five groups, and discuss each group below.

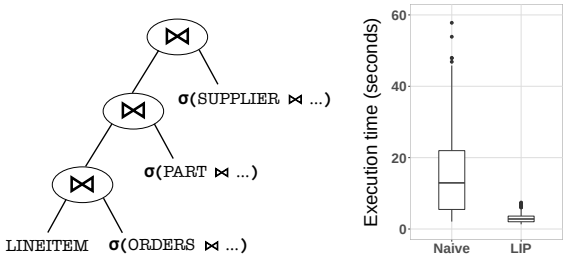
Bloom filters. First introduced in [10], Bloom filters have been used extensively in distributed database systems to minimize I/O and network transmission costs [30, 4, 12], as well as in semi-join optimization [9, 47, 30, 4, 12, 22, 43] to accelerate joins. In this chapter, we use bloom filters in LIP and also explore the impact of its parameters when used with LIP.

Sideways information passing (SIP). The *semi-join reduction* [9, 47, 30, 4, 12] accelerates a single join by passing a filter from one side to the other. SIP strategies and *magic sets transformation*, first introduced in [8], consider the space of semi-join reductions and associated query rewriting techniques. There has been much follow-on work in this space, including the use of greedy heuristics [15, 36] and cost-based approach [42].

Our proposed LIP strategy can be considered a special case of the general SIP strategies, though with the additional crucial component of adaptive filter reordering. Further, LIP also bears considerable likeness to *adaptive information passing* [25]. However, to the best of our knowledge ours is the first work to focus on the *robustness* benefits of SIP, as well as the ability to get *near-optimal* performance from all plans in the subspace under consideration. We supported these claims using both theoretical and empirical results.

Adaptive reordering of filters. Prior work has shown the benefits from reordering of predicates in relational and stream processing. If selectivities are known exactly, then the optimal filter ordering can be derived directly [24]. Otherwise, learning-based approaches [3, 7] can be used. Our use of adaptive reordering of lookahead filters is inspired by these approaches. However, we focused our implementation efforts on ensuring low-overhead adaptation, as well as cache-sensitive bulk application of the filters. These implementation details are crucial to the fast convergence rate as well as the robustness benefits of LIP.

Robust query execution. The notion of robustness in query optimization has been well studied in the literature [49]. Proposed techniques include correcting cardinality estimates though sampling [5] or runtime feedback [45], dynamically switching between a candidate set of plans at runtime [17], as well as runtime re-optimizations [32, 6]. We consider our work to be complementary to such techniques. Whereas these techniques require a redesign of the query optimizer (in addition to any changes in the execution engine), our proposed LIP approach attempts to entirely avoid changes to the query optimizer, using query evaluation to immunize against selectivity



(a) (b)

Figure 3.9: Left-deep star join subplan in TPCH Query 8, and execution times of all 675 possible query plans.

estimation errors.

Worst-case optimal join algorithms. The result that LIP makes all plans nearly optimal may be considered related to the class of *worst-case optimal join* (WCOJ) algorithms [48, 16, 37] that essentially look at all join relations at the same time. However, we note that there is not much linkage between LIP and the WCOJ algorithms. The WCOJ algorithms particularly target *full conjunctive* queries that are *cyclic*, and try to bound the total evaluation time by the overall output size. However, under a star schema where we evaluate joins between the fact table and multiple dimension tables, the WCOJ algorithms will be suboptimal compared to LIP, due to its overhead in building heavy hash [37] or B-tree [48] indices, or sorting [16] all the relations.

3.6 Conclusions and Future Work

In this chapter we have introduced a query execution strategy called LIP that collapses the space of left-deep query plans for star schema warehouses down to almost a single point near the optimal plan. In addition to this robustness benefit, it also significantly speeds up query execution in this important subplan space. We have demonstrated these claims through theoretical and empirical results. Besides the immediate application of LIP, we believe our work opens a novel approach to the notion of "robustness", one that is focused on query execution strategies possibly tailored to corresponding query plan (sub-)spaces.

In future work, we hope to generalize these ideas about robustness and the specific LIP strategy to more complex schemas and query plan shapes. We will also explore how this new approach to robustness impacts query optimization.

Chapter 4

Additional Query Optimization: Drop Early, Drop Fast

In this chapter, we introduce two additional query optimization techniques. Both techniques improve Quickstep's performance on data warehousing workloads (such as the SSB and the TPC-H benchmarks). Moreover, the second technique is crucial to the performance of the QuickGrail system, as we will introduce in Chapter 5.

Below, we first describe a technique that pushes down certain disjunctive predicates more aggressively than is common in traditional query processing engines. Then, we describe how certain joins can be transformed into cache-efficient semi-joins using *exact filters*. An exact filter can be pushed down as a kind of LIP filter where the original join or semi-join gets completely eliminated.

The unifying theme that underlies LIP and these two query optimization methods is to eliminate redundant computation and materialization using "drop early, drop fast" approaches: aggressively pushing down filters in a query plan to drop redundant rows as early as possible, and using efficient mechanisms to pass and apply such filters to drop them as fast as possible.

4.1 Partial Predicate Push-down

While query optimizers regularly push conjunctive (`AND`) predicates down to selections, it is difficult to do so for complex, multi-table predicates involving disjunctions (`OR`). Quickstep addresses this issue by using an optimization rule that pushes down *partial predicates* that conservatively approximate the result of the original predicate.

Consider a complex disjunctive multi-relation predicate P in the form $P = (p_{1,1} \wedge \cdots \wedge p_{1,m_1}) \vee \cdots \vee (p_{n,1} \wedge \cdots \wedge p_{n,m_n})$, where each term $p_{i,j}$ may itself be a complex predicate but depends only on a single relation. While P itself cannot be pushed down to any of the referenced relations (say

R), we show how an appropriate relaxation of P, $P'(R)$, can indeed be pushed down and applied at a relation R.

This predicate approximation technique derives from the insight that if any of the terms $p_{i,j}$ in P does not depend on R, it is possible to relax it by replacing it with the tautological predicate $\top$ (i.e., `TRUE`). Clearly, this technique is only useful if R appears in every conjunctive clause in P, since otherwise P relaxes and simplifies to the trivial predicate $\top$. So let us assume without loss of generality that R appears in the first term of every clause, i.e., in each $p_{i,1}$ for $i = 1, 2, \ldots, n$. After relaxation, P then simplifies to $P'(R) = p_{1,1} \vee p_{1,2} \vee \ldots \vee p_{1,n}$, which only references the relation R.

The predicate P' can now be pushed down to R, which often leads to significantly fewer redundant tuples flowing through the rest of the plan. However, since the exact predicate must later be evaluated again, such a partial push down is only useful if the predicate is selective. Quickstep uses a rule-based approach to decide when to push down predicates, but in the future we plan to expand this method to consider a cost-based approach based on estimated cardinalities and selectivities instead.

Our experiments show that the partial predicate push-down technique improves the performance of two queries in the TPC-H [46] benchmark. In particular, for TPC-H at scale factor 100, the technique improves Query 07 by a speedup of 6X and improves Query 19 by a speedup of 4X.

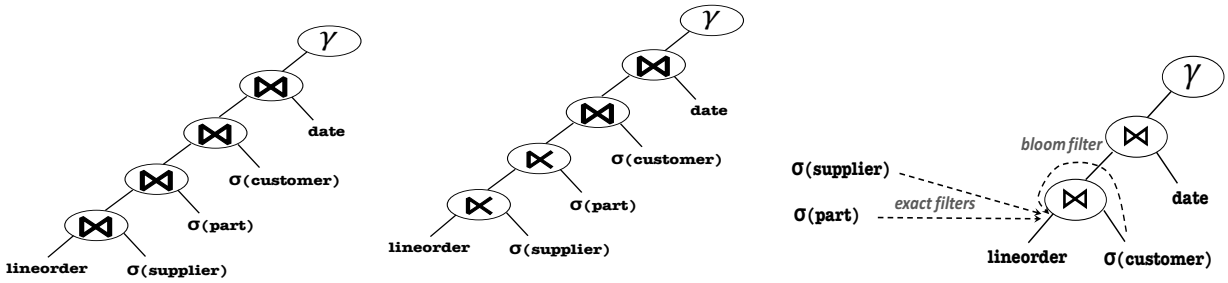
There is a discussion of join-dependent expression filter pushdown technique in [11], but the overall algorithm for generalization, and associated details, are not presented. The partial predicate push-down can be considered a generalization of such techniques.

Note that the partial predicate push down technique is complimentary to implied predicates used in SQL Server [33] and Oracle [40]. Implied predicates use statistics from the catalog to add additional filter conditions to the original predicate. Our technique does not add any new predicates, instead it replaces the predicates from another table to `TRUE`.

4.2 Exact Filters: Join to Semi-join Transformation

A new query processing approach that we introduce in this chapter is to identify opportunities when a join can be transformed to a semi-join, and to then use a fast, cache-efficient semi-join implementation using a succinct bitvector data structure to evaluate the join(s) efficiently. This bitvector data structure is called an *Exact Filter* (EF), and we describe it in more detail below.

To illustrate this technique, consider the SSB [38] query Q4.1 (see Figure 4.1a). Notice that in this query the `part` table does not contribute any attributes to the join result with `lineorder`, and the primary key constraint guarantees that the `part` table does not contain duplicates of the join key. Thus, we can transform the `lineorder` – `part` join into a semi-join, as shown in Figure 4.1b.



(a) Original query plan

(b) Plan using join to semi-join transformation

(c) Final plan using exact filters to eliminate joins

Figure 4.1: Query plan variations for SSB Query 4.1

During query execution, after the selection predicate is applied on the `part` table, we insert each resulting value in the join key (`p_partkey`) into an *exact filter*. This filter is implemented as a bitvector, with one bit for each potential `p_partkey` in the `part` table. The size of this bitvector is known during query compilation based on the min-max statistics present in the catalog. (These statistics in the catalog are kept updated for permanent tables even if the data is modified.) The EF is then probed using the `lineorder` table where the original join gets completely eliminated. The same optimization also applies to the `supplier` table and Figure 4.1c shows the final optimized plan.

The implementation of semi-join operation using EF rather than hash tables improves performance for many reasons. First, by turning insertions and probes into fast bit operations, it eliminates the costs of hashing keys and chasing collision chains in a hash table. Second, since the filter is far more succinct than a hash table, it improves the cache hit ratio. Finally, the predictable size of the filter eliminates costly hash table resize operations that occur when selectivity estimates are poor.

The same optimization rule also transforms anti-joins into semi-anti-joins, which are implemented similarly using EFs.

As with the previous section, we use the SSB 100 scale factor dataset to do the experiments. (The results for the TPC-H dataset is similar.) The LIP and EF techniques together provide a nearly 2X speedup for the entire benchmark. The LIP and semi-join transformation techniques individually provide about 50% and 20% speedup respectively. While some queries do see a slowdown due to the individual techniques, the application of both techniques together always gives some speedup. In fact, of the 13 queries in the benchmark, 8 queries see at least a 50% speedup and three queries see at least 2X speedup. The largest speedups are in the most complex queries (group 4), where we see an overall speedup of more than 3X.

4.3 Conclusions and Future Work

In this chapter, we have described additional query optimization techniques that eliminate redundant computation and materialization under a "drop early, drop fast" theme. The techniques include aggressive push-down of certain disjunctive predicates and cache-efficient semi-joins using exact filters.

There is a never ending search for even more sophisticated query optimization techniques. And looking at optimizing queries with a very large number of joins and aggregation is a direction of future work. Examining if machine learning methods can help in query optimization or if methods like LIP can be extended even further is likely to be a interesting research direction.

