Feedback-Driven Query Optimization: Design and Infrastructure

Yuchen Liang

CMU-CS-25-150
December 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
Jignesh Patel

Submitted in partial fulfillment of the requirements for the Master's degree in Computer Science.

Keywords: database management systems, query optimization, query feedback, cardinality estimation

For my family and friends

Abstract

Query optimizers are critical components in database management systems (DBMSs) that turn a query that might otherwise take hours to run into one that completes in seconds. However, modern data stacks allow applications to generate data files (e.g., Parquet) outside the DBMS's purview that lack statistical summaries. Without information about data distribution, optimizers fall back to ungrounded guesses when computing cardinality and cost estimates needed to select the best plan from an exponential number of candidates. This thesis investigates the design of an adaptive query optimization strategy that leverages query feedback to improve planning for future queries. We first study how an optimizer should memoize query plans during optimization so that it can associate past traces with future queries. To facilitate this investigation, we built a Cascades-style transformational search engine with a memo table that organizes equivalent sub-plans and detects duplicates. We measured the amount of reuse when maintaining the optimization state across queries. Our results showed that future queries can reuse the cached optimization information from previous queries when they exhibit similar subplan patterns. We then analyze runtime feedback collection methods, focusing on integration strategies and the resulting artifacts. Our study reveals a tension between the query coverage provided by runtime artifacts and the collection overhead they entail. Although most DBMSs can provide runtime row-count profiles at the operator level, the optimizer can only use this information for future queries that match the exact predicates. To address this limitation, we present a method that measures the true selectivity of each prefix in the conjunctive predicate during execution, which the optimizer can then leverage to improve planning for a broader range of queries. We implemented this method in PostgreSQL and measured the runtime performance overhead. Our experiments show that fine-grained instrumentation incurs a constant overhead regardless of the number of conjuncts. We conclude the thesis by discussing the design choices and trade-offs of integrating an extensible query optimizer service with existing DBMSs.

Acknowledgments

First and foremost, I would like to thank my thesis committee members, Professor Andy Pavlo and Professor Jignesh Patel. Your guidance and mentorship has motivated me to become a better researcher and love databases even more.

In addition, I would like to thank my friends in the Carnegie Mellon Database Group for their general advice, feedback on research ideas, and collaboration on the project: Sam Arch, Sirui Huang, Chris Laspias, Wan Shen Lim, Ben Owad, Drew Ripberger, Alexis Schlomer, Sarvesh Tandon, Connor Tsui, Chi Zhang, Will Zhang. I would also like to thank my mom and my dad for their support and encouragement throughout my education. Of course, I could not do any of this without Ziyin Zhao.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A DBMS's query optimizer transforms a declarative query (e.g., SQL) that describes *what* result the user wants into an efficient physical execution plan that specifies *how* to retrieve the results. For a given query, the optimizer makes many decisions to minimize execution cost, such as choosing join orderings, algorithms (e.g., hash vs. merge joins), and grouping strategies. To find the best execution plan out of exponential choices, the optimizer relies on cardinality estimates and a cost model to guide its search process and compare the semantically equivalent alternatives [31].

Although researchers have studied how to build DBMS optimizers since the late 1970s [46], they remain one of the most challenging problems in systems research [13], and there are decades of examples of bad query plans. Much of this difficulty comes from the reliance on faulty cardinality and cost estimates. In most DBMSs, such estimates rely on (1) simplifying assumptions (e.g., uniformity, independence) and (2) approximations about data [31]. For the latter, these include statistical summaries (e.g., histograms [25], sketches [9, 16, 41]) and samples [8, 32] that estimate filter selectivity, join sizes, and the number of distinct values [10, 13]. Modern data stacks exacerbate this problem by allowing applications to generate data files (e.g., Parquet) outside the DBMS's purview. Thus, an optimizer may have plan queries that target this data with zero statistical summaries. As a result, the query optimizer falls back to using ungrounded guesses, hoping they are correct, without ever revisiting them. Such problems are why optimizers underestimate cardinalities and generate inefficient plans that are orders of magnitude slower than the optimal [33].

The fundamental challenge in traditional query optimization is that the optimizer cannot recover from bad initial estimates. Once an optimizer chooses a plan based on incorrect cardinality estimates, it executes that plan regardless of how inaccurate those estimates prove to be during execution. More critically, when the user later reruns the same or similar queries, the optimizer makes the same mistakes because it cannot learn from them. This is a missed opportunity for improvement in query optimization, since repeated queries are common in industry workloads. Previous work indicates that in 50% of database clusters in the Redshift fleet, 80% of queries are identical repetitions of queries that have been encountered before [52]. To avoid repeating mistakes, the optimizer must learn from past executions to refine the optimization process. Most adaptive optimization strategies studied by prior work are internal to a specific DBMS and many require highly specialized runtime features [3, 15, 49]. Despite the vast design space, there has been no systematic study of feedback-driven query optimization.

1.1 Thesis Contributions

This thesis investigates the design of an adaptive query optimization strategy that leverages query feedback to improve planning for future queries. In Chapter 2, we first describe the requirements of an extensible query optimizer and the core components that facilitate optimization. In Chapter 3, we study feedback-driven query optimization and key design choices in building an optimizer based on query feedback, which involves feedback collection, feedback exploitation, and state management. Then in Chapter 4, we evaluate the benefit provided by the query feedback and the overhead of our proposed runtime feedback collection method. We will discuss related work in Chapter 5 and conclude the thesis in Chapter 6.

Chapter 2

Extensible Query Optimizers

A query optimizer is composed of a search engine (or plan enumerator) to enumerate candidate plans, a cost model primarily based on cardinality estimates to compare the candidate plans, and a system catalog to provide schema and statistical information necessary for query optimization. Figure 2.1 demonstrates the architecture of an extensible query optimizer. An extensible query optimizer allows developers to adapt any of its components to handle new data types, operators, access methods, or optimization strategies without requiring fundamental changes to the optimizer's core architecture [19, 20]. In this chapter, we describe the core components of an extensible query optimizer as the foundation for building the query feedback infrastructure.

2.1 Plan Enumeration

The plan enumerator explores a sufficiently large search space and generates query plans that are semantically equivalent but apply operations in a different order. Two primary approaches of plan enumeration are transformational search and bottom-up enumeration.

In transformational search, the optimizer starts with an initial logical query plan and applies rewrite rules to generate alternative query plans. Transformation rules rewrite a logical expression into another equivalent logical expression. Since transformation rules are based on the equivalence of relational algebra and its extension, the plan before and after the transformation differs in the sequence of operations, but produces the same output. For example, a plan with only scan, filter, and join operators has the following equivalences:

$$\sigma_p(R \bowtie_q S) \equiv \sigma_p(R) \bowtie_q S \text{ if } \mathcal{F}(p) \subseteq \mathcal{A}(S) \qquad \text{(predicate pushdown)}$$

$$R \bowtie_p S \equiv S \bowtie_p R \qquad \text{(join commutativity)}$$

$$(R \bowtie_p S) \bowtie_q T \equiv R \bowtie_p (S \bowtie_q T) \text{ if } \mathcal{F}(p) \subseteq \mathcal{A}(R) \cup \mathcal{A}(S) \cap \mathcal{F}(q) \subseteq \mathcal{A}(S) \cup \mathcal{A}(T) \qquad \text{(join associativity)}$$

where $\mathcal{A}(R)$ is the set of columns (variables) available in the output of relation R, and $\mathcal{F}(e)$ is the set of free columns (variables) in a scalar expression e.

Implementation rules, on the other hand, specify how a logical operation should be executed using a specific algorithm or access method. For example, a hash join implementation rule can

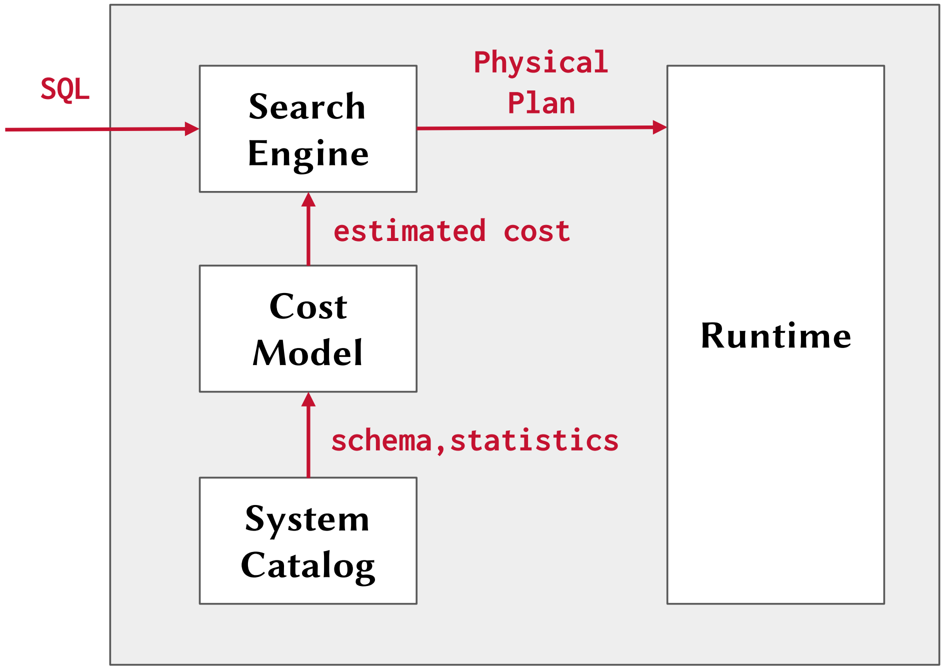


Figure 2.1: Architecture of an extensible query optimizer.

transform a logical join into a hash join operator, specifying that the execution engine will place tuples from the build side of the join into a hash table and check for matches against tuples from the probe side. To avoid duplicate rule application, the transformational search uses a memo table to track equivalent expressions and detect whether the optimizer has already seen them.

