ADAPTIVE QUERY PROCESSING:

DEALING WITH INCOMPLETE AND

UNCERTAIN STATISTICS

by

Pedro G. Bizarro

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2006

For Ana, with love

ABSTRACT

The standard Database Management Systems (DBMS) query processing model picks a single non-adaptive plan and executes it to completion. The chosen plan aims to minimize running time by carefully optimizing the use of secondary storage, memory, and CPU. DBMS optimizers estimate plan costs by using statistics–information describing the datasets, the queries, and the system. When statistics needed to cost plans are not available in the database catalog, the optimizer estimates them by using heuristics and default values. These estimates may contain errors and these errors grow exponentially with the number of estimated statistics derived from other estimated statistics. This may lead to selecting query plans that are sub-optimal by several orders of magnitude. Having more information in the catalog (e.g., histograms) reduces the problem but does not scale with the number of relations and attributes in the database. In addition, several hardware and software trends are making this hard problem harder. For example, the optimization space is increasing exponentially because there are more operators to considerer, larger datasets to manage, and more complex queries to optimize. Thus, optimizers are increasingly more likely to select sub-optimal plans.

In the general case, DBMS optimizers may have insufficient information to choose a single, good, non-adaptive query plan. Instead of focusing on providing more information to the optimizer, we propose several Adaptive Query Processing (AQP) techniques as alternatives or extensions to the non-adaptive architecture employed by today's commercial database systems. Our proposals are targeted to: i) correct or avoid query processing problems due to the use of incorrect and partial information at optimization time and ii) collect information not available at optimization time and dynamically determine and assign different plans for different subsets of the data. The work presented here complements, extends, or supersedes previous AQP proposals.

ACKNOWLEDGMENTS

I also thank my future colleagues at the University of Coimbra, in Portugal, Henrique Madeira and Paulo Marques. They have been extraordinary friends, they have helped me immensely maintain, nurture, and extend my ties with Coimbra, and they have already started helping me transition from student to faculty.

My work would also not be possible without the brotherly support and careful reviews of the closely tied Database Group at Wisconsin, especially Ahmed Ayad, Jennifer Beckmann, Vuk Ercegovac, Alan Halverson, Ameet Kini, Kristen Lefevre, and Eric Robinson.

I thank also my good friends in and around the CS department, Alexey&Wendy, Ameet&Sommya, Christina&Richard, Ian&Irene, Jennifer&Brad, Phillip&Corina, Vuk&Madgalena, Sérgio&Heather, Tânia&Zé Santana, Zé&Julia, Alan&Coffee, Eric&Talking, Giordano&Tiramisu, Hao&Dinners, Hogi&Civ, Jaime&Ice-cream, and Rui&Beer. I now know that all those distractions you provided were just a clever ploy to keep my sanity. It worked!

To my family, I own more than what I can express here. My grand-parents, Vovina and Vô Danilo, have been a great example of energy and hard-work. My parents have been nothing short of fantastic. From my childhood years, they have been patiently nurturing the scientist in me, looking up with me things in multiple encyclopedias, feeding my curiosity, strengthening my method, providing guidance, inspiration, and motivation. They even gave me my first computer and taught me to program before I was 10! And Erika, my little sister, by being the popular girl in school while at the same time winning the Portuguese Math Olympics, taught me that you can eat the cake and have the cherry too. Nicely done!

I thank also my delightful, cheerful, beautiful, vibrant baby boy, Diogo, for helping me put things into perspective. My time is better managed if I squeeze bedtime stories and zoo trips between conference deadlines. And those dreadful paper reviews are less scary when, after all is said and done, I can go home to you, be silly and make you smile. And to think there is another baby on the way!

Finally, I want to thank my wife, Ana. Before I met you I did not know that people like you existed. You give so much. So much support, so much laughter, so much love, so much life. Not only my Ph.D. would not be possible without you by my side, my life would be meaningless. Or said another way:

No frio da noite/na solidão da tempestade/na escuridão da viagem/Eu sou o navio.

Cruzo o oceano resoluto, mas já não sozinho

O trovão, a chuva e a água não me assustam. Porque

no frio da tempestade/na solidão da viagem/na escuridão da noite/tu és o caminho.

Sempre que preciso, estás lá.

Sempre. Ou seria,

no frio da viagem/ na solidão da noite/ na escuridão da tempestade/ um navio a afundar.

Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Users and application programs typically use the language SQL to access information in a relation database system (DBMS). As a declarative language, SQL allows the user to specify *what* information from the database is needed without having to specify *how* to retrieve it. It is the responsibility of the DBMS to translate each SQL query into an efficient execution plan.

As shown in Figure 1, SQL queries are first translated into a DBMS internal representation of the query called a *logical plan* through a module called a *parser*.
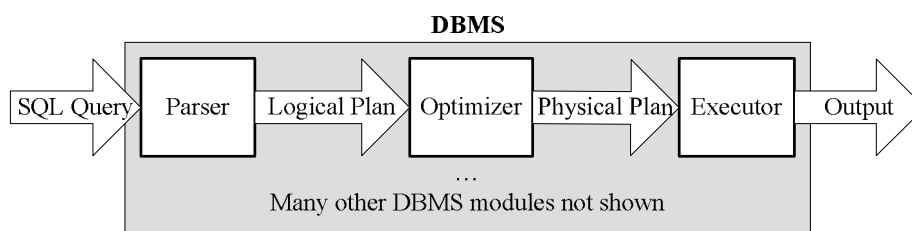
Figure 1 – Query processing overview

Logical plans specify what to do and not how to do it. There are many different alternatives to execute the query specified by a logical plan. Each of those alternatives, called a *physical plan*, has an execution cost. *Query optimization*, performed by the *optimizer*, is the name given to the task of selecting, for each logical plan, the corresponding physical plan with the lowest cost. Finally, *query execution*, performed by the *executor*, takes the chosen optimal physical plan, executes it to completion, and returns the output to the user. The entire process of parsing, query optimization, and query execution is called query processing.

To select an execution plan, the optimizer considers, or *enumerates*, many physical plans (henceforth simply called plans). A plan is a tree of operators. *Leaf oprators* retrieve records from a single relation and pass those records to their parent operator. Almost always, *intermediate oprators* retrieve and join records from two children operators and pass the joined records to their parent operator. (The set of records output by an operator is called an *intermediate relation*; relations are called *base relations* to distinguish them from intermediate relations.) Each operator represents one algorithm. Algorithms that retrieve records from a single base relation–e.g., file-scan (FS) and index-scan (IS)–are called *access methods*. Algorithms that join records between relations–e.g., nested-loops join (NLJ) and sort-merge join (SMJ)–are called *join methods*. Each enumerated plan is different from the rest in one or more operators or in the shape of the tree (i.e., in the way the operators are connected). Figure 2 shows three plans for executing the same query: joining relations R, S, and T. Plan 2 is different from Plan 1 in the operators marked gray. Plan 3 has the same operators of Plan 1, but the inputs to the top operator are reversed.
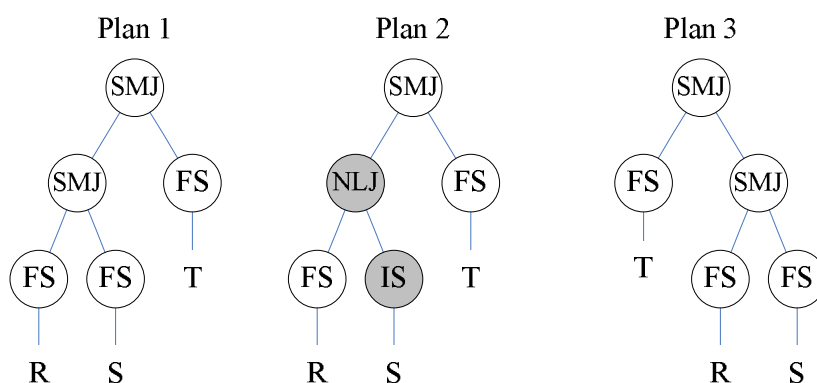
Figure 2 – Three different plans joining relations R, S, and T

The cost of a plan is the sum of the costs of the operators that compose it. The task of the optimizer is to enumerate alternative physical plans, estimate their cost, and *pune* all plans except the one with minimal cost. The exact cost of each operator, and therefore of the entire plan, is very hard to compute. Indeed, the cost of each operator may depend on:

- The state of the system running the DBMS (e.g., the amount of memory available to hold pages of base and intermediate relations)

- The parameters of the SQL query

- The sizes of base relations and intermediate relations

- The data distributions of values in the relations

To correctly cost and prune plans, the optimizer needs to obtain all the information mentioned above. Each piece of information used during costing is called a *statistic*. Some statistics can be obtained from the DBMS. For example, the DBMS might know much memory is available at optimization time. Other statistics can be obtained from the query itself. Still, others can be obtained from the *database catalog*. The database catalog (or simply, catalog) is a repository of meta-information in the DBMS, describing, among others, sizes of base relations and (approximations of) data distributions.

However, the catalog is far from complete: most DBMS data distributions are not represented there. Other times, queries are not fully specified. That is, there are queries, referred to as *prametric queries*, with placeholders for parameters that are known at execution time but not at optimization time. In addition, the state of the system at execution time may be different from the state of the system at execution time. In any of these situations, the optimizer needs to cost plans without having access to all the statistics needed to accurately estimate their cost. Whenever this happens, the optimizer uses a number of heuristics to estimate the unknown statistics which are then used to estimate the cost of operators, and the cost of plans.

1.1. Query Optimization Problems

Sometimes, the inputs to the optimizer may be incorrect or partially missing. When that happens, correctly estimating the cost of all operators in all enumerated plans is virtually impossible and the

optimizer may select a plan that is sub-optimal. Below we describe scenarios that can lead to sub-optimal plan choices.

1.1.1. Statistics not in the Catalog, Heuristics Incorrect

Based on the query, the optimizer determines which statistics to retrieve from the catalog: sizes of base relations, sizes of records, distributions of values, etc. The optimizer then estimates the cost of operators and plans using those statistics. However, frequently, statistics needed to compute costs of operators are not stored in the catalog. When this happens, the optimizer estimates the missing statistics by using a combination of heuristics. For example, if a distribution of values is unknown, it is typically assumed to be uniform. Likewise, if the relationship between two distributions is unknown, they are typically assumed to be independent. However, frequently the assumptions do not hold and the resulting estimates may be incorrect. Throughout the query optimization process, estimated statistics may be used to produce other estimated statistics. For example, to estimate the size of the join of relation R with S, the optimizer will likely estimate the sizes of R and S separately. If the sizes of R and S were incorrectly estimated, the estimate of the size of the join of R with S will also likely be incorrect. It has been shown that, not only this propagation of errors happens, in fact, the propagated error grows exponentially with the number of statistics estimated from other estimated statistics [51]. As the errors in estimated statistics grow, the optimizer is more likely to select a sub-optimal plan.

1.1.2. Parametric Queries

Parametric queries are queries with one or more input parameters that are unknown at optimization time. Different input parameter values may yield different results and may require different optimal plans. Not knowing what values to expect, the optimizer guesses those values and proceeds with optimization.

Again, due to unknown statistics, in this case the input parameters to the parametric query, the optimizer may produce sub-optimal plans.

1.1.3. Detailed Statistical Information not Considered

Another prevalent assumption used during query processing is that there is a single efficient plan for a given query based on current statistical properties of the data. However, different subsets of the data can sometimes have very different statistical properties. In such scenarios it can be more efficient to process different subsets of the data using different plans. Even if catalogs contained statistics at the level of detail necessary, current optimizers are not designed to consider this scenario and may use sub-optimal plans.

1.1.4. Changing Statistics

Finally, traditional query processing has been extended to consider potentially infinite, continuous queries over streams of data. Throughout the life of such long running queries, the conditions of the *Data Stream Management System* (DSMS) and the characteristics of the data can change, possibly changing the optimality of running plans. Dealing with the changing environment typical of DSMSs has been the focus of other work [5, 12, 20, 21, 27, 39, 67] and will not be addressed in depth in this thesis. However, in both DBMS and DSMS, the problems and the architectured solutions have major points in common: incorrect, missing, or changing information lead to the execution of sub-optimal plans, sub-optimality is detected at run-time, and a new optimal plan is generated to replace the sub-optimal plan being executed [10]. Because of these common points, some of the solutions developed in this thesis are also applicable to DSMSs [see Chapter 2, or reference 15].

1.2. Trends Exacerbating Query Processing Problems

The traditional DBMS architecture described above has been widely successful and it has been used virtually by all databases, ever since its introduction, almost three decades ago [79]. However, while the basic DBMS architecture has remained the same, there have been spectacular improvements in hardware and software: the average improvement in price-performance since 1985 is 68%/year (beating Moore's Law at 58%/year) [42]. For example, the best hardware and software combination of 1985 could sort 0.05 GB/$, while the best combination in 2005 could sort 1660 GB/$, a 33200-fold increase in 20 years [75].

With so much increased processing capacity, DBMSs are able to handle larger and larger databases. For example, in 1983, the pioneer Wisconsin Benchmark [14] had 3 relations totaling less than 5 MB while in 2006, the TPC-H [85] site reports systems handling up to 10,000 GB, a two-million-fold increase in dataset size over 20 years.

There have been also many new operators; in 1979 optimizers only had to consider a handful of operators when building plans [79], while in the latest SQL Server DBMS, the optimizer has close to 100 operators to consider [65]. Query size has also increased substantially in the last two decades. Initial systems aimed at joining two or three relations efficiently [14, 79]. Nowadays it is common for real-life DBMS to process queries joining 20 relations.

In summary, datasets have been growing exponentially with time, DBMSs have become very complex systems with dozens of operators, and it is common for queries to access dozens of tables. These trends present significant optimization challenges. The optimization space increases exponentially with the number of relations joined and with the number of operators to consider. The number of statistics needed to correctly cost all possible plans grows exponentially with the number of relations and number of attributes in the relations. Thus, larger and more complex datasets are likely to have more information missing in the catalog. All this makes the job of the optimizer increasingly harder: every optimization call may have to consider and cost thousands of plans, each with tens of operators. In addition, due to better

software and hardware, optimizer mistakes are becoming, not only more common, but also more costly. That is, the percentage difference between an optimal plan and a sub-optimal plan is likely to increase with hardware and software improvements.

Assuming the described trends will continue, we believe query processing will have to be re-architectured in order to be able to deal with the challenges described in Section 1.1. The next section describes the contributions that this thesis makes towards addressing these problems.

1.3. Contributions and Thesis Organization

This thesis proposes several Adaptive Query Processing (AQP) approaches as alternatives or extensions to the non-adaptive architecture employed by today's commercial database systems. Our proposals are targeted to: i) correct or avoid query processing problems due to the use of incorrect and partial information at optimization time and ii) collect information not available at optimization time and dynamically determine and assign different plans for different subsets of the data.

We note that adaptive query processing proposals over DBMSs are not new [5, 26, 28, 40, 46, 55, 56, 58, 61, 63, 83]. The work presented here represents a second-generation of adaptive query processing techniques that complements, extends, or supersedes some of those previous proposals. Our main contributions were developed in four different projects:

- **Content-Based Routing (CBR), described in Chapter 2**. Query optimizers in current database systems are designed to pick a single efficient plan for a given query based on current statistical properties of the data. However, different subsets of the data can sometimes have very different statistical properties. In such scenarios it can be more efficient to process different subsets of the input datasets using different plans. We propose a new query processing technique called content-based routing that eliminates the single-plan restriction in current systems. We present low-overhead, adaptive algorithms that partition input data based on statistical properties relevant to query execution

strategies, and efficiently route individual tuples through customized plans based on their partition. We have implemented CBR as an extension to the Eddies [5] query processor in the TelegraphCQ system [21], and we present an extensive experimental evaluation showing the significant performance benefits of CBR.

- **Proactive Re-Optimization (Rio), described in Chapter 3**. First generation AQP systems used a traditional optimizer to pick a plan, and then reacted to estimation errors and resulting suboptimalities that were detected in the plan during execution. The effectiveness of this approach is limited because traditional optimizers choose plans unaware of issues affecting re-optimization. We address this problem using proactive re-optimization, a new approach that incorporates three techniques: i) the uncertainty in estimates of statistics is computed in the form of bounding boxes around these estimates, ii) these bounding boxes are used to pick plans that are robust to deviations of actual values from their estimates, and iii) accurate measurements of statistics are collected quickly and efficiently during query execution. We present an extensive evaluation of these techniques using a prototype proactive re-optimizer named Rio, implemented in open-source DBMS Predator [81]. In our experiments Rio outperforms current re-optimizers by up to a factor of three.

- **Streaming, Highly Adaptive Run-time Planner (SHARP), described in Chapter 4**. SHARP is a new multi-join, adaptive, relational operator that joins three or more relations of a star-join. SHARP reduces the possible impact of optimizer mistakes by determining which plan to execute independently of optimization estimates. During normal query processing, SHARP collects statistics, and by using a combination of late-binding plan decisions and tuple routing strategies, it is able to change join order and table access methods. However, unlike previous tuple routing operators used for in-memory stream processing, SHARP was designed to process local relations with sizes much larger than available memory. We have implemented SHARP in the open-source DBMS Predator,

and we present an extensive experimental evaluation showing the significant performance benefits that the SHARP operator can provide.

- **Progressive Parametric Query Optimization (PPQO), described in Chapter 5**. Many commercial applications rely on pre-compiled parameterized procedures to interact with a database. Unfortunately, executing a procedure with a set of parameters different from those used at compilation may be arbitrarily sub-optimal. Parametric query optimization (PQO) attempts to solve this problem by exhaustively determining the optimal plans in each point of the parameter space at compile time. However, PQO is likely not cost-effective if the query is executed infrequently or if it is executed with values only within a subset of the parameter space. We propose instead to progressively explore the parameter space and build a parametric plan during several executions of the same query. We introduce algorithms that, as parametric plans are populated, are able to frequently bypass the optimizer but still execute optimal or near-optimal plans. We present an extensive performance evaluation of PPQO using a prototype implementation and SQL Server 2005.

CHAPTER 2

CONTENT-BASED ROUTING

Content-Based Routing (CBR) is a new technique that overcomes the single-plan restriction in current systems. CBR uses low-overhead adaptive algorithms that partition input data and efficiently route tuples through customized plans based on their partition.

2.1. Introduction

The conventional approach to query optimization is to pick a single efficient plan for a query, based on statistical properties of the data along with other factors such as system conditions. In many application domains, different partitions of the data accessed by a query may have very different statistical properties. For example, statistical properties of the observations collected by different sensors in a sensor network environment may be very different [33]. In such cases it can be more efficient to process the different partitions using different plans. In this chapter we propose a new general-purpose query processing technique called Content-Based Routing (CBR) that eliminates the single-plan restriction in current systems. CBR automatically identifies *tuple classes* —partitions of the input data that differ in relevant statistical properties—and processes the query using multiple plans, each of which is customized for an individual tuple class. CBR is low-overhead and it is adaptive, revisiting its decisions as changes in data characteristics are detected.

Adaptive approaches to query optimization have received a great deal of attention recently, with a focus on handling data properties and system conditions that may change while a query is running [5, 12, 20, 21, 27, 39, 67]. Our problem is different: We do not focus on adapting a single plan as data characteristics change, but rather on detecting classes of data characteristics that can be used to route

different data to different plans. Note that even Eddies [5], which can potentially adapt at the tuple granularity, still uses a single plan for (nearly) all tuples at any point of time.

2.1.1. Eddies and Source-Based Routing

Our CBR algorithms are implemented as an extension to Eddies [5]. However, our approach applies to any query processing environment where the data movement can be modeled as streams, e.g., stream systems, regular database systems using iterators [41], and "pull" systems like acquisitional query processors [62]. The Eddy is an operator that routes tuples through a pool of operators until they are processed by all operators or are dropped along the way [5]. The Eddy continuously observes the performance of the operators by collecting statistics at run-time (e.g., selectivity and cost) and routes tuples to the most efficient operator[1]. Since these statistics are potentially changing, the process is automatically adaptive, possibly sending different tuples through different routes.

Without CBR, an Eddy makes routing decisions based on the selectivity of each operator over all tuples the operator has processed recently. Tuples are not differentiated based on content, so all tuples from the same stream source are routed identically. We denote this type of routing as *source-based routing* (SBR).

2.1.2. Motivations for Content-Based Routing

When CBR is added to Eddies, correlations between tuple content and operator selectivity are detected, and they are exploited during routing to eliminate tuples sooner, reduce latency, and improve overall system throughput relative to SBR. Next we motivate CBR using two examples.

Example 2.1. Figure 3(a) is an intrusion detection query for an enterprise network [13, 82]. The lookup table T may contain addresses of subnetworks in the enterprise that are exposed to the public Internet. The byte sequences represent patterns common to a specific type of network attack [13]. Figure 3(b) shows an Eddy for this query with three filter operators–O_1, O_2, and O_3–corresponding to the three conditions, over an incoming stream S of network packets. Operator O_1 performs a prefix-based join on the destination address attribute of incoming S tuples with T. Operators O_2 and O_3 perform the 100-byte and 256-byte sequence matches respectively.

Let c_i denote the current average processing cost per tuple for operator O_i, and let σ_i, $0 \leq \sigma_i \leq 1$, denote the current expected selectivity of O_i.[2] Suppose the following conditions hold for the example: $c_3 > c_1 > c_2$ and $\sigma_3 > \sigma_1 > \sigma_2$. Given these statistics, the Eddy's routing will converge to the ordering O_2, O_1, O_3, i.e., most tuples will follow this route as shown in Figure 3(b).

Now suppose the monitored attack is underway on a subnetwork whose prefix is not in T. (The subnetwork may be secured separately by a firewall.) In this case, σ_2 and σ_3 will be very high, and σ_1 will be very low for packets (tuples) coming from the attacker(s). So, O_1, O_2, O_3 will be the most efficient ordering for processing these "attack packets". For other packets, O_2, O_1, O_3 will remain the best ordering as before. Since an attack happens typically from some group of compromised hosts, CBR can distinguish between the attack and non-attack packets based on the source address, and use the appropriate ordering (Figure 3(c)). Without CBR, the Eddy will continue using the O_2, O_1, O_3 ordering, limiting performance. ∎

---

[1] In reality, the Eddy routes most, but not all, tuples through the route expected to be most efficient (in a process called *exploitation*) and simultaneously routes some few tuples through other routes to discover if any of those other routes has become the most efficient (*exploration*).

[2] Cost is the time spent by the operator processing the tuple. Selectivity refers to the fraction of input tuples passed by the operator.

a) Query: "Track packets with destination address matching a prefix in table *T*, and containing the 100-byte and 256-byte sequences "0xa...8" and "0x7...b" respectively as subsequences"

SQL version using UDFs:
SELECT * FROM packets
WHERE matches(destination, *T*) ← O_1
AND contains(data, "0xa...8") ← O_2
AND contains(data, "0x7...b"); ← O_3

b)

almost all
tuples
follow
this route

Stream of tuples

c)

attack
tuples
follow
this route

non-attack
tuples
follow
this route

Stream of tuples
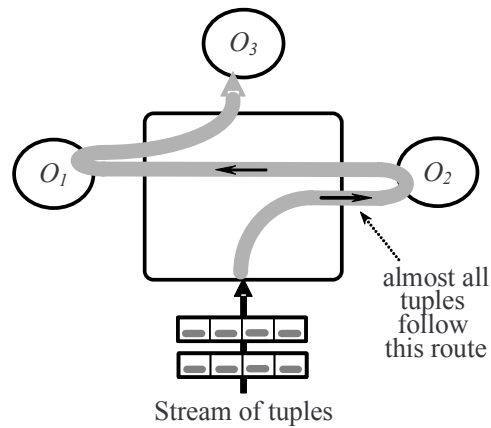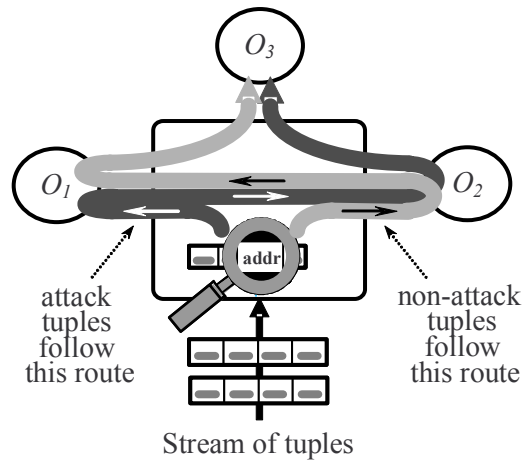
Figure 3 – (a) A continuous query; (b) Eddies; (c) Eddies with CBR

Example 2.2. Consider the following query over a distributed sensor network in a large warehouse building:

SELECT * FROM sensors
WHERE light < 1000 lux[3] AND temperature > 20ºC;

To answer this query, data must be acquired from sensors. However acquiring readings from sensors is a power-consuming operation. Since sensors are power-constrained, one of the main goals of acquisitional systems is to minimize power consumed by data acquisitions [33]. Note however, that sensors that are placed close to windows receive more natural light and likely report higher temperatures than sensors located in interior rooms. Therefore, for those sensors close to windows, the probability that the predicate on light will fail may be higher than that for the temperature predicate. On the other hand, for sensors that are placed in interior locations, the probability that the predicate on light will fail may be lower than that for the temperature predicate. Therefore, instead of using a single fixed order for evaluating the two predicates across all sensors, we may want to use CBR: use different operator evaluation orders depending on the sensor location. For each sensor location, CBR chooses an operator evaluation order that evaluates the most selective operator first. On average, CBR will reduce the number of predicates evaluated per sensor and the number of data acquisitions required, resulting in significant power consumption savings in this setting. ∎

2.1.3. Contributions and Outline

The major contributions of this chapter are:

- In Section 2.3 we define classifier attributes, an important concept in CBR.

---

[3] A value of 1 Lux corresponds to moonlight, 400 Lux to a bright office, and 100,000 Lux to full sunlight.

- In Section 2.4 we present algorithms to automatically and efficiently learn classifier attributes, to partition the underlying data into tuple classes, and to route tuples from these classes optimally through the operators in an Eddy.

- In Section 2.5 we discuss the adaptive nature of our algorithms to handle changes in input data properties and system conditions while the query is running.

- Finally, in Section 2.6 we present an extensive experimental evaluation of CBR using a prototype implementation in TelegraphCQ. Our results show good performance improvements over not using CBR.

2.2. Related Work

Work related to CBR can be grouped into four categories: exploiting correlations among attributes during query processing, adaptive query processing, identifying correlations in large datasets, and computing complex statistical information over data streams.

The work most closely related to CBR is identifying *conditional plans* in an acquisitional query processing system [31, 33]. Like CBR, a conditional plan partitions the input data and processes each partition with a different plan. The approach taken in [33] is to learn a single good conditional plan based on an initial training sample of the data, and then to use this plan unchanged throughout query execution. That initial training is done offline, requires a large amount of collected training data, and learns the conditional plans using complex decision tree building algorithms. On the other hand, CBR uses light-weight machine learning techniques over the streaming tuples that enable a continuously adaptive approach to query processing. Thus, CBR does not require previous knowledge of the data and is not dependent on previous learned models of the world.

While many adaptive query processing systems have been built to date, most of them use a single plan for almost all tuples at any point of time [5, 9, 11, 53, 58, 63]. Some of these systems, including Eddies, on which we have implemented CBR, process almost all of the input tuples using the current best

plan, and the remaining tuples are processed using other plans to track the performance of these plans (to identify plans to change to) or to collect run-time statistics [5, 9]. Ives et al describe a technique that combines hash join and merge join operators to take advantage of mostly-ordered inputs [56]. Tuples following the expected order are routed to the merge join; remaining tuples are routed to the hash join. A final phase joins tuples across the two operators to produce the complete join result. This technique, complementary to ours, can be seen as providing adaptivity within a single join operator while CBR provides adaptivity in a query plan by allowing different join orders.

There has been some recent work on identifying correlations in large datasets. None of this work has been used to identify different plans for processing different partitions of the data for a query. CORDS identifies sets of attributes that are correlated [50]. Deshpande et al use the lack of correlation (independence) among attributes to build compact multi-dimensional histograms [30]. Getoor et al use probabilistic models like Bayesian networks to capture the statistical relationship among attributes so as to compute cardinalities accurately for intermediate results in query plans [36].

There has been work on computing complex statistical information over data streams, for example, decision trees [34], correlated aggregates [37], and histograms [43]. None of this work includes computing correlations between tuple content and selectivities of operators, identifying tuple classes, or finding different plans for different subsets of data.

2.3. Classifier Attributes

Our goal is to identify tuple classes where each class has a different optimal operator order for processing. CBR considers tuples classes that can be distinguished from one another based on tuple content, namely, the attributes in the tuples. In this context, different tuple classes may have different optimal operator orders if the selectivity of one or more operators is correlated with the content of one or more input attributes. Attributes used to distinguish tuple classes are called *classifier attributes*. Informally, an attribute A is called a classifier attribute for an operator O if the content of A is correlated with the

selectivity of *O*. As illustrated by Example 2.3, CBR is based on identifying and exploiting such classifier attributes.

Example 2.3. Consider an input stream *S* processed by three operators O_1, O_2, and O_3. Let *A* be an attribute of tuples in *S* which takes one of three values *a*, *b*, or *c* with equal probability. Table 1 shows the respective selectivities of O_1–O_3 for the tuple classes with *A=a*, *A=b*, and *A=c*, and the overall selectivity of each operator on *S* tuples. Assuming O_1–O_3 have the same execution costs, if only overall selectivities are considered, then the best ordering for *S* tuples is O_1, O_2, O_3. However, note that the selectivity of O_2 is correlated with the value of *A*: the selectivity of O_2 for *A=a* and *A=b* is much lower than O_2's overall selectivity, and it is much higher for *A=c*. Therefore, for tuples with *A=a* or *A=b*, the ordering O_2, O_1, O_3 will outperform O_1, O_2, O_3, while O_1, O_3, O_2 will outperform O_1, O_2, O_3 for tuples with *A=c*.　　■

Table 1 – Content specific selectivities

Value of *A*	σ_1	σ_2	σ_3
A=a	32%	10%	55%
A=b	31%	20%	65%
A=c	27%	90%	60%
Overall	30%	40%	60%

The degree of correlation between two distributions may be specified in a number of ways [66]. We use a specification from Information Theory which is based on the concept of *gain ratio* [66], described next.