Chapter 5

Provenance Graph Analytics Using the Quickstep/Grail Approach

5.1 Introduction

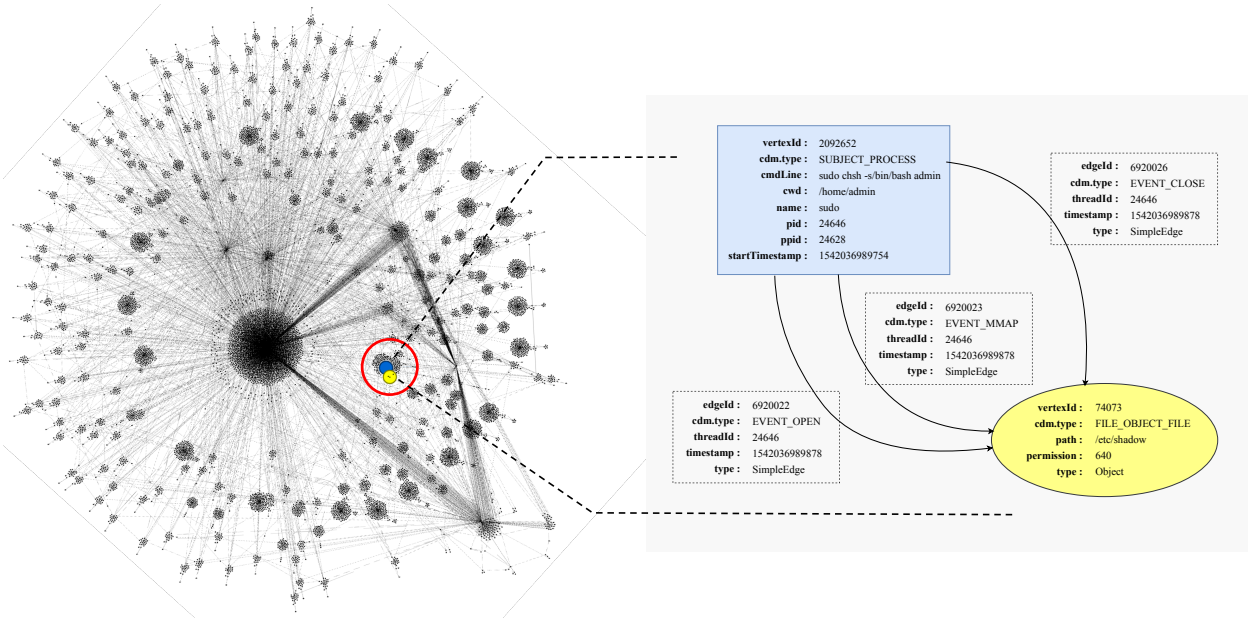
In this chapter, we look into designing an efficient query processing engine for interactive exploration on large provenance graphs.

The larger background lies in the DARPA Transparent Computing (TC) program. The TC program aims to develop technologies that record and preserve the *provenance* of system elements/components and the interactions and causal dependencies among these components, and perform intrusion detection and forensic analysis based on the provenance to counter various types of cyber threats.

One of the technical challenges that the TC program presents is to store and efficiently query high-volumes of provenance data structured in a graph data model called *provenance graph*. The provenance data arrive in high velocity. A single server can produce 100 million provenance graph vertices and edges per day, and in typical scenarios we want to query the graph within an one-week or one-month span.

The basic types of queries to the provenance graph are *filter*, *lineage* and *path* queries. The filter queries find vertices/edges that satisfy certain constraints (predicates) on annotations. The lineage queries find all ancestors or descendents in the graph starting from a collection of vertices. The path queries find all possible vertices and edges on any path between two collections of vertices.

Unfortunately, existing systems cannot even effectively handle the basic lineage and path queries. On one hand, due to the power-law nature of real-world provenance graphs, one extra hop on the graph may geometrically expand the number of combinations of possible paths. Thus for systems using traditional path traversal algorithms (e.g. [1, 2]) it is totally unpredictable whether a single



(a) (b)

Figure 5.1: The left figure (a) shows an example provenance subgraph containing 4996 vertices and 12207 edges, which is part of an overall graph of 10 million vertices and 33 million edges. The right figure (b) shows the detailed annotations of 2 vertices connected by 3 edges from (a).

lineage query may come back in a few seconds, or just do not finish (DNF) for one day. On the other hand, systems designed for handling large-scale power-law graphs (e.g. [19, 31, 20]) mainly use the Gather-Apply-Scatter (GAS) computation model that is effective for data mining and machine learning workloads. It is unclear whether the GAS model can be adapted well to handle the path queries and more generally the pattern matching queries.

In this chapter, we propose a system called *QuickGrail* that have efficient implementations of the filter, lineage and path operations, together with fast manipulation of intermediate result graphs. We then use these basic operations as building blocks to further efficiently support a class of graph pattern matching queries. With all these functionalities, the QuickGrail system can both server as a layer to retrieve data for upstream analysis, and server as a tool for interactive exploration on provenance graphs.

5.2 Preliminaries

We begin this section by introducing *property graphs*, which is the underlying logical data model used for representing graphs throughout the chapter. Then, we scope our focus to a domain of graphs called *provenance graphs*, and refine a space of queries that our system needs to support for

provenance graph analysis.

5.2.1 The Property-Graph Data Model

Property graph is a general and widely used logical data model for structuring graph data. The data model is supported by many existing graph database systems, such as Neo4j, Apache Giraph, and GraphLab (now Turi).

A property graph is simply composed of a collection of vertices and directed edges, where each vertex and each edge is associated with a set of key-value pairs called *properties* (or *annotations*). Besides, an optional *schema* can be defined to constraint the property domain, such as the possible combination of key names and data types.

Figure 5.1b shows an example property graph. The vertices and edges each contain property pairs such as "`name:sudo`" and "`threadId:24646`". Note that the two vertices have different sets of property keys and we visualize them with different shapes and colors.

Besides the syntactic representation, the properties in Figure 5.1b are also associated with certain domain-specific semantics that are interpreted as *provenance* information. In the next sections we introduce provenance graphs and the related analytical demands in detail.

5.2.2 Provenance Graph

Provenance is a term that is used within a wide range of fields, such as Art, Law, Archaeology, Anthropology, etc., and Computer Science. Though the precise meaning varies in different contexts, the term roughly means the history (origin, relation, causality, lineage) of objects and activities.

There are different ways of organizing provenance information. One standardization effort is the Open Provenance Model (OPM) [35]. OPM defines *provenance graph* as the data model for structuring provenance information, which can be viewed as a property graph constrained by a schema on property key names and value domains. In this thesis we consider provenance graphs that conform to a slight variance of OPM.

Here we still use Figure 5.1 as an example. Note that there are two vertices in the graph. One vertex stands for a *process* and the other stands for a *file*. The three edges record activities (*open*, *mmap*, *close*) of the process on the file, at different time points.

5.2.3 Querying a Provenance Graph

The queries to a provenance graph can be high-level, vague and descriptive, such as

> *Detect in an enterprise-level system provenance all potentially compromised accounts and track related suspicious activities*. (4.2.A)

Or they can be low-level, explicit and operational, such as

> *Find all the vertices that have remote IP address as 128.104.222.140 and show all descendants of these vertices that are within 2 hops where the edges have timestamps within time interval [t1, t2].* (4.2.B)

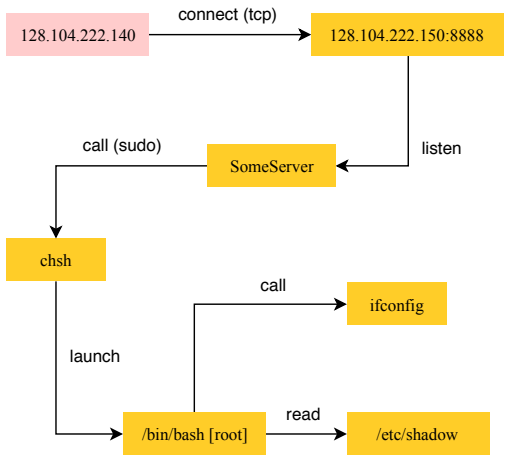
In this thesis, we will focus on a collection of low-level style operations on provenance graphs. The operations allow a human analyst to filter the graph to yield (possibly very large) *subgraphs* as intermediate results, and do subgraph manipulations such as graph *union*, *intersection*, *subtraction*, as well as finding paths among subgraphs. The operations should be fulfilled in a timely manner so that the analyst can interactively and iteratively dig through a provenance graph using typical *trial-and-error* and *divide-and-conquer* methods.

Below Listing 5.2 shows an example script that does the analysis described by (4.2.B). The script is written in the *QuickGrail* language which we will introduce in detail in Section 5.3. The analysis is done step by step, where at each step an operation is performed to either derive a new graph from existing graphs, or to examine intermediate results so that a human analyst can decide the subsequent exploration actions.

```
# From the base (i.e. overall) graph,
# find all vertices that have remote IP address as 128.104.222.140,
# and denote the result (all-vertices) subgraph as $ip.
$ip = $base.getVertex(remoteAddress = '128.104.222.140';

# Show summarized information about graph $ip,
# e.g. number of vertices, annotation statistics.
describe $ip

# Get a subgraph out of the base graph by selecting edges
# that have timestamps between 100 and 200.
$slice = $base.getEdgeWithEndpoints(timestamp >= 100 and timestamp <= 200)

# Within $slice, find all descendants that are within 2 hops of each vertex in $ip.
$descendents = $slice.getLineage($ip, 2, 'desc');

# Export graph $descendents in a format that is ready for visualization,
# and mark those vertices which are also in graph $ip with red color.
visualize $descendents $ip.attr('color', 'red');
```

Listing 5.2: A script that performs analysis (4.2.B) using QuickGrail language.

Moreover, on top of the primitive operations, we also consider a class of powerful *pattern matching queries* that make things easier for digging out complex connection patterns. Though it is known that the general graph pattern matching problem is NP-hard, we alleviate the problem by allowing false positive results when a query is not *acyclic*.



(a)

```
with $base max depth 5
match remote ->* server ->* serverProcess
             ->* chsh ->* bash
  and bash ->* ifconfig
  and bash ->* shadow
  and remote.remoteAddress = '128.104.222.140'
  and server.localAddress = '128.104.222.150'
  and server.localPort = 8888
  and serverProcess.* LIKE '%SomeServer%'
  and chsh.* LIKE '%chsh%'
  and bash.* LIKE '%/bin/bash%'
  and ifconfig.* LIKE '%ifconfig%'
  and shadow.* LIKE '%/etc/shadow%'
as $matchedGraph;
```

(b)

Figure 5.3: Example graph pattern that represents a potential infiltration into a server, and the corresponding pattern matching query written in QuickGrail language.

Figure 5.3a shows an example connection pattern that represents a potential infiltration into a server, and Figure 5.3b is the corresponding pattern matching query written in *QuickGrail* that extracts all matched vertices and edges from the underlying provenance graph. The pattern matching query can be read as

> *Find within the base graph all possible combinations of vertices {remote, server, serverProcess, chsh, bash, ifconfig, shadow}, such that "remote" has a path to "server", and "server" has a path to "serverProcess", and . . . , and vertex "remote" has annotation "remoteAddress" with value "128.104.222.140", and . . . , where each path has a max depth of 5. Finally union all the possible result vertices and the paths among them as specified by the query into result graph* `$resultGraph`.

Note that each edge in pattern 5.3a is usually an "abstract" edge that is indeed a *path* without the semantic information (such as "connect" and "listen"), so that the example query does not contain edge predicates. More details on pattern matching queries will be described in Section 5.3.1.