In bottom-up enumeration [38, 39, 46], the optimizer builds the query plan incrementally by combining smaller components into large expressions. To avoid duplicate work, bottom-up enumeration typically uses a DP table to keep track of solutions to smaller subproblems, which can be used to solve larger ones.

Both transformational search and bottom-up enumeration can find the best execution plan, provided the cardinality estimates and cost model are accurate.

2.2 Cost Model

The cost model estimates the resource consumption of individual physical operators in a query plan, then combines their costs to calculate the total plan cost [13]. Using these estimates, the optimizer compares all candidate plans generated by the plan enumerator and selects the one with the lowest execution cost.

The cost model generally represents an operator's execution time as a function of three factors: (1) the amount of data processed, (2) the cost per unit of data, and (3) runtime settings. The amount of data processed depends on the number of input rows (i.e., input cardinalities) and the size of each tuple. The cost per unit of data varies with the hardware configuration. For example, the unit cost of a sequential scan differs between a hard drive and a fast SSD. Runtime configuration factors, including buffer pool sizes and degree of parallelism, can further influence execution performance. Among these factors, the amount of data processed typically has the greatest impact on overall execution time. Since intermediate cardinalities remain unknown until execution, the optimizer must rely on estimates to approximate them. We will discuss them in the next section.

2.3 Cardinality Estimation

The cardinality estimator takes a query expression as input and estimates the number of output rows. As stated in the previous section, computing accurate cardinality estimates is crucial to obtaining accurate cost estimates, which directly influence decisions about join ordering, access path selection, and the choice of physical operators. Traditionally, a query optimizer derives cardinality estimates from (1) precomputed statistics and (2) simplifying assumptions to apply these statistical summaries to obtain estimates [7, 46].

For example, PostgreSQL estimates the cardinalities of the base table using a combination of histograms, most common values, and the number of distinct values [31, 33]. To reduce the collection overhead, these column statistics are computed on a sample of base relations. PostgreSQL assumes conjunctive predicates on a single table are independent and multiplies individual selectivities to obtain the overall selectivity of predicates on a single table. To estimate the cardinality of a join, PostgreSQL uses the following formula:

$$|R \bowtie_{x=y} S| = \frac{|R| \cdot |S|}{\max(\mathrm{dom}(x), \mathrm{dom}(y))} \tag{2.1}$$

where R and S are arbitrary relational expressions and $\mathrm{dom}(x)$ is the number of distinct values of column x. The cardinality computation formulas in PostgreSQL makes the following assumptions:

1. **Uniformity.** All attribute values except for the most common values are uniformly distributed.
2. **Independence.** Predicates on different attributes are independent (i.e., no correlation).
3. **Containment.** The set of distinct values in the attribute with fewer distinct values is fully contained within the set of distinct values in the attribute with more distinct values when joining two attributes.

However, these assumptions often fail in real-world datasets, particularly when correlated attributes, skewed distributions, or complex predicates involving user-defined functions are present [30, 31, 33]. When cardinality estimates deviate significantly from actual runtime cardinalities, the optimizer may select suboptimal plans, resulting in long-running queries that are orders of magnitude slower than optimal.

2.4 System Catalog

The system catalog in a DBMS stores schema metadata, such as table and column information, as well as internal bookkeeping data. The query planner uses information from the system catalog to perform rewrites and make cost-based decisions. For example, the query planner consults the system catalog to determine which indexes are available and considers them when selecting base table access methods. In addition, knowing the PK-FK relationship allows the cardinality estimator to produce a more accurate estimate of the number of distinct values, since, by definition, the values in the primary key column must be unique, and the PK-FK join satisfies the principle of containment.

In addition, the system catalog maintains statistical summaries on columns in the base table. As discussed in the previous section, the optimizer relies on these statistical summaries to estimate the cardinality of intermediate results. Statistical summaries become stale when a significant portion of the data changes. To obtain up-to-date information, traditional DBMSes allow users to refresh statistical summaries via a manual command (e.g., `ANALYZE`) or as an automated background job.

Chapter 3

Feedback-Driven Query Optimization

As noted in Chapter 2, the traditional query optimization paradigm optimizes a query from scratch only based on the static information it has, without learning from previous query instances. Feedback-driven query optimization uses information observed at execution time to repair incorrect statistics and cardinality estimates of query plans for future queries. In this chapter, we conduct an extensive study of the design choices in the key components of an optimization framework built on the query feedback infrastructure: (1) feedback collection, (2) feedback exploitation, and (3) state management.

3.1 Feedback Collection

Feedback collection is the first component in a feedback-driven query optimizer. The design of this component involves considering runtime integration, the artifacts produced, and the scope of collection. We will discuss each of these areas in this section. For the rest of the section, we consider the following criteria when evaluating a collection method:

1. **Flexibility:** The ability of the collection method to capture diverse types of feedback information.
2. **Runtime overhead:** The additional computational cost that feedback collection imposes on query execution.
3. **Coverage:** How broadly collected feedback applies across different queries beyond those from which it was initially collected.
4. **Accuracy:** To what degree the collected feedback reflects the ground truth.
5. **Engineering overhead:** The implementation and maintenance complexity required to integrate the collection method.

3.1.1 Runtime integration

Although query feedback is a feature for query optimization, the optimizer relies on the execution engine to gather runtime information. We categorize the integration methods as offline exploration and online instrumentation.

Offline Exploration

Offline exploration treats feedback collection as a separate process from regular query execution. In this approach, the optimizer acts as a client to the execution engine and runs separate, dedicated SQL queries to gather statistics and runtime information. This approach offers maximum flexibility by leveraging the capabilities of the runtime systems to compute any expressions supported by SQL or DBMS extensions.

Offline exploration also requires minimal engineering overhead. First, the decoupled architecture allows separation of concerns, enabling engineers to develop the optimizer independently without understanding or modifying the execution engine's internals. In addition, this approach has a well-defined contract with the execution engine. By using only the stable, backward-compatible standard query interface, the optimizer ensures that it does not depend on any execution engine internal features subject to removal in a future version upgrade.

Despite excellent flexibility and low engineering overhead, offline exploration incurs high runtime costs because the optimizer must run additional queries to gather feedback about the dataset. The optimizer also needs to ensure that executing these exploratory queries does not cause regressions in actively running customer queries. For DBMS with separation of storage and compute, the optimizer can run exploration queries on a separate compute node to eliminate its impact on customer queries.

Online Instrumentation

The online instrumentation approach integrates feedback collection into the execution of user queries. Rather than running separate queries, this approach instruments the query execution engine as queries naturally execute. Compared to offline exploration, online instrumentation applies only to queries similar to those they have seen before.

The engineering effort required to integrate online instrumentation into runtime systems depends on the system's implementation. Compilation-based systems can directly lower the feedback logic to native code, and for simple operations, it is even possible to use hardware counters to achieve even better performance. For interpretation-based systems, several integration strategies are available. The first strategy is to model the feedback collection procedure as a per-tuple callback. The idea here is to do extra work inside the callback beyond the computation needed to obtain the correct results. If the execution engine already has an instrumentation infrastructure, it is possible to integrate more advanced collection methods directly with it. For example, PostgreSQL's `EXPLAIN ANALYZE` utility already captures operator-level row counts, the actual time spent within an operator, and the buffer usage for each operator.

The second strategy is to model the feedback-collection procedure as a user-defined operator or function. Essentially, the operation will forward the function input to the output while performing some instrumentation and recording the observation. Compared to the first approach, the UDF-based instrumentation imposes greater engineering overhead, as engineers must understand how to manipulate query expressions in the targeted database systems. Further, implementing UDF-based feedback naively on the SQL layer could prevent optimization opportunities. For example, the execution engine might not push UDF predicates down to the base table, thereby missing the chance to prune data blocks. To properly implement UDF-based feedback collection,

	Runtime overhead	Engineering overhead	Information Gained	Accuracy
Operator-level row count	Lowest	None	Low	Exact
Fine-grained selectivity feedback	Low	Low	Medium	Exact
Statistical Summaries	High	Medium	High	Close to Exact
Statistical Summaries (sampled)	Medium	High	High	Less

Table 3.1: Comparison of produced artifacts

the query optimizer will run an additional pass that adds these collection operators/functions to the best plan produced by previous passes. Thus, although the callback method requires implementing feedback as special logic, it requires the least modification of existing systems and therefore has a lower engineering complexity.

3.1.2 Produced Artifacts

With runtime support, the optimizer can now gather observations or artifacts to improve future optimization. Artifacts differ in the amount of information they provide and the runtime overhead required to obtain it. Table 3.1 presents a overview of a non-comprehensive list of artifacts that the optimizer could find helpful.