Let *R* be a random sample of tuples processed by an operator *O*. (We assume all operators are filters; an extension to non-filter operator is discussed in Section 2.4.5.) Let σ be the overall selectivity of *O* for tuples in *R*. Each tuple in *R* belongs to one of two *classes*: tuples that *O* passes and tuples that *O* drops. The *entropy* [66] of *R*, which is an information-theoretic metric used to capture the information content of *R*, is defined as:

$$Entropy \ (R) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{1}$$

where c is the number of classes in R and p_i is the fraction of R belonging to class i. In our case $c=2$, corresponding to the tuples passed and dropped by O, so $p_1=\sigma$ and $p_2=1-\sigma$ respectively. Therefore:

$$Entropy \ (R) = -\sigma \log_2 \sigma - (1-\sigma) \log_2 (1-\sigma) \tag{2}$$

Let A be an attribute of tuples in R. Let $v_1,\ v_2,\ ...,v_d$ be the distinct values of A in R. The information gain of A with respect to R, which represents the increase in information about R gained by knowledge of A, is defined as [66]:

$$Informatio\,nGain(R, A) = Entropy \ (R) - \sum_{i=1}^{d} \frac{|R_i|}{|R|} Entropy \ (R_i) \tag{3}$$

Here, R_i is the subset of R with $A=v_i$ and $|R|$ ($|R_i|$) is the number of tuples in R (R_i). Gain ratio is a normalized representation of information gain [66]:

$$SplitInfor\ mation\,(A) = -\sum_{i=1}^{d} \frac{|R_i|}{|R|} * \log_2 \frac{|R_i|}{|R|} \tag{4}$$

$$GainRatio\,(R, A) = \frac{Informatio\,nGain(R, A)}{SplitInfor\ mation(A)} \tag{5}$$

Gain ratio is used widely in decision-tree learning algorithms (e.g., ID3 [66]) to determine the attribute that best classifies a given data set. Since classifier attributes serve a similar purpose in our case, our formal definition of a classifier attribute is based on gain ratio.

Definition 3.1 (Classifier Attribute) An attribute A is a classifier attribute for an operator O if for any large random sample R of tuples processed by O, we have $GainRatio(R, A) > \gamma$, for some threshold γ. ■

Example 2.4. We revisit Example 2.1. Let Table 1 now represent the selectivities computed from random samples R_1, R_2, and R_3 of tuples processed by operators O_1, O_2, and O_3 respectively. Since A takes one of values a, b, or c with equal probability, the samples will contain tuples with $A=a$, $A=b$, and $A=c$ in

roughly equal proportion. We can use Equations (2) – (5) to compute the gain ratio of attribute A with respect to R_1, R_2, and R_3: $GainRatio(R_1, A) = 0.33$, $GainRatio(R_2, A) = 0.63$, and $GainRatio(R_3, A) = 0.37$. Notice that $GainRatio(R_2, A)$ dominates the others because of the strong correlation between the selectivity of O_2 and the content of A. ∎

Our definition of classifier attributes extends to *classifier attribute sets* where the selectivity of an operator is correlated with a set of attributes instead of with any single attribute in that set. That is, tuple classes in the input may be determined by a set of attributes instead of a single attribute. We do not consider classifier attribute sets in this work; instead we focus on single-attribute classifiers. Note, however, that CBR considers multiple single-attribute classifiers when making routing decisions. While some of our techniques extend directly to classifier attribute sets, we defer a detailed exploration of this issue to future work.

2.4. Learning Routes Automatically

We are now ready to consider the problem of learning good content-based routes automatically for the CBR framework introduced in Section 2.3. We will consider a single input stream S with tuples having attributes C_1, C_2, …, C_k that are processed by operators O_1, O_2, …, O_n, and describe our *Content-Learns* algorithm to learn good content-based routes automatically in this setting. For now we will consider all operators O_1, O_2, …, O_n, and for each operator, we consider all attributes C_1, C_2, …, C_k as potential classifier attributes for CBR. In Section 2.4.4 we will present heuristics to prune the space of attributes and operators that we consider for CBR. Content-Learns consists of two continuous, concurrent steps:

1. **Optimization**: In this step, for each operator $O_\ell \in O_1$, …, O_n, if one or more attributes in C_1, …, C_k are classifier attributes for O_ℓ, then we keep track of the best classifier attribute for O_ℓ. Informally, we identify the attribute in C_1, …, C_k based on whose content we can make the best routing decisions with respect to O_ℓ. The operator-attribute combinations identified during optimization are used for

CBR by the routing step as described in Section 2.4.2. Details of the optimization step are described in Section 2.4.1.

2. **Routing**: In this step we perform CBR using the current operator-attribute combinations identified by the optimization step. If the selectivity of operator O_ℓ is not correlated with the contents of any attribute, then we do not use any O_ℓ-attribute combination but instead make routing decisions regarding O_ℓ using the selectivity of O_ℓ alone. Our routing algorithm for CBR is described in Section 2.4.2.

2.4.1. The Optimization Step of Content-Learns

The goal of optimization is, for each operator $O_\ell \in O_1, \ldots, O_n$, to identify the best classifier attribute for O_ℓ in $C_1, \ldots, C_k$. We cycle through the operators in a round-robin fashion, so each operator is considered periodically. When we consider operator O_ℓ, termed *pofiling* O_ℓ, we identify the best classifier attribute for O_ℓ. To identify the classifier attributes for O_ℓ, we have to measure the gain ratio of $C_1, \ldots, C_k$ based on a random sample of tuples processed by O_ℓ; recall Section 2.3. To collect this random sample R when O_ℓ is profiled, the Eddy routes a fraction of input tuples to O_ℓ before they are routed to any other operator, and notes whether O_ℓ dropped each such tuple or not. (Note that we profile operators using tuples straight from the input stream. However, in some scenarios it may make sense to profile tuples after they have been filtered by some operators. We can extend our profiling to track such conditional selectivities as in [9] which we intend to do as future work.)

Our profiling technique requires the specification of two parameters: a probability P for sampling an input tuple so that it will be routed first to O_ℓ, and a sample size to fix $|R|$. Once R has been collected, we can compute $GainRatio(R, C_j)$ for each $C_j \in C_1, \ldots, C_k$, to determine the classifier attributes for O_ℓ. If there are two or more such attributes, then the attribute with maximum gain ratio is the best classifier attribute for O_ℓ. Details of our implementation for profiling O are outlined next.

Let D_j denote the domain of potential classifier attribute C_j. For each C_j we choose a **partitioning function** f_j that partitions D_j into d partitions. If C_j is a discrete-valued attribute, we choose a hash function that maps any $v \in D_j$ to one of d buckets. If C_j is a continuous-valued attribute, we maintain running estimates of $\max(D_j)$ and $\min(D_j)$ and use a range-partitioning function to map any $v \in D_\ell$ into one of d partitions. Without loss of generality, let $v_1, v_2, \ldots, v_d$ denote the d partitions of each domain. (Note that, e.g., partition v_1 of domain D_1 is not necessarily the same as partition v_1 of domain D_2.)

Content-Learns maintains the following run-time data structures, as shown in Figure 4.
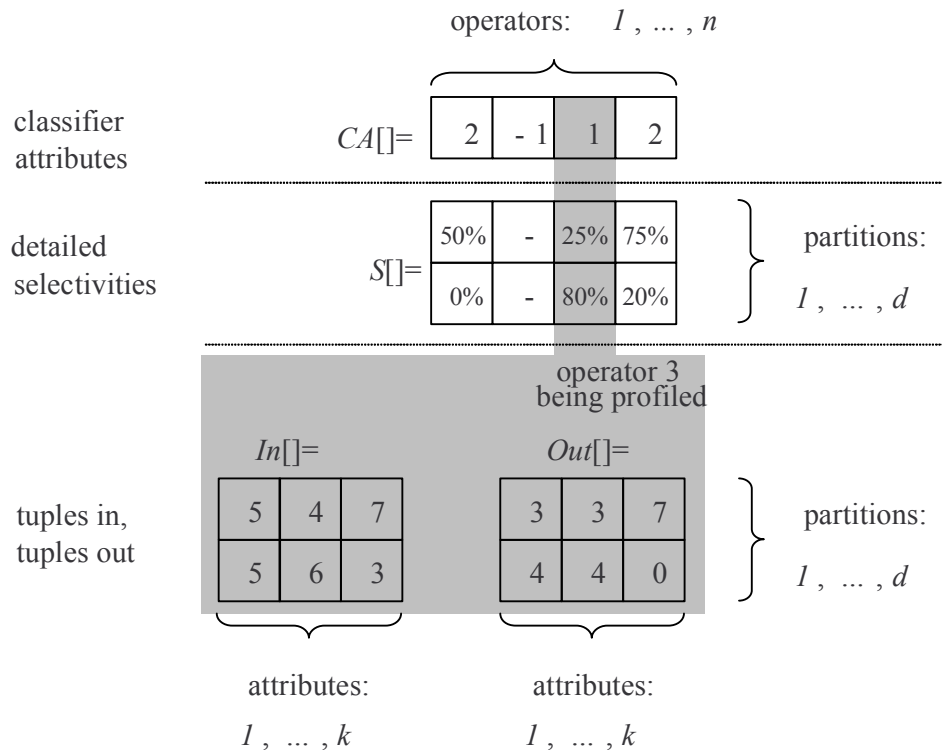
Figure 4 – Run-time data structures

1. **Classifier Attribute Matrix, *CA*[].** CBR keeps an array that, for each operator O_ℓ, stores the attribute index of the best classifier attribute, i.e., the attribute with highest gain ratio for O_ℓ. If O_ℓ has no classifier attributes, CBR assigns $CA[\ell] = -1$. CBR recomputes $CA[\ell]$ after R random sample tuples

are used to profile operator O_ℓ. In Figure 4, the classifier attribute for operator 3 (marked in gray) is attribute 1.

2. **Tuples In, *In*[], and Tuples Out, *Ot* [], Matrices**: These matrices track which tuples in which partitions of all attributes pass (increments both *In*[] and *Out*[] entries) or fail (increments only *In*[] entries) the operator being profiled. For each one of the *R* random sample tuples, *k* entries are updated in each one of these matrices. The entries to be updated are (j, i), with $j = 1,\ldots,k$, and $v_i = f_j(t.C_j)$.

3. **Detailed Selectivities Matrix, *S*[]**. Each column in this matrix stores the running selectivities for an O_ℓ–C_j operator–classifier-attribute pair. Entries in the matrix are updated at two different times:

   (i) *Run-time:* Each time a tuple passes or fails an operator, one entry in this matrix is updated.[4] For a tuple *t* being processed by O_ℓ, the column to update in the matrix is l, and the row is $v_i = f_j(t.C_j)$, with *j* being the index of the classifier attribute for O_ℓ, i.e., $j = CA[\ell]$.

   (ii) *Initialization:* After profiling operator O_ℓ has been completed and its classifier attribute C_j found, CBR updates O_ℓ's column: $S[\ell,i] \leftarrow Out[j,i]/In[j,i]$, with $i=1,..,d$. If $In[j,i]=0$, then $S[\ell,i] \leftarrow W[\ell]$, where $W[\ell]$ is the overall selectivity of operator O_ℓ as described next.

4. **Overall Operator Selectivities, *W*[]**. This matrix (not shown in Figure 4) is non-CBR specific information and it is kept both by CBR and by the non-CBR implementation in TelegraphCQ. $W[\ell]$ tracks the recent overall selectivity of operator O_ℓ over all tuples processed by O_ℓ.

Once we have collected the random sample *R* of tuples processed by operator O_ℓ while profiling O_ℓ, we can compute *GainRatio*(R,C_j) (Equation (5)) for all $C_j \in C_1,\ldots,C_k$ using matrices *In* and *Out*. From

---

[4] The formula used to update selectivity after a tuple is known to pass or fail an operator is: selectivity = selectivity * α + pass * (1- α), where selectivity is a percentage between 0 and 100, pass is 100 if the tuple passes the operator or 0 if it is dropped, and $\alpha = 0.95$.

Equation (2), *Entropy* (*R*) depends only on the overall selectivity of O_ℓ over *R*, which is the number of output tuples over all tuples profiled: $\left(\sum_{i=1}^{d} Out [j,i]\right)\Big/|R|$ for any *j*.

Similarly, *Entropy* (R_i) in Equation (3) for *InformationGain*(*R*,C_j) depends only on *In* and *Out*. Finally, |R_i| in Equations (3) and (4) for *InformationGain*(*R*,C_j) and *SplitInformation* (C_j) is equal to *In*[*j*,*i*].

So far we have seen how the classifier attributes for O_ℓ can be determined by profiling O_ℓ. If there are one or more such attributes, then the attribute with maximum gain ratio, denoted C_{max}, is the best classifier attribute for O_ℓ. Note that after computing gain ratio values for $C_1,\ldots,C_k$ while profiling O_ℓ, we may realize that O_ℓ has no classifier attributes. In either case, we move on to profile the next operator in our round-robin schedule.

2.4.2. The Routing Step of Content-Learns

In this section we describe how we extended the original Eddy routing algorithm to incorporate the operator-attribute combinations identified in the optimization step for CBR. This algorithm routes tuples to operators according to a probability that is inversely proportional to the operators' selectivities (stored in matrix *W* in our implementation). We call this algorithm *Source-Based Routing* (*SBR*).[5]

When an Eddy using Content-Learns has to route a tuple *t* to one of operators $O_1,\ldots,O_n$, the Eddy routes *t* to the operator with minimum value σ, where σ is defined as follows for an operator O_ℓ:

- If O_ℓ is tagged with classifier attribute C_j, then σ is the expected selectivity of O_ℓ for tuples *t'* with $f_j(t'.Cj)=f_j(t.C_j)$, which is equal to *S*[ℓ,*i*] where $f_j(t.C_j)=v_i$ and *j*=*CA*[ℓ]. (We have used the same notation as in Section 2.4.1.)

---

5 We call this algorithm Source-Based Routing because without looking at the content, an Eddy treats all tuples coming from the same source the same way.

- If O_ℓ is not tagged with a classifier attribute, then σ is $W[\ell]$, the expected overall selectivity of O_ℓ, which is the same value as used by the SBR algorithm.

Intuitively, for operators that have a classifier attribute, CBR uses the content-specific selectivity of the operator while making routing decisions. The content-specific selectivity is available from the selectivity matrix for the operator. For operators that do not have a classifier attribute, CBR uses the overall selectivity of the operator across all tuples as done by SBR.

2.4.3. Overheads and Benefits of CBR

There are two forms of overhead associated with CBR: the *routing overhead* of evaluating content-based conditions while making routing decisions, and the *learning overhead* of learning and maintaining good routes automatically. The routing overhead was designed to be very low, as it is incurred each time a tuple is routed by the Eddy. The learning overhead is amortized across a large number of tuples as this overhead is incurred once after |R| sample tuples are observed. Section 2.6.8 presents experiments where the overheads of CBR can be observed to be very low.

The benefit of CBR comes from finding routes that drop tuples sooner. As such, the benefit of CBR is proportional to the percentage of time that a query spends evaluating operators. In Section 2.6.7 we explore the performance of CBR while varying operator costs.

2.4.4. Pruning Operators and Attributes

So far we considered all attributes and all operators as potential candidates for CBR. We now describe some heuristics to prune this space. These heuristics likely reduce the learning overhead of CBR significantly without any noticeable effect on the quality of content-based routes.

CBR applies when optimal operator orderings differ across input tuple classes. If an operator is very cheap or very selective relative to the other operators, or both, then its position will mostly remain

unchanged across the orderings. This intuition translates into an effective pruning heuristic where we do not consider very inexpensive or very selective operators for CBR. Similarly, we can ignore operators that are very expensive or not very selective with respect to the other operators because their position is likely to remain unchanged across those orderings as well.

Similar to pruning operators, there are some effective heuristics to prune the attributes considered for CBR. For example, we can ignore monotonically increasing (or decreasing) attributes such as timestamps or sequential identifiers which typically are generated synthetically. Discrete-valued attributes with large domains, e.g., a comments string attribute, can also be ignored. (It is advisable to ignore long attributes as classifier attributes for CBR to keep routing overhead low.) While it is not hard to detect such attributes automatically, the required information often is available from the schema definitions.

2.4.5. CBR for Non-Filter Operators

We have focused so far on filter operators that either pass or drop an input tuple. This class does not capture, for example, non-foreign-key join operators, limiting the scope of our techniques. However, our techniques apply to non-filter operators with one simple modification. We have used the filter property of an operator only to compute entropy in Equation (2) which contributed to the gain ratio value used to identify classifier attributes. The two-class notion of passed and dropped tuples is meaningless for non-filter operators whose "selectivity"–the expected number of tuples produced per input tuple–can be any non-negative real number. Our real purpose here is to quantify the skew in content-specific operator selectivities with respect to the overall selectivity. Gain ratio is one proven technique to quantify this skew. There are other techniques, e.g., variance, which apply to non-filter operators. Therefore, our techniques for CBR apply to non-filter operators provided the gain-ratio-based test for classifier attributes is replaced by an appropriate test that applies to non-filter operators.

2.5. Adaptivity

Since the Eddies architecture has been designed to support adaptive processing, a relevant question to ask is how our extensions to support CBR in Eddies affect adaptivity. Adaptivity refers to the ability of the system to find an efficient plan quickly for the new data and system characteristics when these change. The changes in the data stream characteristics that can affect routing decisions are changes in operator selectivities and changes in correlations between attributes and operators' selectivities.

CBR increases both the learning overhead and the routing overhead of Eddies. Fundamentally, reducing run-time overhead is at odds with improving adaptivity [21]. The approach we have adopted is to keep run-time overhead as low as possible while being as adaptive as the SBR routing policy in TelegraphCQ.

To be as adaptive as SBR, CBR keeps the operator selectivity matrix W up to date. Note that W is common across both policies. In exchange, CBR settles for slower adaptivity with respect to changes in classifier attributes by profiling only one operator at a time. This design decision may fail to detect a new correlation if the classifier attribute for an operator changes between two of its profiling phases. However, in spite of this decision, CBR is designed to never be less adaptive than SBR. Example 2.5 illustrates why.

Example 2.5: CBR as adaptive as SBR. Consider that CBR finds C_j to be the classifier attribute for O_ℓ. Then, when routing tuple t, CBR assumes the selectivity of O_ℓ to be $S[\ell,i]$, with $v_i=f_j(t.C_j)$. However, it may be the case that the correlation between C_j and O_ℓ no longer holds since O_ℓ was last profiled due to one of two reasons:

- *No attribute is correlated with O_ℓ.* If this is the case, then the selectivity of O_ℓ is given by $W[\ell]$ and not $S[\ell,i]$. However, if C_j is not actually correlated to O_ℓ, then all entries $S[\ell,i]$, with $i=1,...,d$ will quickly converge to $W[\ell]$ (because CBR updates entries in $S[]$ as frequently as those in $W[]$).

- *Another attribute is correlated with O_ℓ.* If this is the case, then we have an argument for more aggressive content-based routing statistics tracking (e.g., profiling multiple operators simultaneously as done in [8]), not less. In any case, given that C_j is not correlated with O_ℓ, all entries $S[\ell,i]$, with $i=1,...,d$ will still quickly converge to $W[\ell]$. ∎

The assumption behind the current CBR design is that operators' selectivities change more frequently than the correlations between operators and tuple content. As such, selectivity is tracked continuously (quick to detect changes) while profiling is performed only for a sample of the tuples (slower to detect changes). For example, in the real-life dataset, we observed changes in selectivity from 1% to 96% in one operator while the best classifier attribute for that operator stayed the same (Section 2.6.6).

2.6. Experimental Evaluation

We now describe an experimental evaluation of our CBR techniques using a prototype implementation in TelegraphCQ [21]. We evaluate the CBR prototype using both synthetic and real life datasets. The synthetic dataset is used to evaluate CBR by varying parameters hard to control in a real-life dataset: skew, selectivity, and aggregate selectivity. The real-life dataset is used to evaluate CBR's adaptivity and performance under varying operator costs and overhead.

2.6.1. Datasets

The prototype implementation of CBR was evaluated with the synthetic and real-life datasets described below:

- **_Stream-Star_**: We created a synthetic benchmark, Stream-Star, based on a star schema. Instead of a central fact table, we used a data stream S.[6] Our experiments use N-way join queries of the following form which join incoming S tuples with N dimension tables d1, d2, …, dN:

SELECT *
  FROM stream S, d1, d2, …, dN
WHERE s.fkd1 = d1.pk   // Operator Op1
   AND s.fkd2 = d2.pk   // Operator Op2
 …
   AND s.fkdN = dN.pk;   // Operator OpN

  Each stream consisted of 100,000 tuples. Depending on the query, between two and eight dimension tables containing 10,000 tuples each are used. Stream _S_ contains tuples with a single classifier attribute, _attrC_, which is correlated with the selectivities of all operators. (We note that in the real-life dataset described next, different operators can have different correlated attributes and these correlations can change, appear, or disappear with time. CBR worked equally well in both settings.). Our stream generator is able to produce tuples with any kind of non-independence between the classifier attribute _attrC_ and the selectivity of the join operators. For example, it can generate a stream with the characteristics shown in Table 1 (on page 17).

- **_Lab_**: The _Lab_ dataset is a trace of readings from 54 sensors in the Intel Research, Berkeley Lab. The readings were taken between end of February and beginning of April of 2004. The schema consists of one single stream, _sensors_. Tuples in the stream have attributes _light_, _humidity_, _temprature_ , _voltage_, _sensorID_, and timestamp information (year, month, day, hour, minute, and second) [33]. We cleansed this dataset by removing tuples with missing values or impossible values (e.g., negative humidity) that sometimes happen when the sensor batteries run low. There are 2.2 million records in the

---

[6]   A star schema was chosen for two reasons: (i) queries over streams normally refer to one single stream source that joins with zero or more local tables; and (ii) data stream applications have streams that represent facts, e.g., traffic information, which then join with dimensions, e.g., speed sensors and cars.

cleansed dataset. For this dataset the readings are sent to TelegraphCQ in generation order, as they would if the tuples were being collected from the sensors in real-time.

2.6.2. Algorithms, Metrics, and Default Values

Section 2.4.2 described most of the details of our implementation of CBR in TelegraphCQ, *Content-Learns* (*Learns* in the figures), and the non-content-based SBR algorithm in TelegraphCQ. To illustrate the differences between the learning overhead and the routing overhead of CBR, in the Stream-Star experiments we include a routing algorithm called *Content-Knows* (*Knows* in the figures) which does not need to learn classifier attributes automatically. Instead, Content-Knows is a theoretical bound that simulates a routing policy that is "told" which attribute is the best classifier for each operator and what is the best routing order for each class.

In addition to the running time, we also use the number of routing calls as a performance metric. The number of routing calls shows a clear picture of the quality of the routing algorithm: a bad routing algorithm will miss opportunities to route a tuple to the most selective operator, e.g., a tuple may be routed several times before being dropped. In addition, the improvement in routing calls due to using Content-Learns instead of SBR acts as a ceiling in the improvement we can expect in total running time.

Unless otherwise stated, the default values used in the experiments are the ones listed in Table 2 and the results are averages over five runs.

Table 2 – Defaults used in experiments and graphs

Parameter	Defaults	Comment
P	6%	Tuple sampling probability
|R|	150 tuples	Sample size to compute GainRatio
d	24	Number of buckets in hash partitions
Confidence	95%	Confidence intervals in graphs

2.6.3. Varying Skew

In this section we use the Stream-Star dataset to show how CBR performs in the presence of skew among the content-specific selectivities of operators. We set the stream to have as many tuple classes as joins. (Each tuple class is identified by a unique value of attribute *attrC*.) Skew was created by setting the selectivity of one operator to *A*, and setting the selectivity of the all other *N-1* operators to *B*, as shown in Table 3.

  A was varied from 5% to 95% with *B* varying accordingly such that the overall aggregate selectivity remained constant at 5%. (Section 2.6.4 reports experiments where selectivities are chosen randomly and Section 2.6.5 reports experiments where the aggregate selectivity is varied.) There were 8 other attributes in tuples in the stream not correlated with the selectivities of the operators. Thus, Content-Learns must learn that, among all these attributes, *attrC* is the best classifier attribute for all operators. The *N*-way join query was run for two, four, six, and eight join operators. Due to space constraints, we only show results for two and six joins in Figure 5 and Figure 6.

  Note that when *A<B* (negative skew), a good routing policy should exploit the selectivity skew by routing tuples first to the lower selectivity operator corresponding to *A*. When *A>B*, a good routing algorithm will avoid the operator with selectivity *A* and route tuples through all the other operators first.

  Overall, the higher the skew between *A* and *B*, especially when *A<B*, the greater the extent by which Content-Learns outperforms SBR. At most, Content-Learns outperforms SBR by performing 67.8% fewer routing calls (with eight operators and the largest skew). Across all experiments, when *A<B*, Content-Learns required on average 26.9% fewer routing calls and when *A>B*, Content-Learns required 10.2% fewer routing calls. That is, it is more useful to know which operator is different by being more selective than it is to know which operator is different for being less selective. This happens because more selective operators will appear earlier in operator orderings affecting more tuples and having greater performance impact than less selective operators that appear later in the operator order.

Table 3 – Selectivities for class/operator pairs

	Op1	Op2	…	OpN
Class 1	A	B	…	B
Class 2	B	A	…	B
…	…	…	…	…
Class N	B	B	…	A
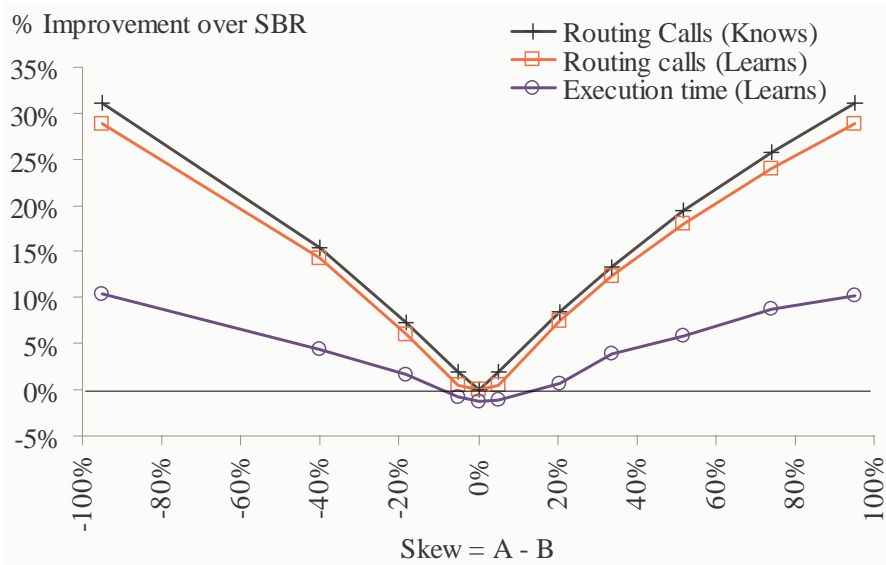
Figure 5 – Improvement with varying skew (2 joins)
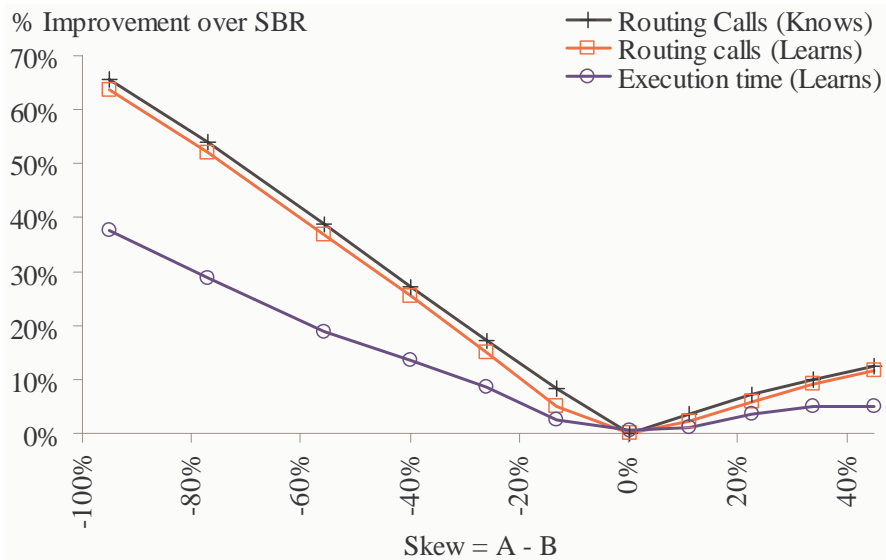
Figure 6 – Improvement with varying skew (6 joins)

2.6.4. Varying Selectivities

In Section 2.6.3, the choice of selectivities made routing tuples to operators difficult for SBR because all operators appeared to be equally selective. Each operator had selectivity *A* for one class of tuples and *B* for all other classes. Thus, in all cases, to SBR, all operators appeared to have a selectivity of (A + B * (N-1))/N, for the N-way join query.