5.2.4 Main Challenges

By now, we have introduced a collection of operations/queries for interactively analyzing the provenance graph data. Our main challenge is to build a system (named *QuickGrail*) that supports these analytical queries with *high performance*, *robustness* and *scalability*. The three characteristics are closely related and are all critical towards achieving interactiveness:

- *High performance* means that under a typical workload, an individual operation should be completed within a few seconds whenever possible. This is usually the response time bound for a human analyst to keep attention and maintain productivity when involved in an interactive session.

- *Robustness* requires that the processing time of a batch of queries be *linear or sublinear* to (each of) the graph size, query size and certain configurable parameters, so that the overall query processing time is bounded and predictable. Specifically, one counterexample is general join processing in relational databases, where the sizes of intermediate states and output could grow much larger than the size of input.

- *Scalability* means that we can retain high performance by investing more computational resources when the workload volume grows. In this thesis, we focus on a scale-up setting where the system scales on a single machine with regard to the number of available CPU cores and the size of main memory.

Here is an overview of our approach to achieve the required system characteristics. We build the QuickGrail system on top of the Quickstep RDBMS. Graph operations and pattern matching queries are finally translated into SQL queries to be executed by Quickstep. Specifically:

(a) Predicates on graph properties are translated into Quickstep scans plus subgraph union/intersection/subtraction operations.

(b) Path queries and subgraphs operations are translated into iterative or nested semi-joins in Quickstep, which are then evaluated with the help of LIP (Lookahead Information Passing) and EF (Exact Filters).

(c) Each individual operation in class (a) and (b) are inherently fast, robust and scalable by virtue of the guarantees from Quickstep's scan and semi-join operators. Meanwhile, we encapsulate the operations in (a) and (b) as *Grail Low-level Instructions* (GLL instructions).

(d) Graph pattern matching queries are categorized as *easy* (acyclic) queries and *hard* queries. We build an optimizer that transforms an easy pattern matching query to a sequence of GLL instructions where the number of instructions is linear to query size (i.e. number of atomic clauses in the query), so that the overall processing time is bounded and predictable.

For hard queries that fall into the category of arbitrary graph pattern matching, note that the problem is in general NP-hard so exact answers are hard to get. We allow the system to return false positive results and allow user to specify a *number-of-iterations* parameter k, so that the overall processing time is linear to k times the query size, and the larger the k the

more accurate the results. Finally, even though false positives exist, the results are usually greatly reduced in size compared to the original graph, and can be fed as input into other systems which are designed to handle hard queries well but not capable of working with very large graphs.

5.3 Our Approach

In this section, we will describe in detail our techniques of building an analytical processing system *QuickGrail* that delivers high performance, robust and scalable querying on provenance graph data.

Figure 5.4 shows the overall QuickGrail architecture. The system is built on top of the Quickstep RDBMS and graphs are stored inside Quickstep as relational tables. Each QuickGrail query may derive new subgraph tables that are stored in Quickstep, or export existing subgraph tables in specified formats as result to the user. A simple *garbage collection* mechanism is utilized to remove unreferenced subgraph tables on a per-query basis.

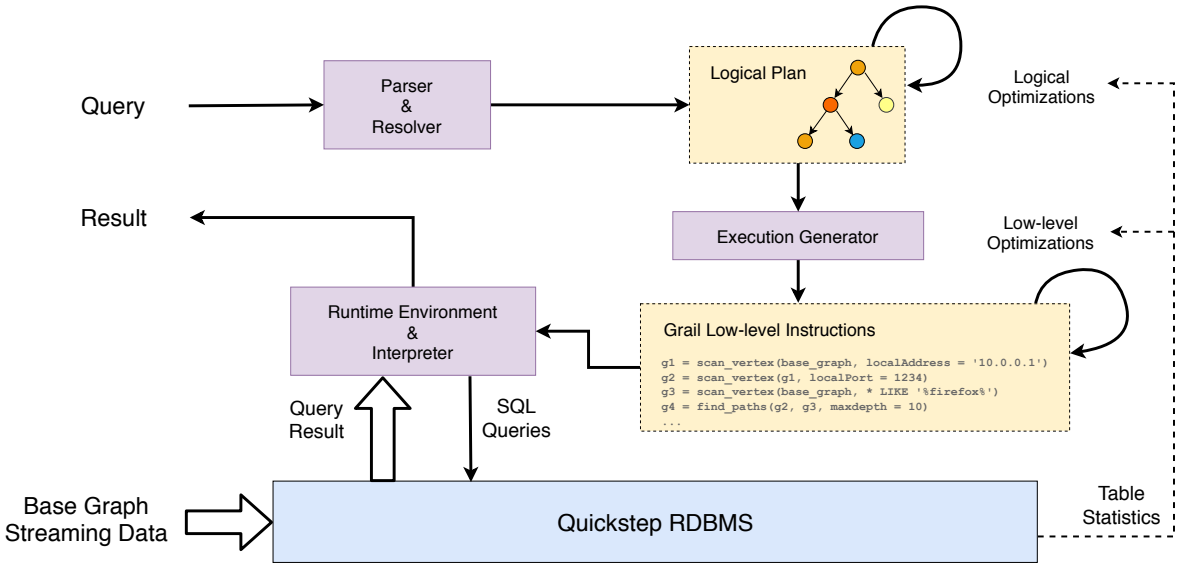


Figure 5.4: The QuickGrail architecture.

During query processing, the system first parses a QuickGrail query into a logical plan. The logical plan is then transformed and optimized mainly to make the query's graph pattern matching components ready for code generation. After that, the execution generator traverses the logical plan trees and generates a sequence of *Grail Low-level* (GLL) instructions accordingly. The GLL instructions in general conform to the *Single Static Assignment* (SSA) form and thus many standard data flow optimizations (such as deadcode elimination, common subexpression elimination,

strength reduction, copy propagation) can be conveniently applied. In fact, a number of useful optimizations can be done either at the logical plan stage or at the GLL stage, but optimization rules for the later stage are easier to write.

Finally, the system has an *interpreter* that executes the GLL instructions in sequencial order. Each instruction has its implementation in QuickGrail's host language (e.g. Java or Python) that may issue one or more SQL queries to the Quickstep RDBMS via socket or RPC connection, and retrieves Quickstep responses as necessary.

The contents in this section are organized as follows. We first introduce the QuickGrail language by examples. Then describe the underlying storage of the base graph and the derived subgraphs in Quickstep. We then list a core set of the GLL instructions and present the underlying implementations in detail. We also analyze the time cost of each GLL instruction by diving deep into Quickstep's execution. After that, we briefly describe how to translate QuickGrails' graph expressions into a sequence of GLL instructions and show a core part of the translation algorithm. Finally, we explain how to handle graph pattern matching and discuss some of the query optimization techniques.

5.3.1 The QuickGrail Language

In this subsection we introduce the core set of QuickGrail queries. The key entities in the QuickGrail language are *subgraphs*, where most of the queries take subgraphs as input and yield a new subgraph as result. The result subgraph can be persisted and referenced by a *graph variable*.

In the QuickGrail language model, we assume that there is a single *base* graph that represents the ground data which grows in an append-only manner. All subgraphs are derived from this base graph and the vertices and edges must be exact subsets of the base graph, that is to say, no mutations. Throughout the section we will use words "subgraph" and "graph" alternatively and they actually both refer to subgraphs of the single base graph.

Furthermore, to better explain the semantics of QuickGrail operations, we briefly formalize the notion of the base graph and the subgraphs, as follows.

Below we organize the queries into five categories and walk through them by examples.

Filter vertice and edges by annotations

This category of queries creates a new subgraph by applying an *annotation predicate* to the input graph's vertices or edges. The predicate can be value comparisons, string pattern matchings, or compositionally the logical conjunction, disjunction and negation of predicates. Listing 5.6 shows two examples.

```
# From the base graph, find all edges that have property "timestamp" value between
# 100 and 200, and denote the result graph as $a.
$a = $base.getEdgeWithEndpoints(timestamp > 100 and timestamp < 200);

# From the subgraph $a, find all vertices in $a that have any annotation value
# matching regular expression pattern 'pid12\d+', and denote the result graph as $b.
$b = $a.getVertex(* REGEXP 'pid12\d+');
```

Listing 5.5: Example vertex filtering and edge filtering queries.

Subgraph operations

There are only three subgraph operations: *union* ($+$), *intersection* ($\&$) and *subtraction* ($-$). As the name indicates, each of the operations just applies the corresponding *set operation* to the underlying sets of vertices and edges, respectively, from the input graphs. Listing 5.6 shows an example.

```
# Get subgraph $e by union $a with the intersection of $b and $c,
# then subtract the vertices and edges from $d.
$e = $a + ($b & $c) - $d;
```

Listing 5.6: Example subgraph operations.

Note that graph subtraction may yield *dangling edges* in the result graph, and we consider a graph with dangling edges *valid* in QuickGrail. The user may use subsequent queries to either remove dangling edges or attach endpoint vertices.

Path queries

A path query finds within an *overlay graph* all paths from one set of *source vertices* to another set of *destination vertices* within a specified max path length, and returns the overall collection of vertices and edges on those paths as a result graph.

Note that the result of a QuickGrail path query is different from that of a typical relational path query, where the later one's result is a table of (*source*, *destination*) vertex pairs.

Moreover, one special case of a path query where one argument includes all the vertice in the overlay graph is the *lineage* query, as Listing 5.7 shows.

Pattern matching queries

Pattern matching queries allow users to specify multiple annotation / path constraints in one place, and find results that satisfy all the constraints. Listing 5.8 shows an example.

```
# Within the overlay graph $g, find all the possible paths from any vertex in $a
# to any vertex in $b within 5 hops, and union these paths as the result graph $c.
$c = $g.getPaths($a, $b, 5);

# Within the overlay graph $c, find all descendants of $a that is within 1 hop.
# Note that the result graph also include paths along the lineage and this query
# is actually equivalent to "$d = $c.getPath($a, $c, 1);"
$d = $c.getLineage($a, 1, 'desc');
```

Listing 5.7: Example path query and lineage query.

```
1  with $U
2  max depth 5
3  match a -[e]->* b
4    and b ->{1..3} c
5    and a.remoteAddress = '128.104.222.140'
6    and (c.cmdLine LIKE '%/bin/bash%' or c.name = 'ssh')
7    and e in $V
8  as $matchedGraph
9  set $a = {a}, $b = {b}, $c = {c};
```

Listing 5.8: Example pattern matching query.

There are 3 groups of components in a pattern matching query: the *global constraints*, the *body*, and the *output specifications*. Below we introduce these components using the example query in Listing 5.8.

The global constraints are query-wide and are applied to all components in the body of a query. Specifically, in Listing 5.8 lines 1-2, "*with $U*" says that the query should only look at subgraph $U, and "*max depth 5*" says that all path components in the body constraints have a default max depth of 5 unless explicitly overwritten.

The body constraints (Listing 5.8 lines 3-7) constitute the major part of the query that comprise of logical compositions (i.e. conjunction, disjunction, negation) of atomic constraints on *free variables*. Each free variable, depending on its surrounding context, stands for *a vertex* or *a set of edges* to be pattern matched. For example, in Listing 5.8, a, b, c are vertex variables and e is an edge variable.

Each atomic constraint can be a path specification, an annotation predicate, or a containment constraint. In Listing 5.8, lines 3-4 are path specifications saying that vertex a should have a path of any length through edges e to vertex b, and vertex b should have a path of length between 1 and 3 to vertex c. Lines 5-6 are annotation predicates on vertices a and c, and line 7 is a containment constraint saying that edges e should be from subgraph $V.

