Hermann Owren Elton
Olaf Rosendahl

Adaptive execution techniques that avoid running sub-optimal query plans

A survey

Pre-study for Master's thesis in Computer Science
Supervisor: Norvald H. Ryeng
December 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU

Abstract

Sub-optimal query plans in database management systems (DBMS) have long plagued the realm of database performance optimization. This literature survey defines sub-optimal query plans and presents three different strategies to handle the sub-optimal query plan problem. First, the survey presents re-optimization strategies, which empower the DBMS to continually reassess and enhance query plans as new information and statistics become available. The survey presents various approaches proposed to dynamically adapt query execution plans in real time.

Furthermore, the survey delves into adaptive runtime strategies that can flexibly adjust query execution plans to align with evolving conditions. This section evaluates the challenges and opportunities related to adaptive runtime strategies.

After this, multi-plan optimization is presented, which deviates from the traditional single-plan approach by generating multiple alternative query execution plans for a given query. This enables the DBMS to select the most appropriate plan based on real-time execution statistics, data distribution, and dynamic workload variations.

Lastly, the survey presents how different commercial database systems have implemented strategies to solve the problem of suboptimal query plans.

By synthesizing existing research and providing an overview, this survey can be used as a resource for researchers, database administrators, and developers seeking to enhance query performance and minimize the impact of sub-optimal query plans in database management systems.

Sammendrag

Suboptimale spørringsplaner i databasesystemer (DBMS) har lenge vært en utfordring for optimalisering av databaseytelse. Denne litteraturstudien definerer suboptimale spørringsplaner og presenterer tre ulike strategier for å håndtere problemet med suboptimale spørringsplaner. Først presenterer undersøkelsen re-optimaliseringsstrategier, som gir databasesystemer mulighet til å kontinuerlig reevaluere og forbedre spørringsplaner etterhvert som ny informasjon og statistikk blir tilgjengelig. Undersøkelsen presenterer ulike tilnærminger foreslått for dynamisk tilpasning av utførelsesplaner for spørringer i sanntid.

Videre utforsker undersøkelsen adaptive kjøretidsstrategier som fleksibelt kan justere planene for utførelse av spørringer for å tilpasse seg utviklende forhold. Denne delen evaluerer utfordringene og mulighetene knyttet til adaptive kjøretidsstrategier.

Etter dette presenteres multiplanoptimalisering, som skiller seg fra tradisjonelle enkeltplantilnærminger ved å generere flere alternative spørringsutførelsesplaner for en gitt spørring. Dette gjør det mulig for databasesystemet å velge den mest passende planen basert på statistikk, datadistribusjon og dynamiske arbeidsbelastningsvariasjoner registrert under kjøring i sanntid.

Til slutt presenterer undersøkelsen hvordan ulike kommersielle databasesystemer har implementert strategier for å løse problemet med suboptimale spørringsplaner.

Ved å sammenstille eksisterende forskning og gi en oversikt, kan denne undersøkelsen benyttes som en ressurs for forskere, databaseadministratorer og utviklere som ønsker å forbedre databaseytelsen og minimere påvirkningen av suboptimale spørringsplaner i databasesystemer.

Contents

Chapter 1

Introduction

In this chapter, we present our motivation for this literature study and the background for it in Section 1.1. Then our goals and research questions for the study are presented in Section 1.2. Finally, an overview of the structure in the rest of the literature study is described in Section 1.3.

1.1 Background and Motivation

In the ever-evolving landscape of data management and retrieval, the optimization of query plans within databases stands as a critical frontier. Databases serve as the backbone of countless applications and systems, processing vast amounts of information to deliver timely and efficient results to users. However, as the scale and complexity of data continue to grow, the need for dynamic and adaptive query optimization techniques increases. This literature study does a deep dive into the problem of suboptimal query plans, to explore and evaluate various strategies and methodologies that address the inherent challenges of query plan optimization. By reviewing and analyzing existing research in this domain, we seek to provide insight into different techniques that handle the detection and correction of suboptimal query plans during execution. In particular, properties such as implementation complexity, overhead, efficiency, and pipelining support have been looked at in more detail for each technique.

What exactly is a sub-optimal plan? When a SQL query is sent to a database, it is parsed, re-written, and sent to the query optimizer which finds a query plan that can be executed. The query optimizer utilizes assumptions about existing data in the database such as data independence and uniformity to estimate cardinalities. The cardinalities are then used

in cost estimation which allows the optimizer to compare different plans and select the cheapest which will give the best execution-performance. Because cardinalities, costs, and therefore query plans, are based on estimates, errors can lead to not selecting the query plan which would be best if the cardinality estimates were replaced with actual values [1]. A sub-optimal query plan is therefore a query plan that is not the most efficient way to execute a query.

The motivation behind this study is rooted in the widespread impact of database query performance on both individual users and organizations at a large scale. Inefficient query plans can lead to slow application response times, increased resource utilization, and decreased user satisfaction. Moreover, inefficient query plans can significantly impact the cost-effectiveness of database operations, particularly in cloud-based environments where computational resources are billed on a pay-as-you-go basis. As such, optimization of query plans not only enhances user experiences but also has profound implications for cost savings and resource efficiency. This literature study aspires to provide a comprehensive overview of the methodologies and approaches that have emerged to tackle query plan optimization challenges, shedding light on their practical applicability and the potential benefits they offer in today's data-driven world.

This literature study is divided into chapters that go in-depth on different types of techniques that handle sub-optimal plans. Parts like how the techniques detect sub-optimalities, update the execution strategy, and then continue execution are described for each technique. We have divided the techniques into "Re-optimization", "Adaptive run time strategy", and "Multi-plan". Re-optimization techniques generally detect sub-optimalities by observing actual data/statistics and tell the query optimizer to re-optimize with the actual statistics if the actual data is too far off compared to the estimates. Adaptive run time strategies do not re-optimize, but can continuously adapt themselves during execution by observing the actual data. Multi-plan techniques are based on making the query optimizer generate multiple plans that cover all optimal plans based on different actual statistics, which allows selecting the optimal query plan without re-optimizing after observing some actual data.

In addition, this study also includes descriptions of how some commercial database systems have implemented and use strategies that handle the execution of sub-optimal plans. We have looked at Oracle DB, Microsoft SQL Server, IBM DB2, PostgreSQL, and MySQL, and seen how their implementations can resemble the theoretical and proposed strategies that are presented in this study.

1.2 Goals and Research Questions

The main goal of this literature study is to assess and compare various techniques for detecting and preventing the execution of sub-optimal query plans in databases, intending to improve query performance and resource utilization.

1. **What are the prevailing techniques and algorithms for re-optimizing query plans in database systems?**
   This question seeks to compile an extensive list of existing approaches, including but not limited to adaptive query optimization, and cost model refinement, to provide a comprehensive overview of the available techniques.

2. **How do these techniques differ in effectiveness, applicability, and resource requirements?**
   This question focuses on evaluating the strengths and weaknesses of each technique, considering factors such as query types, database workloads, and scalability. It aims to provide information on which techniques are best suited for specific scenarios.

3. **What are the practical implications and real-world use cases of these techniques?**
   This question delves into the practical deployment of strategies that prevent the execution of sub-optimal query plans, examining their impact on query performance, resource consumption, overall system stability, and actual usage in commercial database systems.

1.3 Structure

The remainder of this literature study is structured as follows: In Chapter 2, we present research within re-optimization of query plans where the query optimizer generates a new and updated plan after execution has started. Chapter 3 presents techniques for adaptively changing query plans during execution. In Chapter 4, we present algorithms that utilize multiple query plans generated by the query optimizer to find the best possible plan during execution. Chapter 5 presents some commercial database system that has implemented and utilized techniques that handle sub-optimal plans. We present our thoughts about potential future work within this field of study, based on our findings in the literature study in Chapter 6. Finally, Chapter 7 presents our conclusions and directions for future research.

Chapter 2

Re-optimization

Re-optimization is one way of handling the sub-optimal query plan problem. Database systems that implement re-optimization monitor the execution of a query plan, comparing the conditions of the chosen query plan, such as cardinalities, with the expected conditions. If the conditions differ too much, the query plan can be classified as sub-optimal and the plan is re-optimized with the actual statistics. The new plan is then executed and can increase performance. Re-optimization is a conceptually simple solution to the sub-optimal plan problem, and [2] presents how re-optimization can give similar results to perfect cardinality estimates in the PostgreSQL optimizer. This shows that re-optimization can give promising results. In this chapter, we look at how to determine when a query plan should be re-optimized and different techniques for re-optimization in Section 2.1. The re-optimization techniques are divided into techniques that do not reuse intermediate results found before the decision to re-optimize is made in Section 2.2, and those that do in Section 2.3.

2.1 Determine whether to re-optimize

Deciding if the current plan is worth re-optimizing based on the current state of query execution is one of the essential steps in implementing and supporting re-optimization within a database system. This is because re-optimizing query plans comes with some cost, and these costs might not outweigh continuing execution of the current sub-optimal query plan. Therefore, finding the right conditions where triggering re-optimization is the right choice is critical.

Cardinality estimate ranges

Cardinality estimates are one of the most important statistics that influence how the optimizer makes decisions about the construction of the query plan [3]. These estimates say how many rows a query plan or sub-part of a query plan returns. When performing re-optimization using deviations from these estimations as an indication of when to re-optimize could be a good strategy to determine if the query plan is sub-optimal. The problem is how far of a deviation is considered sub-optimal. There are multiple different ways to find the best deviation limit [4, 5].

In [4] the author proposes a method to calculate a range of cardinality estimate error where the current plan is still optimal. These ranges can be used to determine whether the current execution state of a query plan is still within an optimal state based on its current cardinality values. The proposed method calculates the optimal range of an edge within an optimal query execution plan. An edge in this context is the path that the query plan takes to the next leg of the query plan. This means that the algorithm calculates the range in which a given path within a query plan is still optimal. Then the method is invoked recursively to calculate the ranges for all edges in the plan, and accumulating these into an overall upper and lower bound. The method initiates by using the same inputs and assumptions as the optimizer, ensuring consistency in the calculation when the enumeration for the initial optimization and optimality range calculation is deterministic, thus enumerating the same plan space. Next, the algorithm shifts its focus to feasible plans enumerated through dynamic programming. To calculate accurate optimality ranges, it enumerates only those plans that comprise subplans deemed optimal at a certain point, in alignment with Bellman's principle of optimality.

Bellman's principle suggests that when solving a complex problem, you can break it down into smaller sub-problems. To find the overall optimal solution, you should make decisions at each step that lead to the best possible outcome not just for that step, but for all subsequent steps as well. This recursive approach allows you to build the solution incrementally, ensuring that at each stage, you are making the best decision considering the future [6].

If one can follow Bellman's principle of optimality, it will accelerate convergence and allow for effective pruning, if applicable, but the algorithm works with any deterministic enumeration strategy. The method additionally restricts this to subplans that attain optimality within the current optimality range, as it does not engage with intersection points that lie beyond this range. This methodology cultivates a robust pruning strategy,

markedly reducing the number of enumerated plans. The algorithm works for optimization algorithms that guarantee finding the cost-optimal plan, like bottom-up and top-down dynamic programming, or transformation-based optimization. It also works for those that enumerate a subset of the necessary plan space, given the enumeration is deterministic.