Operator-level row count

Operator-level row count is the simplest form of feedback artifact and serves as the foundation for improving cardinality estimate via query feedback. This approach instruments query operators to track the number of tuples that flow through them during execution, providing the ground-truth cardinality estimation with minimal implementation complexity and negligible runtime overhead. The instrumentation procedure only needs to increment counters at operator boundaries. However, operator-level row counts have an inherent limitation when dealing with complex predicates. When a filter operator contains multiple predicates combined in a conjunctive clause, the observed row count reflects only the joint selectivity of applying all filters together. Thus, although operator-level row counts can indicate that a cardinality estimate was incorrect, they cannot pinpoint which specific predicates caused the misestimation. Due to coarse-grained coverage, the optimizer can leverage this feedback information only when encountering the same predicate.

Fine-grained Selectivity Feedback

To address the limitation of coarse-grained operator-level row count, the optimizer can instrument the queries to track the number of tuples that satisfy each predicate in the conjunctive clause. This fine-grained approach provides more insight into data correlation on a per-predicate level. However, obtaining such detailed feedback requires careful consideration of the semantics of runtime evaluation. In particular, most data systems employ short-circuit evaluation for conjunctive predicates, where the evaluation stops as soon as any condition evaluates to false. Thus, the optimizer can only obtain the number of tuples that satisfy each prefix of the conjunctive clause. For example, this approach would obtain not only the final cardinality after applying a filter with condition $A \wedge B \wedge C$ but also the intermediate cardinalities after applying predicate A and $A \wedge B$. Reordering the conjunctive predicates or turning off the short-circuit evaluation increases the coverage of feedback, but at the cost of increased computational overhead. Integrating this approach with DBMSes that adaptively reorder predicate evaluation at runtime (e.g., DuckDB) also requires care. On the implementation side, this approach requires the instrumentation infrastructure to keep more counters and increment them after evaluating each term in the conjunctive clause.

Advanced Statistical Summaries

Beyond simple cardinality tracking, the query instrumentation system can construct more sophisticated statistical summaries that provide richer insights into data distributions. For example, building HyperLogLog [16] sketches on intermediate results can estimate the number of distinct values (also called set cardinality) with only a few kilobytes of space. With more accurate distinct-value estimates for columns involved in the join keys and the group-by clause, the optimizer can significantly improve cardinalities for these operators in future queries. Other advanced summaries include Count-Min sketches [9] for frequency estimation, t-digests [14] for quantile approximation, SpaceSaving sketches [36] for capturing the most frequent elements, and multidimensional histograms [25] to detect correlations between columns. These structures enable the optimizer to learn skewed data distributions and cross-relation correlations that traditional statistical analysis on base relations cannot capture. However, building advanced statistical summaries incurs higher computational costs during query execution and consumes more memory. Building these summaries from a sample of data could reduce this overhead, at the expense of slightly lower accuracy.

3.1.3 Scope of Collection

Due to runtime overhead, the optimizer may restrict to collect feedback only on a subset of queries and choose where to collect them. To reduce overhead, the optimizer could collect information only for columns that appear in selection predicates, join keys, and group-by columns. In this case, the instrumentation system only accesses the columns relevant to estimating cardinalities for each tuple. Another strategy is to collect feedback from the pipeline breaker. Here, the instrumentation system exploits the fact that the system must materialize tuples anyway, thereby reducing overhead. However, the query plan may have a limited number of pipeline breakers depending on its structure. For example, there are no pipeline breakers in a left-deep plan after building all hash tables for the join operations. Additionally, the optimizer could use previous feedback to narrow down the collection points, collecting computationally expensive artifacts only where significant estimation errors occur. Lastly, for latency-sensitive queries, the optimizer can turn off feedback so the DBMS can satisfy the customer's requirement.

3.1.4 Fine-grained Selectivity Feedback in PostgreSQL

To understand the requirement and challenges of integrating feedback collection into an existing DBMS, we add support for the fine-grained selectivity feedback in PostgreSQL [42]. This section discusses the implementation of the collection method.

Background: PostgreSQL Expression Evaluation and Instrumentation

During the executor initialization of a particular physical plan, PostgreSQL compiles expressions tagged to the relational operators (e.g., computed columns, predicates) into a small virtual machine code. If just-in-time (JIT) compilation is enabled, PostgreSQL will then compile the expressions into machine code using LLVM [44].

The unmodified PostgreSQL supports operator-level instrumentation and has counters for the number of rows removed by a filter (i.e., the `qual` clause). These counters are then made available to the users using the `EXPLAIN ANALYZE` command [43]. However, PostgreSQL does not track the conditional selectivity of individual terms in the conjunctive filter clause. For example, the instrumentation system only tracks the joint selectivity $\mathrm{Sel}(\sigma(R.x) \wedge \sigma(R.y))$, but not $\mathrm{Sel}(\sigma(R.y) \mid \sigma(R.x) = \mathrm{True})$.

Implementation

To obtain fine-grained selectivity estimates, we made minimal changes to PostgreSQL while implementing the core instrumentation logic as a PostgreSQL extension. First, we expose an additional plugin point (`ExecInitQual_hook`) to support customization of the predicate evaluation initialization logic. During executor initialization, PostgreSQL calls this hook, and we are able to instrument the virtual machine code. Second, we extend the instrumentation data structure with per-term counters to store fine-grained observations during execution.

The standard PostgreSQL compilation procedure adds an `EEOP_QUAL` instruction for each term in a conjunctive predicate, which specifies a jump to `EEOP_DONE_RETURN` for early termination if the result is false or null. Our modified logic instruments the evaluation path by inserting an `EEOP_CONST` instruction that store the 1-based index of each predicate term before for each `EEOP_DONE_RETURN` instruction. We then redirect the `EEOP_QUAL` instructions to jump to these index-storing constants rather than directly to the terminating instruction. If a tuple satisfies all predicates, the return value is set to 0. Figure 3.1 demonstrates the pseudocode for compiling and instrumenting an example query.

To collect feedback with minimal overhead and preserve correct boolean semantics, we wrap PostgreSQL's generated evaluation function with a custom `pg_feedback_evalfunc`. This wrapper intercepts the integer return value from the instrumented expression: it returns true if the value is 0 (all predicates passed), and false otherwise. When a predicate fails (non-zero return), the wrapper increments the appropriate per-term counter, recording which predicate in the conjunction caused the tuple to be filtered. This design preserves compatibility with PostgreSQL's JIT compilation path—the wrapper operates on the compiled evaluation function regardless of whether it was JIT-compiled or interpreted.

After done with execution, we expose the instrumentation results through the `EXPLAIN ANALYZE` framework by implementing the `explain_per_node_hook` [40]. We display the

```
SELECT * FROM orders WHERE total_amount < 200 AND status in ('Cancelled');
```

```
current_value = …
if (current_return_value = false) {
  set current_return_value = 1
  jump to the return instruction
}
current_value = …
if (current_return_value = false) {
  set current_return_value = 2
  jump to the return instruction
}
…
// Passed all predicates
set current_return_value = 0
DONE_RETURN
```

```
bool wrapper_func(expr, eval_func) {
   index = instrumented_eval_func(expr)
   Instr_counters[index-1]++;
   return index != 0
}
```

Figure 3.1: Pseudo-code showing fine-grained selectivity instrumentation in PostgreSQL.

```
feedback_db=# set pg_feedback.enabled = true;
SET
feedback_db=# EXPLAIN ANALYZE SELECT * FROM orders WHERE total_amount < 200 AND status in ('Cancelled');
                                                QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Seq Scan on orders  (cost=0.00..56.00 rows=153 width=69) (actual time=0.028..1.049 rows=137.00 loops=1)
   Filter: ((total_amount < '200'::numeric) AND ((status)::text = 'Cancelled'::text))
   Rows Removed by Filter: 1863
   Buffers: shared hit=26
   Rows Removed at each Qual Step:
     (total_amount < '200'::numeric): 1235
     ((status)::text = 'Cancelled'::text): 628
 Planning:
   Buffers: shared hit=74 dirtied=1
 Planning Time: 1.795 ms
 Execution Time: 1.138 ms
(11 rows)
```

Figure 3.2: An example of running `EXPLAIN ANALYZE` on a query with fine-grained selectivity feedback.

number of rows filtered at each step, providing the conditional selectivity at each position in the conjunction. Figure 3.2 presents an example of running `EXPLAIN ANALYZE` on a query with fine-grained selectivity feedback. The optimizer can then consume the output in JSON format.

3.2 Feedback Exploitation

After receiving feedback from the runtime, the optimizer needs to decide how to use it. This section discusses ways for the optimizer to integrate feedback into future optimization. Table 3.2 provides a comparison between plan-level exploitation and statistics-level exploitation.

	Plan-Level Exploitation	Statistics-Level Exploitation
Coverage	Low, only applies to matching sub-plans	High
Accuracy	High	Low, relies on estimation

Table 3.2: Comparison of plan-level and statistics-level exploitation

Plan-level exploitation directly injects cardinalities or selectivities observed during actual query execution. The strength of this method lies in its precision: since the feedback comes from executions, there is no estimation error for the cached components. However, the input can only be applied when future queries contain the same sub-plans or sub-expressions, making it challenging to generalize insights to new or modified workloads.

On the other hand, statistics-level exploitation uses runtime feedback to improve the query optimizer's underlying statistics and data structures. For example, the optimizer can choose better histogram configurations based on observed data skews, adjusting bin boundaries or bucket sizes to better capture the actual data distribution. Although cardinality estimates derived from these statistics are less accurate, applying corrective feedback at the statistics level allows more queries in the workload to benefit from it.