　　We continue to use the Stream-Star dataset in the following experiments. Each query was run against 50 different streams. Attribute *attrC* was correlated with the selectivities of the operators. However, this time we assigned random selectivities to each operator. As before, we included additional attributes (constants, sequences, and foreign keys) whose content was not correlated with any of the selectivities of the operators. Figure 7 shows that Content-Learns is very effective at learning the right classifier; out of the 16 million routing calls, Content-Learns used the wrong classifier only 6.4% of the time.
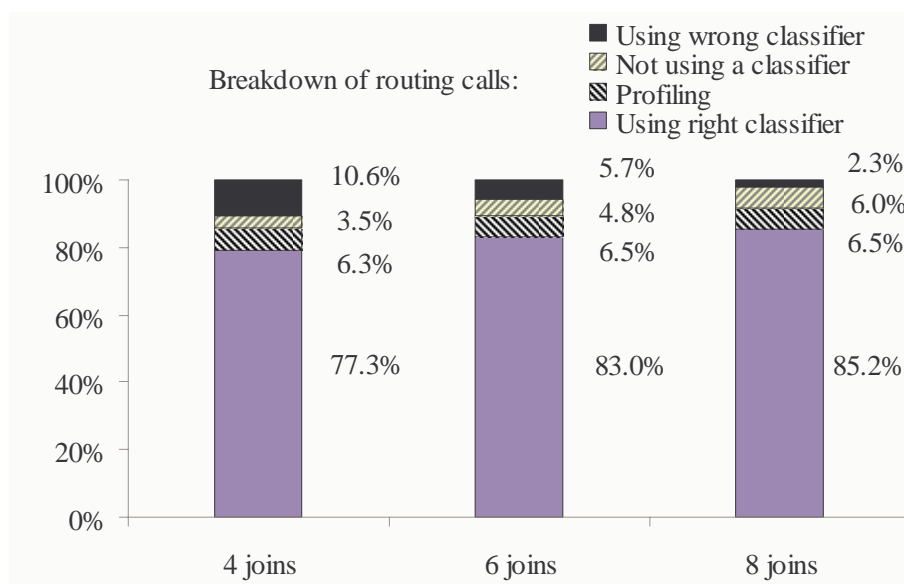
Figure 7 – Breakdown of routing calls

% Improvement over SBR
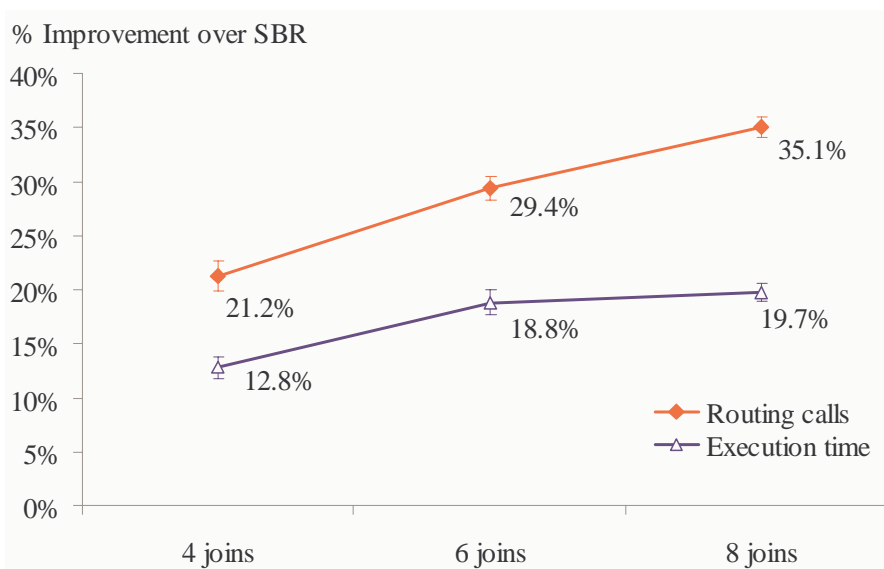
Figure 8 – Improvement with random selectivities

Figure 8 shows the improvement of Content-Learns over SBR both in terms of routing calls and total execution time. Note that the larger the number of operators involved, the more opportunities are available for improvement.

2.6.5. Varying Aggregate Selectivity

In Section 2.6.3 the overall aggregate selectivity was kept at 5%. In Section 2.6.4 the operator selectivities were randomly selected without any guarantee on the aggregate selectivity. On average, the aggregate selectivity was 8% across all streams. This section explores the space of aggregate selectivities from 5% to 35%. For this experiment, we ran a 6-way join query over Stream-Star with the operators having random selectivities under the restriction that the overall aggregate selectivity was kept at some predetermined value. The aggregate selectivity is varied in Figure 9. Each point in the plot represents the average improvement of CBR over SBR for 50 streams of 100,000 tuples each.
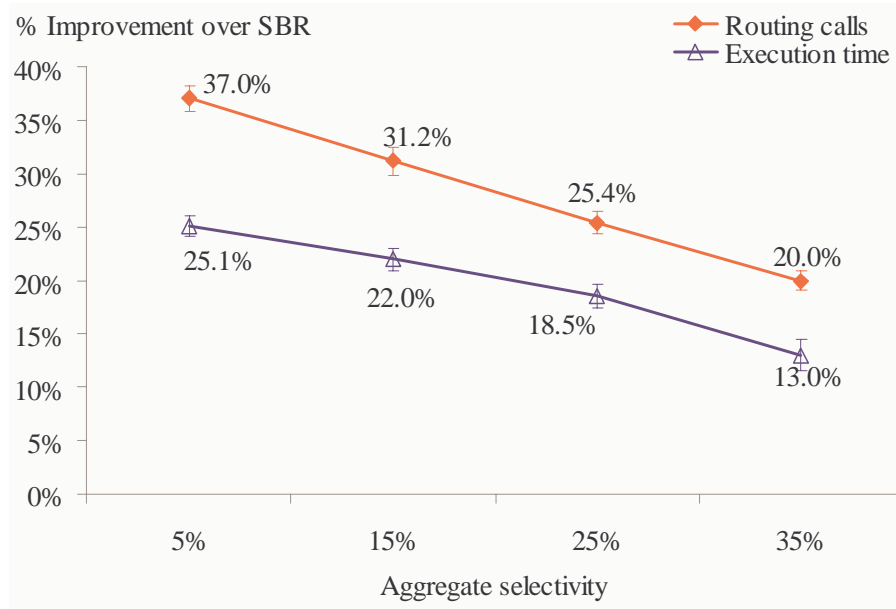
Figure 9 – Improvement with varying aggregate selectivity (6 joins)

2.6.6. Adaptivity Experiments

In this and subsequent sections, we will use the real-life *Lab* dataset. In the *Lab* dataset the best classifier attributes for operators change as time progresses. Query Q1 is used to illustrate how CBR adapts in the presence of variations of selectivity and variations of correlation.

SELECT * FROM sensors WHERE light>500   (Q1)

For example, the amount of light varies with the time of day in the obvious way: during the day there is more light than during the night. However, the predicate that evaluates "light>500" may actually be correlated with *sensorID* and not with, say, *hours*. This happens because some sensors are placed in illuminated areas like windows or in offices, while others are placed in hallways with less human activity and light. Furthermore, if the operator that checks if light>500 evaluated to true for, say, sensor 7, at 12h34pm, then it is very likely that it will evaluate to true for the same sensor 1 minute later. During the night, when it is dark and when people have left the building, the operator that tests for light will almost always have zero selectivity. When that is the case, no attribute can be found to be correlated with the

operator; that is, if the selectivity of an operator is 0% (or 100%), then all attributes have zero information gain ratio.

Figure 10 shows the result of running query Q1 for three days and nights of data. The top part of the figure shows the selectivity of the predicate; note that during the day the selectivity does not reach 100%, thus, some sensors are in darker areas than others. In the middle of the figure, we show what attribute is most correlated with the selectivity of the operator for each moment in time. *sensorID* is almost always the best classifier attribute. Sometimes, especially during transitions night-day or day-night, the attribute *hours* is the best classifier attribute. In three other moments, one of the other attributes was found to be the best classifier. In all other periods not covered by any of the black lines from "sensorID", "hours", and "All others", CBR could not find any attribute correlated with the selectivity of the operator (because its selectivity was 0%). Finally, the lower part of the Figure 10 shows how the information gain of attribute *sensorID* varies with time. Although Figure 10 is indicative that data characteristics in the stream change dramatically and that CBR is able to adapt to them, queries with only one operator (like query Q1) do not require good routing policies.

To evaluate the adaptivity of CBR on the *Lab* dataset, we ran queries similar to query Q2 below:

```
SELECT  * FROM sensors                                    (Q2)
WHERE  light        BETWEEN lowL AND highL
   AND temperature  BETWEEN lowT AND highT
   AND humidity     BETWEEN lowH AND highH
   AND voltage      BETWEEN lowV AND highV;
```

For each attribute, the parameter lowX was randomly chosen from among the lowest 25% values in the attribute's domain and the parameter highX was randomly chosen from the highest 25% values in the domain.

For 50 different random Q2 queries, we obtained on average an improvement of 8% in routing calls, 5% in total execution time, 7% in time spent evaluating operators, and 18% in routing calls needed until a tuple is dropped. The results are positive but modest. Two reasons explain why CBR does not provide greater improvements:

(i) There are overheads in TelegraphCQ unrelated to routing or operator execution [29], for instance, the IO required to get the stream tuples from the network and the overhead of queuing those tuples before they get to the Eddy. These overheads limit the benefit we can obtain from a better routing policy. In Section 2.6.7 we explore operators with higher execution costs and show that as operator costs increase, CBR's performance improves.

(ii) CBR can only obtain improvements when the selectivities of the operators are not close to 0% or 100%. As seen in Figure 10 there are large intervals in the dataset where the selectivities of operators stay very close to 0% or very close to 100%. The selectivity graphs for the other operators (not shown) have similar intervals very close to 0% or to 100%. For Q2, this happened 57.2% of the time, CBR yields improvements only on the other 42.8% of the time.
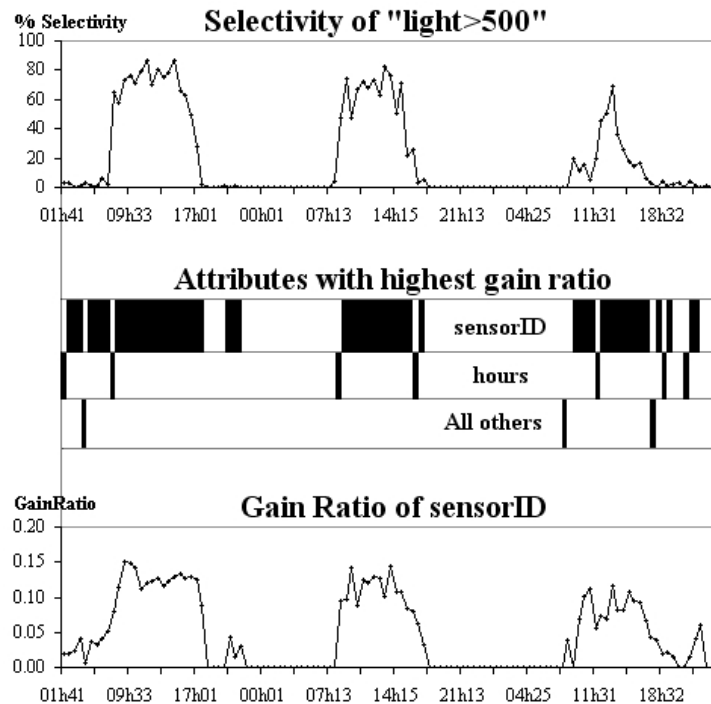
Figure 10 – Change in selectivity, best classifiers, and gain ratio

2.6.7. Varying Operator Cost

In this section we vary the time it takes an operator to process a tuple and report the corresponding CBR's performance improvements. There are two motivations for exploring the space of higher operator costs: (i) there are applications where operator costs can be very high (for example, [33] reports operator costs, expressed in terms of power consumption, with cost differences of two orders of magnitude between operators) and (ii) the implementation of TelegraphCQ we used has overheads [29] that overshadow operator costs. By increasing the operator costs, we decrease the weight of these overheads in the overall execution time.

Figure 11 shows the improvement in performance from using Content-Learns in queries like Q2. The improvement in the number of routing calls remains constant throughout and is shown only for reference. The improvement in execution time improves as the operator cost increases. The increase in operator cost was obtained by running CPU intensive computations every time a tuple had to be processed by an operator.
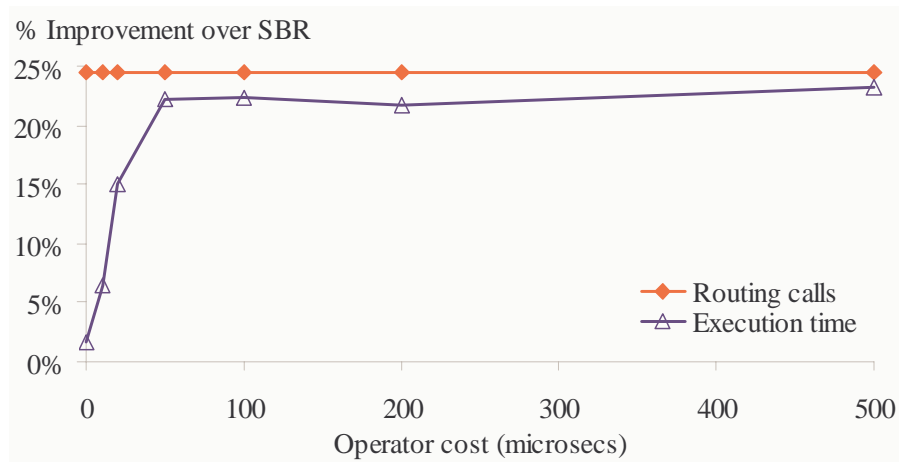
Figure 11 – Improvement with varying operator cost

2.6.8. Run-Time Overhead Of CBR

As mentioned in Section 2.4.3, CBR has two overheads: routing overhead and learning overhead. We instrumented the code to determine the time spent by each of these overheads. The routing overhead was

measured as the time taken by the function that performs routing decisions (the algorithm of Section 2.4.2). The learning overhead was measured as the time taken for updating the data structures described in Section 2.4.1 together with the time spent computing the best classifier attributes for each profiled operator. We also instrumented the SBR version to report its routing and updating overheads (although SBR does not determine classifier attributes, it spends time updating statistics as well). Figure 12 reports, per routed tuple[7], these overheads, in microseconds, for both SBR and CBR policies for the experiments of Section 2.6.4 (*Stream-Star* dataset). For both policies, the total overhead (routing together with learning and updating statistics) was around 4-5% of the total execution time.
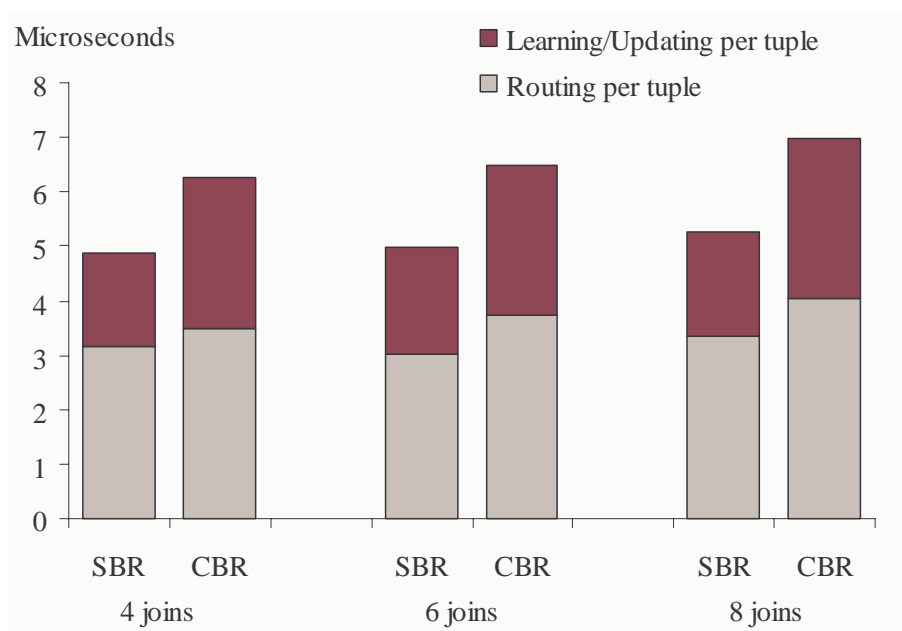
Figure 12 – Per tuple overhead

---

[7] Per tuple overhead is computed as total overhead divided by the number of routing calls. Note that the number of routing calls is equal to the number of times the Eddy has to route tuples.

In addition, we also measured the worst case scenario for CBR: when the routing policy is irrelevant, as is the case for queries with one operator only. If there is just one operator, no benefit can be gained from different routing policies. Thus, differences in total execution time must be from overhead and not from better decisions. For this experiment we run query Q1 from Section 2.6.6 over the *Lab* dataset (without using the operator delays mentioned in the previous section) for both CBR and SBR. The average over 10 runs of query Q1 shows that, when no benefit is possible, CBR is about 0.8% worse than SBR in total execution time.

2.7. Conclusions

In this chapter we proposed a new concept: assigning different query execution plans for subsets of data with different statistical properties. As such, we developed a new query processing technique called content-based routing that eliminates the single-plan restriction in current systems. We showed how the adaptive architecture of a data stream management system, TelegraphCQ, can be extended with content-based routing to enable the system to exploit correlations between tuple content and operator selectivities.

Our most important contribution was to show that content-based learning and routing can be simultaneously inexpensive and adaptive while still achieving significant performance improvements. We presented the Content-Learns algorithm which learns good content-based routes automatically, and we showed that the overhead of maintaining the extra statistics and computing classifier attributes is negligible when compared to a non-CBR algorithm.

Our prototype implementation indicates that CBR can improve execution time by up to 35% when compared with routing based on operator statistics alone. For all queries with more than one operator, CBR yielded better results than SBR, both in the number of routing calls as well as in absolute running time. In addition, the performance comparison between Content-Learns and Content-Knows showed that Content-Learns learns classifier attributes correctly in real time.

CHAPTER 3

PROACTIVE RE-OPTIMIZATION

While addressing some of possible mistakes done by traditional optimizers, first generation AQP systems still use a traditional optimizer to pick a plan. The effectiveness of this approach is limited because traditional optimizers choose plans unaware of issues affecting re-optimization. For example, previous AQP approaches are more likely to start execution with a sub-optimal plan. In addition, detection of sub-optimality can be inefficient, re-optimization can lead to lost and repeated work, and convergence to optimal plan can be slow. We address this problem using proactive re-optimization, a new approach that is less likely to start with a risky plan, it is more likely to detect sub-optimalities faster and is more likely to converge to the optimal plan in less steps.

3.1. Introduction

As described in Chapter 1, most query DBMS use a sequential, non-adaptive approach to query processing–the optimizer enumerates plans, computes the cost of each plan, and picks the plan with lowest cost which is then executed [79]. This approach relies heavily on the accuracy of estimated statistics of intermediate subexpressions to choose good plans. It is a well-known problem that errors in estimation propagate exponentially in the presence of skewed and correlated data distributions [24, 51]. Such errors, and the consequent suboptimal plan choices, were not a critical problem when datasets were smaller, queries had only a few joins and simple predicates, and hardware resources were limited. In the last two decades, datasets, query complexity, and the hardware resources to manage DBMS have grown dramatically. Query optimizers have not kept pace with the ability of database systems to execute complex queries over very large data sets.

Several techniques have been proposed to improve traditional query optimization. These techniques include better statistics [70], new algorithms for optimization [25, 49, 52], and adaptive architectures for execution [5]. A very promising technique in this direction is *re-optimization*, where the optimization and the execution stages of processing a query are interleaved, possibly multiple times, over the running time of the query [56, 58, 63, 87]. Markl et al show that re-optimization can improve the performance of complex queries by an order of magnitude [63].

Current re-optimizers take a *reactive* approach to re-optimization: they first use a traditional optimizer to generate a plan, and then track statistics and respond to estimation errors and resulting suboptimalities detected in the plan during execution. Reactive re-optimization is limited by its use of an optimizer that does not incorporate issues affecting re-optimization, and suffers from at least three shortcomings:

- The optimizer may pick plans whose performance depends heavily on uncertain statistics, making re-optimization very likely.

- The partial work done in a pipelined plan is lost when re-optimization is triggered and the plan is changed.

- The ability to collect statistics quickly and accurately during query execution is limited. Consequently, when re-optimization is triggered, the optimizer may make new mistakes, leading potentially to thrashing.

In this chapter we propose *proactive re-optimization* to address these shortcomings. We have implemented a prototype proactive re-optimizer called *Rio* that incorporates three new techniques:

- *Bounding boxes* are computed around estimates of statistics to represent the uncertainty in these estimates.

- The bounding boxes are used during optimization to generate *robust* and *switchable* plans that minimize the need for re-optimization and the loss of pipelined work.

- Random-sample processing is merged with regular query execution to collect statistics quickly, accurately, and efficiently at run-time.

Our experimental results demonstrate that proactive re-optimization can provide up to three times improvement over a strictly reactive re-optimizer. The rest of this chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 uses a series of examples to illustrate the problems with reactive re-optimization, and Section 3.4 shows how proactive re-optimization addresses these problems. Section 3.5 describes the Rio implementation and Section 3.6 presents an experimental evaluation.

3.2. Related Work

In previous work, we classify adaptive query processing systems into three families: plan-based, routing-based, and continuous-query-based [10]. In this chapter we focus on plan-based systems, the more closely related to Rio being ReOpt [58] and POP [63]. Other related projects include Ginga [69], Tukwila [53], query scrambling [87], and corrective query processing [56]. ReOpt and POP use a traditional optimizer to pick plans based on single-point estimates of statistics. These reactive re-optimizers augment the chosen plan with *checks* that are verified at run-time. The query is re-optimized if a check is violated.

The use of intervals instead of single-point estimates for statistics has been considered by least-expected-cost optimization (LEC) [25], error-aware optimization (EAO) [89], and parametric optimization [49, 52]. LEC treats estimated statistics as random variables to compute the expected cost of each plan. Unlike LEC, Rio does not assume knowledge about the underlying distribution of statistics. Instead, Rio computes the uncertainty in these estimates based on how they were derived. Like Rio, EAO considers intervals of estimates and proposes heuristics to identify robust plans. However, the techniques in EAO assume a single uncertain statistic (memory size) and a single join. Furthermore, LEC and EAO do not consider re-optimization or the collection of statistics during query execution. Therefore, these techniques use execution plans that were picked before the uncertainty in statistics is resolved. Parametric

optimization identifies several execution plans during optimization, each of which is optimal for some range of values of run-time parameters. Parametric optimization, along with the choose-plan operator [40], enables the optimizer to defer the choice of plan to run-time. Switchable plans and switch operators in Rio are similar. However, unlike choose-plan operators, switch operators may occur within pipelines. Furthermore, parametric optimization does not consider uncertainty in estimates, collection of statistics during execution, robust plans, or re-optimization.

Rio combines the processing of random samples of tuples with regular query processing to obtain quick and accurate estimates of statistics during execution. This approach differs from previous uses of random samples, e.g., providing continuously-refined answers in an online manner [47], computing approximate query results [1, 22], or building base relation statistics from samples [24]. Robust cardinality estimation (RCE) uses random samples for cardinality estimation, to deal with uncertainty, and to explore performance-predictability tradeoffs [6]. However, RCE does not consider re-optimization. Furthermore, RCE does not consider techniques such as merging random-sample processing with regular query execution, or propagating random samples through joins.

3.3. Problems With Reactive Re-Optimization

In this section we present a series of examples to highlight the problems with current approaches to query re-optimization. One known problem with traditional optimizers, e.g. [79], is that they rely frequently on outdated statistics or invalid assumptions such as independence among attributes. Consequently, they may choose suboptimal query plans that degrade performance by orders of magnitude [24, 63]. Example 3.1 illustrates this problem.

Example 3.1. Consider the query "select * from R, S where R.a=S.a and R.b>K1 and R.c>K2". Assume the database buffer-cache size is 200MB, $|R|$=500MB, $|S|$=160MB, and $|\sigma(R)|$=300MB, where $\sigma(R)$ represents the result of the "R.b>K1 and R.c>K2" selection on R. However, because of skew and

correlations in the data distributions of R.b and R.c, the optimizer underestimates $|\sigma(R)|$ to be 150MB. With this incorrect estimate, the optimizer would pick plan P13a for this query (Figure 13). P13a is a hash join with $\sigma(R)$ as the build input and S as the probe. (We use the convention that the left input of a hash join is the build and the right input is the probe.) However, since $|\sigma(R)|$ is actually 300MB, Plan P13a's hash join requires two passes over R and S. Plan P13a is suboptimal because plan P13b, which builds on S, finishes in one pass over R and S. ∎
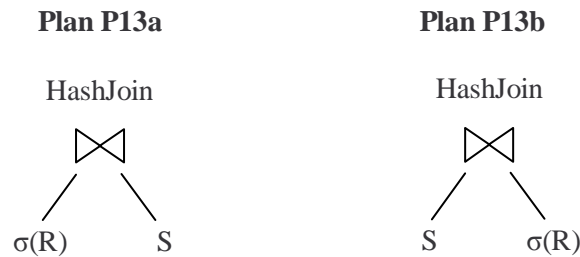
Figure 13 – Two plans for the σ(R)⋈S query

Re-optimization can avoid problems similar to the one in Example 3.1. Current systems that use re-optimization first use a traditional optimizer to pick the best plan, and then add *check oprators* to the chosen plan. The check operators detect sub-optimality during execution, and trigger re-optimization if required. For example, the check-placement algorithm used by POP computes a *validity range* for each plan [63]. Let P be a left-deep plan. The root operator of P is a binary join operator with subtree D and base relation R as inputs. Let |D| denote the result size of D. POP defines the validity range of P as the range of values of |D| for which P has the lowest cost among all plans P', where P' is logically equivalent to P, P' is rooted at an operator with the same inputs D and R, and P' gives the same interesting orders as P.

During execution, each check operator collects statistics on its inputs. If these statistics satisfy the validity ranges for the plan picked by the optimizer, then execution proceeds as usual. Otherwise, re-optimization is invoked to choose the best plan based on the statistics collected. The reuse of intermediate

results that were materialized completely in a previous execution step is considered during re-optimization. Example 3.2 illustrates the overall technique.

Example 3.2. Consider the scenario shown in Example 3.1 and Figure 13. A re-optimizer like POP will choose the same plan (P13a) as a traditional optimizer. Additionally, POP will compute validity ranges for the chosen plan. For example, a validity range for P13a is 100KB≤|σ(R)|≤160MB. If |σ(R)|<100KB, then it is preferable to use an index nested-loops join with tuples in σ(R) probing a covering index on S. If |σ(R)|>160MB, then plan P13b is optimal. In this example, the check |σ(R)|≤160MB will fail during execution, invoking re-optimization.   ■

3.3.1. Limitations of Single-Point Estimates

Although re-optimization preempts the execution of the suboptimal plan P13a in Example 3.1 when |σ(R)|>160MB, it incurs the overhead of calling the optimizer more than once and the cost of repeating work. For example, the (partial) scan of R in plan P13a (of Figure 13) until re-optimization is lost and must be repeated in P13b. The optimizer may be better off picking plan P13b from the start because P13b is a *robust plan* with respect to the uncertainty in |σ(R)|; see Figure 14.

When |σ(R)|≤Memory, both plans finish in one pass and involve the same amount of IO. However, when |σ(R)|>Memory, only P13b finishes in one pass.

Current re-optimizers do not account for robustness of plans since they consider *single-point estimates* for all statistics needed to cost plans. (To arrive at these single-point estimates, optimizers are often forced to make assumptions like uniformity and independence [79].) Non-robust plans may lead to extra optimizer invocations and wasted work, as we will show in Section 3.3.3.
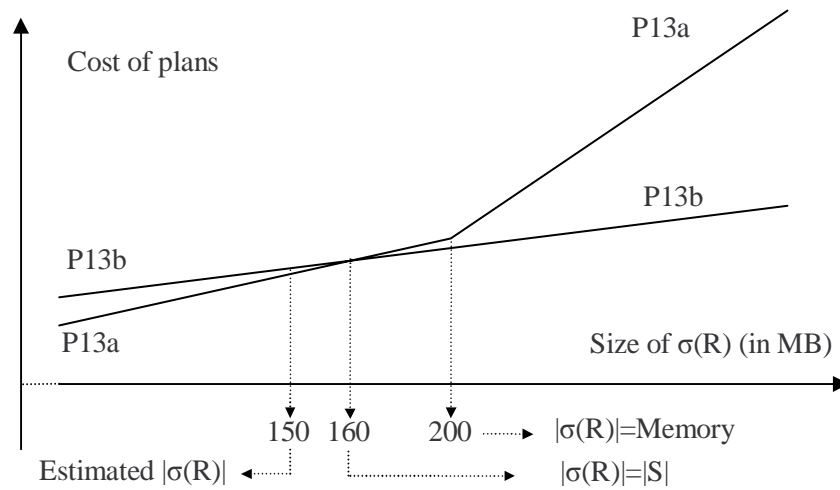
Figure 14 – Cost of plans P13a and P13b as |σ(R)| varies

3.3.2. Limited Information For Re-Optimization

Current re-optimizers make limited effort to collect statistics quickly and accurately during execution. For instance, the validity check in Example 3.2 will fail when $|\sigma(R)|=160MB$, and re-optimization will be invoked. However, the optimizer does not know $|\sigma(R)|$ accurately at this point–it only knows that $|\sigma(R)|\geq160MB$–which may cause it to chose a suboptimal plan again. Example 3.3 illustrates an extreme instance of the thrashing that can result.

Example 3.3. Consider the query "select * from R, S, T where R.a=S.a and S.b=T.b and R.c>K1 and R.d=K2". Assume that the sizes of the tables are known accurately to be $|R|=200MB$, $|S|=50MB$, and $|T|=60MB$. Further assume that $|\sigma(R)|=80MB$, but that the optimizer underestimates it significantly as 40KB.[8] Based on these statistics, the optimizer chooses plan P15a shown in Figure 15.

A reactive re-optimizer may compute validity ranges for plan P15a as shown by the gray boxes in this plan. For example, the validity range for the index nested-loops join between $\sigma(R)$ and S in P15a is

|σ(R)|≤100KB. This validity-range check will fail at run-time, triggering re-optimization. Plan P15b will be picked next with a validity range as shown in Figure 15. This check will fail and re-optimization will be triggered again, and so on until the optimal plan P15d is chosen finally. ∎

3.3.3. Losing Partial Work in a Pipeline

In addition to the multiple re-optimization steps as illustrated in Example 3, current re-optimizers also lose the partial work done by a pipeline in execution when re-optimization is triggered. For example, plan P15c in Figure 15 has a pipeline PPL2 (enclosed with dotted lines) that scans R, probes S in HashJoin1, and builds joining tuples into HashJoin2. The validity-range check before HashJoin2 will fail before pipeline PPL2 finishes, and the partial work done by this pipeline will be lost. On the other hand, work done by completed pipelines, like PPL1–scanning and building S–can be reused. However, in this example, the build of S in plan P15c cannot be reused in plan P15d because the hash tables are built on different join attributes.

3.4. Proactive Re-Optimization

In this chapter we propose *proactive re-optimization*, a new paradigm for query re-optimization. Proactive re-optimization addresses the problems with current reactive approaches that were illustrated in Section 3.3. A proactive re-optimizer incorporates three new techniques:

- Computing *bounding boxes*–intervals around estimates–as a representation of the uncertainty in estimates of statistics.

- Using bounding boxes during optimization to generate *robust plans* and *switchable plans* that avoid re-optimization and loss of pipelined work.

---

[8] A recent paper from IBM reports cardinality estimation errors on real datasets that exceed six orders of magnitude [63].

- Using randomization to collect statistics quickly, accurately, and efficiently as part of query execution.

Plan P15a

IndexNL2

Re-optimized when
σ(R)>100KB

IndexNL1   T

|σ(R)|≤100KB

σ(R)   S

Plan P15b

HashJoin2

...