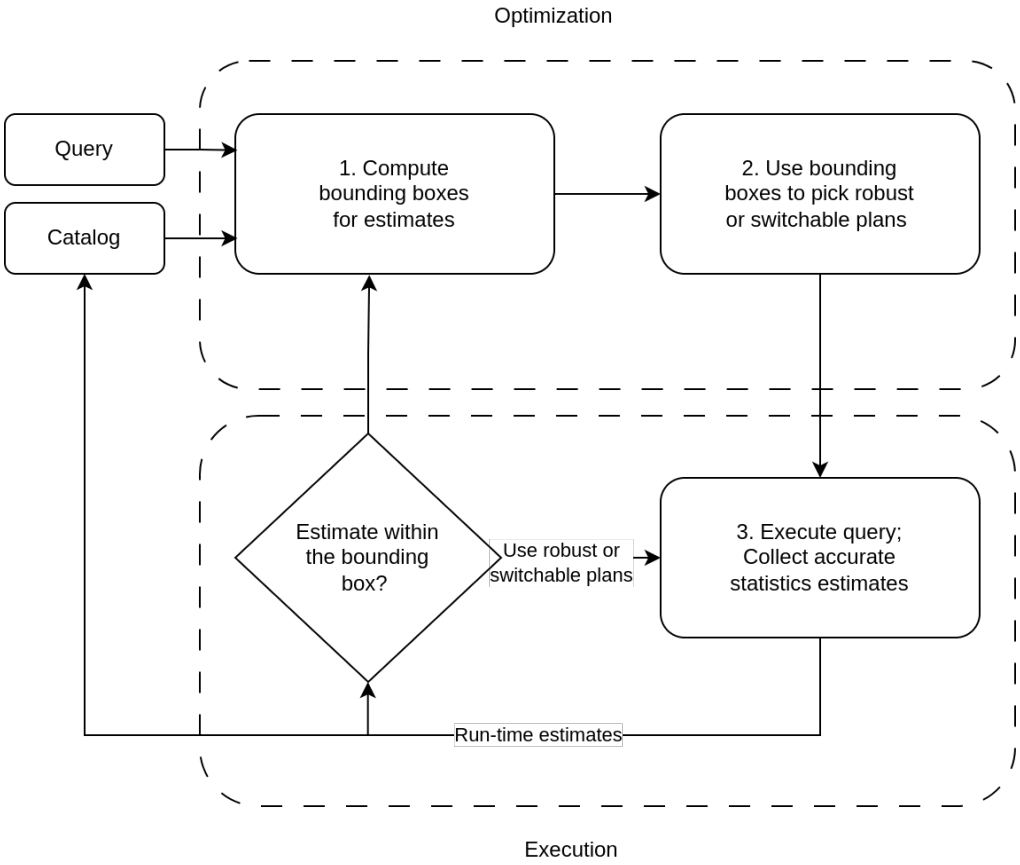
The author also states that their methods could be used to optimize methods for query optimization such as mid-query re-optimization [5] and parametric queries [7], which we will come back to later in this survey. The author concludes the paper by stating that further work would be to address assumptions made within the paper and validate that the method proposed calculates optimal ranges for other query processing applications.

Re-optimization cost

It is very important to also look at the cost of re-optimizing a query plan during execution, and the implications of implementing re-optimization. Since re-optimization reinvokes a previous step within the database query-handling process, there will be some added overhead to the overall response time of the query. It is therefore important to consider this when determining when to re-optimize, but also how to ensure the next optimization gets the query plan more right than the previous run. There are also different aspects of what causes extra cost and other techniques handle it differently [8, 9].

Consider the following query: "`SELECT * FROM R, S, T WHERE R.a = S.a AND S.b = T.b AND R.c > K1 AND R.d = K2`". We have accurate information about the sizes of the tables: $|R| = 200MB$, $|S| = 50MB$, and $|T| = 60MB$. Additionally, we assume that the size of $\sigma(R)$ is known to be $80MB$, but the optimizer significantly underestimates it as $40KB$. Consequently, the optimizer selects an optimal plan based on these statistics. A reactive re-optimizer can calculate a validity range for this plan. For example, the validity range for the index nested-loops join between $\sigma(R)$ and S in the given plan is $|\sigma(R)| \leq 100KB$. However, this validity-range check fails during runtime, prompting re-optimization. Subsequently, the next best plan is selected, but its validity range check also fails, leading to another round of re-optimization, and so forth [8]. This process continues until the completely optimal plan is chosen, producing a lot of overhead. In [8] the author tries to solve this problem by proposing three new techniques. First, poor statistics is addressed by computing bounding box intervals as a representation of uncertainty in statistics. Then these bounding boxes are used to produce robust plans and/or switchable plans that avoid re-optimization and loss of pipelined work. Lastly, they use randomization

to collect statistics quickly, accurately, and efficiently as a part of query execution. This method is visualized in Figure 2.1.1



Source: [8]

Figure 2.1.1: Proactive re-optimization

The implementation details for the database system Rio that was used to test proactive re-optimization needed the following operators to be implemented to work.

- Hybrid hash join: A join algorithm that processes tuples from two input subtrees
- A switch operator used to implement and use switchable plans
- Buffer operators that could collect and delay processing until necessary statistics become available.

- Operators that can scan the Materialized views to find what can be reused.

The execution engine needed the following implementations

- The ability to restart and stop execution to re-optimize.
- An in-memory catalog to track collected statistics and materialized views.
- An inter-operator communication mechanism that made it possible for query plan operators to signal to its parents.

Randomly collecting statistics could be a way to give the optimizer more information for subsequent invocations. This will give the optimizer more information about the runtime state of the query. What if one encounters a join algorithm that one finds sub-optimal, but one does not have adequate information about the joining tables of the join? This could still lead to a sub-optimal plan if the optimizer is reinvoked.

The author of [9] introduces a new entry within the query plan called a CHECK operator. This operator acts as a checkpoint that can be inserted into the plan before the operators that can potentially suffer if certain conditions are true. CHECK can then trigger re-optimization, if needed given the state of the query execution. This allows re-optimization only to be injected where needed and not for every single operation within a plan. This allows re-optimization to be inserted strategically within a query plan. Each CHECK has a condition that indicates the cardinality bound within which a plan is valid. The validity range is computed through a novel sensitivity analysis of query plan operators, then CHECK triggers re-optimization if the CHECK condition is violated. The article proposes five different flavors of CHECK operators. Lazy checking (LC), Lazy checking with eager materialization (LCEM), Eager checking without compensation (ECWC), Eager checking with buffering (ECB), and Eager checking with deferred compensation (ECDC). The different versions implement the check operator in different ways, allowing different types of functionality. The different types can be explained like this

- Lazy Checking (LC): is positioned above materialization points, carrying a minimal risk, primarily related to context switching. However, the potential for re-optimization is similarly minimal, transpiring solely at materialization points.

- Lazy Check with Eager Materialization (LCEM): incorporates pairs of CHECK and materialization on the outer portion of the Nested Loop Join (NLJN). While it introduces an additional risk related to materialization overhead, it also offers an opportunity for re-optimization at both materialization points and NLJN outers.
- Eager Check with Buffering (ECB): Implement a Buffer CHECK on the outer portion of NLJN. While it poses a significant risk owing to the absence of precise cardinality for the sub-plan below the ECB, it also allows for re-optimization at any moment during materialization, presenting a notable opportunity.
- Eager Check without compensation (ECWC): is placed beneath materialization points, carrying a substantial risk due to the potential discarding of an arbitrary amount of work during re-optimization. Nevertheless, this position also furnishes the possibility for re-optimization anywhere below a materialization point.
- Eager Check with deferred compensation (ECDC): similar to the ECWC, this CHECK operator also carries a substantial risk of forfeiting considerable work during re-optimization. However, it provides the possibility for re-optimization at any location within the plan of an SPJ-query (Select Project Join).

The main metrics to evaluate CHECK are the risk and opportunity of re-optimization at the checkpoint. An additional metric is its usability in pipelined plans, i.e., query execution plans that do not have any operators that block row processing, but stream all rows directly to the user to reduce the time that that user has to wait before seeing the query's first results. Re-optimization in this case might be triggered after some results have already been returned. Without buffering or compensating for those rows, re-optimization will result in unexpected duplicates, which is inconsistent with the semantics of the original query. The paper handles this by not writing the intermediate results to disk, but by using a temporarily materialized view as a pointer to actual objects from the current execution. If this view is reused, the attributes of this in-memory object are altered to accommodate the new plan. For example, the internal IDs for each column of this scan might change when the plan is modified. Moreover, re-optimization occurs within the same transaction as the initial partial execution, retaining all previously acquired locks. Consequently, all persisted results maintain their transactional accuracy during re-execution, ensuring that no work is forfeited.

2.2 Re-optimization of the entire plan

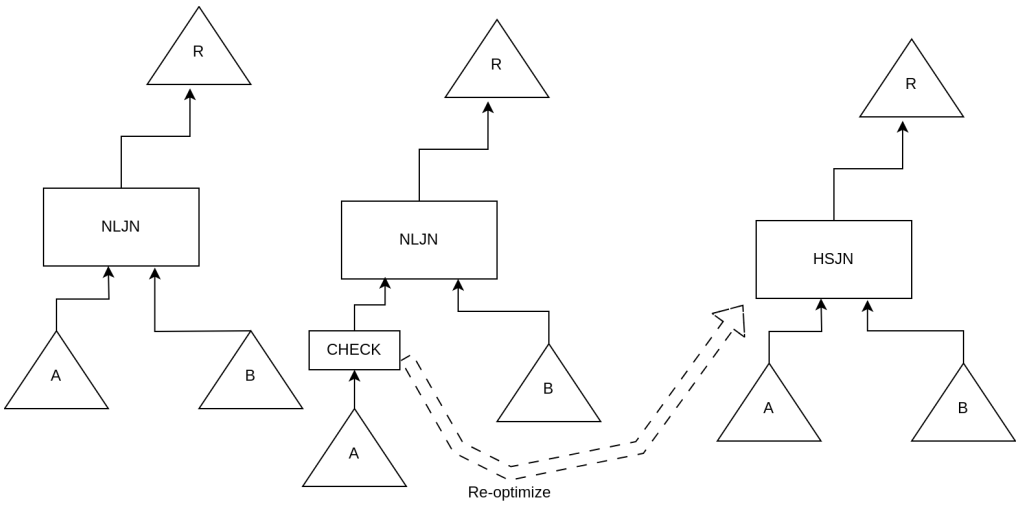
After deciding that a plan should be re-optimized, the next step is to re-optimize the given plan. In this section, we will focus on papers that present ways of executing re-optimization by re-optimizing the entire plan again. This is a simple way of after detecting a sub-optimality within the query plan, re-envoke the optimizer equipped with new information or another way of ensuring more optimal plan making, and then running the newly produced plan.

MONSOON

There are different ways of thinking about executing re-optimization, and how to avoid the optimizer simply picking the same query plan again. The MONSOON optimizer [10] has a way of handling user-defined functions (UDFs) by implementing re-optimization. A UDF is a function or an operation that is defined outside of the scope of the database system. The problem with UDFs is that the optimizer does not have access to the implementation of the UDF; this makes it difficult to estimate the cost of a query that uses a UDF, and this can lead to suboptimal query plans. MONSOON handles this problem by creating a query plan that does not take the UDF into account and runs this query plan while collecting statistics on the UDF. After collecting these statistics, MONSOON re-optimizes the query with the new information, which creates a new plan that handles the UDF. Looking at how MONSOON handles UDF, we can derive a way of re-optimizing that uses runtime-collected statistics to re-optimize the query plan with real information about cardinality or other information that can help the optimizer pick a better plan. MONSOON also presents a way of collecting the necessary statistics needed to perform the re-optimization. They use a Markov Decision Process (MDP) [11] to decide how to interleave execution and when to collect statistics. The paper's findings suggest that MONSOON is a promising approach to improve the performance of queries with partially obscured predicates. The multi-step optimization and execution approach allows MONSOON to gradually refine the query plan as more information becomes available. The Markov decision process helps MONSOON to make optimal decisions about when to execute the query plan and when to collect statistics.

Robust Query Processing through Progressive Optimization

In the previous section, we presented the CHECK operator [9] that is used to improve the optimizer to handle sub-optimal plans. This paper also presents how they perform re-optimization when a query execution satisfies the CHECK operator conditions. When a query execution hits a checkpoint, it collects the actual cardinality estimates for the given state of the execution and records these estimates so that they can be used to guide the optimizer into picking a better plan when re-optimizing.