3.2.1 Discussion: Resolving observation with estimation

Using estimated and observed selectivity together during optimization is a challenging problem and may not necessarily lead to a better plan. For example, if the optimizer severely underestimates plans that have not been executed before, these plans might be more attractive to the optimizer than plans costed with the observed cardinality. Further, as we observe feedback from the runtime, there may be multiple opportunities that we can take advantage of. For example, if we have selectivity feedback for $a = 3 \wedge b > 4$ and $b > 4 \wedge c < 2$ and can obtain selectivity estimates in the individual columns a, b, and c, then to estimate the selectivity of $a = 3 \wedge b > 4 \wedge c < 2$, we have to decide whether we want to compute the estimate as actual_sel$(a = 3 \wedge b > 4) \cdot$ est_sel$(c < 2)$ or est_sel$(a = 3) \cdot$ actual_sel$(b > 4 \wedge c < 2)$. With this observation, we must combine statistics in a probabilistically consistent way. The Maximum Entropy approach by Markl et. al [35] is a potential solution to this problem.

3.3 State Management

To use execution-time observations to improve future queries, query optimizers need memory to retain information from previous executions and associate this feedback with entities in the query plan (operators, scalar expressions, and columns). Given a new query plan, we also need to detect if part of it is similar to some executed sub-expression to exploit the feedback. This section discusses design decisions for managing the query optimizer's state across queries.

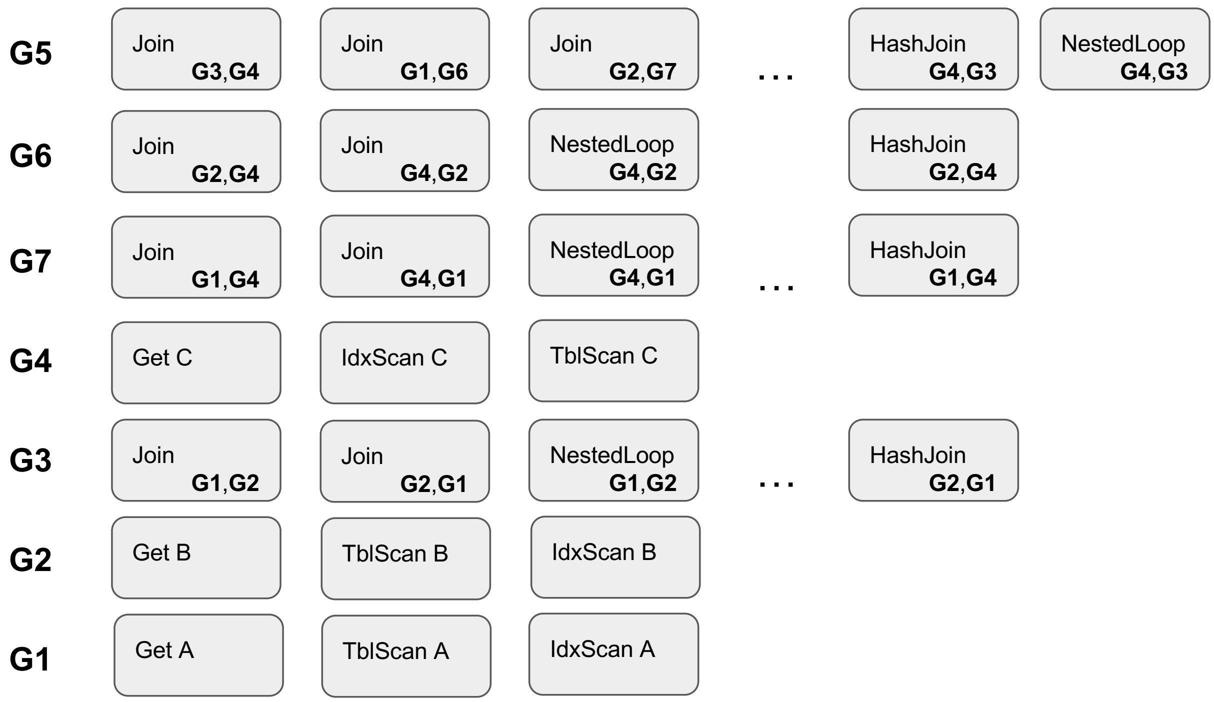


Figure 3.3: The internal structure of a Cascades memo table.

3.3.1 Detecting Reuse Opportunities

Here, we outline two approaches for detecting reuse opportunities from past optimizations.

Tracking Plan Structure (Syntactic)

In this approach, the optimizer stores the plan subexpressions and assigns unique ids to all used relational operators and scalar expressions. An example of plan-structure-based tracking is the memo table in the Cascades optimizer [18]. Figure 3.3 shows the internal structure of a Cascades memo table that detects duplicates and maintains a hierarchy of plan expressions. The optimizer can associate the collected feedback with the memo group id of a sub-plan since all plan expressions within a group are logically equivalent. One downside of this approach is that the stored feedback only applies to plans that are equivalent to the executed sub-plans.

Tracking constraints (Semantic)

This approach tracks constraints rather than the plan structure. The core idea is that the optimizer will record the observation based on the constraints imposed by the sub-expressions (value domain ranges, joined columns). Consider the two queries in Listing 1.

We observe that the result of $Q2$ is a subset of the result of $Q1$. The feedback for $Q1$ can serve as upper bounds for $Q2$, since the tuples passing through operators in $Q2$ are a subset of those in $Q1$. We would not reuse bounds that are too loose, but this approach could tighten estimates beyond what histogram buckets and pre-computed statistics capture.

```sql
-- Q1
SELECT * FROM t1, t2
WHERE t1.id = t2.id AND t1.a < 3 AND t2.b > 2008

-- Q2
SELECT * FROM t1, t2
WHERE t1.id = t2.id AND t1.a < 2 AND t2.b > 2009
```

Listing 1: Two queries Q_1 and Q_2 with the results of Q_2 containing the results of Q_1.

The implementation of this approach is similar to view matching algorithms [17], except that we are caching statistics instead of tuples. The challenge of this approach is scalability: since we have feedback for all queries, the number of ”views” we need to match could be very high. This means that for practical use, we have to optimize the lookup time using some specialized data structures and potentially bin the observations.

3.3.2 Base table columns vs. Arbitrary computed expressions

As with feedback collection, we also need to decide what to track in our optimization state. The base table columns are pre-defined in the catalog schema, making them natural candidates for tracking. On the other hand, tracking feedback on arbitrary computed expressions helps gather observations on expressions that traditionally lack statistics, such as user-defined functions (UDFs). In practice, tracking computed expressions on demand may work best. Under this hybrid model, the optimizer tracks feedback on base table columns by default to keep the state manageable, while allowing state to be tracked for specified computed expressions when needed.

3.3.3 Invalidation

When the state of the DBMS changes, the optimizer must invalidate previously cached states so it does not use stale observations. Query optimization state invalidation can be either logical or physical. Logical updates modify the database schema and metadata using Data Definition Language (DDL). Adding new columns does not affect existing feedback, but dropping columns removes all feedback involving the dropped column. Physical updates modify the actual data content using Data Manipulation Language (DML). `INSERT` and `DELETE` statements invalidate feedback for all columns in the table, whereas `UPDATE` statements invalidate feedback only for the columns they modify. Ideally, the optimizer should track DML statements directly to determine when to invalidate statistics and provide feedback, such as based on the proportion of modified rows. Without this capability, the optimizer can trigger invalidation when new statistics arrive, since database management systems typically regenerate statistics only after substantial data modifications.

Chapter 4

Evaluation

We now evaluate the benefit of reusing the optimization state and the overhead of our proposed feedback collection method. We report additional results in Appedix A.

4.1 Savings on Optimization Time

In this section, we evaluate the savings on the optimization time by maintaining the query optimization state across queries. Our evaluation employs a star schema, where a central fact table is connected to multiple dimension tables. Queries against a star schema typically join the fact table with multiple dimension tables, making them well-suited for evaluating optimizer performance as the number of joins increases. To observe the maximum benefit of reusing optimization state, we design a synthetic microbenchmark that measures the time it takes the optimizer to optimize queries as the number of dimension tables in the schema increases.

Figure 4.1 shows the time it takes to optimize star schema join queries with an increasing number of dimension tables, ranging from 1 to 13. The y-axis displays the optimization time on a logarithmic scale. As shown in the figure, the optimization time increases exponentially as the join graph includes more dimension tables. Although the optimizer maintains the optimization time within the millisecond range for queries with fewer than 10 joins, the optimization overhead quickly becomes prohibitive, escalating to over a minute for queries with 13 dimension tables. Maintaining optimization state across queries allows the query optimizer to reduce the optimization time to milliseconds, since it can directly return the cached best plan. This shows that maintaining the optimization state could improve end-to-end response time for previously-seen query fragments, whereas the traditional approach must restart optimization from scratch each time.