The semantics of a pattern matching query is to find all possible combinations of bindings to each free variable such that the constraints are all satisfied. In fact, despite of the global constraints, the edge constraint and the path length constraints, we can express Listing 5.8 as a first-order logic (e.g. datalog) query:

```
Q(a, b, c) :- Path(a, b), Path(b, c),
              VertexAnnotation(a, 'remoteAddress', '128.104.222.140')
              VertexAnnotation(c, 'cmdLine', ca),
              ( ca LIKE '%/bin/bash%' ; ca = 'ssh' ).
```

But again note that the output of a QuickGrail pattern matching query is not like $Q(a, b, c)$ which is a relation of all possible combinations of free variables. Instead, it returns the overall collection of vertices and edges on all involved paths as a result graph (Listing 5.8 Line 8), and projections of the result graph on each of the free variables (Listing 5.8 Line 9). In fact, this weakened result format is one key design choice to make pattern matching queries *computationally tractable*, because results in relational form such as $Q(a, b, c)$ can have as many as $|V|^m$ rows, where $|V|$ is the number of vertices in the result graph, and m is the number of free variables in a query. In our practical workloads, the ad-hoc results during exploratory analysis can often contain more than millions of vertices so that even $m = 2$ is not tractable for the $|V|^m$ bound.

5.3.2 Relational Storage

The QuickGrail system stores four types of conceptual entities in the underlying Quickstep RDBMS: the *base graph*, the *subgraphs*, the *auxiliary annotations*, and a *symbol table*. Each entity may be represented by one or more tables. Below we go through them in detail.

Storage of the base graph

QuickGrail represents the base graph with four relational tables: *vertices*, *edges*, *vertices_annotations*, and *edges_annotations*. Listing 5.9 shows the detailed SQL schema definition. Each vertex or edge in the base graph simply takes one row in the *vertices* or *edges* table, and may use multiple rows in the *vertices_annotations* or *edges_annotations* table to encode the associated annotations, respectively. Moreover, the two annotation tables use dictionary encoding to compress the columns so that the storage redundancy from duplicated *id*'s and *key*'s are minimized.

The way that QuickGrail structures the two annotation tables is also known as the *entity-attribute-value* representation or *vertical* representation, as opposed to the *horizontal* representation where annotation keys become names of multiple columns and then all annotation values of

```
CREATE TABLE vertices (
  "id" integer PRIMARY KEY
);

CREATE TABLE edges (
  "id" integer PRIMARY KEY,
  "src" integer FOREIGN KEY REFERENCES vertices(id),
  "dst" integer FOREIGN KEY REFERENCES vertices(id)
);

CREATE TABLE vertices_annotations (
  "id" integer FOREIGN KEY REFERENCES vertices(id),
  "key" text,
  "value" text
);

CREATE TABLE edges_annotations (
  "id" integer FOREIGN KEY REFERENCES edges(id),
  "key" text,
  "value" text
);
```

Listing 5.9: SQL schema definition of the base graph.

a vertex or edge are packed into one row. We note that there are certain pros and cons with each choice, and we made the current choice due to the vertical representation's support for efficient *wildcard scans*, e.g. "find all vertices where *any annotation* matches pattern `'%ssh%'`", which are very often seen in our target workloads.

Nevertheless, QuickGrail decouples annotation tables from all GLL instructions except the two filter instructions *scan_vertices* and *scan_edges* (details about GLL instructions will be introduced in Section 5.3.3). Thus it suffices to just implement one more version of the two GLL instructions to support both vertical and horizontal representations of annotation tables, to tackle respective querying demands. We consider this support and related optimizations as future work.

Storage of the subgraphs

A QuickGrail subgraph is simply represented as two collections of ids referencing the base graph's *vertices* and *edges* tables. Listing 5.10 shows the detailed SQL schema definition.

There can be many subgraphs stored in Quickstep and each subgraph is associated with a unique name. The name is allocated automatically by QuickGrail (typically as auto-increment numbers) and in Listing 5.10 the example subgraph's name is "1234".

```
CREATE TABLE subgraph_vertices_1234 (
  "id" integer FOREIGN KEY REFERENCES vertices(id)
);

CREATE TABLE subgraph_edges_1234 (
  "id" bigint FOREIGN KEY REFERENCES edges(id)
);
```

Listing 5.10: SQL schema definition of a subgraph with name “1234”.

Auxiliary annotations

The auxiliary annotations for a subgraph are recorded with two tables, as Figure 5.11 shows. The two tables cannot be queried and are used only for attaching additional information to subgraph vertices and edges that control output formatting. That is, for example, the *color* and *shape* of each vertex in a subgraph that is exported in JSON format to be visualized by certain frontend tools.

```
CREATE TABLE aux_vertices_annotations_1234 (
  "id" integer FOREIGN KEY REFERENCES vertices(id),
  "key" text,
  "value" text
);

CREATE TABLE aux_edges_annotations_1234 (
  "id" integer FOREIGN KEY REFERENCES edges(id),
  "key" text,
  "value" text
);
```

Listing 5.11: SQL schema definition of auxiliary annotations for a subgraph.

Symbol table

The symbol table simply stores key-value string pairs. It is used to persist certain states in Quickstep so that the QuickGrail server becomes *stateless*. The states include all the mappings from *graph variables* to *subgraph names*, and an *auto-increment counter* for allocating subgraph names.

5.3.3 Grail Low-level Instructions

The core set of GLL instructions are listed in Table 5.12. In this subsection we will describe the detailed implementation of each instruction with pseudo-code algorithms that emit SQL queries.

Besides, we will also explain how the emitted SQL queries are evaluated in Quickstep and analyze the underlying performance. Note that a key point for understanding performance is that each GLL instruction only contains *sem-joins* but no regular *joins* at all.

Category	**GLL Instruction**	**Explanation**
Scan	*scan_vertices*(G_{in}, P_{sql}, G_{out}) *scan_edges*(G_{in}, P_{sql}, G_{out})	Filter subgraph G_{in} on its vertex or edge annotations with a SQL predicate P_{sql}, and get result subgraph as G_{out}.
Subgraph ops	*union*(G_{in1}, G_{in2}, G_{out}) *intersect*(G_{in1}, G_{in2}, G_{out}) *subtract*(G_{in1}, G_{in2}, G_{out})	Union / intersect / subtract subgraphs G_{in1} and G_{in2}, and get result subgraph as G_{out}.
Path	*find_paths*($G_{overlay}$, G_{src}, G_{dst}, d_{max}, G_{out}) *find_ancestors*($G_{overlay}$, G_{start}, d_{max}, G_{out}) *find_decendants*($G_{overlay}$, G_{start}, d_{max}, G_{out})	Find within the overlay subgraph $G_{overlay}$ all paths from vertices in subgraph G_{src} to vertices in subgraph G_{dst}, with path length no greater than d_{max}, and get the overall collection of vertices and edges on those paths as result subgraph G_{out}. Here *find_ancestors* and *find_decendants* can be viewed as special cases of *find_paths* where either G_{src} or $G_{dst}$is equivalent to $G_{overlay}$.

Table 5.12: The core set of Grail low-level instructions.

Scan operations on annotations

The implementation of the two scan operations *scan_vertices* and *scan_edges* are shown as Algorithm 2. Note that there are actually separate scripts for each operation. But here for conciseness we just put them in one place using a "switch" statement.

Now we elaborate a discussion on performance. Conceptually there are three stages in Quickstep for evaluating the SQL query in Algorithm 2.

1. Join the subgraph table with the base graph's annotation table to concretize the annotation key-value pairs.
2. Filter the key-value pairs using predicate P_{sql}.
3. Project and deduplicate on the *id* column and insert into the result subgraph table.

With the exact-filter (EF) technique as described in Section 4.2, we remark that Step 1 will be transformed into an EF probe operation where Quickstep builds a bit-vector from the subgraph table and uses it to filter the base graph's annotation table before evaluating predicate P_{sql} in Step 2. So that the semi-join operation is completely eliminated and the overall CPU cost of the first two

Algorithm 2: Scan operations

Input: *operation* – scan operation name
G_{in} – name of the input subgraph
P_{sql} – text representation of the filtering predicate
G_{out} – name of the output subgraph

switch *operation* **do**
  case "*scan_vertex*" **do**
    executeSQL(`INSERT INTO subgraph_vertices_`G_{out}
    `SELECT DISTINCT id FROM vertices_annotations`
    `WHERE` P_{sql} `AND id IN (SELECT id FROM subgraph_vertices_`G_{in}`);`)
  case "*scan_edge*" **do**
    executeSQL(`INSERT INTO subgraph_edges_`G_{out}
    `SELECT DISTINCT id FROM edges_annotations`
    `WHERE` P_{sql} `AND id IN (SELECT id FROM subgraph_edges_`G_{in}`);`)

steps is roughly

$$(C_{\textit{build-ef}} + C_{\textit{predicate}}) \cdot |G_{in}| + C_{\textit{probe-ef}} \cdot |A_{base}|$$

where $C_{\textit{predicate}}$ is the average cost of evaluating the filtering predicate P_{sql} on a single annotation row, $C_{\textit{build-ef}}$ is the average cost of inserting a value into a bit-vector, $C_{\textit{probe-ef}}$ is the average cost of probing a value in a bit-vector, $|G_{in}|$ is the total number of rows in the input subgraph's vertices (or edges) table, and $|A_{base}|$ is the total number of rows in the base graph's *vertices_annotation* (or *edges_annotation*) table. Meanwhile, the overall cost of Step 3 is roughly

$$(C_{\textit{deduplicate}} + C_{\textit{materialize}}) \cdot |G_{out}|$$

where $C_{\textit{deduplicate}}$ is average cost of obtaining a deduplicated integer *id* and $C_{\textit{materialize}}$ is the average cost of materializing an integer *id*.

In fact, the evaluation time of a scan operation is either dominated by the predicate term $C_{\textit{predicate}} \cdot |G_{in}|$ that is bounded by the size of the input subgraph, or dominated by the probe term $C_{\textit{probe}} \cdot |A_{base}|$ that is a constant, depending on how large the input subgraph is. In whatever situation, the cost factors $C_{\textit{predicate}}$ and $C_{\textit{probe-ef}}$ are sufficiently small so that every scan query can be finished within a bounded and predictable time, i.e. within a few seconds under typical workloads.

Subgraph operations

The implementations of the subgraph operations *union*, *intersection*, and *subtraction* are shown as Algorithm 3. Again for conciseness we put the three operations in one place using a "switch"

Algorithm 3: Subgraph union, intersection and subtraction

Input: *operation* – subgraph operation name
G_{in1} – name of the first input subgraph
G_{in2} – name of the second input subgraph
G_{out} – name of the output subgraph

foreach C_{type} in {"*vertices*", "*edges*"} **do**
  switch *operation* **do**
    case "*union*" **do**
      executeSQL(`INSERT INTO subgraph_`C_{type}`_`G_{out}
      `SELECT id FROM subgraph_`C_{type}`_`G_{in1}
      `UNION`
      `SELECT id FROM subgraph_`C_{type}`_`G_{in2}`;`)
    case "*intersection*" **do**
      executeSQL(`INSERT INTO subgraph_`C_{type}`_`G_{out}
      `SELECT id FROM subgraph_`C_{type}`_`G_{in1}
      `WHERE id IN (SELECT id FROM subgraph_`C_{type}`_`G_{in2}`);`)
    case "*subtraction*" **do**
      executeSQL(`INSERT INTO subgraph_`C_{type}`_`G_{out}
      `SELECT id FROM subgraph_`C_{type}`_`G_{in1}
      `WHERE id NOT IN (SELECT id FROM subgraph_`C_{type}`_`G_{in2}`);`)

statement.