Source: [9]

Figure 2.2.1: Proactive re-optimization

Figure 2.2.1 shows how the CHECK operator works, the original plan uses nested-loop join (NLJN) if the cardinality estimates that were used to make this decision are wrong, it can have detrimental effects on the execution time of the query. By inserting a checkpoint right before this join, the query execution can check if the cardinality estimates are correct and that it is safe to continue the original plan. In Figure 2.2.1 we see that the checkpoint has triggered re-optimization because the cardinality estimates did not match the actual cardinality, making NLJN a bad join algorithm. Re-optimization is therefore triggered, equipped with the newly recorded cardinality estimates, and therefore picks Hash-Join in the newly produced plan.

2.3 Re-optimization with re-use

We have looked at how to decide when to re-optimize, and how one can perform re-optimization in the sense of what one should do to not arrive at the same plan as originally deduced. The strategies for re-optimization we have looked at so far re-optimize the whole query plan if specific conditions are met during execution. This means that the results produced during the first execution are lost when performing the re-optimization, and when the new plan is created the execution starts from scratch. In this section, we will look at techniques for re-optimizing that retain or re-use the data that has already been fetched during the original execution [12, 13, 5, 14].

Tukwila

The Tukwila system [12] is an advanced database system for data integration. Tukwila combines many different adaptive query processing techniques to handle uncertainties that come with data integration. The Tukwila optimizer is designed in such a way that if not enough information or statistics is available, the optimizer creates an incomplete query plan that needs to be fleshed out during execution when more information is available. The optimizer also formulates the corresponding rules for event-condition-action. These rules describe two critical aspects, the circumstances and procedures for runtime adjustments to specific operators when necessary, and the conditions to evaluate at materialization checkpoints for identifying opportunities to optimize again. The query optimizer preserves the state of its exploration space when invoking the execution engine, enabling it to efficiently resume optimization incrementally when necessary. In Tukwila, operators are structured into segmented units known as "fragments". Once a fragment reaches its conclusion, pipelines come to a halt, results become tangible, and the remaining plan can undergo re-optimization or be rescheduled. This allows the system to keep hold of the data that it has already fetched, and only re-optimize the remaining query plan. Tukwila also implements different rules that trigger events that start different adaptivity processes. Tukwila is overall a system that implements a lot of different adaptivity concepts but for re-optimization, the biggest outtake is how it uses fragments which allows for the re-use of fetched data and only re-optimizing the given remaining plan.

Tukwila was a system based on adaptivity and handling a constantly changing environment. This means that in addition to re-optimization, it has other features to handle sub-optimal plans. In [13] the authors propose a method for incrementally re-optimizing query plans in the presence of

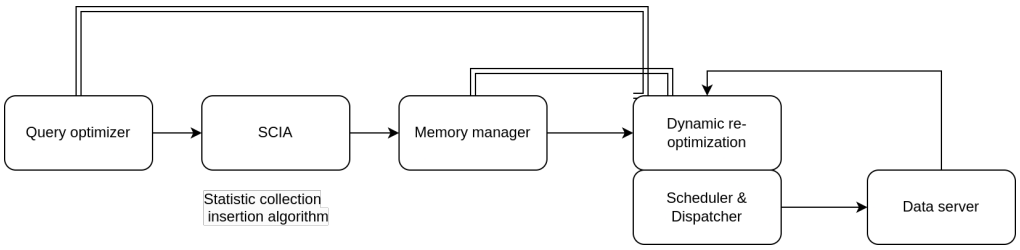
runtime changes. The proposed method is based on a cost-based optimizer that can be incrementally updated with new cost information. The authors show that their method can be used to improve the performance of queries in dynamic environments, such as stream processing systems. The re-optimization is with effect for the current execution of the query. The proposed method uses a cost-based optimizer that can be incrementally updated with new cost information. This allows the optimizer to re-optimize the query plan even after the query has started executing. The authors of the paper argue that this approach is more efficient than re-optimizing the query plan for the next time it is executed. This is because the optimizer can avoid re-evaluating the cost of all possible query plans. Instead, it can only re-evaluate the cost of the query plans that have been affected by the new cost information. The method for re-optimization is done as follows, the optimizer starts by creating a cost model for the query. This model takes into account the cost of the operators in the query plan, as well as the cost of accessing the data. Then the cost of all possible query plans is evaluated. The plan with the lowest cost is chosen as the initial query plan. The query starts to execute. As the query executes, the optimizer monitors the runtime statistics. These statistics can be used to update the cost model. If the cost model changes, the optimizer re-evaluates the cost of all possible query plans. The plan with the lowest cost is chosen as the new query plan. The query continues executing with the new query plan.

Tukwila is a whole system developed around being adaptive and handling a constantly changing environment. This makes it hard to just plugin the strategies of Tukwila into any database system. It would be hard to use Tukwila to help a database system that struggles with sub-optimal query plans.

Mid-query re-optimization

In "Mid-query re-optimization" [5], the authors propose that statistics collected at specific points during the execution of a given query can be used to collect information that can be fed back into the system to help execute the query. They proposed an algorithm that starts with inserting statistical collection points into a query plan. These collection points collect statistics such as cardinalities, sizes, and histograms. They are then fed back into the database system to improve optimization and runtime resource allocation. Also at these collection points cardinality thresholds are checked and if the current cardinalities exceed the threshold for that given point in the query plan, re-optimization is triggered. The execution stops, and the query is sent back to re-optimization with all the new statistics collected

throughout execution. The optimizer then returns a new plan with a new cost. This cost and the re-optimization overhead cost are compared to the current plan's cost, and if the new plan is deemed better, the execution continues from its current state with the new plan. This is one of the simpler methods of re-optimization, but it is also the paper that laid a lot of the groundwork for the current re-optimization and other optimization techniques that are in use today, which we will dive deeper into later in this paper. The authors also tested their technique on the TPC-D dataset, and the results for these tests show that the larger and more complex the query becomes, the larger the performance increase is made by performing mid-query re-optimization. The tests also showed that skewed datasets benefit from dynamic re-optimization.



Source: [5]

Figure 2.3.1: Mid-query re-optimization architecture

The Dynamic Re-Optimization algorithm also focuses on keeping the overhead of the algorithm low. They do this by only inserting collection points into the plan where it is deemed to be the most effective. The algorithm for determining the placement of collection points, also referred to as the statistics-collectors insertion algorithm, is executed based on several pivotal factors. Initially, it pinpoints potentially beneficial statistics that might enhance the optimizer's predictions. Subsequently, these statistics are arranged according to their efficiency. A statistic's efficiency is ascertained by its potential for inaccuracy, which refers to its capacity to expose a noteworthy divergence between the predicted and actual performance of a query plan. If two statistics present identical inaccuracy potential, the one impacting a more extensive segment of the query execution plan is deemed more efficient. Upon ordering the statistics, the algorithm commences the elimination of the least efficient statistics from the list. This procedure is perpetuated until the cumulative anticipated time for calculating all the lingering statistics falls below the maximal tolerable overhead. This approach ensures that the algorithm amasses the most effective statistics to aid in optimizing the query execution, while concurrently maintaining the

overhead of statistics collection within a permissible scope. The objective is to strike a balance between the need for precise statistics and the imperative to conserve resource usage.

The previously introduced re-optimization techniques rely heavily on the original plan when performing re-optimization or they are complete database systems that are made for handling constantly changing environments. The simple re-optimization techniques generally use the original as a baseline for re-optimization and add some guiding factors such that the new plan produced by the optimizer will at worst be as good as the original plan. They also normally look at mistakes within a plan and start re-optimization after a point of failure or a point where the true cardinality is too different from the estimated cardinality. What if the problem with the plan is based on the join order of the join that produced a true cardinality that was different from the estimate?

QuerySplit

The authors of [14] introduce a new re-optimization algorithm called "QuerySplit". This algorithm aims to handle the previously mentioned problems with re-optimization by introducing a way of splitting a query plan into different parts. They define a join that produces an unexpected cardinality result as an explosive join and queries that contain these joins as explosive queries. The algorithm tries to split the query plan into parts that are explosive and non-explosive. In this way, they can execute the non-explosive sub-queries first, and postpone the explosive sub-queries to a later stage. The implementation goes as follows. They take in a parsed query and create a split query set that contains sub-queries that cover the original query. They then start a loop that picks out the simpler queries to act on first and postpone the more complex sub-queries for later. On an iteration of the loop, the algorithm picks out a sub-query from the set and based on its cost calculated by a cost function that determines which query is the "simplest". This cost function used the estimated cost generated from the optimizer and the cardinality estimates of the query. This produces a plan that is defined as the simplest sub-query plan at that iteration. This plan is sent to the execution engine. The result of each iteration is materialized in a memory buffer, which is done by setting its output destination to a temporary table in PostgreSQL. This table is sent to the statistic collector where Postgres performs the basic statistic collection. The table is then used in the next iteration to update the rest of the sub-queries that overlap the finished sub-query. This is done until the last entry in the set, which is the final finished result is sent to the user.

The author presents different ways of calculating the cost of a sub-query. They define C(q) as the cost of the query defined by the optimizer, and S(q) as the cardinality of the query. They conclude that the most optimal method of calculating the cost is to use the following formula: $C(q) * S(q)$.

They also propose the sub-queries splitting policy called FK-Center as the most optimal for generating sub-queries. This policy is based on the non-expanding property of the primary-foreign-key joins. We define the concepts of FK-relations, which are relations with at least one foreign-key reference to another relation, and PK-relations, which are relations with primary keys referred by other relations when given a query. The query is then represented by a directed join graph, where each vertex denotes a relation, and each edge denotes a join predicate (single table predicates are connected to the vertices). Each edge in this graph runs from an FK-relation to a PK-relation. The edge becomes bidirectional if the join operation is performed between two relations of the same kind. We eliminate superfluous join predicates, such as those that create cycles in the join graph, to increase efficiency, giving priority to the elimination of bidirectional edges. The directed join graph's vertices are all sequentially explored by the FK-Center approach. They build a subquery centered around each vertex that has at least one outbound edge that includes all the relations to which the vertex refers.

Chapter 3

Adaptive run time strategy

An alternative to re-optimization when a query plan is found sub-optimal is using adaptive operators and strategies that automatically handle sub-optimality by itself. Compared with re-optimization, this avoids the potential overhead that occurs when a query plan is run through the optimizer multiple times. In addition, some of the presented techniques don't require cardinality estimates and can always be used in the query plan. This can simplify the optimizer and therefore increase performance. This chapter first presents two different methods for dynamic path access in Section 3.1. Then, methods that dynamically can switch an operator during execution are presented in Section 3.2. In Section 3.3 multiple techniques for dynamic reordering or operators are presented. Finally, eddies and multiple techniques that expand on and enhance eddies are presented in Section 3.4.

3.1 Adaptive Access Path

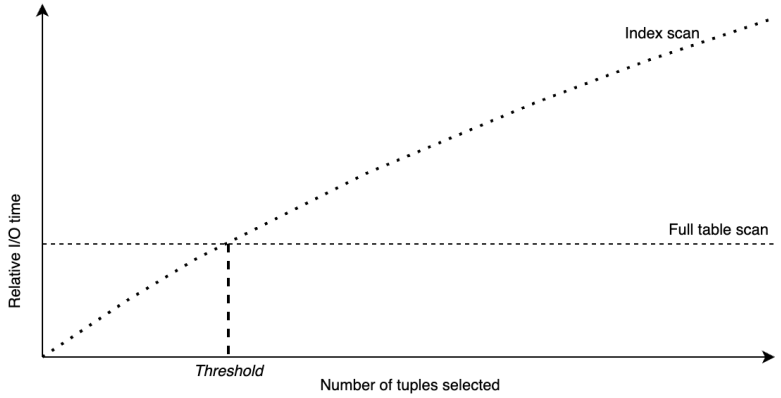
Adaptive Access Path techniques dynamically adjust the access path for retrieving data from a database based on real-time statistics gathered during query execution. Unlike static methods that rely on pre-computed indexes and selectivity estimates, adaptive access paths continuously monitor data distribution and access patterns, enabling them to select the most efficient method for each query.