HashJoin1   T

100KB<|σ(R)|≤50MB

σ(R)   S

Re-optimized when
σ(R)>50MB

Plan P15d

HashJoin2

...

HashJoin1   σ(R)

S   T

Re-optimized when
σ(R)>60MB

Plan P15c

HashJoin2

|σ(R)|≤60MB

HashJoin1   T

Pipeline PPL2

Pipeline PPL1

S   σ(R)
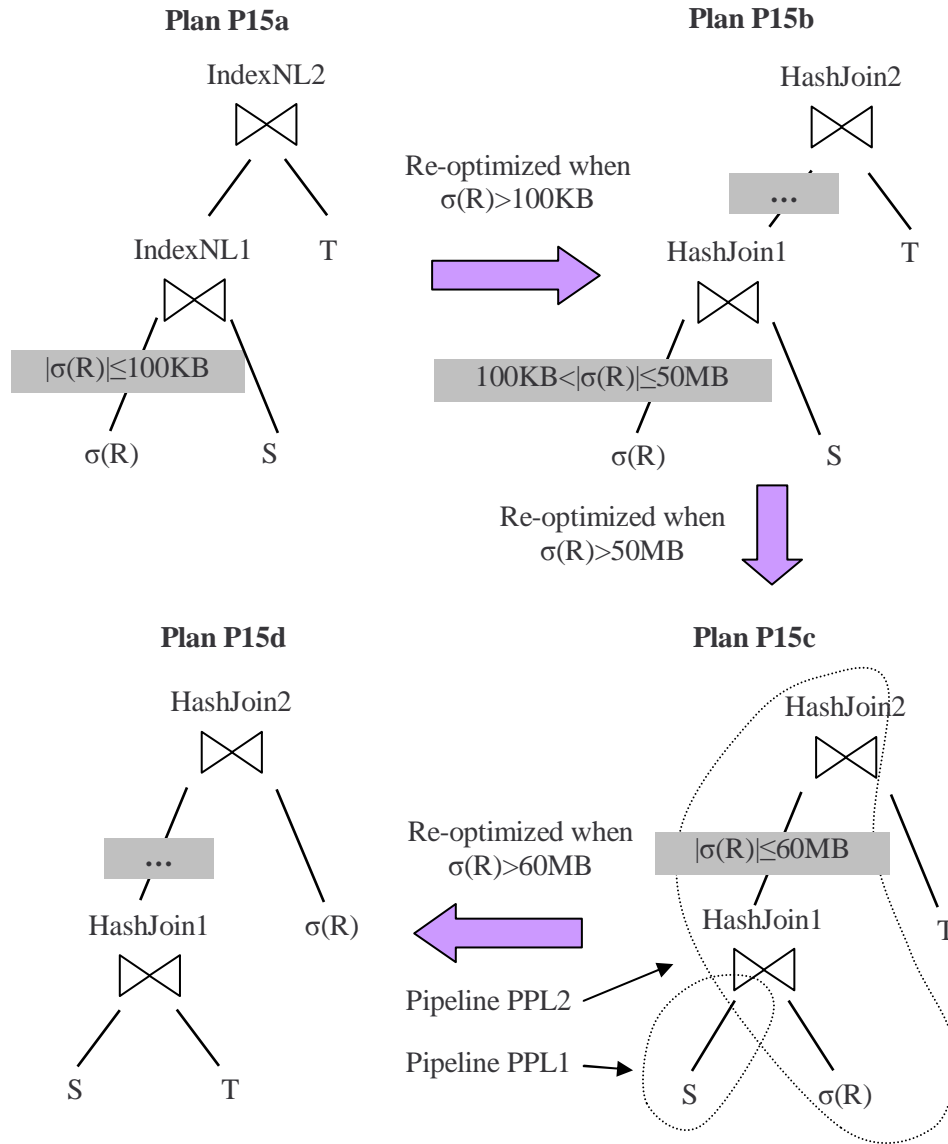
Figure 15 – Thrashing with reactive re-optimization
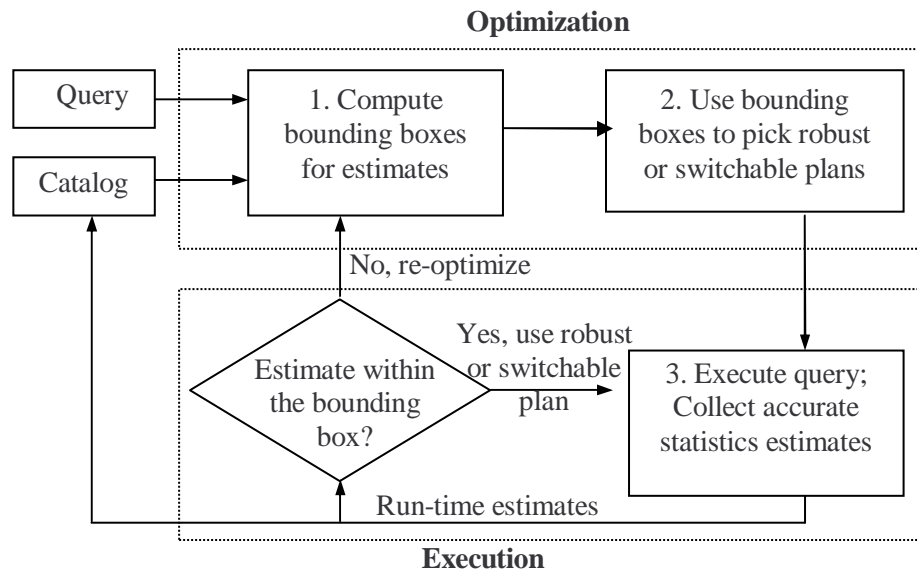
Optimization

Figure 16 – Proactive re-optimization

Figure 16 shows the architecture of a proactive re-optimizer. In Section 3.5 we introduce Rio, our specific implementation of a proactive re-optimizer.

3.4.1. Representing Uncertainty in Statistics

Current re-optimizers compute a single-point estimate for any statistic needed to cost plans. One way to account for possible errors in estimates is to consider intervals, or bounding boxes, around the estimates. If the optimizer is very certain of the quality of an estimate, then its bounding box should be narrow. If the optimizer is uncertain of the estimate's quality, then the bounding box should be wider. There are different ways of computing bounding boxes, e.g., using strict upper and lower bounds [23] or by characterizing uncertainty in estimates using discrete buckets that depend on the way the estimate was derived [58]. Our implementation uses the latter approach as described in Section 3.5.2.

Example 3.4. Consider the scenario from Example 3.1. The costs of plans P13a and P13b depend mainly on $|\sigma(R)|$ and $|S|$. Suppose a recent estimate of $|S|$=160MB is available in the catalog. However, in the absence of a multidimensional histogram on R, $|\sigma(R)|$ must be estimated from the estimated selectivities

of R.b>K1 and R.c>K2 and an assumption of independence between these predicates. This estimate of |σ(R)|=150MB is thus very uncertain. In this case, Figure 17 shows an example bounding box around the single-point estimate (|σ(R)|=150MB, |S|=160MB).   ∎
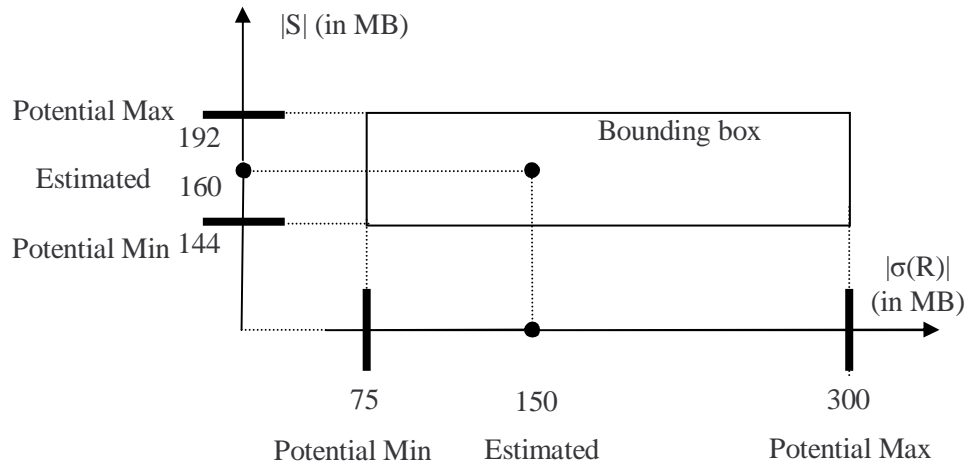
Figure 17 – Bounding box around estimates of |σ(R)| and |S|

3.4.2. Using Bounding Boxes During Optimization

Since current re-optimizers consider single-point estimates only, their plan choices may lead to extra re-optimization steps and to the loss of partial pipelined work if actual statistics differ from their estimates. Bounding boxes can be used during optimization to address this problem. While there is always one plan that is optimal for a single-point estimate, one of the following four cases can occur with a bounding box B:

(C.i) *Single optimal plan*. A single plan is optimal at all points within B.

(C.ii) *Single robust plan*. There is a single plan whose cost is very close to optimal at all points within B.

(C.iii) *A switchable plan*. Intuitively, a switchable plan in B is a set S of plans with the following properties:

a) At each point p in B, there is a plan p in S whose cost at p is close to that of the optimal plan at p;

b) The decision of which plan in S to use can be deferred until accurate estimates of uncertain statistics are available at query execution time; and c) If the actual statistics lie within B, an

appropriate plan from *S* can be picked and run without losing any significant fraction of the execution work done so far.

(C.iv) *None of the above.* Different plans are optimal at different points in *B*, but no switchable plan is available.

A proactive re-optimizer identifies which of the above four cases *B* falls into. Note that a single optimal plan is also robust, and a robust plan is a singleton switchable plan. Example 3.5 illustrates how a proactive re-optimizer can exploit robust plans and switchable plans. Details of how to enumerate and choose robust and switchable plans are given in Section 3.5.

Example 3.5. Consider the scenario from Example 3.1. Figure 18 is the same as Figure 14 except that it considers the bounding box B=[75MB, 300MB] for $|\sigma(R)|$. As seen, plan P13a is optimal for the estimated $|\sigma(R)|$=150MB, but not in the entire bounding box. While plan P13b is not optimal for the estimated $|\sigma(R)|$, P13b is robust because its cost is very close to optimal at all points in *B*. Therefore, picking plan P13b would be a safe option. However, as we will see in Section 3.5, P13a and P13b (which are hybrid hash joins with build and probe reversed) are switchable. It is preferable to pick the switchable plan P={P13a, P13b} instead of the robust P13b because P is guaranteed to run the optimal plan as long as $|\sigma(R)|$ lies within *B*. ∎
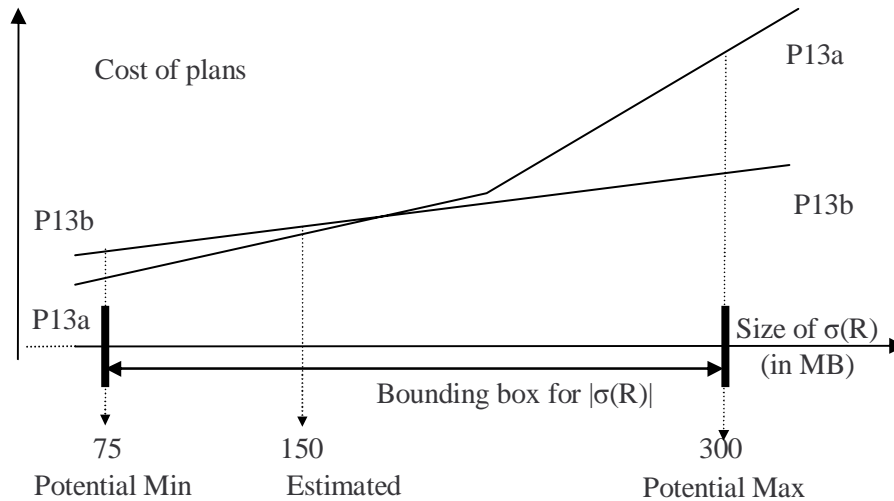
Figure 18 – Robust and switchable plans

3.4.3. Accurate Run-Time Statistics Collection

As seen in Example 3.3, the lack of accurate run-time statistics collection can lead to thrashing during re-optimization. In general, accurate run-time estimates are needed to pick the right plan from a switchable set, to detect when to trigger re-optimization, and to pick a better plan in the next optimization step.

For efficiency, we hide the cost of collecting accurate statistics by combining statistics collection with regular query execution. Furthermore, for early detection of the need to re-optimize, the run-time estimates must be computed both quickly and accurately. We achieve these goals by using a new technique of merging the processing of random samples of tuples along with regular query execution. Example 3.6 illustrates this approach. Implementation details are given in Section 3.5.4.

Example 3.6. Consider Example 3.3. Assume that the optimizer had picked the suboptimal Plan P15a which contains a pair of index nested-loops joins with $\sigma(R)$ as the outer input. Suppose tuples in R are physically laid out in random order on disk. Then, once 5% of the R tuples have been scanned and processed, a fairly accurate estimate of the selectivity of σ is available. Thus, $|\sigma(R)|$ can be estimated reliably. This estimate enables a proactive re-optimizer to detect quickly that P15d is the optimal plan, thereby avoiding the thrashing problem in reactive re-optimizers. ∎

3.5. Proactive Re-Optimization With Rio

Section 3.4 presented an overview of proactive re-optimization without providing specifics about the implementation. We now describe our prototype proactive re-optimizer Rio.

3.5.1. Building Rio

Rio was built using the Predator DBMS [81] by extending it as follows:

- Equi-height and end-biased histograms were added [70].

- Predator has a traditional cost-based dynamic-programming optimizer [79] which we refer to as *TRAD*. We added:

  - A *Validity-Ranges Optimizer* (VRO), our implementation of the algorithms used by POP [63].

  - *Rio*, our proactive re-optimizer.

  - Uncertainty buckets and rules from [58] to generate and propagate uncertainty buckets during query optimization.

- The following operators were added:

  - A hybrid hash join operator [59] that processes tuples from two input subtrees. At most one of the subtrees is a deep subtree and at least one is a subtree with one base relation. Either subtree can be the build input of the hash join. Thus, this operator enables us to consider arbitrary linear plan shapes, e.g., right-deep join trees like plan P22c in Figure 22. Recall our convention that the left input to the hash join is the build and the right input is the probe.

  - A switch operator to implement switchable plans.

  - Operators to read random samples from base relations and to generate random samples of joins as part of query execution.

  - Buffer operators to buffer tuples and delay processing in a pipeline until the statistics necessary to choose among the set of plans in a switch operator have been collected.

- ▪ Operators to scan previously materialized expressions for reuse after re-optimization. Materialized expressions that may be reused include completed builds of hash joins and the sorted temporary files created by a sort operator.

- ▪ The original validity-ranges algorithm [63] uses checks on buffers to trigger re-optimization when the buffers overflow or underflow. In our VRO implementation, validity ranges are checked by buffer operators placed appropriately in the plan which buffer and count incoming tuples. The buffer operators trigger re-optimization if any validity range is violated.

- Execution engine:

  - ▪ The ability to stop query execution midway, re-optimize, and restart execution.

  - ▪ An in-memory catalog to track statistics collected at run-time as well as expressions materialized as part of query execution. The optimizer consults this catalog during re-optimization.

  - ▪ An inter-operator communication mechanism based on punctuations [86] that, e.g., allows an operator C to signal to its parent operator that C has generated a 1% random sample of its output.

3.5.2. Computing Bounding Boxes

Recall that a proactive re-optimizer uses bounding boxes instead of single-point estimates for statistics needed to cost plans. Currently, Rio restricts the computation of bounding boxes to size and selectivity estimates. For each such estimate E, a bounding box B is computed using a two-step process:

- An uncertainty bucket U is assigned to the estimate E

- The bounding box is computed from the (E, U) pair

To compute U, we adopted a technique from [58] that uses a set of rules to compute uncertainty. In the original approach [58], the value of U belongs to a three-valued domain {*small*, *medium*, *large*} that characterizes the uncertainty in the estimate E. The value of U is computed based on the way E is derived. For example, if an accurate value of E is available in the catalog, then U takes the value *small* that denotes

low uncertainty. In Rio, we augmented the domain of U to an integer domain with values from 0 (no uncertainty) to 6 (very high uncertainty).

A bounding box B of an estimated value E is an interval [lo, hi] that contains E. The uncertainty value U is used to compute the values lo and hi as shown in Figure 19. Example 3.7 illustrates the computation of uncertainty buckets and bounding boxes for our running example.

Example 3.7. Consider the scenario from Example 3.1. The optimizer needs to cost plans P13a and P13b which depend on $|\sigma(R)|$ and $|S|$. Recall that σ represents R.b>K1 and R.c>K2. The single-point estimates for $|S|$ and $|\sigma(R)|$ are E_S=160MB and E_R=150MB respectively. Assume that E_S was obtained from the catalog. Therefore, our rules adapted from [58] for derivation of uncertainty set U_S=1 (low uncertainty in E_S). From Figure 19, the bounding box for E_S is B_S=[144, 192]. On the other hand, assume that the estimate E_R was computed from the estimated selectivities of R.b>K1 and R.c>K2 based on the assumption that these predicates are independent (no multidimensional histogram was available). Thus, the uncertainty in E_R is high. Accordingly, our rules for derivation of uncertainty set U_R=5. From Figure 19, the bounding box for E_R is B_R=[75, 300]. ∎

```
ComputeBoundingBox(   Inputs: estimate E, uncertainty U
                      Outputs: lo, hi) {
  Δ+ = 0.2;        // increment step
  Δ- = 0.1;        // decrement step
  hi = E * (1 + Δ+ * U);
  lo = E * (1 - Δ- * U);
}
```

Figure 19 – Computing bounding boxes for an (E, U) pair

3.5.3. Optimizing With Bounding Boxes

The TRAD optimizer enumerates and groups plans based on their join subset (JS) and interesting orders (IO) [79]. For each distinct (JS, IO) pair enumerated, TRAD prunes away all plans except the plan with the lowest cost, denoted *BestPlan*. The cost of each plan is computed based on estimated statistics.

VRO takes the same steps as TRAD initially, so VRO will find the same optimal plan (*BestPlan*) for each (JS, IO) pair. However, VRO then adds validity ranges on the inputs to the join operators in *BestPlan* [63]. Consider a join operator O with inputs R_D and R_B, where R_D is the deep subtree input and R_B is the base relation input. The validity range of O is the range of values of $|R_D|$ where operator O has the lowest cost among all join operators with the same inputs R_D and R_B, and giving the same set of interesting orders as O. The validity range of O is computed by varying $|R_D|$ up (and down) until the cost of O is higher than that of some other join operator with the same inputs R_D and R_B and giving the same set of interesting orders as O. The *Newton-Raphson method* can be applied to the join cost-functions to compute validity ranges more efficiently than linear search; see [63].

Unlike TRAD and VRO, Rio computes bounding boxes for all input sizes used to cost plans. Then it tries to compute a switchable plan (which may also be a single robust plan or a single optimal plan) for each distinct (JS, IO) pair based on the bounding boxes on the inputs to the plan. If Rio fails to find a switchable plan for a (JS, IO) pair, then it picks the optimal plan for (JS, IO) based on the single-point estimates of input sizes (*BestPlan*), and adds validity ranges like VRO.

Rio computes switchable plans in two steps. First, it finds three seed plans for each (JS, IO) pair. Then, it creates the switchable plan from the seed plans as described next.

3.5.3.1. Generating the Seed Plans

In traditional enumeration, plan cost is computed using single-point estimates of statistics. In Rio, the enumeration considers three different costs for each plan, C_{Low}, C_{Est}, and C_{High}. Cost C_{Est} is computed using the single-point estimate of statistics exactly like in traditional enumeration. Cost C_{Low} (C_{High}) is computed at the lower left corner (upper right corner) of a bounding box as illustrated in Figure 20.

Rio augments the (JS, IO) pair used during traditional enumeration with an extra *cost bucket* CB that takes values *Low*, *Estimated*, or *High*. Like the interesting order concept, the cost bucket defines which plans and costs are comparable during cost-based pruning, e.g., a plan P for (JS, IO, CB=*Low*) is pruned

if and only if there exists a plan P' for (JS, IO, CB=*Low*) with a lower cost C_{Low} than P. For each distinct (JS, IO) pair, Rio enumerates and prunes plans for the three triples (JS, IO, CB=*Low*), (JS, IO, CB=*Estimated*), and (JS, IO, CB=*High*). The plans that remain after pruning are the three plans corresponding to the minimum C_{Low}, C_{Est}, and C_{High} for (JS, IO).
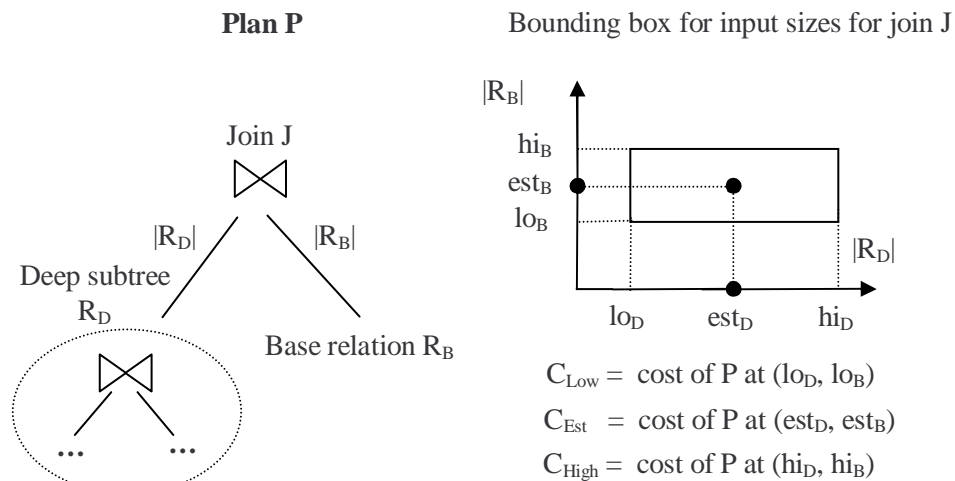
Figure 20 – Computing plan costs

Note that the best plan for (JS, IO, CB=*Estimated*) is the same plan as computed by TRAD for (JS, IO). Also, the addition of the extra cost bucket guarantees that the optimal plan for the estimated statistics will not prune away plans that are optimal at the upper right or lower left corners of the bounding boxes for input sizes. For each (JS, IO) pair, we end up with three seed plans from which a switchable plan will be created:

- *BestPlanLow*, the plan with minimum cost C_{Low}

- *BestPlanEst*, the plan with minimum cost C_{Est}

- *BestPlanHigh*, the plan with minimum cost C_{High}

3.5.3.2. Generating the Switchable Plan

Given the seeds *BestPlanLow*, *BestPlanEst*, and *BestPlanHigh*, one of four cases arises:

(C.i)   The seeds are all the same plan.

(C.ii) The seeds are not all the same plan, but one of them is a robust plan.

(C.iii) The seeds are not all the same plan, and none of them is robust, but a switchable plan can be created

        from the seeds.

(C.iv) We cannot find a single optimal plan, a single robust plan, or a switchable plan from the seeds.

In Case (C.i), the single optimal plan is the switchable plan. (Recall that an optimal plan is also robust and a robust plan is a singleton switchable plan.) In Case (C.ii), the optimizer checks if any of the seeds is a robust plan. A necessary test to determine whether *BestPlanLow* is robust is to check whether (i) cost C_{Est} of *BestPlanLow* is close to (e.g., within 20% of) C_{Est} of *BestPlanEst*, and (ii) cost C_{High} of *BestPlanLow* is close to C_{High} of *BestPlanHigh*. Intuitively, we are testing whether *BestPlanLow* has performance close to optimal at the estimated point and at the upper corner of the bounding box as well. While this test is not sufficient to guarantee robustness–because we do not check all points in the bounding box–Rio currently labels a plan as robust if it passes this plan-robustness test. If one of the seeds passes this test, then Rio uses that seed as a singleton switchable plan.

Example 3.8. Consider the scenario from Example 3.1. As seen in Figure 21, *BestPlanLow*=*BestPlanEst*=P13a and *BestPlanHigh*=P13b. The cost of P13a is not within 20% of the cost of P13b at the upper corner of the bounding box ($|\sigma(R)|$=300MB). Thus, P13a is not a robust plan within the bounding box. On the other hand, P13b is within 20% of the cost of P13a both at the estimated point ($|\sigma(R)|$=150MB) and at the lower corner of the bounding box ($|\sigma(R)|$=75MB). Therefore, P13b passes the plan-robustness test.   ∎
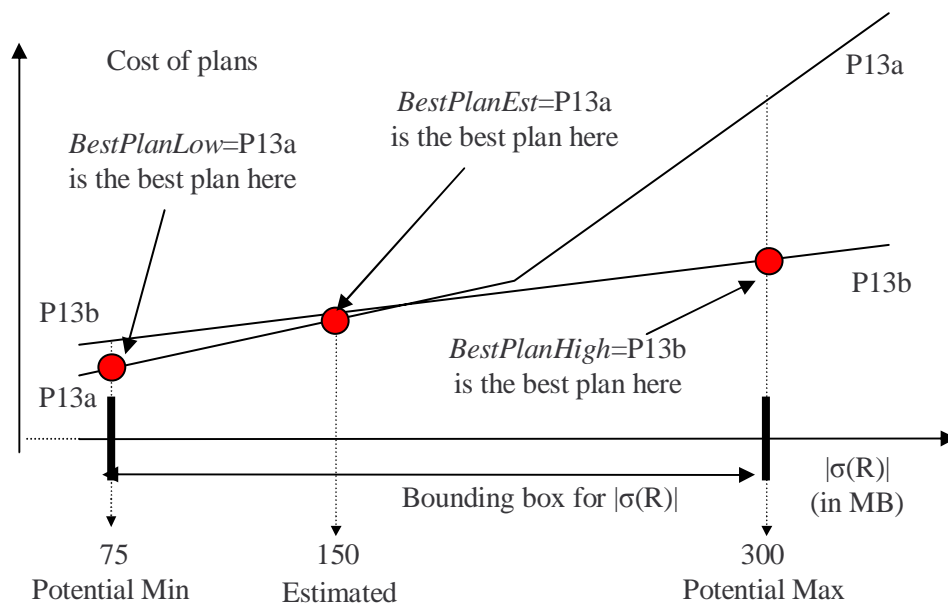
Figure 21 – Finding a robust plan in |σ(R)|'s bounding box

If none of the seeds is a single optimal plan or a single robust plan (Case (C.iii)), then the optimizer tries to find a switchable plan. A switchable plan for a (JS, IO) pair is a set of plans S where:

(i)   All plans in S have a different join operator as the root operator. (Hybrid hash joins with the build and probe reversed are treated as different operators.)

(ii) All plans in S have the same subplan for the deep subtree input to the root operator.

(iii) All plans in S have the same base table, but not necessarily the same access path, as the other input to the root operator.

Figure 22 contains an example of a switchable plan with three member plans for (JS={R,S,T}, IO=∅). Any two members of a switchable plan are said to be *switchable with each other*. In Section 3.5.4 we illustrate how the switchable plan chooses one of its members at execution time.
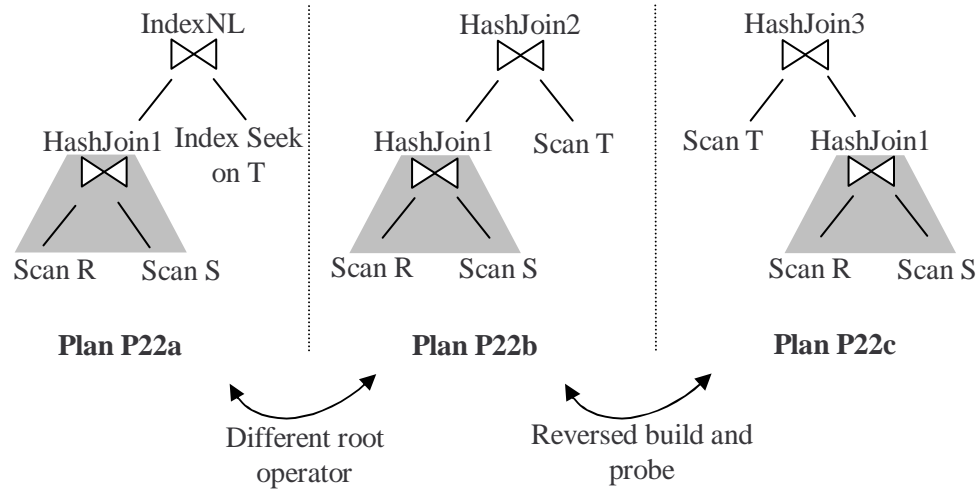
Figure 22 – Possible members of a switchable plan

If the seed plans for a (JS, IO) pair have the same subplan for the deep subtree, then the seeds themselves constitute a switchable plan. If these subplans are different, then Rio picks one of the seed plans, say *BestPlanLow*, and enumerates the set SW_Low of all plans that are switchable with *BestPlanLow* based on Conditions (i)—(iii) of switchable plans above. Then, among the plans in SW_Low, Rio finds the plan, *planMinEst* , with minimum cost at the estimated statistics point, and the plan, *planMinHigh* , with minimum cost at the upper right corner of the bounding box. If C_{Est} of *planMinEst* is close to (e.g., within 20%) C_{Est} of *BestPlanEst*, and C_{High} of *planMinHigh* is close to C_{High} of *BestPlanHigh*, then {*BestPlanLow*, *planMinEst* , *planMinHigh* } is a switchable plan. If not, Rio tries the same procedure with the two other seed plans.

Example 3.9. Suppose *BestPlanLow*=plan P22a, *BestPlanEst* =plan P22b (from Figure 22), and *BestPlanHigh*=plan P23 (from Figure 23) for R⋈S⋈T with no interesting orders. The subplan for the deep subtree of the outer join is different between P22a and P23, so they are not switchable. Thus, Rio enumerates SW_Low, which contains plan P22c. If C_{High} of plan P22c is close to that of P23, then {P22a, P22b, P22c} is a switchable plan.   ∎

If these techniques fail to find a switchable plan (Case (C.iv)), then Rio picks *BestPlanEst*–the optimal plan for the single-point estimates–and adds validity ranges, just like VRO.
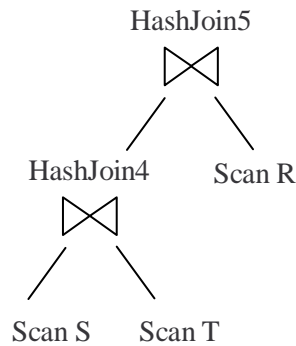
Plan P23

Figure 23 – Plan P23

3.5.4. Extensions to the Query Execution Engine

A switchable plan S defers the choice of which member plan to use for a join until the uncertain input sizes can be estimated accurately at run-time. S ensures that no (partial) work done by the pipeline containing the join is lost whenever the actual input sizes lie within the corresponding bounding box. Our implementation of switchable plans uses the following operators and communication framework:

- A *switch operator* that corresponds to the chosen switchable plan. This operator decides which member plan to use based on run-time estimates of input sizes, and instantiates the appropriate join operator and base relation access path.