It is easy to see that all the three operations have time complexities that are linear to the total size of the input graphs. More importantly, with the EF technique, we can eliminate the semi-joins (i.e. the *IN* clause) in the SQL queries, so that the evaluation time for *intersection / subtraction* operation is roughly

$$C_{probe\text{-}ef} \cdot |G_{in1}| + C_{build\text{-}ef} \cdot |G_{in2}|$$

which can be an order of magnitude faster than an ordinary hash semi-join's

$$C_{probe\text{-}ht} \cdot |G_{in1}| + C_{build\text{-}ht} \cdot |G_{in2}|$$

where $C_{build\text{-}ef}$ is the average cost of inserting a value into a bit-vector, $C_{probe\text{-}ef}$ is the average cost of probing a value in a bit-vector, $C_{build\text{-}ht}$ is the average cost of inserting a value into a hash table, and $C_{probe\text{-}ht}$ is the average cost of probing a value in a hash table.

Path queries

The evaluation process of a path query is divided into three stages, as Algorithm 4 and (continued) Algorithm 5 show.

The first stage creates and initializes four temporary tables *memo_edges*, *curr_vertices*, *next_vertices*, *reachable_vertices* for storing intermediate states.

The second stage is an iterative process that retrieves all reachable vertices backwardly starting from the destination vertices. The iterative process is essentially a breadth-first traversal on the overlay graph. All reachable vertices are stored into the *reachable_vertices* table and all edges that have been traversed are stored into the *memo_edges* table.

By doing an intersection between the *reachable_vertices* table from Stage 2 and the original source vertices, we now obtain a subset of source vertices $S_{reachable}$ that have at least one path to some of the destination vertices. Then the third stage traverses forwardly on the *memo_edges* table starting from $S_{reachable}$, and retrieves all vertices and edges that are forwardly reachable as final results. Moreover, a constraint on *depth* is applied in Algorithm 5 to omit paths that have lengths longer than the specified max depth.

It is easy to see that the overall evaluation time of a path query is bounded by

$$d_{max} \cdot (T_{stage2} + T_{stage3})$$

where T_{stage2} and T_{stage3} are the per-iteration evaluation time of the SQL queries in Stage 2 and Stage 3, respectively.

Here we borrow the discussion from the previous *scan operations* and *subgraph operations* parts, and just remark that the SQL queries in the iterations are optimized with the EF technique so that the per-iteration evaluation time is linear to the size of the base graph's edge table. Therefore, the overall evaluation time of a path query is bounded by

$$c \cdot d_{max} \cdot |E_{base}|$$

where c is some constant factor, d_{max} is the max depth, E_{base} is the base graph's edge table.

The key point about the time bound is that $c \cdot |E_{base}|$ is very robust under real workloads. For every depth d, a user can gain empirical experience about the maximum possible time cost that will grow linearly with d, regardless of the path query characteristics (e.g. input graph sizes and potential output graph size). In Section 5.4 we will see that this robustness is not the case for the graph database system Neo4j, where the same path query can finish within 1 second for depth 3, but cannot finish within 1 hour for depth 5.

Algorithm 4: Find all paths between two collections of vertices

Input: $G_{overlay}$ – name of the overlay subgraph
G_{src} – name of the subgraph of source vertices
G_{dst} – name of the subgraph of destination vertices
d_{max} – max path depth
G_{out} – name of the output subgraph

/ Stage 1. Initialize the intermediate tables. */*

executeSQL(
```
CREATE TABLE memo_edges(src INT, dst INT, depth INT);
CREATE TABLE curr_vertices(id INT);
CREATE TABLE next_vertices(id INT);
CREATE TABLE reachable_vertices(id INT);
INSERT INTO curr_vertices SELECT id FROM subgraph_vertex_G_dst;
```
)

/ Stage 2. Start from the destination vertices and do backward flooding. */*

$d_{actual} \leftarrow d_{max}$

for $D \leftarrow 1$ **to** d_{max} **do**

executeSQL(
```
INSERT INTO memo_edges
SELECT src, dst, D FROM edge
WHERE id IN (SELECT id FROM subgraph_edge_G_overlay)
  AND dst IN (SELECT id FROM curr_vertices);
TRUNCATE TABLE next_vertices;
INSERT INTO next_vertices
SELECT DISTINCT src FROM edge
WHERE id IN (SELECT id FROM subgraph_edge_G_overlay)
  AND dst IN (SELECT id FROM curr_vertices);
TRUNCATE TABLE curr_vertices;
INSERT INTO curr_vertices
SELECT id FROM next_vertices
WHERE id NOT IN (SELECT id FROM reachable_vertices);
INSERT INTO reachable_vertices SELECT id FROM curr_vertices;
```
)

$C \leftarrow$ executeSQL(`SELECT COUNT(*) FROM curr_vertices;`)

if $C = 0$ **then**

$d_{actual} \leftarrow D$

break

/ The code for Stage 3 (forward tracing) is continued in Algorithm 5 due to space constraint. */*

Algorithm 5: Find all paths between two collections of vertices (continued)

/ Stage 3. Start from all source vertices that are backward reachable from the destination vertices,*
** trace through the "memo_edges" table and accumulate the final result vertices and edges. */*

executeSQL(
```
TRUNCATE TABLE curr_vertices;
TRUNCATE TABLE next_vertices;
INSERT INTO curr_vertices
SELECT id FROM subgraph_vertices_G_src
WHERE id IN (SELECT id FROM reachable_vertices);
INSERT INTO subgraph_vertices_G_out SELECT id FROM curr_vertices;
```
)

for $D \leftarrow 1$ **to** d_{actual} **do**

executeSQL(
```
INSERT INTO next_vertices
SELECT DISTINCT dst FROM memo_edges
WHERE src IN (SELECT id FROM curr_vertices)
  AND depth + D - 1 <= d_actual;
INSERT INTO subgraph_edges_G_out
SELECT id FROM edge
WHERE src IN (SELEXT id FROM curr_vertices)
  AND dst IN (SELEXT id FROM next_vertices)
  AND id IN (SELECT id FROM subgraph_edges_G_overlay);
TRUNCATE TABLE curr_vertices;
INSERT INTO curr_vertices
SELECT id FROM next_vertices
WHERE id NOT IN (SELECT id FROM subgraph_vertices_G_out);
INSERT INTO subgraph_vertices_G_out SELECT id FROM curr_vertices;
```
)

$C \leftarrow$ executeSQL(`SELECT COUNT(*) FROM curr_vertices;`)

if $C = 0$ **then**

break

5.3.4 Generate GLL Instructions from Graph Expressions

In this subsection we briefly describe how graph expressions in the QuickGrail language gets compiled into GLL instructions. That is, for example, we can write a single composite expression as

```
$d = $a + ($b & $c).getVertex(pid = 100)
```

but the underlying evaluation would require multiple GLL instructions to achieve the corresponding semantics. In fact, assume that `$a`, `$b`, `$c` are mapped to subgraphs G_1, G_2, G_3 in the symbol table, respectively, then QuickGrail will translate the expression into a sequence of GLL instructions as

intersect(G_2, G_3, G_4)

scan_vertices(G_4, `"key"='pid' and "value"='100'`, G_5)

union(G_1, G_5, G_6)

set_symbol(`'$d'`, G_6)

Algorithm 6: Generate GLL instructions from QuickGrail expressions

Input: R – graph expression tree root

Output: result subgraph name G_{result}
and a sequence I of GLL instructions that generates G_{result}

$I \leftarrow []$

Function GraphExpressionCodeGen*(E)*

- **switch** E.type **do**
  - **case** "*union*" or "*intersect*" or "*subtract*" **do**
    - $G_{in1} \leftarrow$ *GraphExpressionCodeGen*(E.firstOperand)
    - $G_{in2} \leftarrow$ *GraphExpressionCodeGen*(E.secondOperand)
    - $G_{out} \leftarrow$ *allocateNewGraphName()*
    - $I \leftarrow I \cup \{$ *GLL instruction with name as E.type and arguments* ($G_{\text{in1}}, G_{\text{in2}}, G_{\text{out}}$) $\}$
    - **return** G_{out}
  - **case** "*getPath*" **do**
    - $G_{overlay} \leftarrow$ *GraphExpressionCodeGen*(E.overlay)
    - $G_{src} \leftarrow$ *GraphExpressionCodeGen*(E.firstOperand)
    - $G_{dst} \leftarrow$ *GraphExpressionCodeGen*(E.secondOperand)
    - $d_{max} \leftarrow E$.thirdOperand
    - $G_{out} \leftarrow$ *allocateNewGraphName()*
    - $I \leftarrow I \cup \{$ *find_paths*($G_{overlay}, G_{src}, G_{dst}, d_{max}, G_{out}$) $\}$
    - **return** G_{out}
  - **case** "*getVertex*" *or* "*getEdge*" **do**
    - $G_{in} \leftarrow$ *GraphExpressionCodeGen*(E.overlay)
    - **return** AnnotationPredicateCodeGen*(*E.type , G_{in}, E.firstOperand*)*
  - **case** "*variable*" **do**
    - **return** *lookupSymbolTable(*E.name*)*

/* Note: function "AnnotationPredicateCodeGen()" is shown in Algorithm 7 due to space constraint. */

$G_{result} \leftarrow$ *GraphExpressionCodeGen*(R)

return G_{result}, I

Algorithm 7: Generate GLL instructions from QuickGrail expressions (continued)

```
Function AnnotationPredicateCodeGen(E_type, G_in, P)
    switch P.type do
        case "and" do
            G_mid ← AnnotationPredicateCodeGen(E_type, G_in, P.firstOperand)
            return AnnotationPredicateCodeGen(E_type, G_mid, P.secondOperand)
        case "or" do
            G_in1 ← AnnotationPredicateCodeGen(E_type, G_in, P.firstOperand)
            G_in2 ← AnnotationPredicateCodeGen(E_type, G_in, P.secondOperand)
            G_out ← allocateNewGraphName()
            I ← I ∪ { union(G_in1, G_in2, G_out) }
            return G_out
        case "not" do
            G_mid ← AnnotationPredicateCodeGen(E_type, G_in, P.firstOperand)
            G_out ← allocateNewGraphName()
            I ← I ∪ { subtract(G_in, G_mid, G_out) }
            return G_out
        case "comparison" do
            K ← the annotation key name in comparison predicate P
            P ← replace all occurrences of K in P with string constant "value"
            G_out ← allocateNewGraphName()
            inst_name ← "scan_vertex" or "scan_edge" according to E_type
            I ← I ∪ { inst_name(G_in, "key = K and T", G_out) }
            return G_out
```

The translation process is analogous to the process of allocating (an unlimited amount of) virtual registers and generating SSA IRs in typical compilers. In QuickGrail, a graph expression is first parsed as an *expression tree*, where each tree node belongs to one type of QuickGrail operations (such as *union*, *intersect*, *getVertex*, *getPaths*), and children of a tree node are operands to the operation. Then, the execution generator translates the expression tree in a bottom-up manner, where the visit to a tree node returns a subgraph that stands for the result of evaluating the expression rooted at that tree node. Algorithm 6 and (continued) Algorithm 7 show details about this recursive translation process.

5.3.5 Graph Pattern Matching

In this subsection, we introduce how QuickGrail evaluates pattern matching queries. Specifically, QuickGrail translates each pattern matching query into a sequence of m GLL instructions, where m is linear to the number of atomic constraints in the query.

To ease the discussion while still capture the core ideas, we begin with a simplified form of pattern matching queries that are essentially *conjunctive queries*. Later we will explain how to generalize the techniques to support a rich collection of functionalities, such as disjunction, negation, and path depth constraints.