Adaptive Selection of Access Path

In "Adaptive Selection of Access Path and Join Method" [15], an approach is presented to choose the access path adaptively. The authors argue that

"If the distribution of attribute values is with a high degree of skew, the number of tuples selected can vary substantially from the value estimated by the attribute selectivity" [15, p. 250]. They acknowledge that using histograms and other summary statistics can improve cost estimates compared to attribute selectivity, but they're problematic in queries with variables as the selected access path is chosen for average performance which can be suboptimal for individual queries. The paper argues that delaying the decision of access path-strategy until more information is available at run time can improve performance substantially.

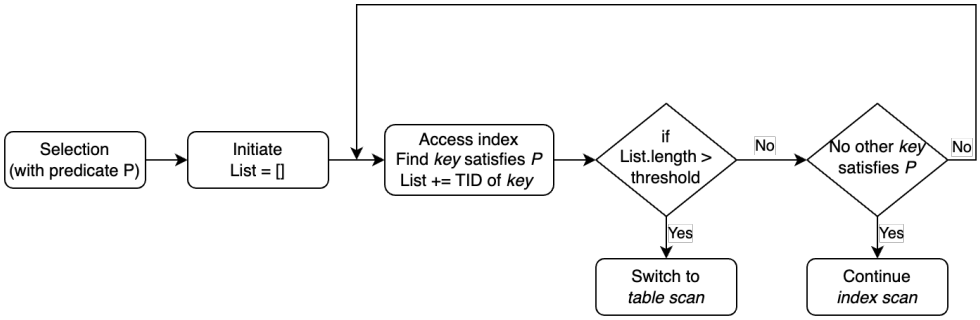
In order to adaptively select the access path, an algorithm is presented. The main strategy of this algorithm is to use the information of an index to find the selectivity and distribution of the tuples that satisfy a given predicate. With this information, a selection on using index scan or full table scan be done before actually retrieving tuples from disk, but without relying on cardinality estimations in the query optimizer. To explain how this is possible we can start by looking at the cost of a full table scan, which is the I/O cost of accessing all the pages in the relation. By using sequential access the I/O cost can, according to [16], be speeded up compared to random access. If for example, the increase in speed is by a factor of 5, the number of pages an index can lookup with random access must be less than a fifth of the total amount of pages in order to be faster than using a full table scan. The theoretical difference in performance based on the number of selected tuples can be seen in Figure 3.1.1. The authors did not implement the algorithm in an actual database, therefore no actual tests have been conducted to verify the suggested performance improvements.



Source: [15]

Figure 3.1.1: Comparisons of the I/O costs for full table scan and index scan

To be as efficient as possible, the algorithm therefore defines a variable of the threshold, which is the maximum amount of pages that can be retrieved before a full table scan is beneficial. Then the index is retrieved and the search for the next key satisfying the predicate is started. When a key is found, its tuple-id (TID) is added to a buffered list in memory. The search continues if the number of pages containing the TIDs is below the threshold. If the threshold is not exceeded and no more keys matching the predicate are found, use the list of TIDs to retrieve the tuples from the disk. If the threshold is exceeded while searching the index, the index search is stopped and a full index scan is started. The extra overhead of running an index scan and then switching to a full table scan is minimal compared with selecting non-optimal access paths which can be high for large relations. A prerequisite for the presented technique is that an index for the given predicate exists, if not, a full table scan will always be necessary. The full flow of this algorithm is illustrated in Figure 3.1.2.



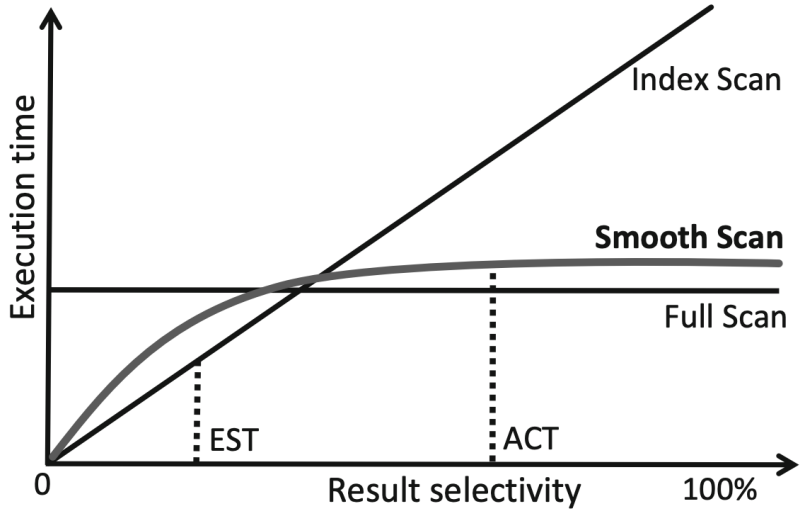
Source: [15]

Figure 3.1.2: Flow diagram of progressive access path selcction

Smooth Scan

Another adaptive access method technique named Smooth Scan [17, 18] has also been proposed. In [17], the authors argue that the main problem of re-optimizing query plans is that it is based on a binary decision. When a cardinality threshold is used to determine if re-optimization is necessary, a single extra tuple can cause a major change in performance. After the strategy switch, the total execution time is extended by the time it takes to perform a full scan. This means that a query that is run twice, but with one more tuple in the result set when run the second time may have a large difference in execution time.

Smooth Scan is therefore proposed as a solution which, instead of re-optimizing and switching access path, uses a morphing strategy to morph between index access and full scan by continuously monitoring the observed selectivity during run time. From [17]: "The core idea behind Smooth Scan is to *gradually* transform between two strategies, i.e., the index look-up and full table scan, maintaining the advantages of both worlds. The main objective is to provide *smooth behavior* so that at no point during execution an extra tuple in the result causes a performance cliff. Smooth Scan *morphs* its behavior incrementally, and continuously, causing only gradual changes in performance as it goes through the data and its estimation about result cardinality evolves." As seen in Figure 3.1.3, the cost of Smooth Scan is worse than Index Scan and Full Scan when they are selected and actually optimal, but ensures near-optimal performance regardless of the cardinalities.



Source: [17]

Figure 3.1.3: Smooth Scan target behavior

How does Smooth Scan achieve gradual adoption? The operator has three different modes during its lifetime, with each mode performing more work since the selectivity increases. The first mode is Index Scan (assuming that an index can be used as an access path). As the tuples are retrieved, Smooth Scan monitors the cardinality and switches to the next mode if it exceeds a given threshold. The second mode is called Entire Page Probe and addresses a fundamental challenge of the Index Scan – the potential for multiple references to the same disk page, which can lead to redundant, random I/O accesses. In this mode, Smooth Scan takes a proactive stance by analyzing all records on each heap page it loads, in order to identify

qualifying tuples and mitigate the need for repeated page accesses. This trade-off involves investing CPU cycles to read additional tuples from each page, a decision justified by the significant CPU savings compared to the cost of I/O operations, which translates to orders of magnitude more CPU instructions. The access pattern in this mode remains sequential within each page.

If the result cardinality continues to grow, Smooth Scan shifts to Flattening Access mode. In this mode, it optimizes query processing by replacing random I/O cost and instead reads additional adjacent pages from the heap sequentially. The final mode is called Flattening Expansion and is an ever-expanding variation of Flattening Access. Initially, it fetches one extra page for each page it accesses. Then, when a rise in result cardinality is detected, Smooth Scan progressively increases the number of pages, which is prefetched by a factor of 2. In essence, when result cardinality increases, Smooth Scan not only adapts but morphs more aggressively towards a full table scan strategy.

Smooth Scan can be implemented with different trigger points. One alternative is to be integrated into existing query stacks to increase robustness, activating morphing when the result cardinality surpasses the optimizer's estimate, signaling potential inaccuracies in the chosen access path. Another alternative is to monitor operator execution in real-time, aiming to maintain performance within defined service-level agreements, starting morphing when a Service Level Agreement (SLA) violation is anticipated. An eager approach exists as well, where Smooth Scan starts morphing from the first tuple, guaranteeing a worst-case scenario of page accesses equal to the total number of heap pages. This approach eliminates the need to record tuples produced before morphing and therefore reduces overhead. The eager approach was chosen for running tests in the paper because it has the lowest overhead, with test results showing that the worst-case scenario creates only 14% overhead compared to the optimal behavior.

The paper describes how Smooth Scan was implemented in PostgreSQL. In essence, it did require only a few changes since the access path was replaced with Smooth Scan, leaving the upper query plan layers unchanged. Performance-wise Smooth Scan is especially efficient compared to plain PostgreSQL when the selectivity of a query is low. When the selectivity increases to above 5%, the performance advantage of Smooth Scan disappears and a small overhead is introduced instead.

3.2 Swap operators

A significant contributor to the sub-optimality of query plans is choosing the wrong join algorithm. If the optimizer thinks the join tables are smaller, it might choose NLJN as the join algorithm for a query. If the cardinality estimates then turn out to be wrong NLJN can affect the query's runtime in a negative due to the nature of how NLJN works. A technique for handling this problem could be to swap out the poorly chosen join algorithm during query execution if and when it turns out to be a bad choice. The goal of swap operators is to dynamically swap operators that are detected as sub-optimal during execution in favor of more optimal operators.

Adaptive join operator

An adaptive join operator that can change the join algorithm adaptively throughout the query execution is presented in [19]. The proposed method always begins with a Symmetric Hash Join (SHJ) to produce the first result tuple as soon as possible and then changes to the Bind Join (BJ) if it estimates that bind joins will be more efficient for the rest of the process. The operator continues to switch between SHJ and BJ as needed to minimize the cost of joining. Being able to adaptively change the join algorithm reduces the negative effects that poor cardinality estimates have on a query plan since the execution engine can adapt to a poor estimate by changing to a safer join algorithm for the given true cardinalities.

The paper also proposes a way of performing these adaptive queries for a multi-join query, and that algorithm goes as follows. Once all tuples from a given relation, referred to as Ri, have arrived, the algorithm proceeds to assess the anticipated time remaining if the adaptive join operator were to switch to a bind join for every other relation that shares a common attribute with Ri. The algorithm selects the relation, denoted as Rj, with the minimum estimated bind join cost and proceeds to compare the expected remaining time if it alters the join method to a bind join for the combination of Ri and Rj with the expected remaining time if the operator persists with a multi-way symmetric hash join involving all the relations. To gauge its performance, AJO underwent comparative evaluations against SHJ and BJ. The results revealed that, in terms of response time, AJO closely matched SHJ. However, in completion time, AJO significantly outperformed SHJ, boasting speeds between 1.4 and 2.2 times faster.

In contrast to BJ, AJO delivered even more remarkable results. It not only accelerated response times, achieving speedups from 11 to 34.3 times faster

but also improved completion times by 2.8 to 6.2 times. This highlights AJO's capability to provide optimal response times akin to Symmetric Hash Join while expediting completion times. Compared to Bind Join, it substantially enhances response times while also advancing completion times across most scenarios. Notably, AJO's performance exhibited consistency across various data sizes and arrival rates, underscoring its effectiveness for both single- and multi-join queries. This positions it as a promising solution for optimizing linked data query processing, promising more efficient utilization of web-based runtime environments.

A blog post from Databricks about speeding up Spark SQL at runtime [20] describes a feature for "Dynamically switching join strategies". It doesn't go much into detail but shows how it can change the join strategy from sort-merge join to broadcast-merge join after observing accurate join relation sizes at runtime, which is quite similar to the technique presented above.

Proactive re-optimizaiton

Switchable plans are also discussed in [8]. In the context of this paper, a switchable plan refers to a collection of plans designed to accommodate varying input sizes. These switchable plans share a common structure. Each has a distinct join operator as the root operator, the same subplan for the deep subtree input to the root operator, and the same base table. However, they may differ in terms of the access path for the other input to the root operator.