- A *buffer operator* that buffers tuples until it can compute an input-size estimate needed by the switch operator.

- *Randomization-aware operators* that prefix their output with a random sample of their complete output.

- An *inter-oprator communication mechanism* based on punctuations [86] that allows operators to send size estimates and to demarcate random samples in their output stream.

3.5.4.1. Implementing Switchable Plans

For a switchable plan chosen by Rio during optimization, the execution-plan generator creates a switch operator and a buffer operator. Figure 24 shows these two operators generated for the switchable plan in Figure 22. Note that the buffer operator is placed above the common subplan for R ⋈ S (marked in gray in both figures). The switch operator is placed above the buffer operator.
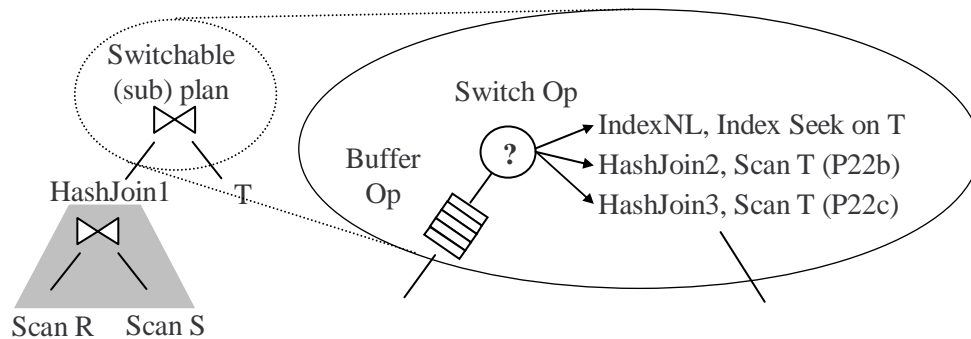
Figure 24 – Implementation of switchable plan from Figure 22

During query execution, the buffer operator buffers tuples from the deep subplan until it gets an *end-of-sample punctuation eos(f)*. (Generation of such punctuations is described in Section 3.5.4.2.) Punctuation *eos(f)* signals that the set of tuples buffered so far is an *f* % random sample of the output of the deep subplan. Based on the number of buffered tuples *n*, $100n/f$ is a fairly accurate estimate of the final output cardinality of R⋈S. The switch operator uses this cardinality estimate to compute the total input size of R⋈S, and instantiates the appropriate member plan.

Rio currently uses only the size of the deep subtree input R_D to the join to choose the best member plan. In terms of Figure 20 (repeated below in Figure 25a), this limitation means that for a switchable plan $P=\{P_{lo}, P_{est}, P_{hi}\}$, where plans P_{lo}, P_{est}, and P_{hi} were chosen for (lo_D, lo_B), (est_D, est_B), and (hi_D, hi_B)

respectively (Figure 25a), Rio has to choose among P_{lo}, P_{est}, and P_{hi} based solely on the estimate of $|R_D|$.

P_{lo} is picked if $|R_D| \in [lo_D, (lo_D + est_D)/2[$, P_{est} is picked if $|R_D| \in [(lo_D + est_D)/2, (est_D + hi_D)/2[$, and P_{hi} is picked if

$|R_D| \in [(est_D + hi_D)/2, hi_D]$. If $|R_D| < lo_D$ or $|R_D| > hi_D$, then the switch operator triggers re-optimization after

adding the collected estimate of $|R_D|$ to the catalog (Figure 25b).

a) Bounding box for switchable plan P={P_{lo}, P_{est}, P_{hi}}
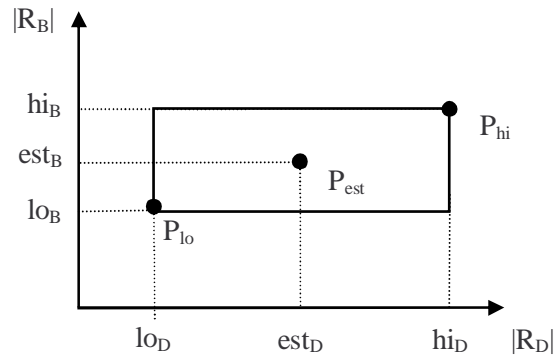
b) Deciding between re-optimization, P_{lo}, P_{est}, and P_{hi} based on $|R_D|$
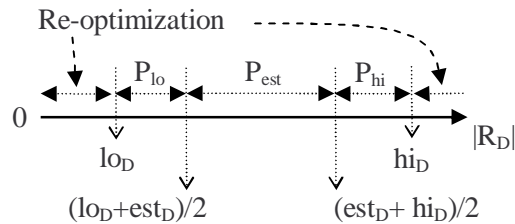
Figure 25 – Bounding box (a); and which plan to choose (b)

3.5.4.2. Random-Sample Processing During Execution

To generate *eos(f)* punctuations required by buffer operators, we altered the regular processing of some of Predator's operators so that, with minimal overhead, they can prefix their output with a random sample of their entire output. Each such operator *O* first outputs an *f* % random sample of its entire output. (*f* is a user-defined parameter.) Next, *O* generates an end-of-sample punctuation *eos(f)* to signal the end of the sample. Finally, *O* sends its remaining output tuples. As shown in Figure 26, tuples output as part of the random sample are not generated again.

Normal output order without randomization

| a | b | c | d | e | f | g | h | i | j |

Output order with randomization

| h | a | e | b | c | d | f | g | i | j |

A random sample of the output

Followed by the rest of the output

Emits punctuation *eos(30%)*
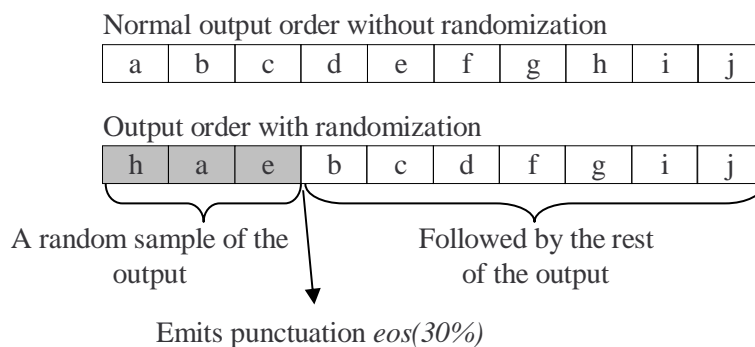
Figure 26 – Random samples in the operator output

Reordering the output of an operator O is not an option if any of the operators above O in the plan depend on the order of O's output. Thus, random sample generation seems inapplicable to operators such as sorts and ordered scans from B-trees. However, there are ways around this problem. For example, the buffer operator above O can regenerate the order using a merge of the initial sample with the later output. Furthermore, blocking operators[9] like sorts provide simpler ways of estimating input sizes without requiring random samples or buffering.

Next we describe how *eos(f)* punctuations are generated by table scans and certain join operators. Note that our techniques never transform a non-blocking operator into a blocking operator.

3.5.4.3. Randomization in Table-Scan Operators

We developed two techniques to enable a scan operator over a table T to first return a random sample of tuples from T:

(i) If tuples in T are laid out in random order on disk, a sequential scan will produce the tuples in the desired order. Whether T has a random layout pattern or not can be a physical property of the table, enforced when the table is created and updated. Additionally, such a layout pattern can be detected

---

[9] A blocking operator reads all of its input before producing any output.

using the Kiefer-Kolmogorov-Smirnov test when *runstats* is invoked to collect statistics on *T*; see [18]. This additional statistic can be maintained in the catalog.

(ii) An *f* % random sample of *T*, denoted *T_sample* , can be maintained explicitly as a separate table, e.g., using the techniques from [38]. Each tuple in *T* contains an extra bit to denote whether the tuple is also present in *T_sample* or not. At run-time the table scan first returns tuples from *T_sample* , followed by an *eos(f)*. Then it scans *T*, returning all tuples not contained in *T_sample* . Note that having tuples duplicated in *T_sample* and *T* allows indexes over *T* to be built and used without any changes. The storage overhead is minimal.

3.5.4.4. Randomization in Join Operators

Adding randomization to the nested-loops join operators–tuple, block, and index–was straightforward. These operators simply pass on the *eos(f)* punctuations from their outer input, and ignore *eos(f)* from their inner input. A join sample produced in this fashion is a true random sample of the join if the outer table's join column is a foreign key referencing the inner table [1].

To produce a random sample first from a hybrid hash join, we made the following modifications to the standard algorithm:

(i) First, tuples from the probe input are read into memory until an *eos(f)* punctuation is received. These tuples represent an *f* % sample of the complete probe input. The join operator inserts these tuples into an in-memory hash table.

(ii) Next, the build input is read and partitioned completely. In addition, as these tuples are being processed, they are immediately joined with the in-memory sample of the probe input. Joining tuples are sent in the join output. At the end of this phase, an *eos(f)* punctuation (using the value of *f* received from the probe) is generated, and the in-memory sample is discarded. The tuples output so far correspond to taking an *f* % sample from the probe and joining it with the complete build. This sample is guaranteed to be a true join random sample if the probe input's join column is a foreign key referencing the build input [1].

(iii) The scan of the probe input, which was paused after the *eos(f)* in Step (i), is resumed. The tuples are partitioned and joined with the memory-resident build partitions.

(iv) The on-disk partitions are joined to complete the join.

3.6. Experimental Evaluation

In this section we describe an extensive experimental evaluation of the Rio prototype. We compare Rio with the traditional optimizer (termed TRAD in Section 3.5.1) and with the Validity-Ranges re-optimizer (termed VRO in Section 3.5.1) under a variety of conditions. In our experiments we used a synthetic data generator provided by IBM. The generated dataset has four tables whose properties are shown in Table 4.

Table 4 – Summary of dataset used in the experiments

Table	Size, # of Tuples	Sample Correlated Attributes
Accidents (A)	420 MB, 4.2 M	accident_with & damage, seat_belt_on & driver_status
Cars (C)	120 MB, 1.7 M	make & model & color
Owner (O)	228 MB, 1.5 M	city & state & country
Demographics (D)	60 MB, 1.5 M	age & salary & assets

All experiments were done on a 1.7 GHz Pentium machine with 2 MB L2 cache, 512 MB memory, and a single 5400 rpm disk. The buffer cache size is 128 MB. Each hybrid hash join operator is allocated a fixed amount of memory which we vary in some of the experiments; the default value is 50 MB. Buffer operators in Rio and VRO are allocated the same amount of memory as a hybrid hash join. The buffers spill to disk when they fill up. B-tree indexes were available on all primary-key attributes. Equi-height and end-biased histograms were available on all integer attributes. The bounding box computation in Rio happens as described in Figure 19 with $\Delta_+=0.6$ and $\Delta_-=0.1$. The cost threshold for robustness tests is 20% (Section 3.5.3.2). The random-sample percentage for size estimation is 1% (Section 3.5.4.2).

3.6.1. Two-Way Join Queries

Our first experiment studies the performance of TRAD, VRO, and Rio with respect to the error in estimates. We use a query joining Accidents (A) with Cars (C) on the car_id attribute. (All joins we consider are foreign key to primary key joins.) There is a selection predicate on A, denoted $\sigma(A)$, of the form A.accident_year>[year], where [year] is a parameter whose value is varied in this experiment. We removed the equi-height histogram on attribute A.accident_year from the catalog to force the optimizer to use the default selectivity estimate of 0.1. Thus, the optimizer always estimates $|\sigma(A)|$=42MB. By varying the value of [year], we vary the error between the estimate of $|\sigma(A)|$ and its actual size.

3.6.1.1. Using Robust Plans

The memory limit for a hybrid hash join was set to 150MB in this experiment. When $|\sigma(A)|$ is less than the size of C (120MB), the optimal plan is a hybrid hash join with $\sigma(A)$ as the build, denoted plan P_{AC}. When $|\sigma(A)|$>120MB, the optimal plan is a hybrid hash join with C as the build, denoted plan P_{CA}. (120MB corresponds to around 180% in Figure 27.) Although B-tree indexes are available on the join attributes, index-nested-loop joins never outperform hybrid hash joins in our setting.

Figure 27 shows query completion times, including both optimization and execution times, for TRAD, VRO, and Rio as we vary the error in the estimate of $|\sigma(A)|$. The error plotted on the x-axis is computed as $|\sigma(A)|_{Actual}$ / $|\sigma(A)|_{Estimate}$ - 1. A positive error indicates an underestimate and a negative indicates an overestimate. Figure 27 also shows the performance of the optimal plan which we determined manually in each case.

Since the optimizer's estimate of $|\sigma(A)|$ is 42MB, TRAD always picks plan P_{AC} which is optimal at $|\sigma(A)|$=42MB. As $|\sigma(A)|$ is increased (and the estimation error increases), the cost of plan P_{AC} increases linearly at a small rate until $|\sigma(A)|$=150MB. ($|\sigma(A)|$=150MB corresponds to an error around 2.5 in Figure 27.) When $|\sigma(A)|$>150MB, the hybrid hash join in plan P_{AC} starts spilling to disk. Because of this extra

IO, the cost of plan P_{AC} increases at a steep rate when $|\sigma(A)|>150MB$, as shown by the plot for TRAD in Figure 27.

VRO always starts with the same plan as TRAD, i.e., plan P_{AC}. However, VRO adds a validity range to the join and verifies this range before starting the join execution. The upper bound of the validity range for the hybrid hash join in plan P_{AC} is 120MB: if $|\sigma(A)|>120MB$, then plan P_{CA} performs better. Therefore, as long as $|\sigma(A)|\leq120MB$, the validity range is not violated and the performance of VRO matches the performance of the optimal plot in Figure 27. When $|\sigma(A)|>120MB$, the validity range is violated and VRO is forced to re-optimize. Plan P_{CA} is picked on re-optimization. VRO cannot reuse the work done by the pipeline in execution in plan P_{AC} when re-optimization was invoked, namely the scan of A and evaluation of $\sigma(A)$ up to that point. This loss of work results in the region in Figure 27 where VRO performs worse than TRAD. However, as the error increases, the re-optimization pays off quickly because when $|\sigma(A)|>150MB$, the join in plan P_{AC} spills to disk while P_{CA} scans A and C only once.
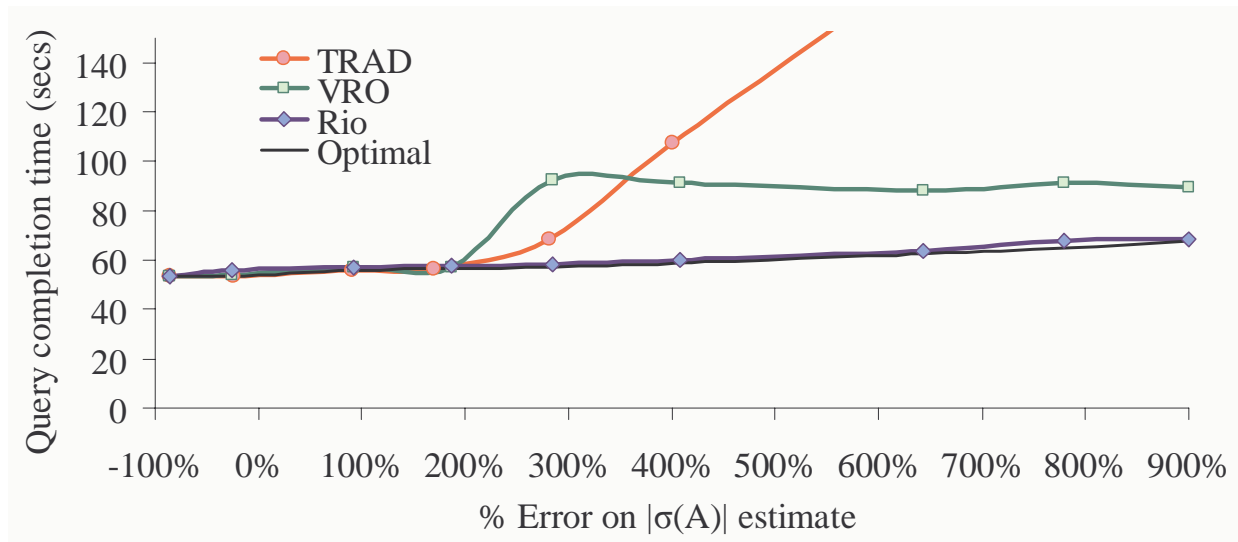
Figure 27 – $\sigma(A)\bowtie C$, 150MB per hash join

Rio first computes bounding boxes for $|\sigma(A)|$ and $|C|$. Since there are no selection predicates on C, the estimate of $|C|$ available from the catalog is accurate. To illustrate robust plans, in this experiment alone

we set Δ_+ and Δ_- in Figure 19 to very high values so that the bounding box on $|\sigma(A)|$ is [0MB, 420MB]. Rio identifies that Plan P_{CA} is a robust plan within this bounding box. (Rio identifies Plan P_{CA} to be a robust plan even if the bounding box is smaller.) Because the bounding box [0MB, 420MB] covers the entire range considered in the experiment, Rio runs Plan P_{CA} at all points in Figure 27. Although Plan P_{CA} is not optimal at all points in the bounding box, note that Rio's performance is close to the optimal plot at all points in Figure 27, showing the robustness of Plan P_{CA}. Since $|C|$ is less than the memory available to the hash join, P_{CA} always finishes in one scan of A and C.

For our default settings of Δ_+ and Δ_-, the bounding box on $|\sigma(A)|$ is [16.8MB, 193.2MB]. In this case Rio used a combination of solutions (re-optimization, switchable plans, and robust plans) to provide near-optimal performance. This graph is omitted because Section 3.6.1.2 shows Rio's performance in a similar situation.

3.6.1.2. Using Switchable Plans

Our next experiment, reported in Figure 28, considers the same query as in the previous section, but now hash joins are allocated only 50MB of memory for in-memory hash partitions. In this experiment, the behavior of Optimal, TRAD, and VRO regarding the choices of plans and re-optimization points are the same as in the previous section. However, Rio behaves differently. Rio computes the bounding box on $|\sigma(A)|$ to be [16.8MB, 193.2MB]. The large width of the box corresponds to the high uncertainty in $|\sigma(A)|$ since this estimate used a default value of selectivity. The bounding box on $|C|$ has zero width since an accurate estimate of $|C|$ is available from the catalog. Rio finds that plan P_{AC} is optimal at $(|\sigma(A)|,|C|)=(16.8MB, 120MB)$, which is the lower corner of the bounding box, and also at the estimated point $(|\sigma(A)|,|C|) = (42MB, 120MB)$. However, for $(|\sigma(A)|,|C|) = (193.2MB, 120MB)$, which is the upper corner of the bounding box, plan P_{CA} is optimal. Furthermore, neither P_{AC} nor P_{CA} is robust in this case. However, Rio identifies that plans P_{AC} and P_{CA} are switchable plans (see Section 3.5.3). Therefore, for this query, Rio starts with a plan containing a switch operator with the two hybrid hash joins

corresponding to P_{AC} and P_{CA} as member plans. Rio estimates $|\sigma(A)|$ during execution. Based on this estimate, Rio chooses one of the two joins or it re-optimizes.

The accident_year attribute in A is not correlated with the layout of A on disk, so a sequential scan of A produces tuples in random order to estimate the selectivity of $\sigma(A)$ (recall Section 3.5.4.2). Rio gets a very accurate estimate of $|\sigma(A)|$ from the default setting of 1% sampling. For example, when $|\sigma(A)|=6MB$ in Figure 28, which corresponds to an error of -85% and lies outside the bounding box, Rio invokes re-optimization. Since the optimizer now has accurate estimates of $|\sigma(A)|$ and $|C|$, it correctly picks plan P_{AC} which is optimal at this point. Note that Rio's performance is very close to that of the optimal plan for $|\sigma(A)| = 6MB$, which shows that the overhead incurred by Rio to sample 1% of A, obtain a run-time estimate of $|\sigma(A)|$, and to re-optimize the query is very small.

When $|\sigma(A)|$ lies within the bounding box computed by Rio, re-optimization is avoided. In this case, the switch operator picks plan P_{AC} or plan P_{CA} appropriately, avoiding loss of work. For example, the switch operator picks Plan P_{AC} when $|\sigma(A)|=32MB$, which corresponds to an error of -26% in Figure 28. Plan P_{CA} is picked when $|\sigma(A)|=160MB$, which corresponds to an error of 284% in Figure 28. When $|\sigma(A)|>193.2MB$, which lies outside the bounding box, Rio will re-optimize with a fairly accurate value of $|\sigma(A)|$ estimated via sampling. In this case, the optimal Plan P_{CA} gets picked. Therefore, Rio's performance is always close to that of the optimal plan for this query.
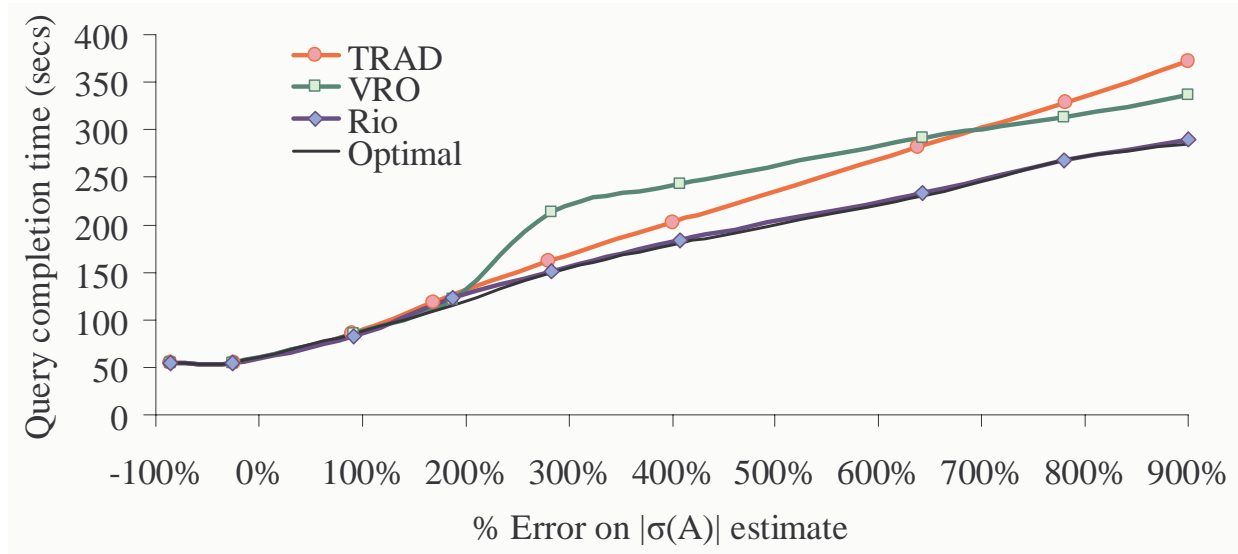
Figure 28 – σ(A)⋈C, 50MB per hash join

3.6.2. Three-Way Join Queries

We now repeat the experiments in Section 3.6.1 with a query joining A, C, and O. There are selection predicates on A.accident_year (σ1) and O.cars (σ2). We removed the equi-height histogram on A.accident_year so that the optimizer uses a default estimate, and we vary the estimation error as in Section 3.6.1. The results are shown in Figure 30. The cardinality of σ2(O) is estimated accurately from an equi-height histogram.

The optimal plan for this query for low values of |σ1(A)| is plan P29a shown in Figure 29. For higher values of A, plan P29b becomes optimal. Plan P29a is also the optimal plan for the single-point estimates of input sizes, hence TRAD always picks plan P29a. Therefore, in the left part of Figure 30, TRAD performs as well as the optimal plan, but its performance deviates more and more from the optimal as the error increases.
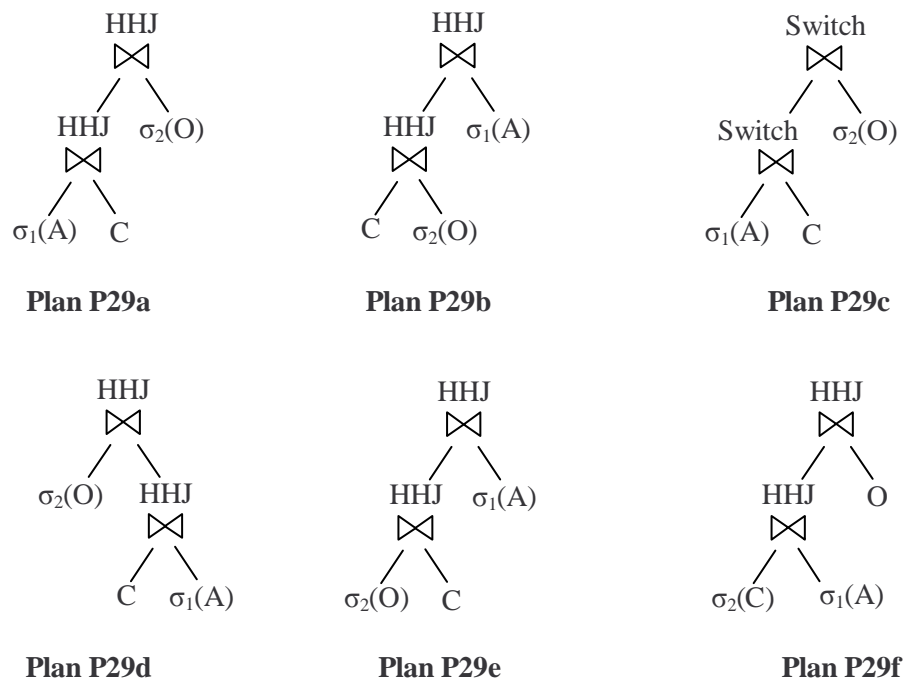
Figure 29 – Plans for A⋈C⋈O used in experiments
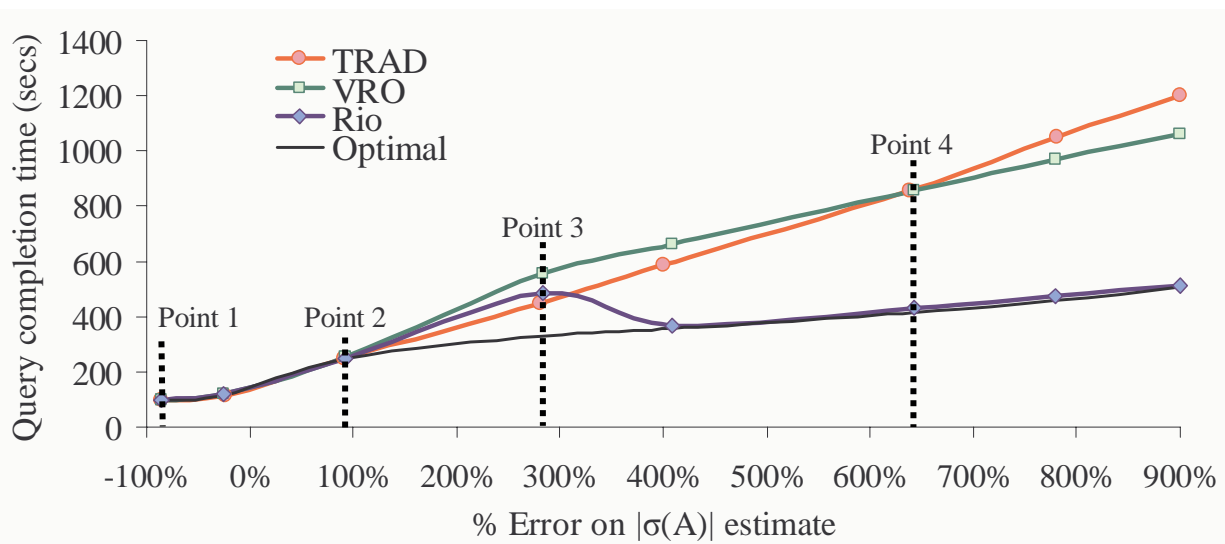
Figure 30 – σ1(A)⋈C⋈σ2(O), 50MB per hash join

Rio starts with plan P29c shown in Figure 29. This plan has two switch operators corresponding to the two joins. (Buffer operators are not shown in Figure 29.) The two member plans in the first switch

operator are (i) hybrid hash join with σ1(A) as build and C as probe, and (ii) hybrid hash join with C as build and σ1(A) as probe. The switch operator will choose between these plans based on a run-time estimate of |σ1(A)| computed from a 1% sample of A. The two member plans in the second switch operator are (i) hybrid hash join with σ1(A)⋈C as build and σ2(O) as probe, and (ii) hybrid hash join with σ2(O) as build and σ1(A)⋈C as probe. The choice between these two plans will be made based on an estimate of |σ1(A)⋈C| from a 1% sample of σ1(A)⋈C obtained by sampling the join (recall Section 3.5.4.4). The bounding box on |σ1(A)| is the same as that in Section 3.6.1. The bounding boxes on |C| and |σ2(O)| effectively have zero width since these estimates are known to be accurate. When |σ1(A)|=6MB (Point 1 in Figure 30 and in Table 5), which corresponds to an error of -85% and lies outside the bounding box, Rio invokes re-optimization and picks the optimal plan P29a. Similarly, when |σ1(A)|=160MB (Point 3 in Figure 30 and in Table 5), which corresponds to an error of 284% and is within the bounding box, both switch operators will pick the base relation input as the build, and execute plan P29d in Figure 29. Thereby, when |σ1(A)|=160MB, Rio avoids re-optimization and the loss of pipelined work which results in the difference of around 72 seconds between Rio and VRO in this case.