Problem formalization

Now we introduce some formalism by considering pattern matching queries represented in the conjunctive form

$$Q = A_1 \wedge \cdots \wedge A_n$$

where each *atomic term* A_i is either a *path formula* or a *domain constraint*.

Here, a path formula is a binary relation of the form

$$\mathit{Path}^{G_{\mathit{overlay}},G_{\mathit{src}},G_{\mathit{dst}}}(x, y)$$

such that for every pair of (x, y) there exists a path from x to y in the overlay graph G_{overlay}, and additionally x is also in graph G_{src} and y is also in graph G_{dst}.

Meanwhile, a domain constraint is a unary relation

$$D^G(x)$$

such that x is in D if and only if vertex x is in graph G.

We further define the (vertex-only) projection graph of query Q on variable x as

$$\mathcal{P}_x(Q) = \{x \mid x \in P(x) \textit{ where } P(x) \text{ :- } Q.\}$$

Then our goal is to efficiently evaluate

$$\mathcal{P}_x(Q), \;\; \textit{for each variable } x \textit{ that appears in query } Q$$

For example, let's consider a simple QuickGrail pattern matching query

```
with $G
match a ->* b ->* c
  and a in $A and b in $B and c in $C
as $R
set $Ra = {a}, $Rb = {b}, $Rc = {c};
```

Then the global constraint and the body of the query can be written into the conjunctive form

$$Q(a,b,c) \text{ :- } \mathit{Path}^{\$G,\$G,\$G}(a,b) \wedge \mathit{Path}^{\$G,\$G,\$G}(b,c) \wedge D^{\$A}(a) \wedge D^{\$B}(b) \wedge D^{\$C}(c) \qquad \text{(Query 5.1)}$$

and our objective is to find result vertices as *$Ra* $= \mathcal{P}_a(Q)$, *$Rb* $= \mathcal{P}_b(Q)$, and *$Rc* $= \mathcal{P}_c(Q)$. Note that the example QuickGrail query also requires finding an overall result graph *$R*. This can be generated "theoretically" by re-evaluating each path query on the result vertices and concatenate the path graphs together, and it is actually done more efficiently in QuickGrail as side-products during the process of obtaining result vertices.

Solve a conjunctive query

In this part we describe how QuickGrail evaluates $\mathcal{P}_x(Q)$ for each variable x in a query Q. In fact, in the case that Q is an *acyclic query*, the problem can essentially be solved by computing the *full reduction* of Q on each term where there exists a classical algorithm [9] that uses only semi-joins and the time cost is linear to the total input table cardinality times the number of atomic terms in Q. Our approach is a variant of the algorithm in [9], nevertheless the additional challenge is that

- The *path formula* is a virtual (a.k.a. IDB) relation whose cardinality can be as large as $O(|V|^2)$, where $|V|$ is the total number of vertices in input graphs. In many cases it is not computationally feasible to materialize such a path relation. Instead, we slightly modify Algorithm 4 to just efficiently compute $\mathcal{P}_x(\mathit{Path}^{\cdots}(x,y))$ and $\mathcal{P}_y(\mathit{Path}^{\cdots}(x,y))$. Thus, traditional semi-joins cannot be done between two path formulae.

Now we elaborate the details of our approach. Given a conjunctive query Q, we first build a *query graph* from Q by creating a node for each variable in Q. Then for each path formula $\mathit{Path}^{\cdots}(x,y)$ in Q we create an edge between nodes x and y. Finally, we attach each domain constraint $D(x)$ in Q as labels to the corresponding node x. Figure 5.13 shows an example graph that is built from Query 5.1.

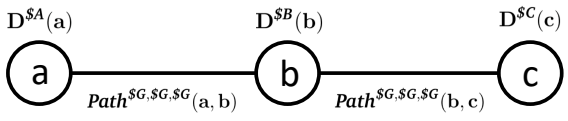


Figure 5.13: Query graph built from Query 5.1.

We define Query Q to be *acyclic* if its query graph is acyclic. In this part we only consider acyclic and fully connected query graphs, which are essentially *trees*. We will explain how to handle cycles and disconnected components later in this subsection.

Given an acyclic and connected query graph, we now choose a node (which node to choose leaves as an optimization task) from the graph to be the *root*, then reshape the graph into an ordinary tree so that each tree node is either a leaf node or an intermediate node that has a number of children. For example, by choosing b as the root we can reshape the graph in Figure 5.13 into a tree where b has two children a and c that are leaf nodes.

Then we do a *bottom-up* (post-order) traversal followed by a *top-down* (pre-order) traversal of the tree. During the bottom-up traversal, we compute for each node x the answers of $\mathcal{P}_x(Q_x)$, where Q_x is the subquery that is comprised of all the nodes and edges rooted at x. During the top-down traversal, we compute for each node x the final answers $\mathcal{P}_x(Q)$, where Q is the overall query. We describe details of each traversal below.

Define Q_x to be the subquery of all nodes and edges rooted at node x in the query tree, we have the recursive definition

$$Q_x = D^{G^x}(x) \wedge \left(\bigwedge_{y \in \mathit{children}(x)} E(x,y) \wedge Q_y(y) \right)$$

where $D^{G^x}(x)$ is the domain constraint for variable x saying that x should be also in graph G^x. Each $E(x,y)$ is the path formula represented by the edge between node x and child node y.

For any queries $Q_1(x), \ldots, Q_m(x)$, from basic set theory it is easy to see that

$$\mathcal{P}_x \left(\bigwedge_i Q_i(x) \right) = \bigcap_i \mathcal{P}_x \left(Q_i(x) \right)$$

Thus we have

$$\begin{aligned} \mathcal{P}_x(Q_x) &= \mathcal{P}_x \left(D^{G^x}(x) \right) \cap \left(\bigcap_{y \in \mathit{children}(x)} \mathcal{P}_x \left(E(x,y) \wedge Q_y(y) \right) \right) \\ &= G^x \cap \left(\bigcap_{y \in \mathit{children}(x)} \mathcal{P}_x \left(E_{y \in \mathcal{P}_y(Q_y)}(x,y) \right) \right) \end{aligned}$$

where $E_{y \in \mathcal{P}_y(Q_y)}(x,y)$ is the original path formula between node x and child node y, plus the additional constraint that y should also be in graph $\mathcal{P}_y(Q_y)$.

The key observation on the above formula is that we have completely eliminated any form of joins between path relations. Note that $E(x,y)$ has the form of either

$$\mathit{Path}^{G_{\mathit{overlay}}, G_{\mathit{src}}, G_{\mathit{dst}}}(x,y) \quad \mathit{or} \quad \mathit{Path}^{G_{\mathit{overlay}}, G_{\mathit{src}}, G_{\mathit{dst}}}(y,x)$$

Then $\mathcal{P}_x\left(E_{y\in\mathcal{P}_y(Q_y)}(y)\right)$ can be efficiently evaluated in QuickGrail as a basic path query that is either

$$\mathit{Path}^{G_{overlay},G_{src},G_{dst}\cap\mathcal{P}_y(Q_y)}(x,y) \quad \mathit{or} \quad \mathit{Path}^{G_{overlay},G_{src}\cap\mathcal{P}_y(Q_y),G_{dst}}(y,x)$$

depending on the actual path direction between x and y.

Therefore, in the first round of bottom-up traversal, we have computed for each variable x the answers $\mathcal{P}_x(Q_x)$, where all the computations can be encoded as GLL *intersection / find_paths* instructions, and the overall number of instructions generated are linear to number of nodes and edges in the query graph.

For the second round of top-down traversal that retrieves for each variable x the final answers $\mathcal{P}_x(Q)$, we just remark that the idea is similarly transforming joins into intersections and push vertex domain constraints into path queries, and omit the description and proof here.

Handle disconnected and/or cyclic conjunctive query graphs

The approach we described in previous subsection assumes that the conjunctive query graph is connected and acyclic. In this subsection we describe how to relax the two restrictions.

The handling of a disconnected query graph is simple. We first evaluate separately each connected component and obtain result vertices for each variable. Then, if any variable has empty results, then the results for all variables are all empty. Otherwise, since each variable must belong to only one connected component, we just collect all variables together and return the results for each of them.

The handling of a cyclic query graph is hard and in general we cannot guarantee to return exact results. We allow users to submit cyclic queries and still evaluate the queries with linear time complexity by making a compromise that the results may contain false positives (but must not omit true results). Our approach is first to let K be a parameter which has default value 2 and is configurable by the user in a pattern matching query. Then we decompose the query graph into components where each component is acyclic, and do K iterations of evaluations on each acyclic component. At the end of each iteration, we intersect the results from all components for each variable.

Support disjunction and negation

Disjunctions are simply handled by *unioning* the results from each disjunctive component. Note that after bringing in disjunction, the top-level conjunctive query may contain composite terms that involve more than two variables, where we actually represent the query graph as a *hypergraph* and we omit the details in this thesis. During the bottom-up and top-down evaluation, if an edge

in the query graph represents a composite term (e.g. a disjunction), then the child node results are pushed into each component of the disjunction, and then recursively evaluate each component and union the results back.

Negations must be supported with care since a naive evaluation may drop true results in our settings. In QuickGrail we support a negated clause only if the clause contains at most one free variable. To evaluate a negated clause for variable x, the clause itself without negation is first evaluated to obtain vertices G, then we do a subtraction between the global constraint graph and G as the results for variable x.

Support path depth constraints

To support path depth constraints, we simply extend a path formula to accept four parameters

$$Path^{G_{overlay},G_{src},G_{dst},d_{max}}(x, y)$$

where the additional d_{max} is the max depth parameter that is properly handled by our implementation of the GLL path instruction. Note that adding this parameter won't affect the nice property that a single path query is *monotonic*. Thus the techniques we described in this section naturally apply to the extended path formula.

Optimize evaluation order of conjunctive queries

Recall that when solving an acyclic conjunctive query we need to choose a node as root for the query tree. Different choices of the root node would result in different execution costs. On the other hand, though this problem looks similar as join order optimization, we note that the plan space is expected to be *robust* due to the semi-join nature of all path queries. Currently for each conjunctive query tree we simply chooses the center node so that the maximum distance from the central node to other nodes is minimized. We consider it a future work to further reason about the cost model and design algorithms to choose the best root nodes.

Eliminate redundant computations

There are two more interesting optimizations that are related to graph pattern matching but implemented within the GLL layer.

First recall that we generate GLL instructions that find results for all the free variables in a pattern matching query. However, a pattern matching query usually only specifies a subset of the free variables to be returned (or even no variable to return, in the case that the query returns the overall matched graph). In this situation, we can do a *live variable analysis* followed by a *dead code*

elimination on the GLL instructions and remove all instructions whose output graphs are not used later.

Besides, during graph pattern matching, many of the generated GLL *intersection* and *union* instructions are redundant. For example, the unoptimized instructions often intersect a graph G with the base graph, which is guaranteed to have the results as G itself. The generated instructions may also intersect graphs G_2 and G_3, where G_3 is previously obtained by intersecting G_1 and G_2 – in this case we can avoid the intersection between G_2 and G_3 and just return G_3. Therefore, QuickGrail does a dataflow analysis that infers the containment relationship among output graphs, and use this information to eliminate unnecessary *intersection* and *union* GLL instructions.

5.4 Experiments

Our datasets come from the DARPA Transparent Computing (TC) program. The datasets have been generated from multiple *engagements*. During each engagement, one or more servers were running with provenance collection middlewares so that system activities were reported as provenance graphs into a central database, and TC program's evaluation team simulated a few attacks to the servers, expecting that the details of the attacks can be recorded by the provenance system and either detected at real-time or revealed by offline analysis. Table 5.14 shows the detailed information of each dataset.