To construct switchable plans, three initial plans are taken into account: BestPlanLow (the plan with the lowest cost, denoted CLow), BestPlanEst (the plan with the lowest cost among estimates, denoted CEst), and BestPlanHigh (the plan with the highest cost, denoted CHigh). These seed plans serve as the foundation for generating a switchable plan.

It is essential to emphasize that any two members within a switchable plan are considered switchable with each other. The switchable plan dynamically selects one of its members during execution, guided by gathered statistics and estimates. This adaptability enables it to respond effectively to changes and uncertainties in the data, ultimately optimizing the execution of the query.

Both of these papers present a way of altering the operations within a query plan. This allows the query execution process to handle sub-optimal plans on its own without having to invoke the query optimizer again. Test results that are presented in [19] show that AJO is faster than other join

methods that are created to handle the same problem. The technique from [8] was also presented in the re-optimization chapter, but in addition to implementing re-optimization, it also handles one of the bigger problems with re-optimization, having to go back to the query optimization in order to re-optimize, which can lead to slower completion time. Swap operators circumvent this by handling suboptimality during execution.

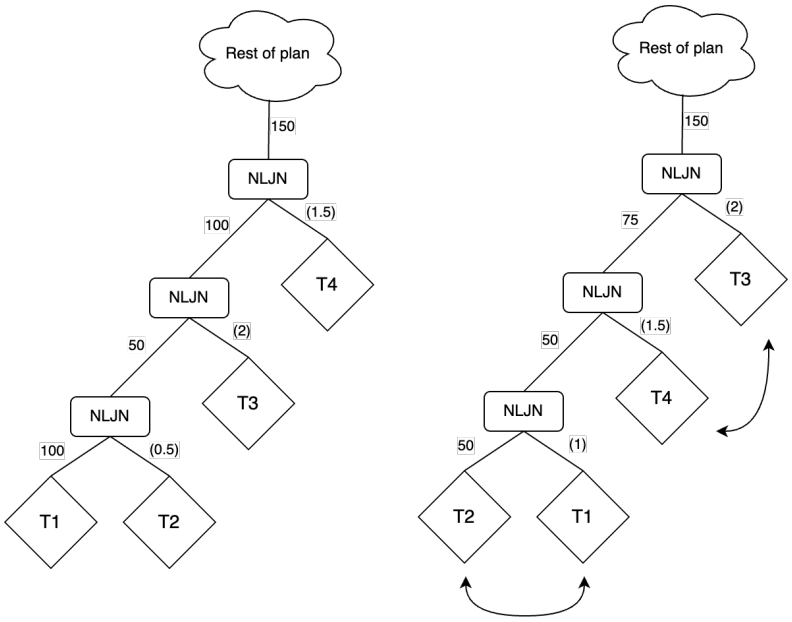
3.3 Dynamic reordering of operators

In the previous section, we presented swap operators that switch out methods within the plan to always use the best algorithms and methods for the given job. When already on the topic of changing operations, why not reorder the logic of the query plan in order to take advantage of runtime statistics and cardinality estimates in order to change the plan into a better plan on the fly? Dynamic reordering of operators handles the sub-optimal query plan problem by changing the order or a query plans operations without invoking the optimizer.

Adaptively Reordering Joins

In [21] the authors present a new technique for dynamically adjusting the join order in pipelined join execution plans in databases. The approach capitalizes on both straightforward statistics and those gathered through runtime monitors. In this way, it can estimate the benefits of changing join orders in real-time, providing more accurate predictions than a static optimizer. Figure 3.3.1 shows an example of how the joins may be reordered.

The strategy uses a "history window" parameter to monitor incoming and outgoing rows, enabling it to adapt to data trends while avoiding short-term fluctuations. Experimental results using this method have shown significant performance improvements for most queries. Moreover, the paper discusses the impact of different history window sizes on join order alterations. It was found that smaller windows resulted in dramatic fluctuations without noticeable performance improvement, whereas larger windows (over 500 rows) led to stable performance and reordering.



Source: [21]

Figure 3.3.1: Example of a nested-loop join plan with original and re-ordered plan

This technique supports changing pipelined join execution by changing the pipeline itself. This also minimizes bookkeeping. In addition, the presented technique that prevents duplication is effective and enables support for both driving and inner tables. Although the paper only presents the technique on nested-loop joins, the authors argue that "it is not difficult to see that this technique can be extended to pipelined hash joins as well".

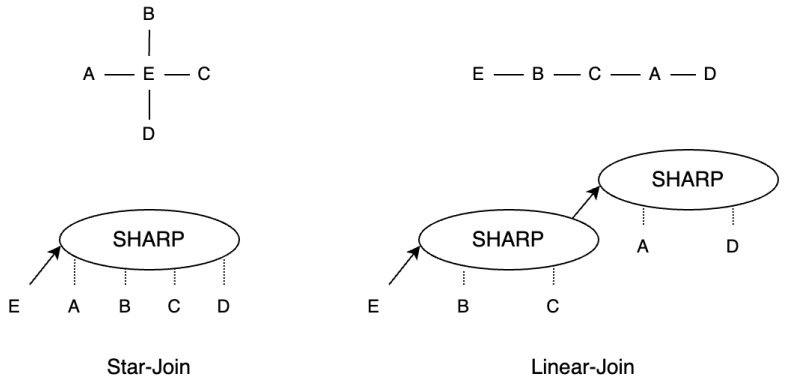
Lastly, the technique was tested with a larger number of joins, further proving its effectiveness. While this approach is primarily focused on nested-loop joins, the paper suggests it could be extended to pipelined hash joins. Future work includes exploring the integration of this technique with re-optimization techniques and the dynamic selection of index scan access paths at runtime.

SHARP

SHARP [22] is depicted as a dynamic multi-join operator, endowed with the capability to modify join sequences in real-time, making decisions rooted in live data observations rather than static predetermined statistics. Such an adaptive mechanism establishes SHARP as an adept tool for intricate

query operations.

Central to SHARP's methodology is tuple routing, presented as a more efficient alternative to the ubiquitous symmetric hash joins. This approach ensures that SHARP operates with greater memory efficiency, even when processing relations that exceed the available memory, a crucial feature when managing datasets larger than memory allowances. It supports star-join and linear-join, as illustrated in Figure 3.3.2.



Source: [22]

Figure 3.3.2: SHARP only supports star-join and linear-join

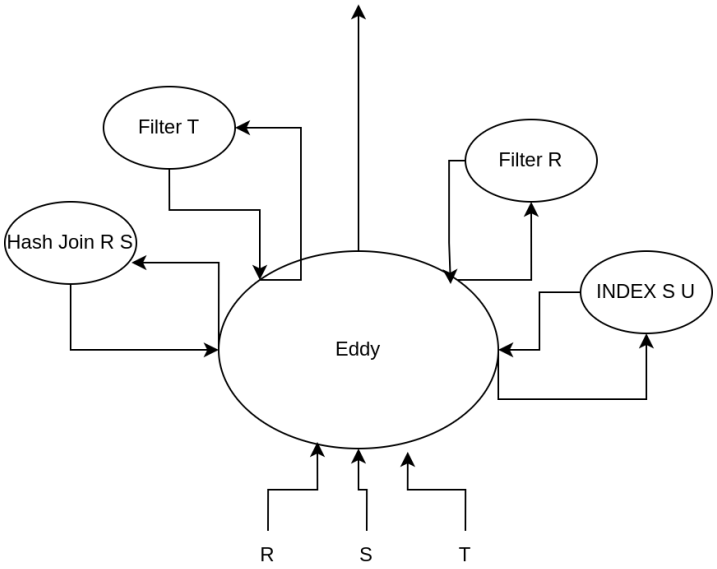
One of SHARP's standout features is its dual-faceted query approach. Post the assessment of all build tables, if conditions suggest that SHARP's second-stage performance is inferior to that of a right-deep hash join tree, it possesses the agility to adapt and function in the same manner as such a tree. This inherent versatility underscores its adaptability and robustness.

Performance-wise, SHARP exhibits several merits. It negates the generation of unnecessary intermediate results, optimizes memory allocation across joins, and importantly, offers a buffer against sub-optimal join sequences as dictated by traditional optimizers, proving its mettle in a variety of situations.

In summary, SHARP is an adaptive and robust query processing technique that can handle large datasets, adapt to changes in data, and optimize memory usage for efficient execution.

3.4 Eddies

A bit more advanced technique is eddies [23]. Eddies offers a dynamic approach to query processing, enabling granular, real-time, and adaptive optimization. This stands in stark contrast to conventional query optimization methods, which typically exhibit broad-based and static strategies. Such static methods might falter in the face of unpredictable environments, especially prevalent in large-scale systems or when handling interactive online queries.



Source: [23]

Figure 3.4.1: Example of an Eddy

Figure 3.4.1 illustrates an eddy functioning within a data pipeline. Data is channeled into the eddy from input relations labeled R, S, and T. The eddy orchestrates the path of data tuples to various operators, each of which operates as an independent thread, thereafter returning tuples back to the eddy. A tuple is propelled to the output only after it has traversed through all the operators. Furthermore, the eddy adaptively determines a unique pathway to guide each tuple through the respective operators.

The primary strength of eddies is their capability to perpetually reshuffle operators within a running query plan. By monitoring tuples as they navigate in and out of operators, eddies can actively modify tuple routes to establish varying operator sequences. This inherent fluidity enables eddies to adjust to evolving system attributes, optimizing query handling based

on the prevailing system conditions.

Furthermore, eddies have the potential to function as a central optimization tool within a query system. This would negate the necessity for intricate coding often integral to classic query optimizers. Alternatively, integrating eddies alongside traditional optimizers can heighten adaptability throughout processing pipelines.

One of the main challenges with Eddies is improving their reaction time. Since eddies adapt on the fly, there can be a delay before they adapt to the new optimal operator order, unlike traditional query optimizers which pre-optimize the query plan. Another challenge that eddies face is adaptively choosing to join orders, especially when some sources are delayed, which requires more research to be effectively addressed.

While eddies perform nearly as well as static optimizers in static scenarios, there is a need for improvement to achieve optimal execution. Future work is suggested to develop policies that can help eddies converge quickly to near-optimal execution in these scenarios. The paper also hints at exploring the parallel load-balancing aspects of the optimization problem, suggesting that balancing load across parallel operators is another challenge for eddies.

The original eddies proposal is a query processing mechanism that continuously reorders operators in a query plan as it runs. It is designed for dynamic environments where resources and workload can change frequently.

STAIR

One of the challenges of the original eddies proposal is that it has to keep track of the history of routing decisions. This is because the routing decisions can have long-term effects on the state of the operators in the query, and the eddy needs to be able to undo the effects of past routing decisions if necessary.

A mechanism called STAIRs (Stateless Adaptive In-memory Routing) that addresses this challenge is presented in [24]. Instead of having the state locked within join operators as traditionally done, STAIRs contain this state. To put it succinctly, a STAIR operator targeting a relation R and an attribute a, symbolized as $R.a$, preserves tuples (which might be intermediate) that have a foundational component from relation R, and is adept at two primary functions.

The standout capability of STAIRs lies in its power to reveal the growing state to the eddy while offering tools for state management. This lets the

eddy reverse past routing choices, essentially negating the constraints of past decisions.

By granting the eddy the privilege of state accessibility and management, STAIRs empower it with heightened adaptability during runtime. This adaptability extends to the re-shuffling of operator sequences anytime during execution. Moreover, the versatile state management granted by STAIRs paves the way for fresh optimization avenues, enabling a balance between computation and storage during query operations.