4.2 Reuse Opportunities Among Queries within a Workload

Although we observe that reusing the optimization state improves query response time on the synthetic microbenchmark, we are interested in how optimization state reuse can reduce computation when optimizing queries for real-world applications. Specifically, we want to investigate

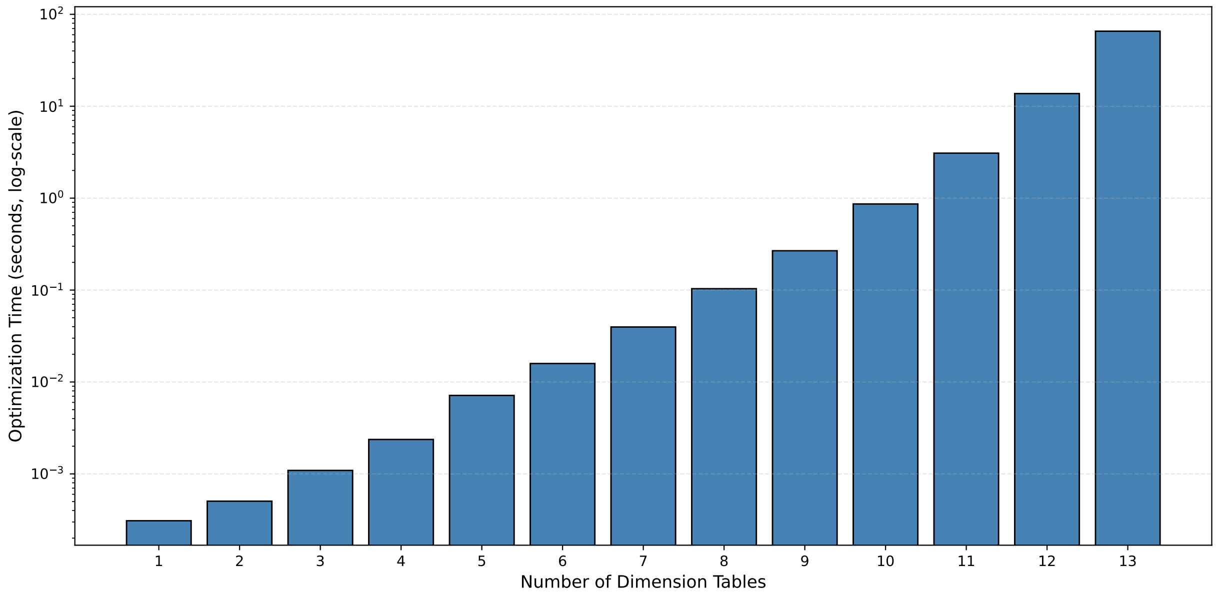


Figure 4.1: Optimization time for star schema join queries with a different number of dimension tables. The time is displayed in log-scale.

the reuse opportunities among queries in a single workload. For this experiment, we use JOB-*light*, a workload derived from the Join Order Benchmark (JOB) [31] that contains 70 of the original 113 queries. Both JOB-*light* and the original JOB workload is based on the IMDB dataset.

We use the number of shared memo groups across queries within JOB-*light* as the metric to measure the reuse opportunity. To understand the characteristics of this workload, We first report the number of memo groups and operators considered when optimizing each query separately. Figure 4.2 shows the number of memo groups in each JOB-*light* query, with the y-axis showing the group count and the x-axis labeled with query ids. We also show the number of operators considered in each query in Figure 4.3. As expected, we see that the number of memo groups and operators is directly related to the number of joins specified in the original SQL query, since the workload contains only SPJ queries.

To attribute memo group reuse to specific queries, we measure, for each pair of queries, the percentage of memo groups from the first query reused by the second.

Figure 4.4 presents a heatmap visualization of the memo group reuse ratio among 70 queries, arranged in a symmetric 70×70 matrix. The y-axis denotes the first query id, whereas the x-axis represents the second query id, with each cell colored according to the reuse ratio between the corresponding query pairs. The color scale ranges from dark purple (0.0, indicating no reuse) to bright yellow (1.0, indicating complete reuse). The bright yellow diagonal line from the upper-left heatmap to the bottom-right corner of the heatmap validates that our optimizer can detect the situation when we are optimizing the exact same query. The bright yellow cells that are off the diagonal represent queries with highly overlapping computation paths.

In addition to measuring the exact memo group reuse ratio, we also collect the templated memo group reuse ratio by replacing all constants in the queries with placeholders. As in the

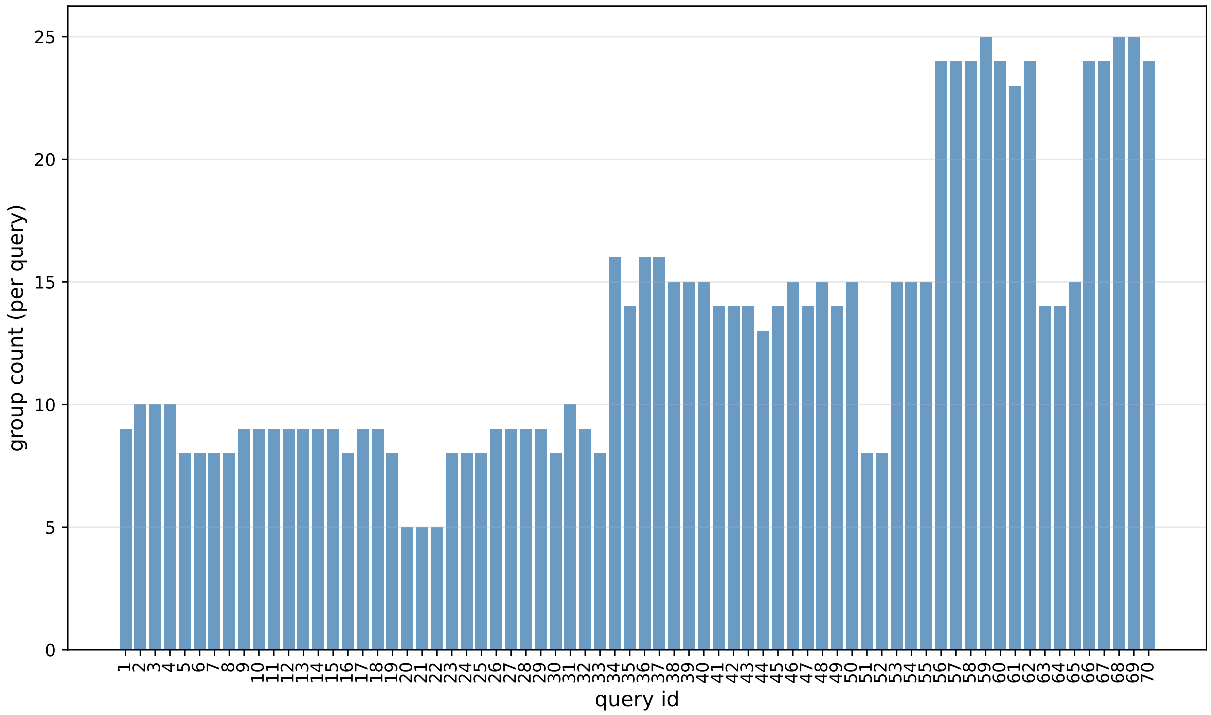


Figure 4.2: Number of memo groups in each JOB-*light* query.

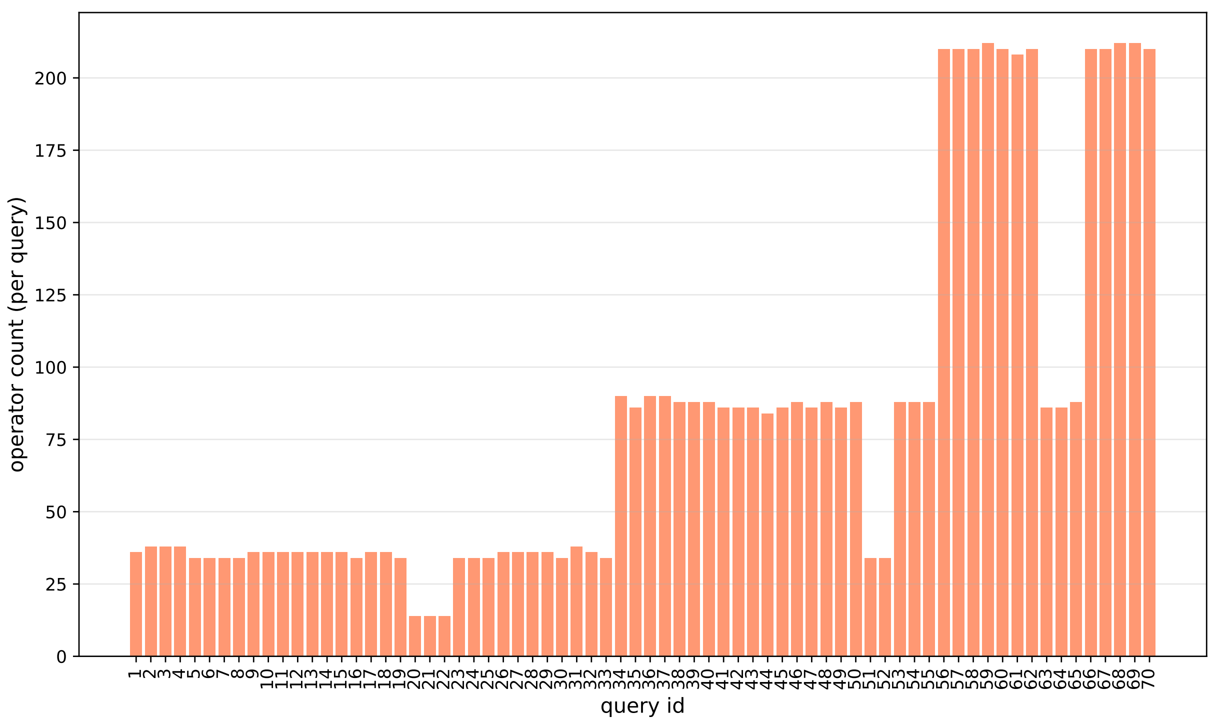


Figure 4.3: Number of operators considered in each JOB-*light* query.