Table 5 – Plans used at points 1, 2, 3, and 4 of Figure 30

Point	|σ1(A)|	TRAD	VRO	Rio	Optimal
1	6 MB	P29a	Inside validity range, runs plan P29a	Outside bounding box, re-optimize, picks plan P29a	P29a
2	80 MB	P29a	Inside validity range, runs plan P29a	Inside bounding box, switch operator picks plan P29a	P29a
3	160 MB	P29a	Outside validity range, re-optimize, picks plan P29d	Inside bounding box, switch operator picks plan P29d	P29b
4	310 MB	P29a	Outside validity range: re-optimize, picks plan P29d	Outside bounding box, re-optimize, picks plan P29b	P29b

The performance of Rio is always close to that of the optimal plan in Figure 30 except for an intermediate range of estimation errors. In this region, Rio picks plan P29d which turns out to be suboptimal compared to plan P29b. This region is a transition region where plan P29d stops being optimal

with respect to plan P29b. Because of an overestimate of the join selectivity of C⋈σ2(O), Rio continues to pick plan P29d as the optimal plan beyond the actual transition point. However, as the error in |σ1(A)| increases, Rio converges to the optimal plan again around an error of 400% in Figure 30.

VRO starts with the same plan P29a as TRAD, but with validity ranges added. When |σ1(A)|≤120MB, none of the validity ranges are violated. (|σ1(A)|=120MB corresponds to around 180% in Figure 30.) When |σ1(A)|>120MB, the validity range on σ1(A)⋈C is violated and VRO is forced to re-optimize. Note that at this point, VRO does not have an estimate of the actual size of |σ1(A)|. Based on the amount of A it has seen so far, VRO always picks plan P29d on re-optimization and adds validity ranges. In addition to the overhead of re-optimization and the loss of pipelined work, the choice of plan P29d illustrates another problem with VRO. VRO gets stuck in a suboptimal plan as the validity ranges in plan P29d will never fail because of an underestimate of |σ1(A)|: there is no better plan to join C and σ1(A) for large |σ1(A)| than the hybrid hash join with σ1(A) as the probe, even though there is a better plan for the entire query. Although Rio could also fail to detect a sub-optimal plan, it is the fact the VRO uses incorrect estimated statistics during the second optimization call that leads it to execute a sub-optimal plan. Since Rio collects accurate statistics at run-time it is less likely to execute sub-optimal plans after re-optimization calls.

A similar situation arises for the second join since σ1(A) is part of the probe input here as well. Hence, as illustrated by the results in Figure 30, VRO performs badly as the estimation error in |σ1(A)| increases. This experiment illustrates one of the pitfalls of reactive re-optimization where the execution plan is decided before the issues affecting re-optimization are considered.

3.6.3. Correlation-Based Mistakes

So far the estimation errors we considered were due to selection predicates on an attribute on which there was no histogram. A more common case of estimation errors is the presence of correlated attributes, which we consider in this section. We use a three-way join query on A, C, and O with selection predicates

σ1(A) and σ2(O). Figure 31 shows the performance of three queries Q1, Q2, and Q3 which have different sets of correlated predicates on A, causing the optimizer to underestimate |σ1(A)| in each case. (Correlations usually lead to underestimates [63].) For example, Query Q2 contains predicates A.accident_with="car", A.driver_status="injured", and A.seat_belt_on="on". |C| and |σ2(O)| are always estimated accurately. Figure 31 indicates that the estimation errors caused by correlated attributes result in performance trends for TRAD, VRO, and Rio similar to those shown in Sections 6.1 and 6.2. The reasons for these trends are also similar to those observed in Sections 6.1 and 6.2. The optimal plan for each query is plan P29e in Figure 29 which Rio picks either because it is a robust plan (Q1) or because Rio discovers the estimation error and the actual estimate quickly because of randomization (Q2 and Q3).
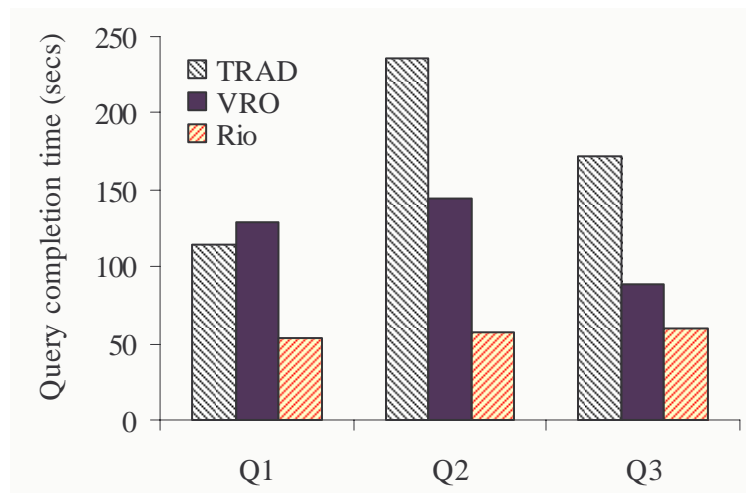
Figure 31 – Errors due to correlated predicates

3.6.4. Thrashing

So far we considered queries where the size of a single input is estimated incorrectly. In this section we consider the performance of VRO and Rio when the size of more than one input is estimated incorrectly. We use a three-way join query on A, C, and O with selection predicates σ1(A) and σ2(C). |σ1(A)| is underestimated significantly because σ is on an attribute with no histograms, while |σ2(C)| is underestimated slightly because the histogram on the corresponding attribute was built from a small

sample of C. For this query, VRO thrashes and takes 690.38 seconds compared to 327.57 seconds for Rio. VRO starts with the optimal plan for the estimated statistics which is similar to plan P29a in Figure 16. Because $|\sigma2(C)|$ is underestimated, VRO computes an incorrect validity range for $|\sigma1(A)|$. This validity range is violated at run-time, and re-optimization picks plan P29f. Since VRO does not have correct estimates of $|\sigma1(A)|$ or $|\sigma2(C)|$ at this point, it computes incorrect validity ranges which fail again. This thrashing results in the factor two slowdown of VRO compared to Rio. Rio invokes re-optimization once for this query when its run-time estimate of $|\sigma1(A)|$ falls outside the bounding box. Because Rio estimates $|\sigma1(A)|$ accurately at run-time using sampling, and also uses bounding boxes to allow for error in the estimate of $|\sigma2(C)|$, it finds the optimal plan in the first re-optimization step.

3.6.5. Increasing Query Complexity

In this section we compare the relative performance of TRAD, VRO, and Rio as we increase the number of joins in the query. The results are shown in Figure 32.
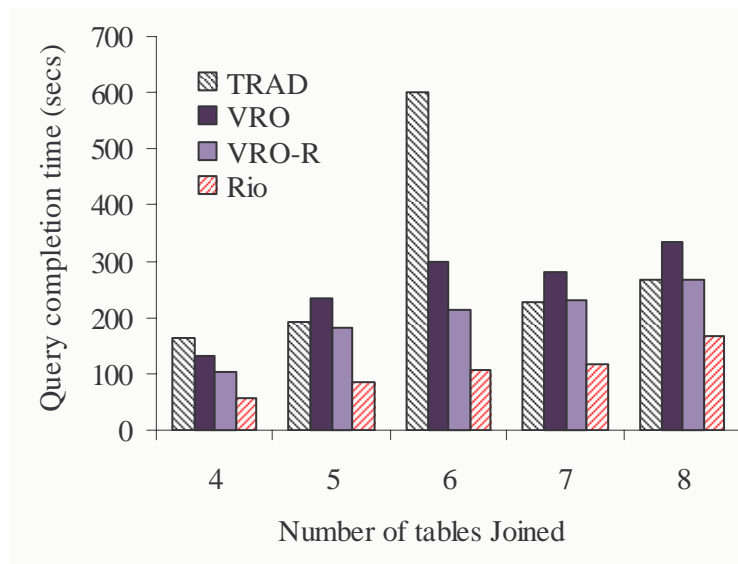
Figure 32 – Increasing query complexity

The dataset provided to us had four tables only (the actual dataset has around 30 tables [63]). For this experiment, we vertically partitioned each table into two and padded each partition with string fields to make it the same size as the original table. Each query had correlated predicates on half of the joined tables. Figure 32 shows the same trends observed in previous sections. The fraction of time spent by Rio and VRO in optimization steps was less than 1.7% in all cases in Figure 32. Roughly, the cost of each optimization phase in Rio is three times the cost of the single optimization phase in TRAD.

Figure 32 also shows the relative performance of VRO-R, which is the validity-ranges optimizer enhanced with our random-sample processing techniques from Section 3.5.4.2. While randomization improved the overall performance of VRO by reducing the time required to trigger re-optimization, the amount of wasted work, and the number of re-optimization steps, Rio still outperforms VRO-R by a significant amount.

3.7. Conclusions

Rio is a second-generation AQP system that improves on previous AQP proposals in several ways. Previous AQP systems, although able to correct some optimizer mistakes, still suffered from the optimizer mistakes in an indirect way. By using a traditional optimizer, those systems were more likely to start execution with a sub-optimal plan. The sub-optimal plan could frequently be detected and replaced by an optimal plan. However, the process could be inefficient if the sub-optimal plan was not quickly detected, or, if it was, query processing work had to be thrown away and repeated. In addition, previous AQP approaches would sometimes only find an optimal plan after multiple re-optimization steps, each with its own potential inefficiencies. With Rio, we extensively re-engineered query optimization and query execution to make the system as insensitive as possible to optimizer mistakes. The optimization module was changed to give priority to robust plans, i.e., plans insensitive to incorrect estimates. This was partially done with a series of heuristics that assign levels of uncertainty to estimated statistics based on the way they were estimated. Given the uncertainties, the optimizer computes bounding boxes around the

estimates, i.e., intervals of estimated low and estimated high values for the statistics. The bounding boxes of statistics are combined during optimization to produce estimated low and estimated high costs for plans instead of using a single value as in traditional query processing. This, in turn, reveals which plans are robust and which plans are risky.

New switch operators were proposed to avoid re-optimization and minimize lost work by deferring the decision of which plan to run until run-time. Switch operators require switchable plans to be found at optimization time. At a high level, a switchable plan is a set of plans that share a common execution prefix, i.e., all plans in the set must start executing the same operator(s) over the same relation(s) and in the same order to be switchable. The common part of the switchable plan set is partially executed first. Then, the switch operator receives statistical information about the data seen in that common execution prefix and chooses the best plan in the switchable plan set to finish execution. While it is unknown how frequently switchable plans can be found for complex optimization spaces, our experiments showed that switchable plans can be used to avoid lost work due to common optimizer mistakes.

Finally, query execution was changed to generate and propagate tuple random samples up the query execution tree. The tuple random samples were used to generate run-time estimates for the switch operator, allow faster detection of sub-optimal plans, and faster convergence to the optimal plan. However, our modifications of join operators are guaranteed to produce a true random sample of the join only in certain situations. In addition, although we suggested some ways to propagate tuple random samples across operators that take advantage of ordered tuple streams, this issue was not addressed in this work. Nevertheless, tuple random samples were generated and propagated at marginal costs during query execution for a variety of operators including some join operators.

These three components of proactive re-optimization, robust plans, switchable plans, and efficient run-time statistics estimation through propagation of tuple random samples, were implemented in the open source database Predator in a prototype called Rio. Rio was compared with previous reactive re-optimization approaches and with traditional non-adaptive approaches. Rio proved to be less likely to

start with a risky plan, faster to detect sub-optimalities, converged to the optimal plan in fewer steps, and

showed total execution times up to three times faster than the reactive re-optimization approach.

CHAPTER 4

ADAPTIVE AND ROBUST QUERY PROCESSING WITH SHARP

This chapter describes SHARP, a new multi-join, adaptive, relational operator that joins three or more relations of a star-join. SHARP reduces the possible impact of optimizer mistakes by determining which plan to execute independently of optimization estimates. During normal query processing, SHARP collects statistics, and by using a combination of late-binding plan decisions and tuple routing strategies, it is able to change join order and table access methods. Unlike previous tuple routing operators used for in-memory stream processing, SHARP was designed to process local relations with sizes much larger than available memory.

4.1. Introduction

As described in Chapter 1, database optimizers cost and choose query plans as if they have precise information about data distributions. However, that is rarely the case. When statistics are not available in the catalog, the optimizer estimates them by assuming that some data distributions are uniform or independent, by using a combination of other (possibly estimated) statistics, or even by using default values [79]. These estimates may contain errors that grow exponentially with the number of estimated statistics derived from other estimated statistics [51] and the chosen plans may be sub-optimal by several orders of magnitude [63]. Having more information in the catalog (e.g., histograms [70]) reduces the problem, but the information needed to correctly cost all possible query plans is likely to increase exponentially as datasets sizes grow, as queries become larger, and as query languages become more complex. If that is the case, then database optimizers may have insufficient information to choose good,

non-adaptive query plans for all queries. Instead, decisions about which query plan to run may have to be made at run-time–using adaptive operators and/or late binding decisions–after some of the data is observed.

One AQP approach is proactive re-optimization, examplified by the Rio prototyped described in the previous chapter. However, in spite of all its benefits, Rio required substantial changes to the query optimization and query execution modules including small modifications in all operators to generate or propagate tuple random samples. Rio is able to take corrective actions and yet avoid losing work but only in the presence of switchable plans–sets of plans that share a common execution prefix. However, it is not clear if switchable plans can be found for complex queries prone to optimizer mistakes. In addition, switchable plans cannot correct sub-optimal join orders, a common optimizer mistake.

By contrast, in this chapter we propose SHARP[10], an AQP strategy fundamentally different from Rio. The proposed SHARP operator is able to correct optimizer mistakes leading to the execution of sub-optimal operators and sub-optimal join orders. In addition, SHARP encapsulates almost all AQP changes needed; the remaining query processing engine is largely unaffected.

However, instead of executing arbitrarily query plans, SHARP is an adaptive, relational operator for processing star-joins with three or more relations. In addition, instead of being able to preempt execution and re-invoke optimization at any moment, SHARP adopts a two-step adaptive approach. First, run-time late-binding decisions determine the driving relation–the first relation to be read and the "outer side" of subsequent joins. Second, tuple routing continuously potentially changes the join order within the orders available after the driving relation was fixed.

Note that, although tuple routing has been previously used mainly for in-memory data processing, SHARP does not keep all the joins completely in memory. This allows SHARP to have both a smaller

---

[10] Streaming, Highly Adaptive Run-time Planner

memory footprint than other adaptive operators [32, 90], and to have an efficient second pass to process relations much larger than available memory.

4.1.1. Contributions and Outline

The main contributions of this chapter are the following:

- In Section 4.3.1, we introduce SHARP, a new, multi-join, adaptive, operator to process star-joins.

- We show that tuple routing policies used in data stream systems can be used in traditional databases processing relations larger than memory. We also provide the first apples-to-apples comparison of three tuple routing policies [5, 8, 29] in the same system. These policies are described in Section 4.3.2.

- We propose a series of late-binding decisions that can opportunistically change the query plan at run-time to improve performance. These decisions, described in Section 4.3.3, are taken after SHARP has seen some tuples, but before deciding on the final execution plan.

- We propose a new multi-join second-stage processing algorithm in Section 4.3.4. This algorithm shows good improvements over alternative techniques and its performance is insensitive to optimizer mistakes.

- As described in Section 4.4, we implement and evaluate a prototype implementation of SHARP in Predator [81]. The results show good performance improvements over plans not using SHARP.

4.2. Eddies and MJoins

The operators most related to SHARP are the Eddy [5] and the MJoin [90], both multi-join adaptive operators using tuple routing. They are described here to provide context for the SHARP contributions in the next Section. Other related work appears in Section 4.5.

4.2.1. Terminology

For an operator *Op* joining two or more relations, we say relation *B* is a *build* relation when tuples from that relation are inserted into some lookup structure (e.g., hash tables). We say relation *D drives Op*, or is a *driving* (or *pobing*) relation for *Op*, if each input tuple t_1 coming from *D* probes the build lookup structures of *Op* and potentially produces output tuples, or schedules t_1 for second-stage processing, before any other tuple t_2 from *D* is processed[11]. A relation may be simultaneously a build and driving relation. In the figures, driving relations are marked with arrowed lines and build relations are marked with dotted lines.

4.2.2. The Eddy

As described in Section 2.1.1, the Eddy [5] is an operator that routes tuples through a pool of operators until they are processed by all operators or are dropped along the way. The Eddy continuously observes the performance of the other operators and routes tuples to the most efficient operator available. The Eddy adapts its routing decisions as the performance of the other operators change, possibly sending different tuples through different routes throughout the life of a query. (However, at any single moment, most tuples follow the same route.) The ability to efficiently change routes (i.e., query plans) relies on operators with *moments of symmetry* [5], moments after which joins can be reordered. The symmetric hash join (SHJ) [91], typically used with Eddies, is an operator with frequent moments of symmetry. Each SHJ consists of two in-memory hash tables[12], one for each relation being joined; tuples from one relation build into its hash table and probe the other. An Eddy with SHJs can then execute several plans, depending on the tuple source and routing policy. For example, the Eddy in Figure 33b executes

---

[11]   For example, in a nested-loops join operator the left input is the driving source, and in a hybrid hash join operator the right input is the driving source.

[12]   Each one of the two hash-tables that composes a SHJ is called a SteM in the Eddies nomenclature [74].

R⋈_aS⋈_bT by sending R tuples to first probe hash table S.a and then probe T.b, T tuples first probe S.b and then R.a, and S tuples have two options: either they first probe T.b and then R.a, or first they probe R.a and then T.b (hash tables are represented as grey rectangles in Figure 33). This design, albeit providing very adaptive plans, introduces a considerable overhead [29]: it requires maintaining two hash tables per join and requires that all joins be completely and simultaneously in memory (e.g., the Eddy of Figure 33b needs to maintain the four hash tables, R.a, S.a, S.b, and T.b in memory). Although the Eddy has the potential to join any number of relations in any order, its memory limitations restrict the Eddy for in-memory processing of *data streams* (possibly infinite, window-bounded, remote tuple sources that deliver tuples at unpredictable and bursty rates).
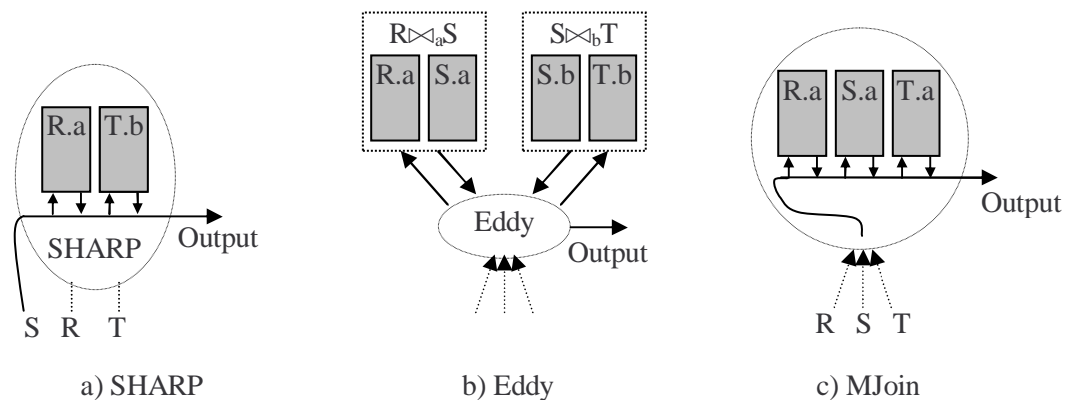
Figure 33 – SHARP, Eddy, and MJoin processing R⋈S⋈T

4.2.3. The MJoin

The MJoin [90] is a completely symmetric multi-way data stream join operator, with one hash table per data stream. As with the Eddy using SHJs, tuples from a particular source build on that source's hash table and probe the others. The MJoin uses fewer hash tables than the Eddy because it assumes that a data stream uses the same joining attribute for the all joins (see Figure 33c). This assumption also allows more join orders in the MJoin than in the Eddy. For example, the MJoin of Figure 33b can process any of the

six join orders (RST, RTS, SRT, STR, TRS, and TSR), being restricted only by the incoming tuple source.

The important contributions of MJoins are i) producing tuples sooner than a tree of binary non-blocking join operators (e.g., SHJs), ii) extending the streaming behavior of SHJs to allow memory overflow, and iii) providing a rate-based cost model of the data stream join problem it addresses.

In contrast, we address the problem of joining local relations. Our goal is to execute plans that are insensitive to optimizer mistakes and our evaluation metric is time to completion. Other differences between the MJoin and SHARP are: the MJoin does not redistribute memory dynamically between joins, requires more memory then SHARP, does not evaluate routing policies, and, for the second-pass, assumes that all relations join on the same attribute.

4.3. SHARP

SHARP is an operator that keeps the inexpensive [29], tuple-routing, run-time adaptivity of the Eddy without incurring the overhead of SHJs [29] and without the requirement that all joins fit completely in memory. The trade-off is that, while Eddies and MJoins can process arbitrary plans, SHARP processes only star-joins and segments of linear-joins as shown in Figure 34. In spite of that, SHARP still has the potential to adaptively decide at run-time which join order to use. In addition to reducing memory usage, not using SHJs also allowed the development of a new technique to process joins between relations much larger than memory.

Queries:

```
        B
        |
A  —  E  —  C
        |
        D
```

A — B — C — D — E

Plans:

Star-Join
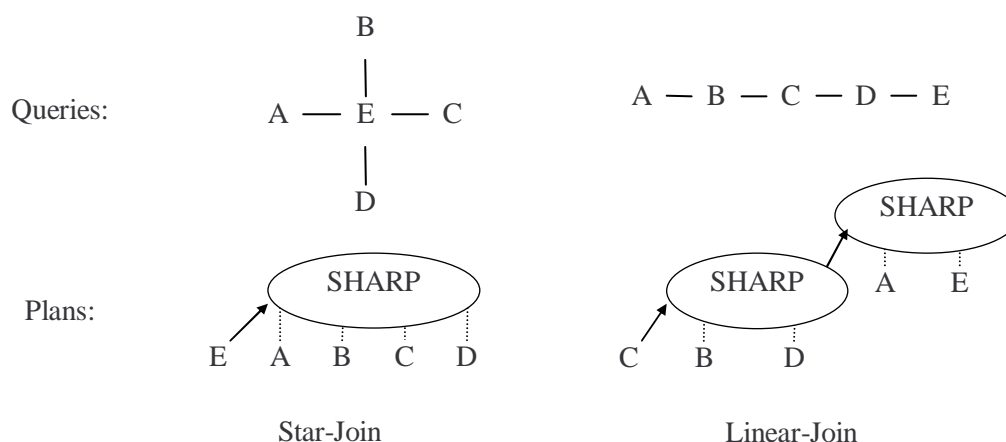
Linear-Join

Figure 34 – SHARPs processing a star-join and a linear-join

SHARP's functionality is described as follows. Section 4.3.1 describes the in-memory behavior of SHARP and minor multi-join improvements. Section 4.3.2 describes three tuple routing techniques implemented in SHARP. Section 4.3.3 introduces late-binding decisions that allow SHARP to change the query plan before tuple routing starts. Then, Section 4.3.4 describes how SHARP processes relations larger than memory. Finally, Section 4.3.5 summarizes SHARP and compares it with Eddies and MJoins.

4.3.1. In-Memory Processing

When a SHARP joins *n* relations, one relation is the single driving relation and all other *n-1* relations are build relations. SHARP starts by reading tuples from the build relations and creates an in-memory hash table for each one[13]. (Processing of build relations bigger than available memory is described in Section 4.3.4) Then, SHARP reads tuples from the driving relation, probes the in-memory build hash tables and outputs join results. Figure 33 (in page 84) shows a SHARP joining $R\bowtie_a S\bowtie_b T$, where R and T are the

---

[13]   An alternative to create an in-memory hash table for each build relation is to use, previously built indexes on the build relations. Depending on the number of driving tables and/or the selectivities of the joins, this alternative could be more effective then creating the in-memory hash tables.

build relations and S is the driving relation. Tuples from the driving relation–henceforth called *driving tuples* –probe the build hash tables in an order specified by an adaptive tuple routing policy, as described in Section 4.3.2.

Note that since driving tuples probe the build hash tables one by one, at any single moment, SHARP executes a linear-join plan; i.e., bushy-tree plans are never executed.

4.3.1.1. Adaptive Redistribution of Memory.

In SHARP, each build hash table is given a memory budget. If the total build size is larger than the budgeted amount, then the hash table must write hash partitions to disk for second-stage processing. However, before writing them to disk, SHARP first loads the remaining build tables into memory until they either consume their entire memory budget or load completely. If any budget is underutilized, SHARP reassigns the available memory to the yet to finish build hash tables. In contrast, the process of redistributing memory across joins is non-trivial for tree-shaped execution plans of binary operators. The process is more difficult because operators lower in the tree cannot obtain excess memory from operators higher in the tree as they have not begun execution.

Note that many memory redistribution policies are possible. For simplicity, SHARP assigns all unused build memory to the first hash table build that did not fit its budget. If that build completes without using all the newly assigned extra memory, SHARP further reassigns it to the next yet to finish build and so on.

4.3.1.2. Multi-Join Optimizations.

SHARP takes advantage of its multi-join nature to obtain two performance benefits. First, SHARP avoids creating some intermediate results: when a tuple *s* from driving relation S, probes build relation R and finds a matching tuple *r*, the resulting *rs* tuple is not generated. Instead, SHARP (like the Eddy's implementation in TelegraphCQ [21]) merely keeps a pointer to *r* and proceeds to probe the other build

relation T using tuple *s*. Then, an *rst* intermediate tuple is generated only if a matching tuple *t* of T is found. If the probe on T fails, no intermediate tuple is ever generated.

Another benefit is the reduction of getNext calls. Consider a traditional plan, using a binary tree of hash joins. Assume tuple *s* from S probes operator Op_1 (the in-memory build hash table for R) and gets a first matching tuple r_1. The resulting intermediate join tuple sr_1 is returned to Op_2 before any more calls are made to Op_1. Even if the sr_1 probe on Op_2 fails, the iterator model will try and get a new tuple from Op_1, r_2. The new intermediate tuple sr_2 will also probe Op_2 and fail because it is using the same *s* component that failed the previous probe on Op_2; see Figure 35. In constrast, when using a SHARP, if a driving tuple probes a build hash table and returns no matches, then any outstanding open probes on other build hash tables are closed and spurious getNext calls are avoided.

These two factors explain why SHARP shows a small performance advantage over trees of binary operators, even in scenarios where its adaptive mechanisms provide no benefit.

4.3.2. Adaptive Tuple Routing Strategies Used

In SHARP, we implemented three routing policies adapted from three previous proposals [5, 29, 8]. The first routing policy, which we call *Continuous* or simply *Cont*, is a modification on the original routing policy in the first Eddies paper [5]: a probabilistic routing mechanism based on lottery scheduling is used to determine where to route tuples next and routing decisions happen each time an operator finishes processing one tuple. The variation is that in *Cont*, we make a routing plan once per each driving tuple, instead of once per probe. In addition, instead of lottery scheduling, we route every *r*-th tuple to a random route. This makes the exploration mechanism independent of the currently estimated best route. For all other tuples *Cont* uses the estimated best routing order. Also, as in the Eddies implementation in TelegraphCQ [21], the selectivity of operators (the join selectivity of build hash tables in our case) is continuously updated after each probe and dropped tuples do not affect the selectivity of operators they do not probe.
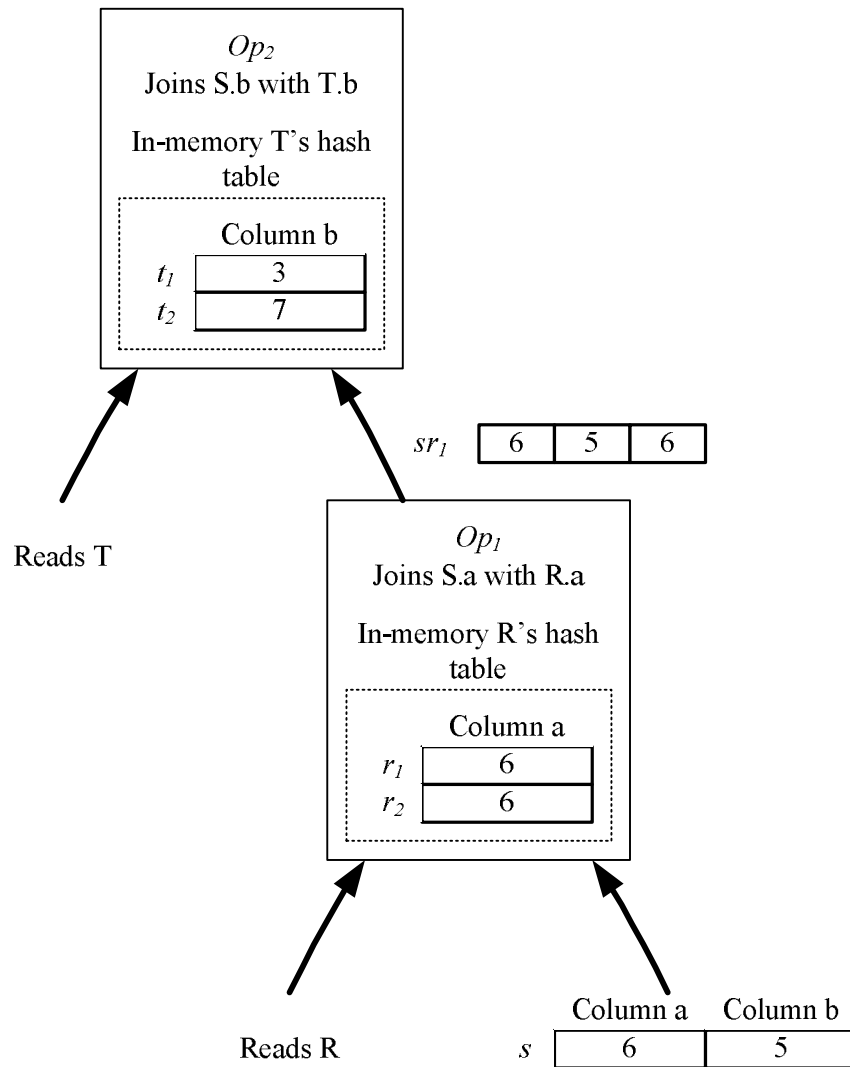
Figure 35 - Tuple _s_ from S probes ρ_1 and ρ_2 in iterator model

Continuous-Batch, or simply _ContB_, the second tuple routing policy implemented, is taken from Deshpande [29]: instead of computing a routing order once per driving tuple, routing orders are computed once per batches of tuples. Policies Cont and ContB minimize the overhead of gathering statistics–tuples are not used to explore operators after being dropped–but they provide no optimality guarantees on the chosen routes: they may take too long to discover new optimal routes or may never discover them.