STAIRs generally tend to outperform Eddies in most scenarios. While both Eddies and STAIRs perform significantly better than the best static plan for a given query, STAIRs often have an edge due to their ability to avoid the initial delay in adapting that Eddies experience. During this delay, an eddy may make less optimal routing choices, which can affect its performance. Eddies and STAIRs are adept at identifying and exploiting the horizontal partitioning of a table. They can route parts of the table through one query plan and other parts through a different query plan, leading to improved performance. However, due to their enhanced flexibility and adaptivity, STAIRs can often make better use of this capability than Eddies.

STEMs

A different extension to eddies called SteMs is presented in [25]. The paper presents a query architecture in which join operators are decomposed into their constituent data structures (state modules, or stems), and dataflow among these stems is adaptively managed by an eddy routing operator [23]. The eddy routing operator navigates tuples between SteMs and additional modules, facilitating the adaptive selection of access methods, join algorithms, spanning trees, and join orders by the eddy. While SteMs provide flexibility, they also necessitate routing constraints to guarantee accuracy.

- Bounded Repetition: This ensures that no tuple can be routed to a given module more than a finite number of times. This prevents infinite loops and helps to manage resources effectively.
- BuildFirst: A singleton tuple from a table T must first be built into the State Module if T has multiple Access Modules or if T has an index Access Module. This constraint ensures that all necessary data are available for querying and processing.
- ProbeCompletion: This constraint states that a prior prober must not be routed to any State Module other than that on its probe

completion table. This ensures the integrity and correctness of the data.

- SteM BounceBack: A State Module must bounce back a build tuple unless it is a duplicate of another tuple that is already in the State Module. Similarly, a State Module must bounce back a probe tuple unless it already contains all matches for the probe that have been cached in other State Modules.

- TimeStamp: When a tuple probes into a State Module and finds a match, the result is returned to the Eddy only if certain conditions are met. This constraint helps maintain the order and chronology of the data.

SteM Bounceback and TimeStamps are enforced within both the State Module and Access Module implementations. Bounded repetition, Build First, and Probe completion must be strictly followed by the Routing Policy Implementor. They guarantee that the routing process does not yield incorrect query results while preserving flexibility and adaptability in query processing.

SteMs facilitate adaptive query processing by encapsulating traditional join algorithm data structures and making them directly accessible to the Eddy, the routing module. They function essentially as a semi-join, managing insert and lookup requests on the encapsulated dictionary data structure for tuples from a table. By partitioning joins into SteMs, the physical operations usually contained within join modules become decoupled. This decoupling allows for performance tuning, nuanced routing adaptation, and work sharing. Employing SteMs enables the Eddy to adaptively determine not only the operator order but also the selection of access methods, join algorithms, and the choice of a spanning tree in the query graph. Consequently, this yields more versatile, adaptive query processing.

Chapter 4

Multi-plan

This chapter presents algorithms that utilize multiple query plans generated by the query optimizer to find the best possible plan during execution. This differs from re-optimization by avoiding query optimization during execution, and instead using the optimizer to generate multiple query plans which cover different optimal strategies based on the actual cardinalities found during execution. Compared to adaptive run time strategies, multi-plan strategies switch to the plan among multiple pre-generated plans that fit the actual cardinalities, instead of adapting the current and only query plan. Different multi-plan strategies can be divided into two groups. Plan bouquets are first presented in Section 4.1, followed by parametric query optimization in Section 4.2.

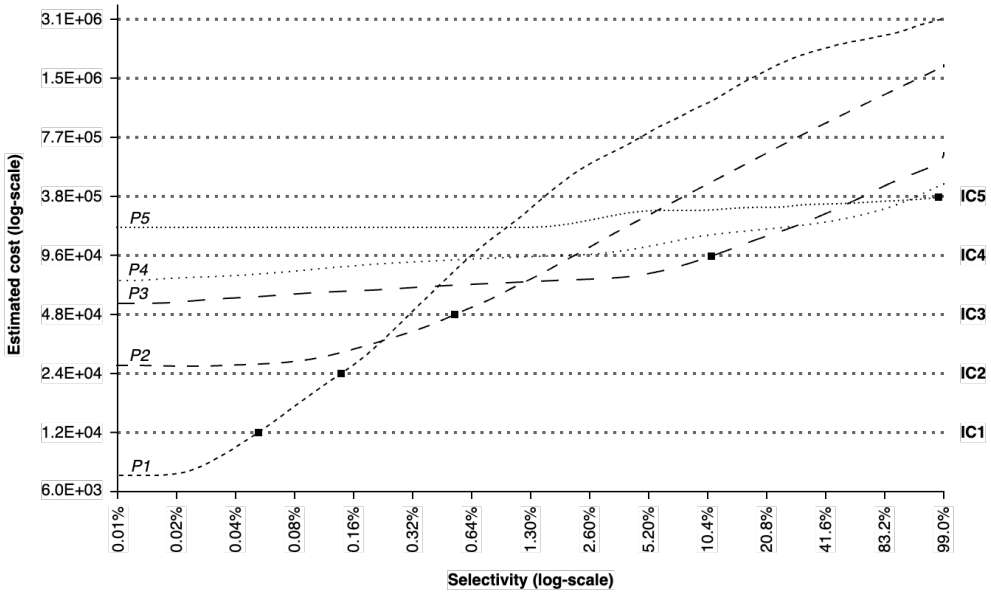
4.1 Plan bouquets

Plan bouquets involve generating multiple query execution plans for the same query and selecting the best plan at runtime based on the specific circumstances of the query. These plans are pre-optimized for different scenarios or parameter values, helping to mitigate the performance variability.

Dutt and Haritsa present a novel plan bouquet algorithm designed for robust query processing in [26], a continuation of [27] by the same authors. Instead of relying on selectivity estimations, which often significantly differ from the actual selectivities, the algorithm generates a limited subset of the optimal plans in the error with a technique that ensures that for each location in the space, at least one of the plans is 2-optimal. 2-optimal means that the performance of the chosen plan will be within a factor of 2 of the

optimal plan for the actual selectivity values. Then, during execution, the "best" plan is found and executed.

To understand how this is possible, let us take a closer look at how the algorithm works step by step for a one-dimensional query. First, at compile-time, the query optimizer is run repeatedly to create a parametric optimal set of plans (POSP) which covers the range of possible selectivities for the predicate, each plan annotated with the range of selectivity where it is optimal. Isocost (IC) steps are then created in intervals and for each isocost level, the best POSP plan is determined. The illustration of this in Figure 4.1.1 shows how the plans P1, P2, P3, and P5 each are best for an isocost-level, while P4 doesn't perform best at any of the levels. The subset of plans which is associated with an isocost-level is the plan bouquet for the query.



Source: [26]

Figure 4.1.1: Plan performance based on selectivity and selected plan for each isocost-level

Then, at runtime, the plan associated with the lowest isocost is executed with two possible outcomes. One is that the partial execution exceeds the isocost level, at which we know that the actual selectivity is above the current level, which leads to a switch to the plan at the next isocost level. If the next level is associated with the same plan as the current level, execution is simply continued. If not, the partial results that have been

produced so far by the current plan are discarded before the next plan is executed. The other possible outcome is that the current plan executes within the isocost level, which means that the actual selectivity has been found, and execution of a 2-optimal plan has been completed.

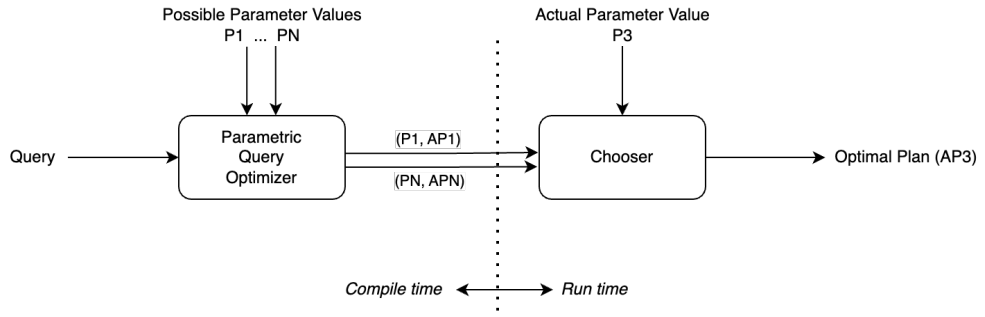
The approach described above for one-dimensional selectivity scenarios can be generalized to work with multiple dimensions. By transforming the "steps", seen in Figure 4.1.1, into isosurfaces with a plan bouquet per isosurface. Then, if it's determined that none of the plans on an isosurface can execute the query within the limits, one simply jumps to a different isosurface with other plans.

The implementation details in [26] present that the implementation of plan bouquets in existing DBMS systems isn't straightforward. Plan bouquets are implemented by creating a "Bouquet Driver" which utilizes the DBMS query optimizer and executor. The database engine must support selectivity injection, abstract plan costing and execution, and cost-limited partial execution of plans.

Regarding performance, the plan bouquet may look slow because it has to both create a lot of POSP plans and at runtime in the worst case execute as many plans as there are in the bouquet. However, without going too far into the technical details, it is as described in [26] possible to beat the performance of the native optimizer for the average cases, while also limiting the worst-code scenarios. This is done with enhancements to the basic algorithm presented above which adds randomization strategies to improve the maximum sub-optimality, replacement of isosurface plan densities, and faster isosurface construction. Plan bouquets can therefore provide performance guarantees with an upper bound of maximum sub-optimality.

4.2 Parametric query optimization

Parametric queries are SQL queries that contain parameters instead of specific values. These parameters are filled with actual values when the query is executed. The overall architecture of parametric query optimization can be seen in Figure 4.2.1. A challenge with parametric queries is that the optimal execution plan may vary significantly depending on the specific parameter values provided, as noted by [28] and [29]. Various algorithms utilizing parametric query optimization have been presented as approaches for this problem [7, 30, 31, 32, 33].



Source: [30]

Figure 4.2.1: Flow-chart of a Parametric Query Optimizer

In [31], the cost of a plan p is modeled as an interval from the lowest cost $l(p)$ to the highest cost $u(p)$ over the parameter space. The algorithm calculates a set of plans where no other plan q has $h(q) < l(p)$. Unfortunately, this technique produces a superset of the optimal set of parametric plans which means that it could generate many more plans than expected in an optimal set of parametric plans. This is both expensive in terms of resource usage and hurts the performance. It also does not provide a way to find regions of optimality.

Several algorithms for generating candidate plans are proposed in [30]. One of these algorithms is to use iterative improvement, which iteratively generates new candidate plans from existing candidate plans. This is combined with a technique that uses randomization which helps the algorithm to achieve an overhead of essentially zero selecting a run-time optimal plan. An issue with this algorithm is however the storage overhead that occurs due to the number of output plan functions from the query optimizer. Another issue with this technique is the lack of guarantees of producing all the optimal parametric plans.

A technique for parametric query optimization that considers optimality regions is presented in [7]. The paper states that "for linear cost functions, each parametric optimal plan is optimal in a convex polyhedral region of the parameter space". This means that, for any linear cost function, the parameter space can be divided into a number of convex polyhedral regions, such that each region has a unique parametric optimal plan. To find the parametric optimal plan for a given query, one can simply determine into which convex polyhedral region the query's parameter values fall. Some issues with this algorithm are the lack of ability to scale to three or more dimensions, and a problem for piece-wise linear cost functions when

memory is added as a parameter.

While [7] only handles queries with a single cost metric, the MPQO algorithm [33] handles queries with multiple cost metrics by using a Pareto optimal approach. MPQO uses Pareto dominance to select the most promising candidate plans. Pareto dominance means that a candidate plan A is said to dominate a candidate plan B if and only if A is better than B in all cost metrics, or A is better than B in at least one cost metric and not worse than B in any of the other cost metrics. At each iteration, MPQO selects the candidate plans that are not dominated by any other candidate plan. Then, new candidate plans are generated by combining the existing candidate plans. The algorithm terminates when it cannot find any new candidate plans that are not dominated by any of the existing candidate plans MPQO is computationally expensive, but argues that it is worth it as it avoids run-time overhead and query optimization.