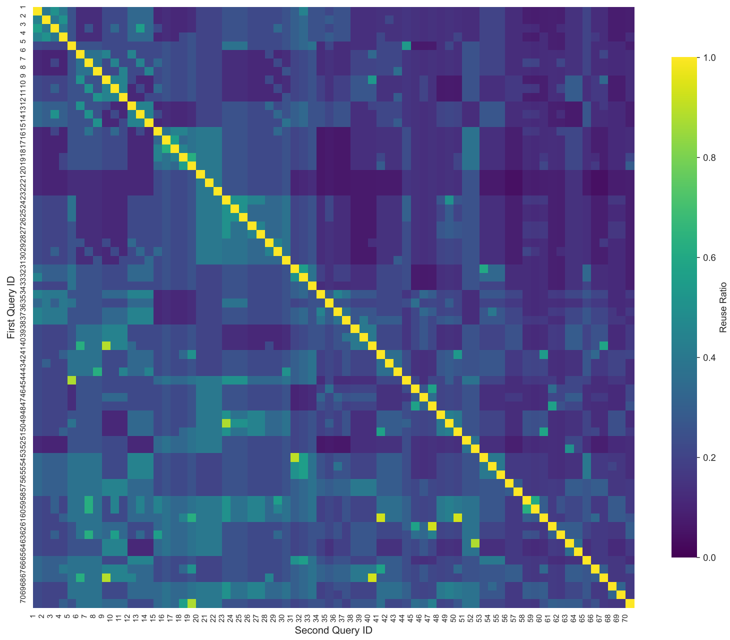


Figure 4.4: Memo group *exact* reuse ratio between each pair of JOB-*light* queries.

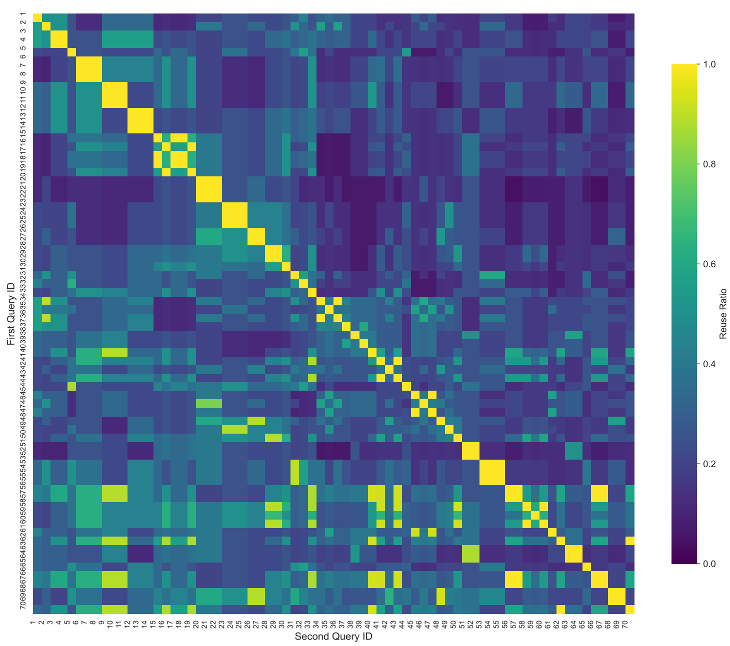


Figure 4.5: Memo group *templated* reuse ratio between each pair of JOB-*light* queries.

previous experiment, we also present the results as a heatmap in Figure 4.5. Compared to *exact* reuse, we see more bright yellow cells and light green cells on the heatmap, which indicates a higher reuse ratio. This effect is expected, as the queries share a lot of base-table predicates in the *templated* reuse experiment.

To understand the factor that decides the reuse ratio, we investigated the queries that have

```sql
CREATE TABLE R (col1 INTEGER, col2 INTEGER, ..., col20 INTEGER);

EXPLAIN (ANALYZE, FORMAT JSON)
SELECT * FROM R
WHERE col1 > 0 AND col2 > 0 AND ... AND col20 > 0;
```

Listing 2: The microbenchmark used to evaluate overhead of fine-grained selectivity collection.

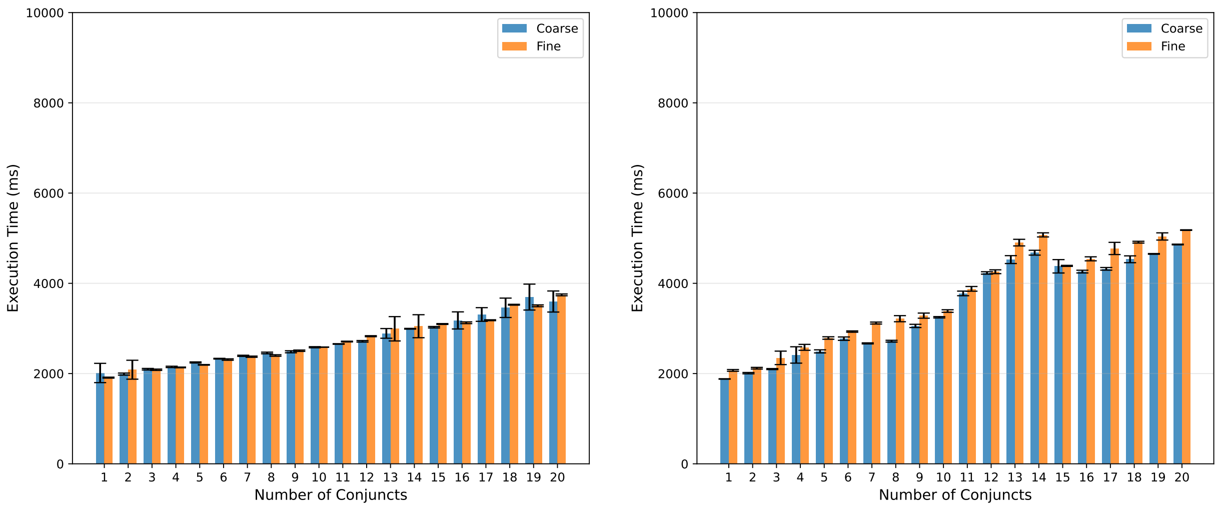
the highest and the lowest reuse ratios. $Q50$ is one of the queries with the highest reuse ratio of 0.93. Although $Q48$ and $Q49$ share the same join graph structure, the highest reuse ratio comes from $Q60$, which have exact mach in table predicates. The queries with lowest reuse ratio all have base table predicates that do not show up in any other queries. This tells us that having matching base table predicates is a key deciding factor whether past optimization results can be reused when managing optimization state using a Cascades-style memo table.

4.3 Overhead of Fine-grained Selectivity Feedback

In this section, we evaluate the runtime overhead of our fine-grained selectivity feedback collection compared to the coarse-grained selectivity feedback collection method on PostgreSQL 18 [45].

Listing 2 shows the schema and query we used for the evaluation. Table R contains 50 million rows and 20 columns, with all positive integers. The query involves a table scan with 1 to 20 conjuncts in the predicate, with all tuples qualifying the condition greater than zero in any column. We run each query with 3 cold runs and 5 hot runs, with R prewarmed before execution. We run all queries with `EXPLAIN ANALYZE` and report the performance of fine-grained vs. coarse-grained feedback with JIT compilation enabled and disabled.

Figure 4.6 reports the execution time in milliseconds for both approaches with (Figure 4.6a) and without (Figure 4.6b) JIT compilation. The error bars are based on the standard deviation of the 5 hot runs. As expected, the performance with JIT compilation is generally superior to the performance of using interpreted predicate evaluation for both fine-grained and coarse-grained selectivity feedback. We see that regardless of the predicate evaluation method, collecting fine-grained selectivity feedback incurs only a constant overhead compared to collecting coarse-grained feedback at the operator level. The result demonstrates fine-grained selectivity feedback as a practical instrumentation strategy.



(a) JIT

(b) Interpretation

Figure 4.6: Execution time of fine-grained vs. coarse-grained feedback on the microbenchmark in Listing 2.

Chapter 5

Related Work

This chapter discusses existing research on query optimization, with a particular emphasis on query optimizer extensibility, learned cardinality estimators, adaptive optimization strategies, and query feedback systems.

5.1 Query Optimizer Extensibility

Database researchers recognized the need for query optimizers to make it easy for system developers to add new functionality [19, 21]. Thus, extensible query optimizers enable such customization through pluggable components: new operators and expressions, transformation rules, cardinality estimators, and cost models. This led to the development of query optimizer generators, frameworks that enable a DBMS implementer to add new operators and new rules for optimizing queries without worrying about the internal optimization strategy.

Volcano [20] is a rule-based optimizer framework. It proposed core concepts, such as the memo table and physical properties, that form the foundation of top-down transformational search. The Volcano optimizer first enumerates a large number of equivalent plans using rewrite rules during the generation phase, then assigns each plan a cost to determine the best candidate. When a rule produces a new equivalent plan, it places the plan into a memo table to track equivalences and avoid redundant computations.