Datasets	Storage Size	# Vertices	# Edges
TC Engagement 2	15 GB	32925145	126763008
TC Engagement 3	79 GB	171756475	888148452
TC Engagement 4.tr1	4.0 GB	9480021	33256223
TC Engagement 4.tr2	5.0 GB	12087984	41391198
TC Engagement 4.het	3.1 GB	6717955	26201034

Table 5.14: Dataset statistics.

We note that TC is a big program where many teams collaborate. We refine our tasks to provenance graph storage and low-level offline analytics (instead of real-time event detection or high-level analyses where machine learning approaches may apply).

More specifically, for each engagement the evaluation team provides a "ground truth" document of attack activities plus its visualized representation (see Table 5.15 and Figure 5.16 as an example). The documents provide important information about the attack but are unfortunately often inaccurate. Note that the inaccuracy is not a deliberately placed challenge. Instead it is due

to differences in the way that the evaluation team views the attack process and the way that the reporter system organizes provenance information. Some connections are of reverse directions and some information may be even not recorded by the reporter system. Thus our TC-specific task is to verify whether each activity described in the ground truth document exists in the collected provenance graph data, and then extract all related connection patterns from the graph.

5.4.1 System Configuration

For the experiments, we use a server with two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors and a total of 160GB ECC memory. Each processor has 10 cores and 20 hyper-threading hardware threads. The machine runs Ubuntu 14.04.1 LTS. Each processor has a 25MB L3 cache, which is shared across all the cores on that processor. Each core has a 32KB L1 instruction cache, 32KB L1 data cache, and a 256KB L2 cache.

5.4.2 Analysis Workflow and Outcome

In this section, we demonstrate the usage of QuickGrail to analyze one attack scenario in Engagement 4. Note that there are multiple attack scenarios in each of the engagements, and we have used QuickGrail to successfully analyze all the attack scenarios in all the engagements. Nevertheless in this thesis it suffices to just go through one analysis in detail and the workflows are similar for analyzing other attacks.

Below Table 5.15 shows the "ground truth" activities provided by the TC program's evaluation team, and Figure 5.16 shows the corresponding pattern diagram.

Time	Session	Actor	Action	Object
09:51	ssh	128.55.12.167	connect (ssh)	[tr1] ssh
09:51	ssh	[tr1] ssh	call	[tr1] netstat
09:51	ssh	[tr1] ssh	write	[tr1] LD_PRELOAD
09:53	ssh	[tr1] LD_PRELOAD	update	[tr1] /lib/libselinux.so
09:54	ssh	[tr1] ssh	call	[tr1] fuser
09:54	ssh	[tr1] fuser	kill	[tr1] /home/admin/launchmyserver.sh
...	...	...	...	...

Table 5.15: Table of activities for one attack in Engagement 4.

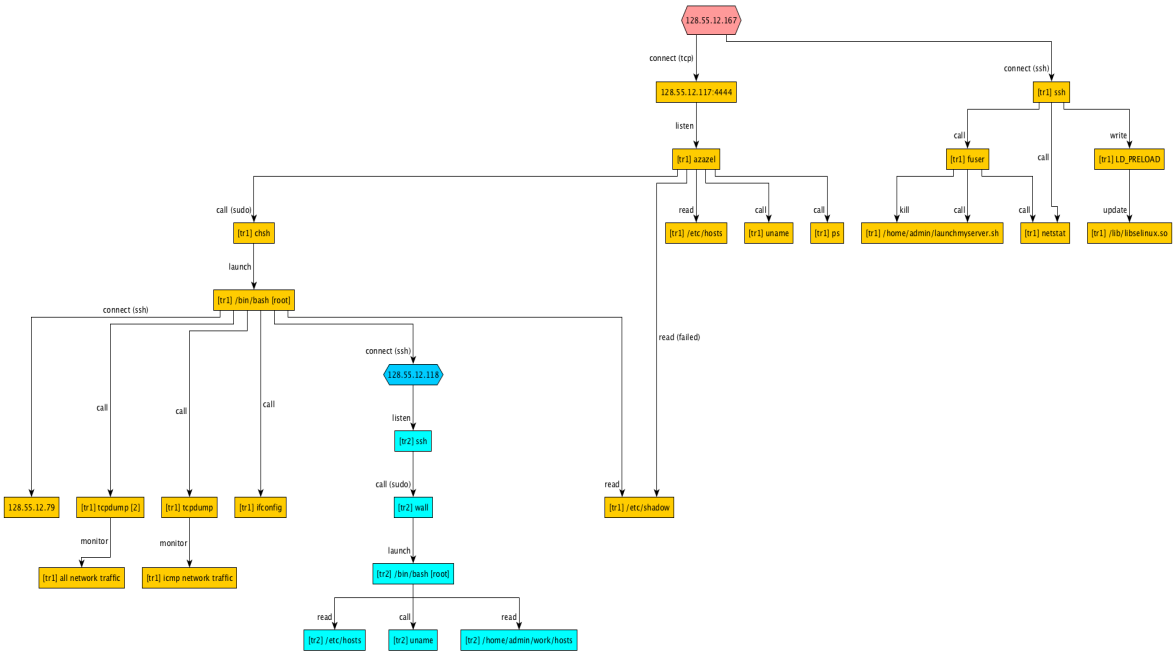


Figure 5.16: The overall diagram for the activities described in Table 5.15.

Now we aim to accomplish three tasks:

1. Verify whether each activity described by ground truth exists in the provenance graph data.
2. If any entity or activity is missing, try to find potential causes (e.g. the entity has a different name in the provenance data, the activity linkage is not as illustrated in ground truth).
3. Finally, extract and visualize all the found entities and activities as a subgraph from the provenance graph.

Table 5.17 shows part of the real workflow of our analysis on the data. It is an interactive and iterative process. We begin with vertex scans on known information, and try to find paths among different groups of vertices. Depending on the result size of a path query, we may want to increase or decrease the maximum path depth to see if we can obtain the desired results or observe something interesting in an *trail-and-error* manner. Meanwhile, we will also apply *divide and conquer* techniques that try to match just part of the overall pattern, and then combine different pieces together using graph union operations.

The final outcome of our analysis is shown as Figure 5.19. Note that some endpoints in the pattern diagram cannot be found in the actual data, so they are not in the result graph.

Queries	Explanation	|V|	|E|	Time (ms)
`$startIp = $base.getVertex(* = '128.55.12.167');`	Find all the vertices that have IP address 128.55.12.167	5	0	381

`$libselinux = $base.getVertex(path like '%libselinux%');`
`$lmserver = $base.getVertex(path like '%launchmyserver%');`
`$ssh = $base.getVertex(name = 'ssh');`
`$etcHosts = $base.getVertex(path = '/etc/hosts');`
...
	Similarly check the existence of all the vertices in Figure 5.16. Note that some queries may return empty results (e.g. "LD_PRELOAD" is not found in any vertex).	6
		1
		34
		1
		...
			0
			0
			0
			0
			...
				395
				386
				415
				367
				...

`$ip2lib = match $startIp ->{..10} $libselinux;`	Check if there are paths from $startIp vertices to $libselinux vertices. Use an initial max path depth of 10.	1075485	6286496	10015
`$ip2lib = match $startIp ->{..3} $libselinux;`	From the previous step we know that there are paths but results are too many. Now reduce the max depth and try again.	6	2320	3987
`visualize $ip2lib.collapseEdge('cdm.type');`	Looks like we have got an interesting result graph $ip2lib. Now visualize it and check the details. Also collapse the edges before visualization since there are too many of them.	6	16	429
`$somePath = match ...`	Similarly check whether paths exist among different pairs of end points, e.g. $startIp to $etcHosts, $startIp to $lmserver, etc. This is a repeated process and the analyzer may spend a few minutes here.			

`$relevantLib = $libselinux & $ip2lib;`
 	
`$libAnc1 = match m ->{..1} $relevantLib;`
`$libAnc2 = match m ->{..2} $relevantLib;`
`$libAnc5 = match m ->{..5} $relevantLib;`
	Since we haven't found the "LD_PRELOAD" keyword where the corresponding vertex should have a path to "/lib/libselinux.so". Now we want to explore what are the ancestor nodes of libselinux.	1
		
		28
		324
		2084408
			0
			
			63
			480
			2776055
				247
				
				511
				788
				2208

`$processes =`
`match a -> b`
`and (a."cdm.type" = 'SUBJECT_PROCESS'`
`or a."cdm.type" = 'SUBJECT_UNIT')`
`and (b."cdm.type" = 'SUBJECT_PROCESS'`
`or b."cdm.type" = 'SUBJECT_UNIT');`
	For some of the paths, from domain knowledge we know that it would be better to look at only "Process" nodes in the path. So we extract a subgraph from the base graph where the nodes all represent processes.	2846074	4347752	5786

Table 5.17: Analysis workflow for tr1 (yellow) part in Figure 5.16. Note that for each command, the $|V|$ and $|E|$ columns stand for the number of result vertices and result edges.

Queries	Explanation	|V|	|E|	Time (ms)
match $startIp −> startIpProc and startIpProc −[$processes]−>∗ p1 −> $libselinux and startIpProc −[$processes]−>∗ p2 −> $lmserver and startIpProc −[$processes]−>∗ p3 −> $etcHosts and startIpProc −[$processes]−>∗ p4 −> $etcShadow and chsh −[$processes]−>∗ startIpProc and $sshIp −> p5 −[$processes]−>∗ chsh and $tcpdump −[$processes]−>∗ chsh and $ifconfig −[$processes]−>∗ chsh and chsh.name = 'chsh' as $result;	After we have confirmed the pairwise connectivity among groups of vertices, we are now ready to put the constraints together. Due to space consideration we just show an all-in-one query that brings in all the constraints. But note that the actual workflow is still an trial-and-error process, where we accumulatively add more and more constraints into one pattern matching query.	41	110	61453

Table 5.18: Analysis workflow for tr1 (yellow) part in Figure 5.16 (continued).