Chapter 5

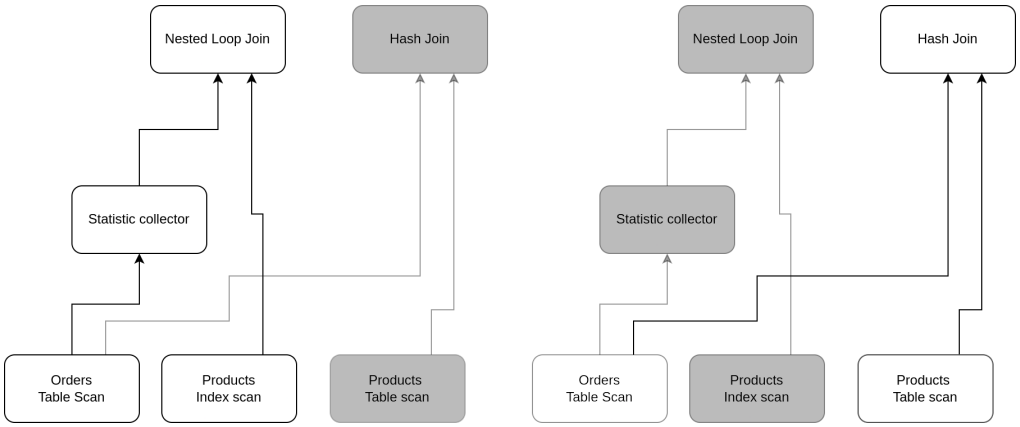
Commercial database systems

In this chapter, we have looked at some popular commercial database systems to see if they have implemented and utilized techniques that handle sub-optimal plans. Some of the techniques do have clear similarities with the techniques we have been through previously in this literature study. We have looked at Oracle DB in Section 5.1, Microsoft SQL Server in Section 5.2, IBM DB2 in Section 5.3, PostgreSQL in Section 5.4, and MySQL in Section 5.5.

5.1 Oracle DB

A technical brief about Oracle DB 19c [34] writes about how Oracle uses adaptive query plans within its system. They use adaptive query plans in order to defer some of the choices of the more complex decisions to runtime when more accurate cardinality estimates are available. This is done by creating a plan that includes statistics collection points much like [5], but they use these points to choose the right operations for the current state of the execution.

The optimizer can dynamically change join methods by creating several sub-plans for various execution plan segments. Consider Figure 5.1.1, where the optimizer chooses an initial default plan for combining the "orders" and "products" tables: nested loops join with index access on the "products" database. There is also a different sub-plan that the optimizer can use to transition to a hash join. In this other strategy, a full table scan is used to reach the "products" table.



Source: [34]

Figure 5.1.1: Example of Oracle DB adaptive query plan

A statistics collector collects execution data and buffers a portion of the incoming rows for the sub-plan during the initial execution. The optimizer chooses which data to gather and how to adjust the plan for different statistical results. It determines an "inflection point", or the statistical value at which both possible courses of action are equally successful. The inflection point is set at 10, for example, if the hash join is favored when the "orders" table scan returns more than 10 rows and the nested loop join is best when it returns fewer than 10 rows. This value is calculated by the optimizer, who also sets up a buffering statistics collector to collect and count up to 10 rows. The join approach is changed if the scan yields at least 10 rows, switching to a hash join, otherwise, it defaults to a nested loops join. Figure 5.1.1 shows how the statistics collector keeps track of and buffers rows produced throughout the "orders" database's entire table scan. The optimizer chooses to apply the hash join because the number of rows from the "orders" database exceeds the optimizer's initial estimate, based on the data collected by the statistics collector.

5.2 Microsoft SQL Server

In the 2017 version of Microsoft SQL server [35], some adaptive query processing methods were introduced.

"Batch Mode Adaptive Join" offers the flexibility of dynamic algorithm selection and the ability to switch query plans to a more optimal join strategy during execution. Specifically, it delays the decision between a

hash join or a nested loop join until the initial input has been scanned. Introducing a novel operator, known as the Adaptive Join operator, plays a pivotal role in this process. It allows for establishing a threshold, which is then compared to the row count of the build join input. If the row count of the build join input is below the threshold, the query plan switches to a nested loop join, as it is deemed more efficient. On the contrary, when the row count exceeds the threshold, the query plan proceeds with a hash join without any alterations. This technique proves particularly beneficial in scenarios where workloads involve frequent shifts between small and large join input scans.

They further introduce "Batch Mode Memory Grant Feedback" as a valuable approach for optimizing memory allocation in query execution. This technique ensures that the allocated memory is sufficient to accommodate all returning rows, offering two key benefits. First, it mitigates the problem of excessive memory grants, which can lead to concurrency issues. Second, it addresses the issue of underestimated memory grants, which can result in costly disk spills. Accurate memory allocation is crucial for maintaining query performance.

In the Batch Mode Memory Grant Feedback process, the optimizer performs a repetition of the workload and re-evaluates the actual memory requirements for the query. Subsequently, it updates the memory grant value for the cached query plan. When an identical query statement is executed in the future, it benefits from the revised memory grant size, leading to improved query performance.

Lastly, they introduced "Interleaved Execution" when SQL Server encounters multistatement table-valued functions (MSTVFs); during the optimization process, it temporarily suspends optimization, executes the relevant subtree first, obtains precise cardinality estimates, and then resumes the optimization process for the remaining operations. This approach ensures that actual row counts are obtained from MSTVFs rather than relying on fixed cardinality estimates, thereby enhancing the efficiency of plan optimizations downstream of MSTVFs. Consequently, workload performance is notably improved.

Additionally, Interleaved Execution allows query plans to adapt dynamically based on revised cardinality estimates, as MSTVFs redefine the unidirectional boundary between the optimization and execution phases of a single query.

5.3 IBM DB2

There are three main parts to AQP support within the DB2 system [36]; Global Statistics Cache (GSC), AQP Request Support, and AQP handler.

The Global Statistics Cache (GSC) functions as a centralized repository within the system, storing statistical data derived from real query executions. Whenever the SQE query engine detects a disparity between the estimated record counts and the observed actual values, it can record an entry in the GSC. This recorded entry serves to furnish the optimizer with improved and precise statistical insights for future optimization processes.

This support operation occurs after the completion of a query. It is executed as a background system task, ensuring that it does not impact the performance of user applications. During this process, the estimated record counts are compared to the actual values obtained. When notable differences are identified, the AQP Request Support saves the observed statistics in the GSC. Furthermore, the AQP Request Support may provide specific recommendations to enhance the query plan for future executions of the query.

The AQP handler is the main component that performs adaptive query processing during runtime. The AQP Handler operates in a separate thread, running in parallel with an active query while monitoring its progress. This handler awakens when a query has been running for at least 2 seconds without yielding any result rows. Its primary function is to examine the real-time statistics generated during the partial execution of the query, diagnose any issues, and potentially rectify joint-order problems originating from inaccurate statistical estimates.

In cases where join order problems are detected, the query can undergo re-optimization using the observed partial statistics, specific recommendations for join order adjustments, or a combination of both. Should this re-optimization yield a new execution plan, the previous plan is terminated, and the query is restarted with the updated plan, provided that the query has not yet produced any result sets.

The main goal of IBM DB2 adaptive query processing is looking for an unforeseen “starvation join” condition. A starvation join arises when a table positioned toward the end of the join sequence filters out a substantial portion of the records from the final result set. Typically, optimizing query performance involves placing the table responsible for eliminating a significant number of rows at the beginning of the join sequence. When AQP detects a table that triggers an unexpected starvation join condition,

it marks this table as the "forced primary table". Information about the forced primary table is preserved for future query optimizations, where it can be leveraged to enhance query performance.

5.4 PostgreSQL

A significant portion of research dedicated to addressing the sub-optimal query plan problem has been conducted within the DBMS PostgreSQL. Techniques such as "Query split" [14], "Smooth Scan" [17, 18], "Plan Bouquets" [26, 27] are all examples of this. The papers that implement their technique in a database system in order to measure results but do not use PostgreSQL, usually implement it in a research database system that is not meant for commercial usage.

Postgres itself does not have any direct form of adaptive query processing. Most of the adaptive query processing is extensions that are either made alongside research or as a proof-of-concept. Postgres professional has an extension that is called Adaptive query processing [37], but it is not defined as we do in this survey. Their implementation is a machine learning-based extension that learns based on previous queries.

5.5 MySQL

There is no information about sub-optimal query plan handling in MySQL that can be found in their official documentation. However, in a blog post from 2014, usage of dynamic range access [38] in MySQL is described. To describe an example of when dynamic range access is used, the following SQL query was used in the blog post: "`SELECT user.user_id, COUNT(message.id) FROM message, user WHERE message.send_date >= user.last_activity GROUP BY user.user_id;`" MySQL must for each user find all messages that the user has not seen. There are two ways this can be done, through a table scan which reads all rows in the table, or through a range scan on the index.

In order to decide between a range scan and a table scan, an analysis of range access is now conducted on the "message" table, utilizing the "user.last_activity" value as input. If the cost calculations indicate that range access is more performant than a table scan, the range access method is employed. This assessment is performed independently for each row in the "user" table. Therefore, if a user recently logged in, it is probable that

the range access method will be utilized to retrieve a few undelivered messages. Conversely, for a who hasn't logged in for a long time, it is likely that a table scan will be employed to retrieve all undelivered messages.

The blog post also shows how range scans can improve performance significantly when "only a small portion of the rows match a non-equality join predicate" [38]. The example shows how the response time drops from ∼44 seconds with table scan to ∼1 second with range scan for the same query.

Chapter 6

Future work

In the course of our survey, several avenues for future research and development have emerged. These potential areas of exploration hold promise for enhancing query optimization processes, ensuring more efficient and effective database management systems.

Although re-optimization techniques were one of the focuses of our survey, more research is necessary to improve and create new tactics. It is imperative to investigate dynamic and adaptive re-optimization techniques that can adjust to shifting query patterns and data distributions over time. Another crucial area for future research is examining the trade-offs between the overhead and the frequency of re-optimization.

How adaptive access method techniques could help reduce the suboptimal query problem was covered in our survey. Subsequent research endeavors should focus on improving current adaptive indexing approaches and creating novel systems that can effectively adapt to changing workload attributes. An area of research that could be interesting is the use of machine learning models to predict access patterns and dynamically modify access ways.

The development of adaptive joining methods requires further study. Join ordering is sometimes one of the leading causes of the sub-optimal query problem. This is why researching how one can perform simple, but effective adaptive join ordering algorithms could show promising results in elevating the sub-optimal query problem.

Eddies' performance at optimizing dynamic queries has shown promise. Subsequent research should focus on eddy-based optimization techniques and investigate their suitability for diverse database structures. Determining the viability of eddy-based techniques in real-world systems will

require examining their scalability and performance in various workload circumstances. Eddies are a complicated operator, and can seamlessly integrate into any database system, so further research could focus on how one could implement such an operator in existing database systems.

One important area of future research is the development of test cases and benchmarks for evaluating the effectiveness of techniques for handling sub-optimal query plans in existing popular database systems like MySQL and PostgreSQL. This would allow researchers and practitioners to compare different techniques and identify the best approaches for different types of queries and workloads.

A notable gap in the current state of research on addressing sub-optimal query plan problems, particularly focusing on re-optimization, adaptive runtime strategies, and multi-plan optimization, is the limited testing and integration within the MySQL database system.

Future research could focus on rigorous testing and integration of these strategies within MySQL. This includes developing dedicated plugins or extensions to facilitate their implementation, along with a comprehensive benchmark against MySQL. To assess real-world applicability and performance implications, future work should place particular emphasis on scalability and compatibility with existing MySQL features.

Query split [14] has an open source implementation of their solution within Postgres, a possible future research project could look into implementing query split within MySQL and see what differs from the implementation and results in PostgreSQL. Another interesting path to take is to see how much of an impact simple swapping of query plan operators has on performance.