An alternative is A-greedy, a routing policy that uses a small percentage of tuples, called _pofile tuples_ , to keep a _pofile window_ : a moving window of pass/fail bits for each operator [8]. Because the

profile window contains information even from otherwise dropped tuples, A-greedy can estimate the selectivities of operators even for routes that it never executes. This information is then used to provide strong guarantees on the optimality of routes that A-greedy selects [8]. However, A-greedy has a higher state update overhead then Cont or ContB and that is why it collects information just after every profile tuple, instead of after every tuple. A-greedy, developed for data stream scenarios where data and system characteristics are expected to change very quickly, computes a new routing order after every profile tuple. Since SHARP is processing local relations instead of data streams, and to lower the overhead of computing routes and to produce the first routing order faster, our implementation of A-greedy, called *Profile*, uses the first n out of every p tuples as profile tuples. Thus, *Profile*, the third tuple routing policy implemented in SHARP, computes a new routing order after every $n+K*p$ tuples, with $K \geq 0$, and uses that routing order for the next p driving tuples. Table 6 summarizes the routing policies implemented.

Table 6 – Routing policies implemented, $p > n > r$, $K \in \mathbb{Z}^*$

Routing Policy	New Route	Update State	Exploration	Optimality Guarantees
Cont	After every tuple	Every tuple	Random route every r tuples	None
ContB	After $n*K$ tuples	Every tuple	Random route every r tuples	None
Profile	After $n+K*p$ tuples	n out of every p tuples	n out of every p tuples probe all builds, even if dropped	Greedy 4-approximation algorithm [8]

4.3.3. Late Binding Decisions

In this Section we describe a series of late binding decisions–decisions made at run-time after some tuples are observed–that change the structure of the query plan executed by SHARP.

The late binding decisions can be made after SHARP loads any build relation, B_i, into hash table t_i, with $i=1..n$, where n is number of builds. While it creates t_i, SHARP also constructs a histogram h_i on attribute $B_i.d$, the attribute of B_i that joins with the driving relation. If t_i fits in memory, then SHARP uses the histogram created, consults the catalog and estimates how many driving tuples, d_i, would join with B_i.

At this point, SHARP can change the query plan in three different ways, summarized in Table 7 and described next, or it can continue the query processing as described in Section 4.3.1. We note that although the late binding decisions can happen after any build table is loaded, in this first prototype, they will happen only once per query: i.e., if SHARP makes late binding decisions after the first build is loaded, then it will not make more late binding decisions after any other build is loaded.

Table 7 – Summary of late binding decisions

Decision	Type of D	Read D	Access D	Buffer D?
Uinl	Base	After B_n	Index	No (stream)
Uibf	Base	After B_i	Index	Yes
Ufbf	Base, intermediate	After B_i	Unchanged	Yes

4.3.3.1. Using an Indexed Nested-Loop (Uinl).

If the driving relation D is a base relation with an index, idx_i, on the attribute(s) $D.a_i$ of D that join with $B_i.d$, then, depending on d_i, on the costs of random and sequential reads, and on properties of index idx_i, it may be better to use an indexed nested-loop to access the driving tuples–with hash table t_i and index idx_i as the outer and inner components of the loop–than to use a file scan on D. At this point (Figure 36a), SHARP makes a cost-based decision[14]. If the file scan is the better access for D, then SHARP does not change the query plan (Figure 36b). Otherwise, driving tuples will be obtained using the indexed nested-loop (Figure 36c),. Note that this change effectively makes B_i the driving relation. In addition, if the indexed nested-loop access is used, then the late binding decision "Using INL and Bloom-Filters", below, is considered before any other build relation is processed.

---

[14] The cost-based decision is similar to the access path selection that happens at optimization. However, during optimization the statistics needed to cost the indexed-nested loop plan may be missing. On the other hand, SHARP computes part of the statistics it needs from observed tuples and is thus less likely to produce incorrect estimates.
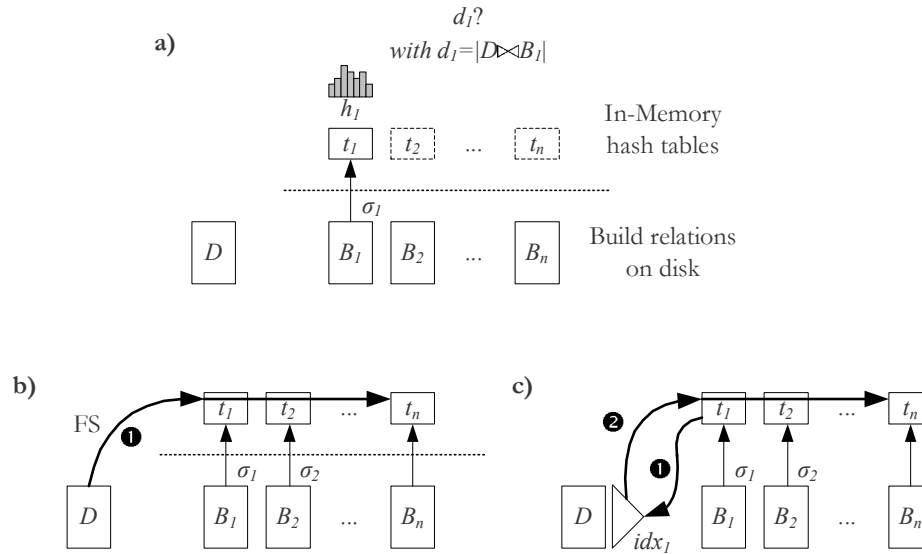
Figure 36 – Late binding decision Using-INL

4.3.3.2. Using INL and Bloom-Filters (Uibf).

Given d_i and the average size of a driving tuple, SHARP computes the total expected size of those d_i tuples, T_i. If T_i is less than the budget given to any build hash table, then SHARP reads all the d_i driving tuples into memory before proceeding to read other builds (Figure 37, ❶). For each driving tuple it loads into memory, SHARP reads attribute a_j that joins with build relation B_j and updates a bloom filter bf_j, for $j=1..n$, $j \neq i$ (Figure 37, ❷). Each bloom filter, bf_j, is a bitmap of length k [16]. When driving tuples are read, attribute a_j is hashed to a value between 0 and $k-1$, and the corresponding bit in bf_j is set. Later, when the other build relations are read, for each tuple, SHARP hashes its join attribute, $B_j.d$, with the hash function used for bf_j. If the bit corresponding to that value in bf_j is 0, then the tuple is dropped, otherwise the tuple is processed normally (Figure 37, ❸). Filtering B_i with its corresponding bloom-filter bf_i should decrease t_i considerably. Finally, driving tuples in d_i continue their path and probe the already filtered in-memory hash tables t_i.
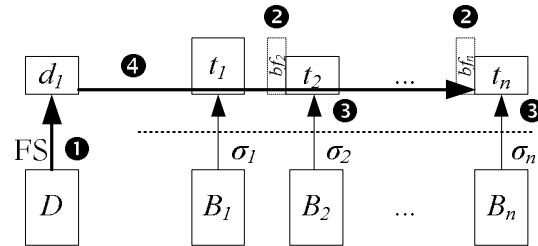
Figure 37 – Late binding decision Using Bloom Filters

4.3.3.3. Using Driving Relation Pre-Filtering and Bloom Filters (Ufbf).

As described in Section 4.3.4, SHARP's if any build relation requires a second pass then the driving relation requires a second pass also. As such, filtering the build relations with bloom-filters may improve performance significantly because it could save both the builds and the driving relations from spilling to disk. Thus, even if SHARP decides not to use an indexed nested-loop to retrieve the driving tuples, it will still check, after building any t_i, if the T_i (the size of all tuples of D estimated to match t_i) fits the budget given to build hash tables. If it does, then, as in the case Uibf above, SHARP reads driving tuples into memory ahead of time. Each driving tuple read probes t_i and if it finds no match, it is dropped. Otherwise, it is kept in an in-memory buffer and it is used to update bloom-filters on the other build relations. Then, build relations are loaded into memory while being filtered by the bloom-filters. Finally, driving tuples are read from the in-memory buffer, and used to probe the builds.

4.3.4. Second-Stage Processing

Even after redistributing memory across joins (Section 4.3.1) and filtering tables using the late binding decisions (Section 4.3.3) it is still possible that one or more relations do not fit in memory. Those portions will have to be temporarily written to disk and processed at a later stage, typically referred to as second-stage. This Section describes SHARP's second-stage processing algorithms.

4.3.4.1. Split Tables into Partitions and Portions.

When SHARP reads build relation B_i, it creates an in-memory hash table t_i with p partitions. If there is no more memory space for B_i tuples, one in-memory partition of t_i is selected, its current records are moved to a temporary file on disk, the partition is marked as frozen, and SHARP continues loading records from B_i. Future records that hash to frozen partitions are held in very small memory buffers and flushed to disk when the buffers fill up. Then, for each t_i, the partitions are assigned to sets of consecutive partitions called *prtions* , such that the size of each portion does not exceed available memory. Figure 38a shows the state of a SHARP after it has completed the build stage. In the example, build hash table t_1 has four portions. Portion 0 is in memory, and the remaining three portions contain the frozen partitions.

After the builds are partitioned, SHARP reads the driving table and partitions it along all n join attributes with the build relations. This multi-dimensional split of D is shown in Figure 38b for the case of two build relations. Note that the split of D is done in terms of portions of the t_i, instead of partitions of t_i.

To split D, incoming driving tuples are routed to some t_i for probing (according to SHARP's routing policy as described in Section 4.3.2). When SHARP probes t_i with driving tuple dt, using join attribute $dt.a_i$, it gets one of three results: "match", "fail", or $t_i(dt.a_i)$, the number of the in-disk partition of t_i that $dt.a_i$ hashes to. We note that a "match" also includes the set of pointers to the dt matching tuples in t_i and implicitly implies that portion $t_i(dt.a_i)$ is in memory (i.e., $t_i(dt.a_i)=0$).

If any t_i probe returns "fail", tuple d_t is dropped; otherwise the tuple is routed to another hash table. If dt is not dropped, there are three cases to consider, corresponding to the dotted, white, and gray portions of D in Figure 38b:

If all t_i probes return "match" (d_t belongs to the dotted portion of D), then the resulting one or more join tuples are output by SHARP and not written for second-stage processing.

If all $t_i(dt.a_i)>0$ (dt belongs to a white portion of D), then it is not known if tuple dt joins or not with any of the builds. Tuple dt is then written to a temporary file for second-stage processing.

If at least one, but not all tables, t_i returned "match" ($t_i(dt.a_i)$=0) then driving tuples dt belongs to a grey portion of D. In this situation, dt tuples can be processed in two ways: *Save Intermediate Tuples* (*SIT*) or *Save Driving Tuples* (*SDT*). In option SIT, SHARP writes to temporary files the intermediate join between dt and its matching tuples (from the hash tables that returned "match"). In option SDT SHARP writes just dt to temporarily files and discards any matching tuples, which are then obtained again during the second-stage processing of dt.

When option SIT is used some D portions (marked gray in Figure 38b) will contain wider, intermediate join tuples, but the probing work will not be lost. When option SDT is used, all D portions contain just driving tuples, but the probing work will be lost and will have to be repeated later by reloading portions of build tables from disk. Depending on the relative sizes of driving tuples and their matching records, and on the selectivity of the joins, either option can be better. Furthermore, the choice between SIT and SDT made for driving tuples for which $t_i(dt.a_i)$=0 can be different of the choices between SIT and SDT for tuples for which $t_j(dt.a_j)$=0, $j{\neq}i$. For example, in Figure 38b, choice SIT can be used for D portions marked 4, 6 and 8, and option SDT can be used for D portions 11 and 20. To simplify the second-stage algorithm, the prototype implementation of SHARP always uses option SDT.

a) End of build stage

b) Multi-dimensional split of *D*
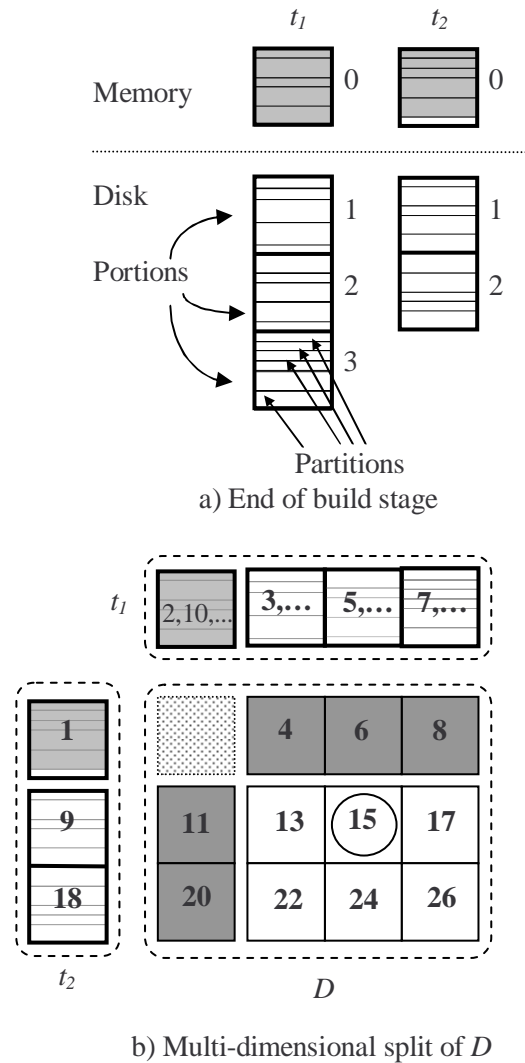
Figure 38 – Second stage processing

4.3.4.2. Second-stage Joins.

After the multi-dimensional split of *D* is complete, SHARP begins the second-stage of the join. First SHARP orders the build hash tables based on the ascending number of portions and, if there is a tie, based on their descending total size. This order is represented by $O_{(i)}$, such that $O_{(i)}$ represents the i-th build hash table in the order. In the example of Figure 38, $O_{(1)}=t_2$ and $O_{(2)}=t_1$ because t_2 has just three portions while t_1 has four portions. Assume also that $|O_{(i)}|$ represents the number of portions in hash table $O_{(i)}$ and that $O_{(i)}.load(k)$ loads portion k, with $k<|O_{(i)}|$ of hash table $O_{(i)}$ into memory, deleting from memory the current

in-memory portion of $O_{(i)}$ and that function Dportion(i_1, i_2, …, i_n) returns the portion of D that corresponds to the i_1-th portion of $O_{(1)}$, to the i_2-th portion of $O_{(2)}$, …, and to the i_n-th portion of $O_{(n)}$. In the example of Figure 38, Dportion(1,2) corresponds to the portion of D marked with a circle.

Then, as shown in the pseudo-code of Figure 39, SHARP executes a series of loops, loading portions of $O_{(1)}$ to $O_{(n)}$ into memory, getting tuples from the corresponding D portion, probing the in-memory portions of $O_{(1)}$ to $O_{(n)}$ and outputting matches. This algorithm loads the on-disk portions of D one time and loads the in-disk portions of $O_{(i)}$ a number of times equal to $\prod_{j=1..i-1}|O_{(j)}|$. The numbers in Figure 38b represent the order in which portions of the example of Figure 38a are loaded.

In contrast, a right deep tree of binary Dynamic Hash Joins (DHJ) [68], reads each input relation just once, but may have to save to and read from disk (during the second-stage processing) the intermediate results multiple times. For example, if no B_i, $i=1,2,3$ fits completely in memory, the execution plan corresponding to the right-deep tree of Figure 40 will need to do a second-pass for each of the joins, saving to and reading from disk part of the intermediate results corresponding to $D\bowtie B_1$, $D\bowtie B_1\bowtie B_2$, and $D\bowtie B_1\bowtie B_2\bowtie B_3$. To minimize the size of those intermediate results, an accurate optimizer estimates the join selectivities between D and B_i, and other things being equal, sets D's join order to be from the most to the least selective B_i. However, join selectivities are hard to estimate correctly and an optimizer may choose an incorrect join order which negatively affects performance.

On the other hand, the performance of SHARP depends only very slightly on the join order defined by the optimizer. If the builds fit in memory, the join order is determined by an adaptive tuple routing policy at run-time. If the builds do not fit in memory, the cost of SHARP's second-stage depends mainly on the order $O_{(i)}$, but this order is determined only after all build tuples are observed; no estimates are needed. However, the performance of SHARP's second-stage suffers from the "curse of dimensionality": if several build relations are much larger than memory, then the repeated readings of the inner most build, $O_{(n)}$, (which is read $\prod_{i=1..n-1}|O_{(i)}|$ times) may dominate the total cost of the join. In Section 4.4 we explore

how the available memory affects the performance of SHARP. It is also shown that even with an amount

of available memory equal to just 10% the size of the largest build table, SHARP's second-stage can still

outperform other methods.

```
for (i₁=0; i₁<|O(1)|, O(1).load(i₁); i₁++)
  for (i₂=0; i₂<|O(2)|, O(2).load(i₂); i₂++)
    …
      for (iₙ=0; iₙ<|O(n)|, O(n).load(iₙ); iₙ++)
        for all tuples dt in Dportion(i₁, i₂, …, iₙ)
          dt probes in-memory portions of O(i), …, O(n);
          output matches between dt and O(i), …, O(n);
        end for;
      end for;
    …
  end for;
end for;
```

Figure 39 – Second-stage pseudo code
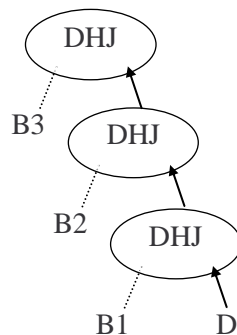
Figure 40 – Right-deep tree of DHJs

4.3.5. Summary of SHARP

SHARP does not use symmetry plans (like the MJoin) or symmetric operators (like the Eddy).and instead

of allowing all relations to be used as builds or probes, the optimizer chooses one single driving relation.

Since only build relations have hash tables, this design reduces the memory footprint of a SHARP to

essentially half of what an Eddy and its SHJs would consume. The routing policy is then responsible for determining the order with which driving tuples probe the build sources.

By having a single driving relation, routing policies in SHARP have fewer routes to choose from than in Eddies (because routing decisions affect only tuples coming from driving relations). However, because SHARP is a pull-based operator–in charge of obtaining new tuples from its sources–we were able to design a series of late binding decisions that, in some cases, are able to promote any of the builds to be the driving relation before the tuple routing stage starts.

In short, an Eddy is able to change execution from any route to any other route at any stage of execution, while a SHARP first determines which source is the driving relation and then continuously adapts the sequence in which the driving tuples probe the build sources. With this two-step adaptive process, SHARP still has the potential to adaptively decide at run-time which one of all[15] possible join orders to use but with a much smaller memory footprint.

4.4. Experimental Evaluation

We now describe an experimental evaluation of the query processing techniques described for SHARP using a prototype implementation in Predator [81]. All results were obtained using a dedicated machine with 512 MB of main memory and a buffer pool of 2000 16-Kbyte pages (hash table builds are kept outside the buffer pool). Results are averages of three cold runs.

We note that the first experimental results (Section 4.4.2) do not evaluate any adaptivity feature of SHARP and exhibit only marginal improvements of a SHARP over competing plans. The purpose of that section, though, is twofold. First, we want to show that, not only is the adaptivity overhead of SHARP very low, but also, even in scenarios where the adaptivity yields no benefit, SHARP still provides a

---

[15]   Ignoring join orders using Cartesian products.

marginal performance advantage due to its multi-join nature. Second, the adaptivity benefits of SHARP compound these initial multi-join improvements.

4.4.1. Datasets

The SHARP prototype was evaluated using two datasets, Star and TPCH, described below. Star, a synthetic dataset we created, allowed us to more easily explore different selectivities, join selectivities and table sizes. TPCH is used to evaluate SHARP's adaptivity and robustness in a widely known benchmark:

- *Star*: We created a synthetic benchmark, Star, based on a star schema, with a central fact table F, and four dimension tables, A, B, C, and D. F has 1,000,000 152-byte records and A, B, C, and D have 100,000 40-byte records. Our experiments use 2-way, 3-way, and 4-way join queries of the following form:

```
SELECT *
  FROM  F, A, B, …, D
 WHERE  F.fkdA = A.pk
   AND  F.fkdB = B.pk
     …
   AND  F.fkdD = D.pk
   AND  σ₁(A)
   AND  σ₂(B)
     …
   AND  σ₄(C);
```

Where σ_i, $i=1,..,4$ represent selection predicates with selectivities between 0% and 100%.

- *TPC-H*: TPC-H is the decision support benchmark from the Transaction Processing Performance Council [85]. We used tables lineitem (L), orders (O), part (P), customer (C), and supplier (S), with scale size 1. We included an extra column in L (a foreign-key to C) to allow star-schema queries using one central fact table (L) and up to four dimension tables (O, P, C, and S). Our queries join either all tables, or all except O (the largest dimension), or all except S (the smallest dimension), and we use varying selection predicates on the dimension tables.

4.4.2. Evaluating Multi-Join Improvements

As described in Section 4.3.1, SHARP has two benefits over plans composed of a tree of binary operators: it can avoid unnecessary intermediate tuple generation and unnecessary getNext calls. We note that these benefits are not due to any adaptive feature of SHARP. Instead, they are a positive side effect of the multi-join nature of SHARP.

In the results shown in Figure 42, table F from schema Star, was joined with two, three, and four dimension tables. All the dimension tables fit in memory, and all join selectivities[16] are 100%. We compare three execution plans: SHARP, SHARP-IR, and RDH. SHARP-IR is a variation of SHARP that generates intermediate join tuples after each successful probe. RDH is a plan composed of a right-deep tree of dynamic hash joins (DHJ) [68], with F as the rightmost table[17] as shown in Figure 41. Both SHARP and SHARP-IR have their run-time statistics collection and adaptive routing policies turned off to ensure that the probing sequence is the same in all plans, and to ensure that we are measuring just the multi-join benefits. (The performance of routing policies in measured in Section 4.4.4.) All three plans use the same hash table implementation code. The benefit of avoiding intermediate results varies between 5% and 15%.

---

[16]   We use the term join selectivity of a build table as the average number of returned records per probing record.

[17]   The right-deep tree plan creates hash tables on the same relations and executes the same number of build and probe operations as SHARP.
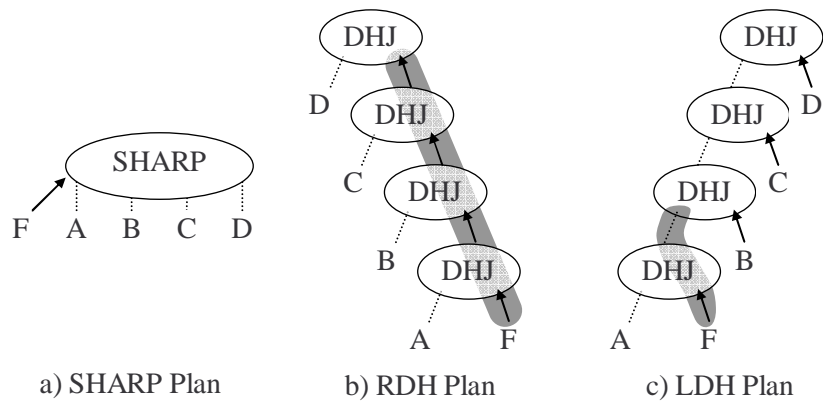
a) SHARP Plan b) RDH Plan c) LDH Plan

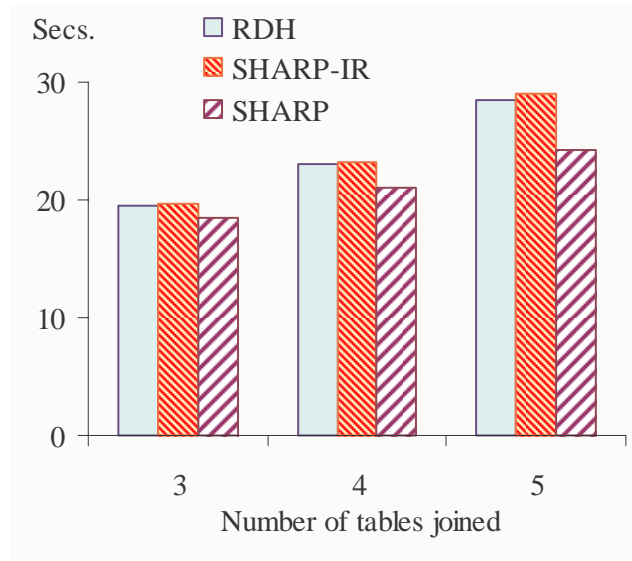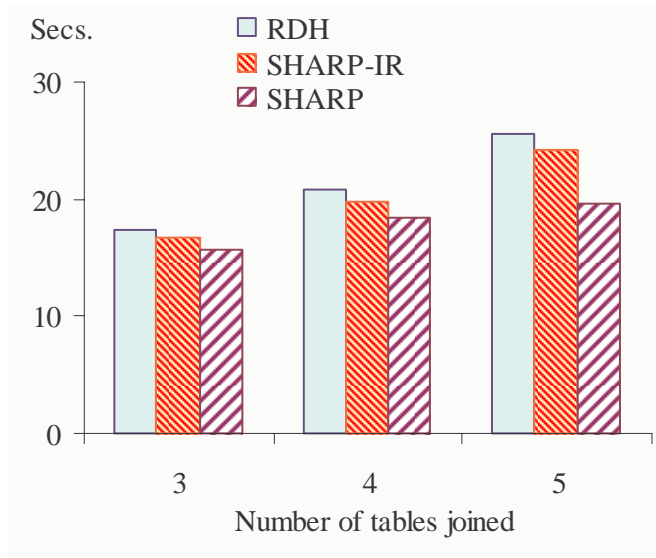**Figure 41 – a) SHARP; b) RDH plan; c) LDH plan**

Figure 42 – Avoiding intermediate results

Figure 43 – Avoiding getNext calls

In a second experiment, we set the selectivity of the last join to 0% (e.g., for the case of three joins, 100% of F tuples join with the first and second dimension, and 0% join with the third dimension). These contrived plans maximize the number of spurious getNext calls made by the RDH plan and give an upper bound on the benefit obtained by avoiding those calls. Figure 43 shows the results for SHARP, SHARP-IR and RDH: SHARP is between 10% and 23% faster than RDH and SHARP-IR is around 5% faster than RDH.

The results show that, even without taking advantage of adaptivity, a tuple routing operator can slightly outperform plans composed of a tree of binary operators.

4.4.3. Redistributing Memory between Joins

For each query, our system gives a pre-specified memory budget for each hash table used to implement hash joins. Traditionally, if building the hash table requires more memory space than the memory budget, some partitions would have to be written to disk and a second-pass on those partitions would be required. In addition, if the hash table requires less memory than the allocated budget, then the unused memory– save some exceptions [28]–is not given to other operators. On the other hand, as described in Section

4.3.1, SHARP loads all its builds into memory before reading the driving relation and is therefore able to redistribute memory between different hash tables.

To evaluate the impact of memory redistribution, table F was joined with selections σ_1 and σ_2 on tables A and B. The selections are such that the size of $\sigma_2(B)$, $|\sigma_2(B)|$, is four times the size of $\sigma_1(A)$, $|\sigma_1(A)|$, and $(|\sigma_1(A)|+|\sigma_2(B)|)/2$ is the memory budget for each one of the two hash tables. Thus, $\sigma_1(A)$ underutilizes its budget while $\sigma_2(B)$ overutilizes it, but on average both fit in memory.

Four different plans were tested, for the four combinations of using SHARP and a RDH and of using two join orders, joining F with A first and then with B, or joining F with B first and then with A. Then, as shown in Figure 44, the size of the combined memory budget was varied from $|\sigma_1(A)|+|\sigma_2(B)|$ (corresponds to 100%) to $2*|\sigma_2(B)|$ (160%).
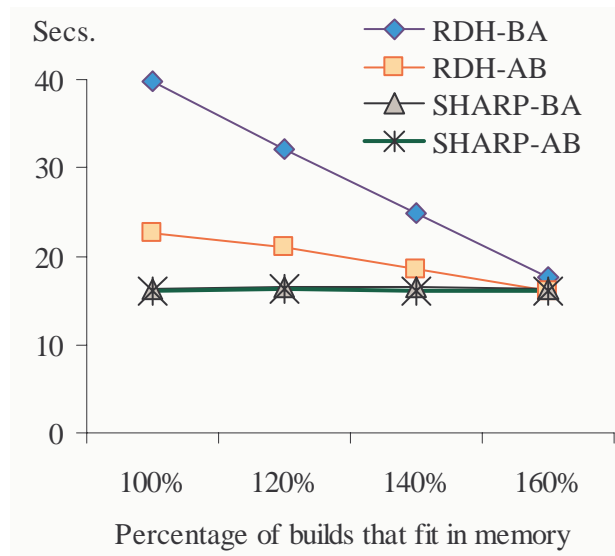
Figure 44 – Redistributing memory

SHARP is able to take advantage of memory redistribution and avoid a second-stage for all amounts of memory tested. On the other hand, because the RDH executes a tree of independent operators, it does not redistribute memory amongst the operators. Thus, the RDH plan only avoids a second-pass in both operators when the budget per hash table is at least as large as the largest hash table. This happens only

for a combined memory budget of $2*|\sigma_2(B)|$, or 160% the size of the $|\sigma_1(A)|+|\sigma_2(B)|$. If more than $2*|\sigma_2(B)|$ of memory is available, the performance of all four plans remains unaffected. If less than $|\sigma_1(A)|+|\sigma_2(B)|$ of memory is available, SHARP also needs a second-stage. Experiments showing the performance of the SHARP second-stage appear in Section 4.4.5.

4.4.4. Comparing Routing Policies

To compare the routing policies described in Section 4.3.2, table F was joined with two, three, and four dimension tables. In each query, the join selectivities were 100% for all joins except for one that was 25%. To highlight the impact of a good routing policy that quickly discovers sub-optimal plans, the initial default join order defined by the optimizer was sub-optimal, executing each of the 100% selectivity joins before the 25% selectivity join.

The experiments explored two variables, the profiling overhead, and the (hash table) probing cost. A higher profiling overhead means that the routing policies spend more time exploring alternative routes, computing new optimal routes, and updating state. The profiling overhead was varied by setting the parameters of Table 6 (in Section 4.3.2) to the following values:

- High Profiling Overhead: $r=10, n=1000, p=10000$

- Low Profiling Overhead: $r=50, n=200, \quad p=10000$

Parameter r, which specifies how frequently the tuple routing policies Cont and ContB explore alternative routes, was set such that $r=p/n$ to ensure fairness in the comparison (the tuple routing policy Profile explores other routes for every n out of p tuples). The probing cost was varied by artificially delaying the probe operation, such that, on average, high probing cost takes about four times longer than low probing cost.