Cascades [18] inherits the core framework of top-down transformational search from Volcano and, in addition, addresses some of its limitations. Cascades improves search efficiency by eliminating the separation between generation and cost analysis, interleaving the costing of enumerated plans with the search for equivalent plans. It decomposes the search process into a series of tasks, each performing a specific action on a memo group or memo expression. Although Cascades uses a stack to track task dependencies, the task-based search algorithm makes it straightforward to parallelize the search by scheduling independent tasks onto different threads. The Orca optimizer [47], built on the Cascades framework, implemented a parallel search. Similar to Orca, our system supports search parallelization through green-thread scheduling without maintaining an explicit stack.

Volcano and Cascades have seen wide industry adoption; notable DBMSs using these architectures include SQL Server [5], Snowflake [11], Databricks [2], CockroachDB [51], and

TiDB [24]. Among all the implementations, the Apache Calcite [4] optimizer has the largest adoption in the open-source community. Unlike any of the other systems, the implementation of Calcite is not tied to any data systems. Instead, it supports optimizing queries that may span multiple query execution engines through the concept of *calling conventions*, which represent the query processing system in which the query fragment will be executed. Our optimizer shares the same goal of extensibility across data systems. Unlike Calcite's static optimization approach, our work incorporates runtime query feedback to adapt optimization strategies.

5.2 Learned Cardinality Estimation

Traditional cardinality estimation methods based on static statistical summaries and simplifying assumptions produce large estimation errors on complex queries. This limitation has motivated research into learning-based cardinality estimators to replace traditional approaches. These techniques use machine learning models to capture complex data distributions and correlations that simple statistical summaries cannot identify. The research community divides estimators into query-driven and data-driven. Query-driven approaches such as MSCN [28, 29] and E2E [50] train models on a set of labeled training queries and their corresponding cardinalities and then use the model to predict the cardinalities of online queries. These query-driven learned estimators are lightweight and have impressive accuracy when the online queries are similar to those in the training set. However, when the testing queries are out of the distribution, query-driven methods lose their robustness.

Data-driven approaches such as DeepDB [23] and NeuroCard [54] train models solely on data to learn the joint distributions of the dataset and then approximate the cardinality based on density estimation. Compared to query-driven methods, data-driven models offer better generalization to unseen queries, since they model the underlying data distribution rather than specific query patterns. Nevertheless, the inference cost of invoking data-driven learned estimators is also more expensive.

Most of the learned estimators use offline exploration as the runtime integration strategy. To fully integrate these methods into an end-to-end optimizer service, learned estimators require substantial training time and computational resources. Our feedback-driven approach does not require a training phase and instead adopts an online learning approach and collects feedback incrementally. The other drawback of learned estimators is that they cannot handle complex query constructs beyond PK-FK joins and simple predicates. Among the five estimators studied by Kim et. al, only 1 of them supports disjunction, 2 of them supports cyclic joins, and only one of them supports distinct-value estimation [27]. Compared to learned estimators, our approach does not pose restrictions on more complex query constructs since we directly track the feedback based on the plan structure. We are interested in integrating learned estimators that support online learning in our future work.

5.3 Adaptive Query Optimization

Adaptive query optimization techniques modify query plans based on information observed at runtime, addressing the fundamental limitation that static optimization relies on potentially inaccurate estimates.

5.3.1 Intra-Query Adaptation

Intra-query approaches adapt query execution while a query is running. Kabra and DeWitt [26] introduced mid-query re-optimization, where the optimizer monitors cardinality estimates during execution and triggers re-planning when significant deviations occur. Xue et al. [53] present a production framework that collects runtime statistics at shuffle boundaries to dynamically re-optimize query plans through join filter injection, join algorithm re-selection, and elastic parallelism adjustment, addressing lakehouse challenges of missing statistics at planning time . Eddies [3] take a more radical approach by continuously routing tuples through different operators based on runtime observations, effectively treating join ordering as an online scheduling problem. Progressive optimization (POP) [34] addresses cardinality estimation errors through runtime monitoring and adaptive re-optimization. POP inserts checkpoint operators into the initial query plan to validate the cardinality estimates against the actual observations during execution. The system triggers re-optimization when detecting significant discrepancies, exploiting both observed cardinalities and partial results already computed.

Although intra-query techniques recover from poor initial plans, they require significant changes to the execution engine to support features specific to adaptive processing techniques. Our work focuses instead on inter-query adaptation, which avoids disrupting individual query execution and making assumptions about the execution engine internals.

5.3.2 Inter-Query Adaptation

Inter-query adaptive optimization improves future queries by learning from past executions. LEO (Learning Optimizer) [49] pioneered this approach in IBM DB2. LEO tracks the true cardinalities during query execution and stores this information in the system catalog. When optimizing subsequent queries, LEO consults the stored feedback to refine cardinality estimates. However, LEO's query feedback has limited coverage on modern queries as it only tracks simple equality and range predicates.

Microsoft SQL Server implements Cardinality Estimation Feedback [37], which adjusts the cardinality estimator's correlation assumptions among the predicates to be fully-correlated, partially-correlated, or full independent, based on the feedback. The system persists these adjustments in the query store and applies them to subsequent executions of the same query template. Plan Stitch [12] extends inter-query adaptation by detecting common sub-plans across different queries and use runtime operator-level statistics to estimate the cardinality of stitched plans. Plan Stitch maintains a library of executed plan fragments with their observed costs and cardinalities, stitching these fragments together when optimizing new queries. Our work on feedback-driven query optimization builds on the inter-query adaptation approach but provides a more systematic investigation of the design space.

TONIC [22] is an optimizer extension designed to improve physical operator selection by leveraging historical query executions. It employs a learning-based approach and consults a case base containing previously executed join orders, each with a summary of the costs for the available physical operators. *TONIC* uses a *Query Execution Plan Synopsis (QEP-S)* to organize historical execution plans in a trie. Once the optimizer determines the join order for a query, *TONIC* identifies the most similar case by searching for the longest prefix match in the trie, and overrides the operator selection of the common operators with the cheapest execution cost. *TONIC* applies query feedback to improve operator selection, which is complementary to our approach to improve cardinality estimates for future queries.

5.4 Query Feedback Systems

The most closely related work to our approach involves systems that explicitly collect and exploit query execution feedback to improve future optimization.

5.4.1 Feedback Collection Approaches

Most DBMSes support operator-level instrumentation. For example, the `EXPLAIN ANALYZE` utility in PostgreSQL tracks the actual row counts, execution time, and buffer usage in each physical operator. This information helps database administrators identify the bottleneck but provides limited utility to future optimization due to its coarse granularity. Our implementation of fine-grained selectivity feedback in PostgreSQL (Section 3.1.4) fills this gap. We demonstrate how to instrument PostgreSQL's executor to track per-predicate selectivities with minimal code changes and quantify the runtime overhead.

5.4.2 Feedback Exploitation Strategies

Previous work differs in how they integrate runtime feedback into subsequent optimizations. LEO [49] and Plan Stitch [12] employ plan-level exploitation, directly caches observed cardinalities, and reuses them when identical sub-plans appear in future queries. Other works, such as self-tuning histograms [1, 6, 48] employ exploitation at the statistics-level and improves the underlying statistical summaries based on runtime observations.

A fundamental challenge, discussed by Markl et al. [35], is combining estimated and observed selectivities in a probabilistically consistent manner. The Maximum Entropy principle provides a theoretically grounded approach to this problem and determines the probability distribution that satisfies all observed constraints.

5.4.3 State Management and Reuse Detection

For feedback systems to benefit future queries, they must store observations in a way that enables efficient lookup and reuse. Cascade-style optimizers [18] provide a natural framework for this through their memo tables, which detect duplicate sub-expressions and organize equivalent plans into groups. Associating feedback with memo groups enables reuse across queries that share a similar structure.

An alternative approach tracks semantic constraints rather than syntactic structure. Goldstein et al. [17] developed techniques for matching query expressions based on the constraints they impose (value ranges, join conditions) to determine whether a materialized view is valid to substitute a sub-plan. Semantic-based feedback tracking employs the same matching algorithm, but instead determines whether the query feedback can apply to a sub-plan.

Chapter 6

Conclusion and Future Work

6.1 Future Work

We identify several directions for future research in feedback-driven query optimization. First, we could design a better memoization strategy to reuse feedback information from query fragments that are not identical. We could start by investigating the materialized view literature and drawing on ideas from view-matching algorithms. We also think it is interesting to test feedback collection methods on OLAP systems (e.g., DuckDB) and see whether the same conclusion holds in the presence of vectorized execution and adaptive filter reordering. In addition, we are interested in developing instrumentation methods that could capture cross-relation correlations. These correlations are harder to detect compared to correlation within the same table due to the physical storage organization of the schema. Lastly, with the feedback-driven query optimization infrastructure, we would like to evaluate the improvement in plan quality that feedback enables. Developing strategies for resolving logs of observations is necessary for using feedback to optimizepagebr queries under time constraint.

6.2 Conclusion

This thesis presents a comprehensive study on the design choices for building a feedback-driven query optimizer. We evaluated feedback collection methods against five criteria: flexibility, runtime overhead, coverage, accuracy, and engineering overhead. We also discussed the design decisions involved in exploiting feedback and managing the optimization state across queries. Our experiments demonstrate the benefit of maintaining optimization state across queries and the viability of integrating more advanced query feedback integration with existing DBMSes with relatively low engineering complexity. Overall, we hope that our work motivates the redesign of the query optimizer component as a stateful service that enables feedback-driven optimization to improve query performance.