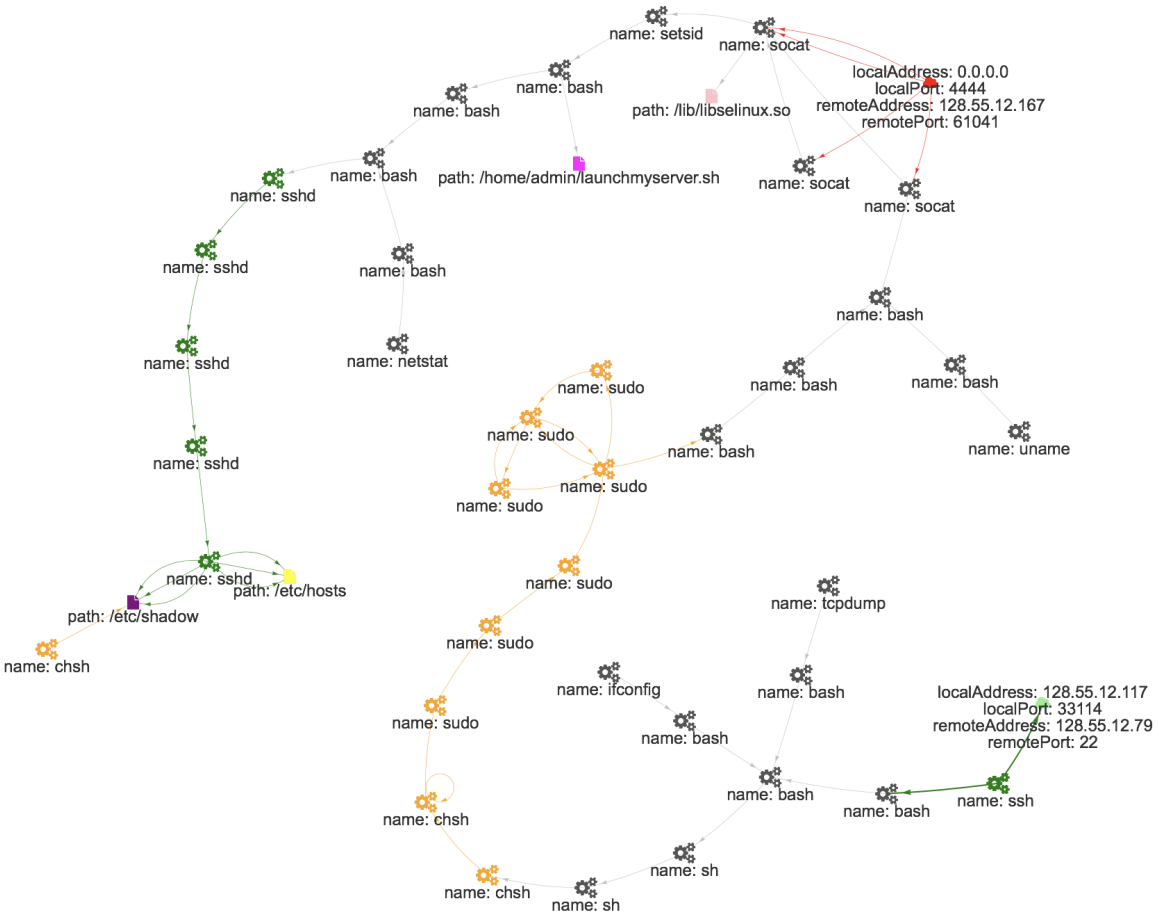


Figure 5.19: Analysis outcome for tr1 (yellow) part in Figure 5.16.

5.4.3 Comparing Performance With Neo4j

In this subsection we compare with Neo4j on the performance of scan, lineage, and path queries. For pattern matching queries, we note that Neo4j failed to finish the evaluation of most queries we used to analyze the TC datasets due to some reasons that would be shown and explained when we look at lineage/path queries below.

Scan operations

For scan operations, we just pick three predicates where each predicate is representative in its own kind. Table 5.20 shows the predicates and performance of both QuickGrail and Neo4j (no indexes are used). Note that the performance difference falls in the range of 16X∼22X, and the speedup comes from Quickstep's efficient and parallel execution of table scans.

Predicate	Kind	Neo4j Time (ms)	QuickGrail Time (ms)
Find all vertices where "remoteAddress" equals '128.55.12.167'	Equal comparison	8888	381
Find all vertices where "path" matches regular expression '.*libselinux.*'	String pattern matching	8848	395
Find all edges where timestamp is between 1542033097 and 1542036549	Range scan	45560	2822

Table 5.20: Performance comparisons on scan operations.

Lineage and path queries

Table 5.21 and Table 5.22 show performance comparisons between QuickGrail and Neo4j on example lineage/path queries, respectively. Note that lineage queries are just special cases of path queries, thus we would have similar observations about performance characteristics in both tables.

In Table 5.21, we show performance of both systems on a lineage query that starts from a single "libselinux.so" vertex and finds all descendants with regard to each specified max depth. The |V| and |E| columns in the table record the total number of reachable descendant vertices and the total number of edges along the way. We can observe that Neo4j is extremely fast for handling initial small depths up to 3. However, increasing the depth to 4 causes Neo4j to spend 802 seconds, and increasing the depth to 5 causes Neo4j to not finish within 1 hour. That is, adding depth by 1 has resulted in an $802134/259 = 3097$X slow down. This drastic degradation of performance is

Max depth	|V|	|E|	Neo4j Time (ms)	QuickGrail Time (ms)
1	28	63	5	625
2	324	480	15	791
3	3451	14642	259	1064
4	75764	153664	802134	1375
5	2084408	2776055	DNF	2210
10	6434609	20816463	DNF	5917
20	6547507	21058826	DNF	8688
50	6547507	21058826	DNF	8712

Table 5.21: Performance comparisons on a lineage query that starts from a “libselinux.so” vertex and finds all descendants up to various max depths. In this table *DNF* means that the query did not finish within 1 hour.

Max depth	|V|	|E|	Neo4j Time (ms)	QuickGrail Time (ms)
1	0	0	6	2499
2	0	0	16	4336
3	0	0	942	5016
4	139	560	270848	6129
5	324	994	DNF	6863
10	1251182	7142131	DNF	11772
20	1405968	7886718	DNF	16114
50	1406785	7889555	DNF	24767

Table 5.22: Performance comparisons on a query that finds all paths from a “ssh” node to any “firefox” node within various max depths. In this table *DNF* means that the query did not finish within 1 hour.

due to the typical path traversing algorithm that is implemented in these systems, which essentially enumerates all possible combinations of paths. Thus by adding depth 1, the time cost is *multiplied* by a factor F that can be as large as the total number of out-edges for all collected vertices. Thus it is not surprising that F can be as great as 3097. In contrast, QuickGrail shows a robust and predictable performance curve. It has fixed overheads even for small depths, but the total time cost keeps growing linearly even when the depth is very large. Note that Section 5.3.3 has explained these performance characteristics for QuickGrail in detail.

Table 5.22 shows a similar performance pattern for path queries and we note that the reasoning is the same as we discussed on lineage queries.

5.5 Conclusions and Future Work

In this chapter, we have introduced a system called *QuickGrail* to support efficient and effective querying on large provenance graphs. The system comes with a domain-specific querying language that fits into interactive exploration on provenance graphs. The functionalities include evaluation of composite filter / lineage / path / pattern matching queries, and flexible subgraph manipulations. The QuickGrail system is designed to be fast, robust and scalable to meet the demands of interactiveness and to improvement the productivity of human analysts.

Nowadays graph analytics is increasingly being used in enterprises. Looking at more sophisticated graph pattern matching, including automatic anomaly detection, and in a scalable way is a direction for future work.

Chapter 6

Summary

In this thesis, we have introduced techniques that facilitate interactive analyses on large volume of data under application scenarios of both traditional data warehousing and provenance graph analytics. We utilize a high-performance RDBMS, Quickstep, as a common computational platform to achieve the objectives.

For traditional data warehousing, we have introduced a novel query execution strategy called LIP for robust query processing. LIP collapses the space of left-deep query plans for star schema warehouses down to almost a single point near the optimal plan. In addition to this robustness benefit, it also significantly speeds up query execution in this important subplan space. We have demonstrated these claims through theoretical and empirical results. Besides the immediate application of LIP, we believe our work opens a novel approach to the notion of "robustness", one that is focused on query execution strategies possibly tailored to corresponding query plan (sub-)spaces.

In addition to LIP, we have proposed two additional query optimization techniques that eliminate redundant computation and materialization under a "drop early, drop fast" theme. The techniques include aggressive push-down of certain disjunctive predicates and cache-efficient semi-joins using exact filters. Our experiments show that the techniques significantly improve the performance of both SSB benchmark queries and TPC-H benchmark queries where applicable.

In future work, we hope to generalize the ideas about robustness and the specific LIP strategy to more complex schemas and query plan shapes. We will also explore how this new approach to robustness impacts query optimization. On the other hand, there is a never ending search for even more sophisticated query optimization techniques. Looking at optimizing queries with a very large number of joins and aggregation is a direction of future work. Examining if machine learning methods can help in query optimization or if methods like LIP can be extended even further is likely to be a interesting research direction.

For provenance graphs analytics, we have introduce a system called QuickGrail that supports

efficient and effective querying on large provenance graphs. The QuickGrail system comes together with an expressive domain-specific query language that allows a human analyst to evaluate complex filter / lineage / path queries and a class of pattern matching queries to yield possibly very large subgraphs as intermediate results, and do set operations such as union, intersection, subtraction on the subgraphs. The intermediate results can be efficiently concretized and be conveniently referenced as inputs for subsequent iterations of exploratory analyses. We have explained the underlying implementations that support all the QuickGrail operations with high performance, robustness and scalability, and demonstrated the claims with experiments. Our experiment datasets come from the DARPA Transparent Computing (TC) program, and we have used QuickGrail to successfully extract the desired patterns for all the attack scenarios in all the TC engagements.

Nowadays graph analytics is increasingly being used in enterprises. In future work, it is a direction for us to consider more sophisticated graph pattern matching, including automatic anomaly detection, in a scalable way.

Bibliography

[1] Neo4j. `https://neo4j.com`.

[2] OrientDB. `https://orientdb.com`.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00*, pages 261–272, New York, NY, USA, 2000. ACM.

[4] E. Babb. Implementing a relational database by means of specialzed hardware. *ACM Trans. Database Syst.*, 4(1):1–29, Mar. 1979.

[5] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 119–130, New York, NY, USA, 2005. ACM.

[6] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 107–118, New York, NY, USA, 2005. ACM.

[7] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD '04*, pages 407–418, New York, NY, USA, 2004. ACM.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, pages 269–284, 1987.

[9] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, Jan. 1981.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13:422–426, 1970.

[11] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *5th TPC Technology Conference, TPCTC*, pages 61–76, 2013.

[12] K. Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB '84*, pages 323–333, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[13] Apache Calcite. `https://calcite.apache.org`, 2016.

[14] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, 6(13):1474–1485, 2013.

[15] M.-S. Chen, H.-I. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *The VLDB Journal*, 6(2):121–131, May 1997.

[16] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 63–78, New York, NY, USA, 2015. ACM.

[17] A. Dutt and J. R. Haritsa. Plan bouquets: A fragrant approach to robust query processing. *ACM Trans. Database Syst.*, 41(2):11:1–11:37, May 2016.

[18] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[20] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[21] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[22] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[23] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[24] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 22(2):267–276, June 1993.

[25] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE '08*, pages 774–783, Washington, DC, USA, 2008. IEEE Computer Society.

[26] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. VERTEXICA: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[27] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[28] D. E. Knuth. The art of computer programming, volume 3: sorting and searching. *Addition Wesley*, 1973.

[29] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB*, 9(3):204–215, Nov. 2015.

[30] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB '86*, pages 149–159, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[32] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD '04*, pages 659–670, New York, NY, USA, 2004. ACM.

[33] Microsoft. Implied predicates and query hints. `https://blogs.msdn.microsoft.com/craigfr/2009/04/28/implied-predicates-and-query-hints/`, 2009.

[34] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[35] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6):743–756, June 2011.

[36] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, pages 627–640, 2009.

[37] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[38] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.pdf`, Jan 2007.

[39] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[40] Oracle. Push-down part 2. `https://blogs.oracle.com/in-memory/push-down:-part-2`, 2015.

[41] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *ACM/SPEC International Conference on Performance Engineering*, pages 361–372. ACM, 2013.

[42] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, pages 435–446, 1996.

[43] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, pages 1196–1207. IEEE, 2013.

[44] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: a modular query optimizer architecture for big data. In *SIGMOD*, pages 337–348, 2014.

[45] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[46] T. P. P. C. (TPC). Tpc benchmark h (decision support) standard specification revision 2.17.1. `http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v2.17.1.pdf`, Nov 2014.

[47] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.*, 9(1):133–161, Mar. 1984.

[48] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, 2014.

[49] S. Yin, A. Hameurlain, and F. Morvan. Robust query optimization methods with respect to estimation errors: A survey. *SIGMOD Rec.*, 44(3):25–36, Dec. 2015.

[50] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *PVLDB*, 8(3):197–208, 2014.