Chapter 7

Conclusions

This literature study has examined techniques for handling sub-optimal query plans in databases during execution. Three main categories of techniques have been identified: re-optimization, adaptive run-time strategies, and multi-plan. Techniques currently in use in commercial databases have also been presented.

1. What are the prevailing techniques and algorithms for re-optimizing query plans in databases?

- Re-optimization techniques detect sub-optimality during query execution and generate and execute a new query plan. This can be done by monitoring the execution of the current plan and collecting statistics, or by using a feedback mechanism from the database system. Re-optimization techniques can be very effective, but they can also be expensive, both in terms of overhead and the potential for performance degradation. Chapter 2
- Adaptive run-time strategies modify the execution of the current query plan based on feedback from the database system. This can involve adjusting the order of operations, changing the operators, or switching to a different path access strategy. Adaptive run-time strategies are typically less expensive than re-optimization techniques, but they may not be as effective in correcting sub-optimality. Chapter 3
- Multi-plan techniques execute multiple query plans simultaneously and then select the best plan based on feedback from the database system. This approach can be effective in handling sub-optimality, but it can also be expensive due to the overhead of executing multiple plans. Chapter 4

2. How do these techniques differ in effectiveness, applicability, and resource requirements?

The choice of technique for handling sub-optimal query plans depends on several factors, including the type of query, the database workload, and the desired performance trade-offs. Re-optimization techniques are typically best suited for queries that are executed infrequently or that have highly variable data distributions. Adaptive run-time strategies are a good choice for queries that are executed frequently or that have predictable data distributions. Multi-plan techniques are best suited for queries that are critical to performance and that can tolerate the overhead of executing multiple plans.

3. What are the practical implications and real-world use cases of these techniques?

Some techniques presented in this survey are full systems that implement strategies that handle the sub-optimal query problem(Section 2.3 - Tukwila) or are complex operators or algorithms that take over multiple stages of the query processing process (Section 3.4 and 2.3 - Query split). These techniques are harder to implement and aren't easy to just implement into existing DBMSs. We have also presented simpler algorithms that insert new operators into a query plan that can attempt to handle sub-optimality (Section 3.2, Section 2.3 - Mid query re-optimization). These algorithms are less complex to implement in existing DBMSs and would give results without a lot of refactoring of logic or implementation details. That said all techniques presented in this survey aren't at all plug-and-play solutions that solve the sub-optimal query problem, but the techniques presented are suggestions on how one can implement a solution to alleviate the problem, and when choosing a strategy it is important to take into account how the DBMS is structured and why it produces sub-optimal query plans.

As presented in Chapter 5, some commercial DBMS implement concepts from some of the techniques presented in this survey. OracleDB 5.1 uses concepts from multi-plans (Section 4) to defer complex query plan choices to execution, by producing multiple plans that can be chosen based on the state of execution. Microsoft SQL server 5.2 can adaptively change its join strategy depending on the state of execution, which resembles the techniques presented in Section 3.2 and 3.3. IBM DB2 5.3 can collect statistics during query execution which can be used to perform re-optimziation similar to what is presented in chapter 2. They use techniques that are similar to "Mid query re-optimizaton" (Section 2.3) where collected statistics are used to trigger re-optimization that can suggest a new plan for the current state of the query execution.

References

[1] Hai Lan, Zhifeng Bao, and Yuwei Peng. "A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration". In: *Data Science and Engineering* 6.1 (Mar. 2021), pp. 86–101. ISSN: 2364-1541. DOI: 10.1007/s41019-020-00149-7. URL: https://doi.org/10.1007/s41019-020-00149-7.

[2] Matthew Perron et al. "How I Learned to Stop Worrying and Love Re-optimization". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1758–1761. DOI: 10.1109/ICDE.2019.00191.

[3] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[4] Florian Wolf et al. "On the Calculation of Optimality Ranges for Relational Query Execution Plans". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 663–675. ISBN: 9781450347037. DOI: 10.1145/3183713.3183742. URL: https://doi.org/10.1145/3183713.3183742.

[5] Navin Kabra and David J. DeWitt. "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 106–117. ISSN: 0163-5808. DOI: 10.1145/276305.276315. URL: https://doi.org/10.1145/276305.276315.

[6] Moshe Sniedovich. "Dynamic programming and principles of optimality". In: *Journal of Mathematical Analysis and Applications* 65.3 (1978), pp. 586–606. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(78)90166-X. URL: https://www.sciencedirect.com/science/article/pii/0022247X7890166X.

[7] Sumit Ganguly. "Design and Analysis of Parametric Query Optimization Algorithms". In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Ed. by Ashish Gupta, Oded Shmueli,

and Jennifer Widom. Morgan Kaufmann, 1998, pp. 228–238. URL: http://www.vldb.org/conf/1998/p228.pdf.

[8] Shivnath Babu, Pedro Bizarro, and David DeWitt. "Proactive Re-Optimization". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: Association for Computing Machinery, 2005, pp. 107–118. ISBN: 1595930604. DOI: 10.1145/1066157.1066171. URL: https://doi.org/10.1145/1066157.1066171.

[9] Volker Markl et al. "Robust Query Processing through Progressive Optimization". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 659–670. ISBN: 1581138598. DOI: 10.1145/1007568.1007642. URL: https://doi.org/10.1145/1007568.1007642.

[10] Sourav Sikdar and Chris Jermaine. "MONSOON: Multi-Step Optimization and Execution of Queries with Partially Obscured Predicates". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 225–240. ISBN: 9781450367356. DOI: 10.1145/3318464.3389728. URL: https://doi.org/10.1145/3318464.3389728.

[11] Rohan Jagtap. *Understanding the Markov Decision Process (MDP)*. URL: https://builtin.com/machine-learning/markov-decision-process (visited on Sept. 21, 2023).

[12] Zachary G. Ives et al. "An Adaptive Query Execution System for Data Integration". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. Ed. by Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh. ACM Press, 1999, pp. 299–310. DOI: 10.1145/304182.304209. URL: https://doi.org/10.1145/304182.304209.

[13] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. "Enabling Incremental Query Re-Optimization". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1705–1720. ISBN: 9781450335317. DOI: 10.1145/2882903.2915212. URL: https://doi.org/10.1145/2882903.2915212.

[14] Junyi Zhao, Huanchen Zhang, and Yihan Gao. "Efficient Query Re-Optimization with Judicious Subquery Selections". In: *Proc. ACM Manag. Data* 1.2 (June 2023). DOI: 10.1145/3589330. URL: https://doi.org/10.1145/3589330.

[15] Y.-H. Lee and P.S. Yu. "Adaptive selection of access path and join method". In: *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*. 1989, pp. 250–256. DOI: 10.1109/CMPSAC.1989.65092.

[16] Robert B. Hagmann. "An Observation on Database Buffering Performance Metrics". In: *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*. Ed. by Wesley W. Chu et al. Morgan Kaufmann, 1986, pp. 289–293. URL: http://www.vldb.org/conf/1986/P289.PDF.

[17] Renata Borovica-Gajic et al. "Smooth Scan: Robust Access Path Selection without Cardinality Estimation". In: *The VLDB Journal* 27.4 (Aug. 2018), pp. 521–545. ISSN: 1066-8888. DOI: 10.1007/s00778-018-0507-8. URL: https://doi.org/10.1007/s00778-018-0507-8.

[18] Renata Borovica-Gajic et al. "Smooth Scan: Statistics-oblivious access paths". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 315–326. DOI: 10.1109/ICDE.2015.7113294.

[19] Damla Oguz et al. "Adaptive Join Operator for Federated Queries over Linked Data Endpoints". In: *Advances in Databases and Information Systems*. Ed. by Jaroslav Pokorný et al. Cham: Springer International Publishing, 2016, pp. 275–290. ISBN: 978-3-319-44039-2.

[20] Wenchen Fan, MaryAnn Xue, and Herman van Hövell. *How to speed up SQL queries with adaptive query execution*. May 2020. URL: https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html (visited on Nov. 5, 2023).

[21] Quanzhong Li et al. "Adaptively Reordering Joins during Query Execution". In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 26–35. DOI: 10.1109/ICDE.2007.367848.

[22] Pedro Bizarro and David Dewitt. "Adaptive and robust query processing with SHARP". In: (May 2006).

[23] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 261–272. ISSN: 0163-5808. DOI: 10.1145/335191.335420. URL: https://doi.org/10.1145/335191.335420.

[24] Amol Deshpande and Joseph M. Hellerstein. "Lifting the Burden of History from Adaptive Query Processing". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 948–959. ISBN: 0120884690.

[25] Vijayshankar Raman, A. Deshpande, and J.M. Hellerstein. "Using state modules for adaptive query processing". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 353–364. DOI: 10.1109/ICDE.2003.1260805.

[26] Anshuman Dutt and Jayant R. Haritsa. "Plan Bouquets: A Fragrant Approach to Robust Query Processing". In: *ACM Trans. Database Syst.* 41.2 (May 2016). ISSN: 0362-5915. DOI: 10.1145/2901738. URL: https://doi.org/10.1145/2901738.

[27] 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*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 1039–1050. ISBN: 9781450323765. DOI: 10.1145/2588555.2588566. URL: https://doi.org/10.1145/2588555.2588566.

[28] G. Graefe and K. Ward. "Dynamic Query Evaluation Plans". In: *SIGMOD Rec.* 18.2 (June 1989), pp. 358–366. ISSN: 0163-5808. DOI: 10.1145/66926.66960. URL: https://doi.org/10.1145/66926.66960.

[29] Guy Lohman. "Is query optimization a "solved" problem". In: *Proc. Workshop on Database Query Optimization*. Vol. 13. Oregon Graduate Center Comp. Sci. Tech. Rep. 2014, p. 10.

[30] Yannis E. Ioannidis et al. "Parametric query optimization". In: *The VLDB Journal* 6.2 (May 1997), pp. 132–151. ISSN: 0949-877X. DOI: 10.1007/s007780050037. URL: https://doi.org/10.1007/s007780050037.

[31] Richard L. Cole and Goetz Graefe. "Optimization of Dynamic Query Evaluation Plans". In: *SIGMOD Rec.* 23.2 (May 1994), pp. 150–160. ISSN: 0163-5808. DOI: 10.1145/191843.191872. URL: https://doi.org/10.1145/191843.191872.

[32] Gennady Antoshenkov. "Dynamic Query Optimization in Rdb/VMS". In: *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 1993, pp. 538–547. DOI: 10.1109/ICDE.1993.344026. URL: https://doi.org/10.1109/ICDE.1993.344026.

[33] Immanuel Trummer and Christoph Koch. "Multi-objective parametric query optimization". In: *The VLDB Journal* 26.1 (Feb. 2017), pp. 107–124. ISSN: 0949-877X. DOI: 10.1007/s00778-016-0439-0. URL: https://doi.org/10.1007/s00778-016-0439-0.

[34] ORACLE WHITE PAPER. "The Optimizer In Oracle Database 19c". In: (2019). URL: https://www.oracle.com/docs/tech/database/technical-brief-optimizer-with-oracledb-19c.pdf.

[35] Joe Sack. *Adaptive Query Processing in SQL Server 2017*. URL: https://cloudblogs.microsoft.com/sqlserver/2017/09/28/enhancing-query-performance-with-adaptive-query-processing-in-sql-server-2017/ (visited on Oct. 16, 2023).

[36] IBM. *How AQP works*. URL: https://www.ibm.com/docs/en/i/7.5?topic=processing-how-aqp-works (visited on Oct. 16, 2023).

[37] Postgres Professional. *Adaptive query optimization*. URL: https://github.com/postgrespro/aqo (visited on Nov. 13, 2023).

[38] Jørgen Løland. *Dynamic range access (and recent changes)*. Apr. 2014. URL: https://dev.mysql.com/blog-archive/dynamic-range-access-and-recent-changes/ (visited on Nov. 14, 2023).