Appendix A

Additional Results

In this chapter, we report additional results gathered for experiments in Chapter 4.

A.1 Reuse Opportunities

In addition to the pairwise reuse ratio reported in Section 4.2, we present the *exact* reuse ratio for each query in the JOB-*light* workload after first running the other 69 queries in Figure A.1. We also report the *templated* reuse ratio in Figure A.2.

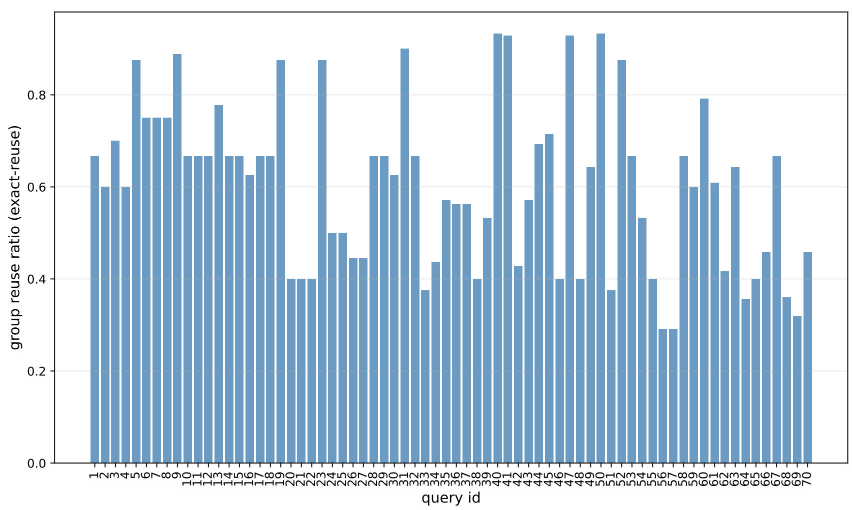


Figure A.1: Memo group *exact* reuse ratio for running each JOB-*light* query after the other 69 queries.

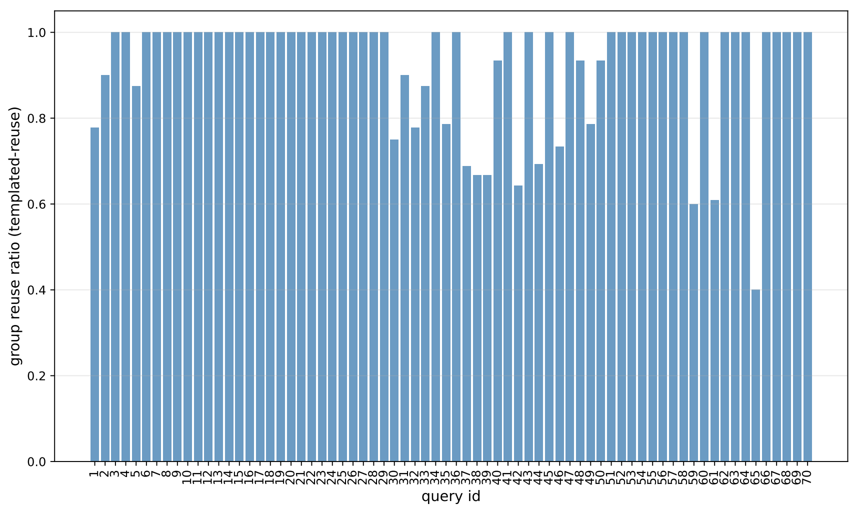


Figure A.2: Memo group *templated* reuse ratio for running each JOB-*light* query after the other 69 queries.

Bibliography

[1] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. *ACM SIGMOD Record*, 28(2):181–192, 1999. 5.4.2

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, 2015. doi: 10.1145/2723372.2742797. 5.1

[3] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000. 1, 5.3.1

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 221–230, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3190662. URL `https://doi.org/10.1145/3183713.3190662`. 5.1

[5] Nico Bruno and Cesar Galindo-Legaria. The cascades framework for query optimization at microsoft. In *Quarantine Tech Talks (2020)*. CMU Database Group, 2020. URL `https://www.youtube.com/watch?v=pQe1LQJiXN0`. 5.1

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 211–222, 2001. 5.4.2

[7] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, 1998. 2.3

[8] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 759–774, 2017. 1

[9] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. 1, 3.1.2

[10] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011. 1

[11] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, 2016. doi: 10.1145/2882903.2903741. 5.1

[12] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek Narasayya. Plan stitch: Harnessing the best of many plans. *Proceedings of the VLDB Endowment*, 11(10):1123–1136, 2018. 5.3.2, 5.4.2

[13] Bailu Ding, Vivek Narasayya, Surajit Chaudhuri, et al. Extensible query optimizers in practice. *Foundations and Trends® in Databases*, 14(3-4):186–402, 2024. 1, 2.2

[14] Ted Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7: 100049, 2021. 3.1.2

[15] Anshuman Dutt and Jayant R Haritsa. Plan bouquets: query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1039–1050, 2014. 1

[16] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, (Proceedings), 2007. 1, 3.1.2

[17] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. *ACM SIGMOD Record*, 30(2):331–342, 2001. 3.3.1, 5.4.3

[18] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18 (3):19–29, 1995. 3.3.1, 5.1, 5.4.3

[19] Goetz Graefe and David J DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 160–172, 1987. 2, 5.1

[20] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*, pages 209–218. IEEE, 1993. 2, 5.1

[21] Laura M Haas, Johann Christoph Freytag, Guy M Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 377–388, 1989. 5.1

[22] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Turbo-charging spj query plans with learned physical join operator selections. *Proceedings of the VLDB Endowment*, 15(11):2706–2718, 2022. 5.3.2

[23] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, March 2020. ISSN 2150-8097. doi: 10.14778/3384345.3384349. URL `https://doi.org/10.14778/3384345.3384349`. 5.2

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang,

Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415535. 5.1

[25] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*, pages 19–30. Elsevier, 2003. 1, 3.1.2

[26] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 106–117, 1998. 5.3.1

[27] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. Learned cardinality estimation: An in-depth study. In *Proceedings of the 2022 international conference on management of data*, pages 1214–1227, 2022. 5.2

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018. 5.2

[29] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1937–1940, 2019. 5.2

[30] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. Analyzing the impact of cardinality estimation on execution plans in microsoft sql server. *Proceedings of the VLDB Endowment*, 16(11):2871–2883, 2023. 2.3

[31] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015. 1, 2.3, 2.3, 4.2

[32] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *Cidr*, 2017. 1

[33] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, 2018. 1, 2.3, 2.3

[34] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670, 2004. 5.3.1

[35] Volker Markl, Peter J Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. Consistent selectivity estimation via maximum entropy. *The VLDB journal*, 16(1):55–76, 2007. 3.2.1, 5.4.2

[36] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*,

pages 398–412. Springer, 2005. 3.1.2

[37] Microsoft SQL Server. Cardinality estimation (ce) feedback. `https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-cardinality-estimation-feedback`, 2025. Microsoft Learn Documentation. Last updated: 2025-07-09. 5.3.2

[38] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*, pages 930–941, 2006. 2.1

[39] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 539–552, 2008. 2.1

[40] pgPedia. Postgresql hooks. `https://pgpedia.info/h/hooks.html`, 2025. 3.1.4

[41] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276, 1984. 1

[42] PostgreSQL Global Development Group. Postgresql. `http://www.postgresql.org`, 2025. 3.1.4

[43] PostgreSQL Global Development Group. Postgresql 18.1 documentation: Explain. `https://www.postgresql.org/docs/18/sql-explain.html`, 2025. 3.1.4

[44] PostgreSQL Global Development Group. Postgresql 18.1 documentation: Chapter 30. just-in-time compilation (jit). `https://www.postgresql.org/docs/18/jit.html`, 2025. 3.1.4

[45] PostgreSQL Global Development Group. Postgresql 18.1. `https://www.postgresql.org/docs/18/index.html`, 2025. 4.3

[46] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 23–34, 1979. doi: 10.1145/582095.582099. 1, 2.1, 2.3

[47] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 337–348, 2014. doi: 10.1145/2588555.2595637. 5.1

[48] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. Isomer: Consistent histogram construction using query feedback. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 39–39. IEEE, 2006. 5.4.2

[49] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001. 1, 5.3.2, 5.4.2

[50] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*,

13(3):307–319, November 2019. ISSN 2150-8097. doi: 10.14778/3368289.3368296. URL https://doi.org/10.14778/3368289.3368296. 5.2

[51] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3386134. URL https://doi.org/10.1145/3318464.3386134. 5.1

[52] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why tpc is not enough: An analysis of the amazon redshift fleet. *Proceedings of the VLDB Endowment*, 17(11):3694–3706, 2024. 1

[53] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, et al. Adaptive and robust query execution for lakehouses at scale. *Proceedings of the VLDB Endowment*, 17(12): 3947–3959, 2024. 5.3.1

[54] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *Proc. VLDB Endow.*, 14 (1):61–73, September 2020. ISSN 2150-8097. doi: 10.14778/3421424.3421432. URL https://doi.org/10.14778/3421424.3421432. 5.2