Figure 45 and Figure 46 show the results for the low profiling overhead cases for the three routing policies implemented, plus a trivial policy, Static, with no profiling overhead, which simply routes tuples

according to the join order defined by the optimizer. The higher profiling overhead cases are not shown because they are very similar to these. As expected, a higher probing cost (Figure 46) affects all policies negatively, but affects the Static policy more than the others because the adaptive policies, detect and avoid the sub-optimal plan while Static continues executing the sub-optimal plan where it performs extra probing operations

The Profile policy was always the best of the three adaptive policies, except when both the profiling overhead and probing costs were high (graph not shown). However, even in this case, it was between 25% and 33% better than the Static policy. On average, we found that, Profile outperformed Static by 24%, ContB outperformed Static by 22%, and Cont outperformed Static by 16%.
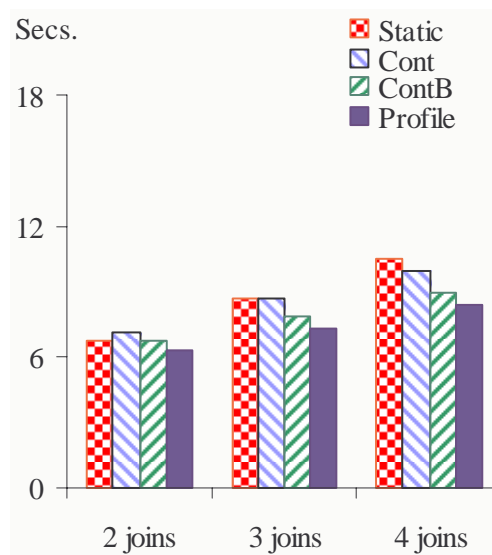
Figure 45 – Low profiling overhead, low probing cost
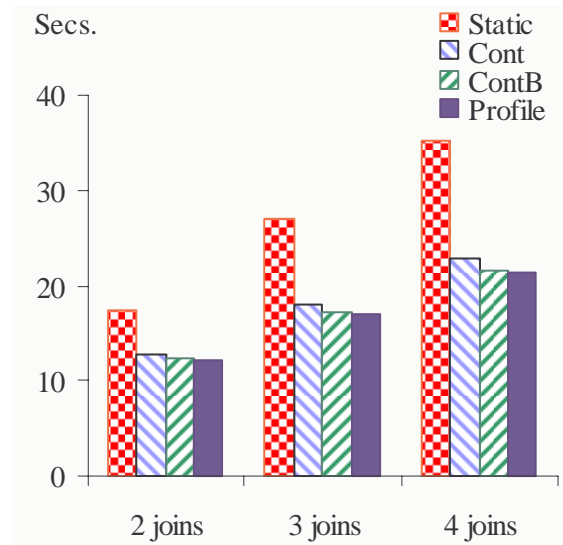
Figure 46 – Low profiling overhead, high probing cost

4.4.5. Evaluating the Second-Stage

To evaluate the performance of the second-stage processing of SHARP, we joined tables F, A, and B in one query and tables F, A, B, C, and D in another query. (Further experiments in the next two sections also evaluate the second-stage.) Note that SHARP only joins relations along existent join predicates. Thus, since all join predicates join F with one different build (F⋈A, F⋈B, F⋈C, and F⋈D), SHARP will never perform cross-products (e.g., it will not do A⋈B) and therefore, it will never execute bushy plans either (e.g., (F⋈A)⋈(B⋈C)).

We do not compare SHARP against bushy plans because, although they are optimal for some queries and datasets, they were never optimal for the queries we tested. SHARP was compared with plans RDH and LDH (see Figure 41 in page 102) which, like SHARP, are linear-join plans.

The amount of memory was varied such that between 10% and 100% of tables A and B in the first query, and A, B, C, and D in the second would fit in memory. We note that both SHARP and RDH are non-blocking, and therefore, their execution pipeline uses the in-memory parts of all the build hash tables simultaneously (see Figure 41b showing RDH's execution pipeline in gray). In contrast, the execution pipeline of the LDH plan, at any moment only manipulates two hash tables (see Figure 41c). Thus, to

ensure the amount of total memory per plan was the same, hash tables in the LDH plan were allowed twice the memory of hash tables in SHARP and the RDH plans. The results are shown in Figure 47 and Figure 48.

Except when the amount of memory is very limited, SHARP outperforms the other two plans. If several build relations are much larger than memory, then the innermost build will be read many times and the performance of SHARP degrades quickly. On the other hand, as shown in Section 4.4.7, if just one or two builds are much larger than memory, and the remaining builds either fit in memory or are not much larger than memory, then the performance of SHARP degrades much more slowly.

To address the exponential degradation problem, the SHARP could convert itself to a RDH plan: after all the builds are read and partitioned, SHARP can easily determine if its performance will degrade quickly or not. At this point, the conversion to a RDH plan is essentially free; all dimension tables are already partitioned with the right hash functions, and no work is lost. We leave this late binding decision as future work.
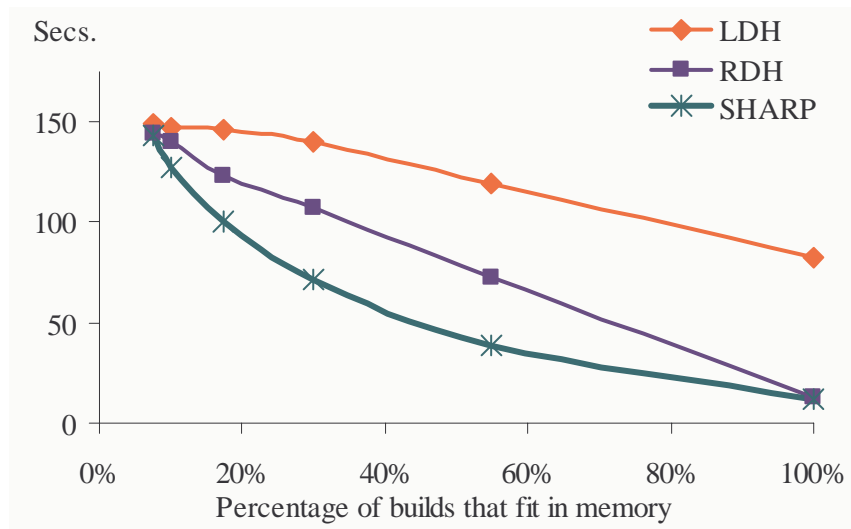
Figure 47 – Evaluating second-stage, 2 joins
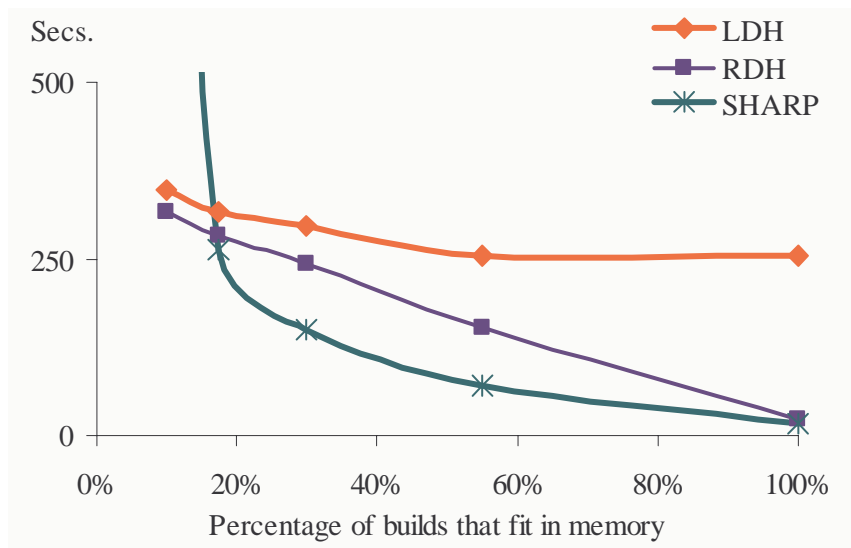
Figure 48 – Evaluating second-stage, 4 joins

4.4.6. Evaluating Late Binding Decisions

To evaluate the effectiveness of the late binding decisions proposed in Section 4.3.3, table F was joined with dimension tables A and B in one query and with dimension tables A, B, C, and D in another. Both queries were run with unlimited memory and with memory limited to 25% the size of the dimension tables. A selection predicate σ was applied to table A and the selectivity of σ was varied from 0.01% to 100%. The execution plan for four joins is shown in Figure 49. Three other plans were also considered, RDH, LDH, and INL. Plans RDH and LDH are similar to the plans b) and c) of Figure 41, but with predicate σ applied on table A. Plan INL is a tree of binary indexed-nested loop joins; i.e., with the same shape as plan LDH, but with the DHJ operators replaced by indexed-nested loop operators.

The three late binding decisions under consideration are Using an Indexed Nested-Loop (Uinl) to obtain the driving tuples, Using Indexed-Nested Loops and Bloom-Filters (Uibf), and Using Driving Relation Pre-Filtering and Bloom Filters (Ufbf). Ufbf was implemented as follows: instead of consulting the catalog to estimate the number of driving tuples matching an in-memory build table–as described in Section 4.3.3–our prototype implementation executes Ufbf every time a build relation had less than 2000 tuples. To simulate the cost of a SHARP using late binding decisions Uinl and Uibf, the plan of Figure 49b was forced and for each point in the graphs, the best time for the plans of Figure 49a and Figure 49b was chosen as being SHARP.

Figure 50 measures the impact of the late binding decisions for the 4-join query with memory restricted to 25% the size of the builds, by comparing SHARP with SHARP-NLB, a version of SHARP where no late binding decisions are allowed. Figure 51 then compares SHARP with plans RDH, LDH and INL for the same query. Figure 52 shows the results for the 2-join query with memory equal to 100% the size of the builds (note that for this query, SHARP and RDH do not need a second-stage but LDH does). These figures show that SHARP is the best plan for a wide range of values of σ, showing the best results in all points of the graphs, except possibly in the range $\sigma \in [0\%, 0.5\%]$ where the INL plan was sometimes better, or in the range $\sigma \in [2\%, 20\%]$ where RDH was sometimes better.

a) SHARP   b) Simulating SHARP with Uinl
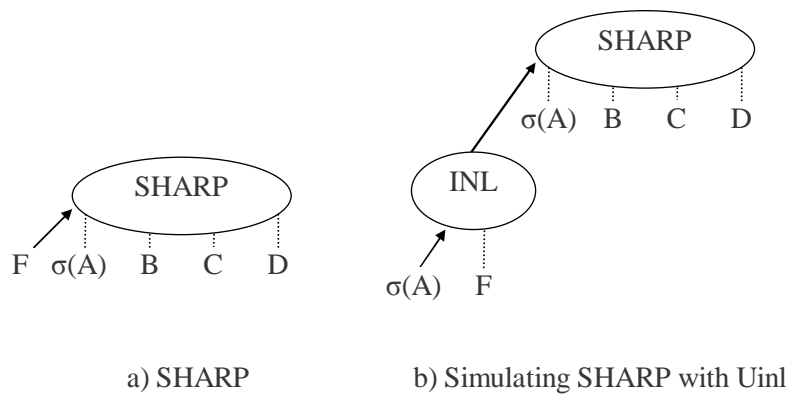
Figure 49 – Plans used to evaluate late binding decisions
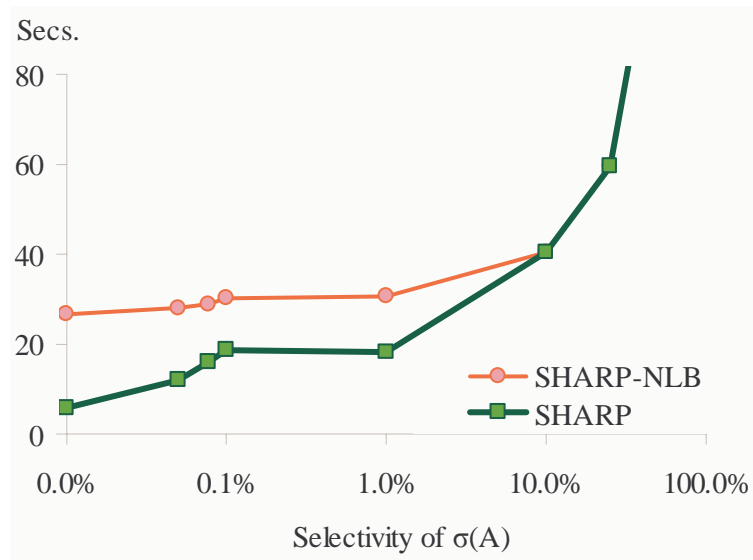
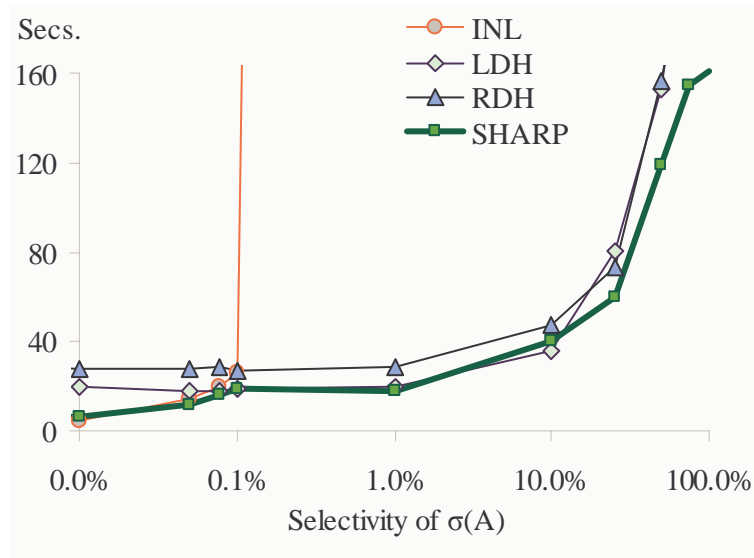**Figure 50 – Late binding evaluation: 4 joins; memory=25% size of builds**

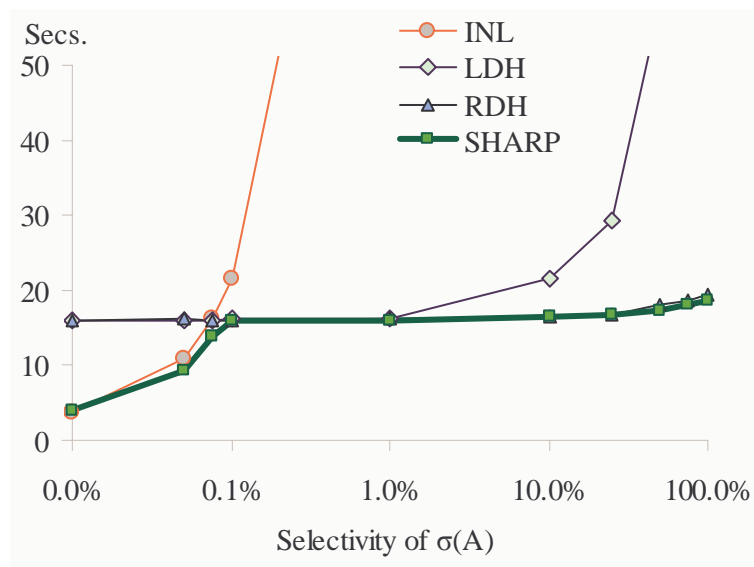**Figure 51 – Late binding evaluation: 4 joins; memory=25% size of builds**

Figure 52 – Late binding decisions: 2 joins; memory=100% size of builds

In the range $\sigma \in [0.01\%, 0.1\%]$, the size of $\sigma(A)$ is so small that it is worthwhile to use late-binding decision Uibf. That is, after reading A tuples into memory, and after filtering them with σ, it is best to use them to lookup the driving tuples with an index then it is to read the driving tuples using a table scan on F. In addition, because the number of driving tuples matching those A tuples is also very small, it is worthwhile to read those few driving tuples into memory and create bloom filters on the builds yet to load instead of proceeding with the unfiltered load of build tables.

In the range of σ from 0.1% to 2%, the number of $\sigma(A)$ tuples was not low enough to use them to obtain driving tuples using an index, but it was still low enough to make late-binding decision Ufbf worthwhile. In that range, it is best to read A tuples into memory, filter them with σ, read and filter F tuples with the $\sigma(A)$ tuples and create the bloom filters on the yet-to-load build tables than it is to proceed with the unfiltered load of build tables.

Overall, Figure 50, Figure 51, and Figure 52 show that it is possible to construct *robust* query plans, that is, plans whose comparative performance is insensitive to optimizer estimates. In the example, regardless of the estimate the optimizer might have had about selection σ, executing SHARP would be a good decision: its performance is either optimal or close to optimal in all points in the graphs.

4.4.7. Evaluating Second-Stage Insensitivity

In order to further evaluate the performance of second-stage processing and robustness of SHARP in the presence of potentially incorrect join orders, SHARP was tested for queries 1, 2, and 3, and the selection predicates shown in Figure 53. Then the optimizer was altered to generate plans with a specific join order.

Figure 54 shows the results of executing Query 1 using two join orders, LPCS (the best join order for RDH), and LCPS (the second best join order for RDH), and with 45MB and 15MB of available memory, for both SHARP and the RDH plan. (The LDH variant was always worse than RDH in this query.)

For the same optimizer-specified join order, SHARP completes the query between 2.5 and 7.5 times faster than an equivalent RDH. The bulk of SHARP's performance benefit comes from memory

redistribution and from the new second-stage processing technique. When there are 45MB of memory available, RDH allocates 15MB for each one of the three DHJ operators. This turns out to be a bad strategy: given the selectivities shown in Figure 53, for Query 1, the DHJs that process the join with S, C, and P would need 2.2MB, 9.2MB, and 33.2MB respectively to avoid writing records to disk. Thus, assigning 15MB to each DHJ operator implies that for one operator, the RDH plan will have to do a second-stage. On the other hand, SHARP–which redistributes memory across joins–is able to complete the joins in just one pass over the relations when 45MB are available. In this scenario, SHARP is between 2.5 and 4.6 times faster than the RDH plan.

When memory is limited to 15MB both the RDH and SHARP plans require a second-stage for relations C and P (and also for whichever intermediate relations are joining with C and P). Although both plans take significantly more time than in the previous scenario, the restricted memory affects RDH more than it affects SHARP. In fact, using the new second-stage processing technique, SHARP is now between 3.6 and 7.5 times faster than RDH.

Query 1

```
SELECT  *
  FROM  L, S, C, P
 WHERE  l_suppkey=s_suppkey
   AND  l_custkey=c_custkey
   AND  l_partkey=p_partkey
   AND  σ1(p_partkey)   ←  75%
   AND  σ2(c_custkey)   ←  25%
   AND  σ3(s_suppkey);  ← 100%
```

Query 2

```
SELECT  *
  FROM  L, C, P, O
 WHERE  l_partkey=p_partkey
   AND  l_custkey=c_custkey
   AND  l_orderkey=o_orderkey
   AND  σ4(o_orderkey)  ←  25%
   AND  σ5(p_partkey)   ←  25%
   AND  σ6(c_custkey);  ←  75%
```

Query 3

```
SELECT  *
  FROM  L, S, C, P, O
 WHERE  l_suppkey=s_suppkey
   AND  l_custkey=c_custkey
   AND  l_partkey=p_partkey
   AND  l_orderkey=o_orderkey
   AND  σ7(o_orderkey)   ←  25%
   AND  σ8(p_partkey)    ←  50%
   AND  σ9(c_custkey)    ←  25%
   AND  σ10(s_suppkey);  ← 100%
```

Figure 53 – Queries and predicate selectivities used in the TPC-H schema
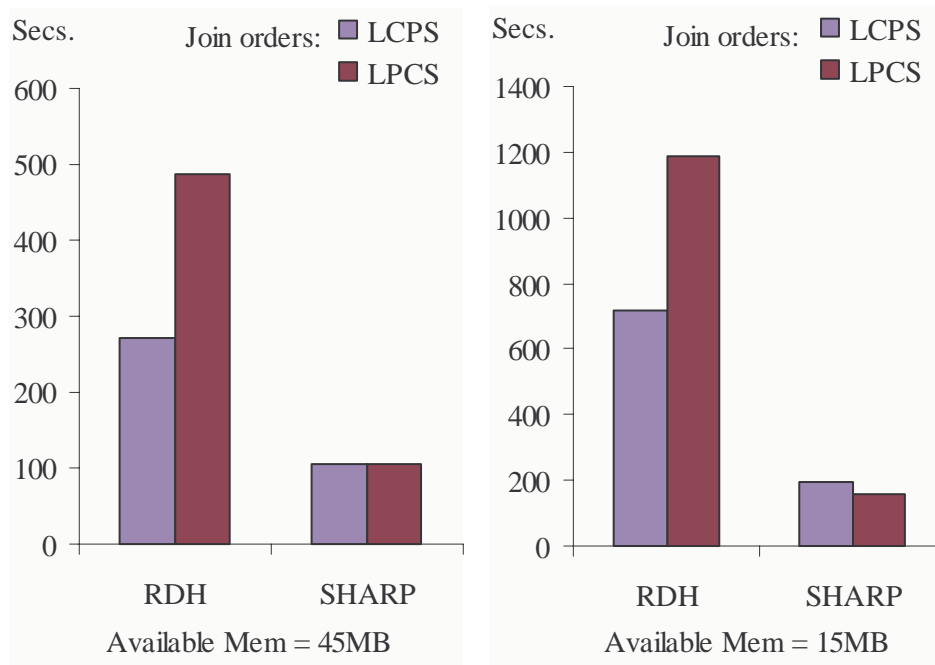
Figure 54 – Execution of Query 1

In addition, not only is SHARP superior to the RDH plan, its performance is also essentially independent of the join order specified by the optimizer. In fact, the order specified by the optimizer affects the SHARP plan in just two ways:

- For the very first few driving tuples, before the first tuple route is computed, SHARP uses the join order specified by the optimizer as the default route. However, as soon as the routing policy produces its first route, the initial join order is forgotten.

- When redistributing unused memory to hash tables, SHARP favors giving memory to hash tables belonging to joins that appear earlier in the optimizer specified join order. This is the reason join order LPCS yields better results than LCPS in the right side of Figure 54.

The RDH plan requires 232MB to avoid second-stage processing in Query 2 and SHARP, taking advantage of memory redistribution, requires only 116MB to avoid second-stage processing for the same query. Query 2 was run with 150MB, 75MB and 30MB of total available memory. The results for different join orders for Query 2 with 30MB of memory appear in Figure 55. For Query 3 the RDH plan requires 309MB to avoid second-stage processing and SHARP requires 110MB. Query 3 was run with both 100MB and 40MB of total available memory. The results for different join orders for Query 3 with 100MB of memory appear in Figure 56. In both cases, second-stage dominates the total execution time of the query.

As shown in Figures 55 and 56, the cost executing the RDH plan depends on the join order specified by the optimizer. Thus, optimizer mistakes may greatly influence the total cost of the RDH plan. However, the second-stage of SHARP has two desirable properties: i) the cost of writing partitions to disk and the amount that is written is independent of the join order specified by the optimizer; and ii) the number of times each partition is read and the order when each partition is read is adaptively determined based on the observed sizes of builds, again, decisions independent of the optimizer.

For the queries tested, the plan with the worst join order using RDH can take twice as long as the plan with the best join order using RDH for the same query. In addition, across all experiments, RDH took, on average, twice as long as SHARP to complete execution.

Note that the results from this section further validate the results of Section 4.4.5: the best join orders for the RDH plan take between 1 to 3 times longer than the best results for SHARP. However, in the presence of optimizer mistakes leading to sub-optimal join orders, using RDH can take up to 7.5 times longer than using the same join orders for SHARP.

Finally, we created a variation of the TPC-H schema, which we call TPC-H-Thin, to explore how the width of the tuples affects SHARP's relative performance. Each table in TPC-H-Thin is a projection of the corresponding table in TPC-H: table L contains only five integer columns, and tables O, P, C, and S contain only two integer columns. Queries 1, 2, and 3 were run again, but this time with much less available memory, such that not all build tables fit in memory. The results are similar to the ones shown in Figures 54, 55, and 56; SHARP is essentially insensitive to the join order specified by the optimizer while the RDH can be up to 3 times longer with one join order than another for the same query. Furthermore, for the queries we tested in TPC-H-Thin, RDH took, on average, 2.2 times longer than SHARP to complete execution.
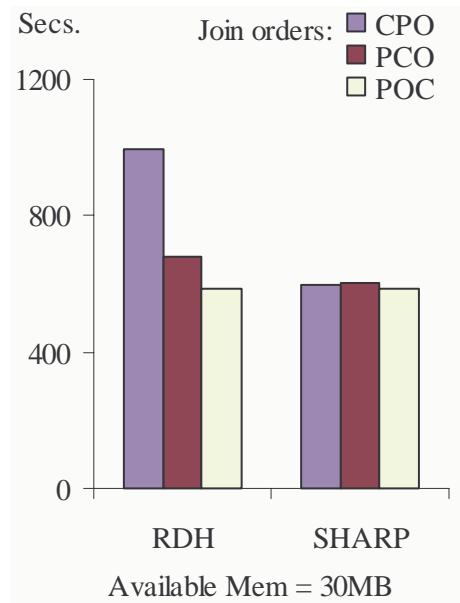
Figure 55 – Execution of Query 2
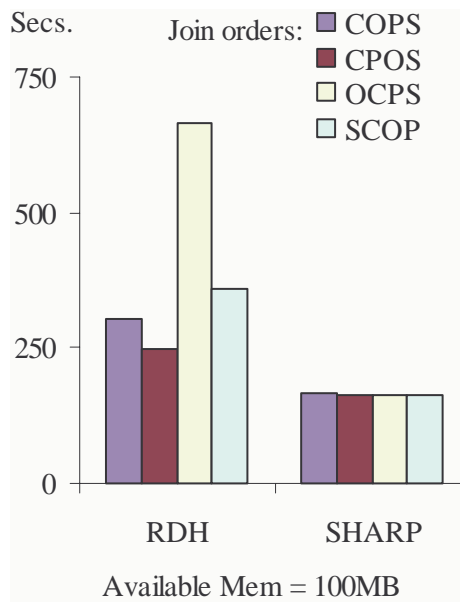
Figure 56 – Execution of Query 3

4.5. Related Work

In addition to the Eddies, MJoin and SHJ operators described in Section 4.2, the work related to SHARP can be grouped in five broad categories: adaptive operators, tuple routing strategies, techniques for processing joins larger than memory, techniques to change the query plan at run-time, and techniques that reduce the need for corrective behavior.

Other adaptive operators: The XJoin [88] is a binary adaptive operator that takes advantage of the non-blocking behavior of SHJ to process push-based remote relations. Although it can process relations bigger than memory, the XJoin schedules the join between out-of-memory relations to mask delays and bursty transfer rates of those sources. In contrast, SHARP uses adaptivity to execute robust plans over pull-based local relations. The operators most related to SHARP's late-binding decisions are the choose-plan operator [26, 40] and the switchable plan operator [11], but neither provides the continuous fine-grained adaptivity of SHARP.

Tuple routing strategies: In SHARP, we implemented three routing policies adapted from three proposals by Avnur [5], Babu [8], and Deshpande [29]. CBR is a tuple routing policy that takes advantage of correlation and skew to make better routing decisions [15]. The implementation of CBR in SHARP is left as future work.

Techniques for processing joins larger than memory: The Dynamic Hash Join [68] is the standard blocking binary hash join algorithm that adaptively freezes partitions to disk as needed and that we extended to join multiple relations simultaneously. Previous proposals extend the SHJ to process relations bigger than memory [54, 88]. However, these techniques were designed for binary joins over remote sources while SHARP processes multiple joins over local relations.

Changing query plan at run-time: In addition to the late-binding operators discussed before, there are proposals that use query re-optimization to correct possible optimizer mistakes [11, 54, 56, 58, 63]. These strategies are orthogonal to SHARP, i.e., a SHARP can be used as an adaptive operator in plans generated by those systems. Other proposals keep the same query plan, but reschedule operators to cope

with unpredictable delivery rates from remote data sources [17, 54, 56, 87, 88, 90] or to improve estimates for online queries [44, 45, 57]. In contrast, SHARP reschedules operators to better distribute memory between in-memory hash tables and possibly avoid a second pass. (A method to redistribute memory in traditional query plans is described in [28].) Finally, some data stream systems periodically determine and change to new query plans [8, 32], and these strategies can be incorporated in SHARP as routing policies (e.g., we implemented [8] in SHARP).

Techniques that reduce the need for run-time corrective adaptivity: Other approaches tackle the problem of insufficient information available to the optimizer by somehow modeling the uncertainty about estimates used at optimization [6, 11, 40, 52, 89]. Optimizers following this approach are more likely to choose robust plans and therefore less likely to need corrective adaptation at run-time. Due to the complexity of the search space, we believe that a combination of some of these techniques, together with adaptive operators like SHARP will prove to be the best approach.

More related work can be found in surveys and other publications with extended discussion of related work [7, 10, 46, 55, 54, 56, 63].

4.6. Conclusions

The observation that tuple routing is not expensive, but symmetric hash joins are [29] led us to design SHARP, a multi-join tuple routing operator without symmetric hash joins (SHJs). To avoid SHJs, we explored a new trade-off: instead of executing arbitrarily query plans, and being able to change from any join order to any other join order at any point during execution, SHARP adopts a two-step adaptive approach. First, SHARP determines which source is the driving relation using late-binding decisions, and second, it continuously potentially changes the probing sequence of the build sources using tuple routing. This two-step adaptive process yields two benefits: it requires less memory than previous adaptive operators and simplifies the design of second-stage processing.

In addition, the performance of the second-stage processing strategy is largely unaffected by estimates made during optimization. The second-stage was shown to be more effective than both left-deep and right-deep trees for a variety of scenarios. Most of the benefit of the proposed second-stage comes from avoiding writing intermediate results multiple times to disk. However, this new second-stage processing technique suffers from the "curse of dimensionality", and thus beyond certain parameters (very little memory, very large build tables, or a high number of joins), we expect its performance to degrade exponentially. Nevertheless, the problem is easily solved: after all build tables are read, if the sizes of memory and tables are such that the SHARP's (second-stage) performance is worse than a right-deep tree of hash joins, then the SHARP can simply execute the same plan a right-deep tree of hash joins would.

In addition, our initial results suggest that, unless the operator processing cost is very high, the A-Greedy [8] tuple routing policy is likely to be the best.

CHAPTER 5

PROGRESSIVE PARAMETRIC QUERY OPTIMIZATION

Many commercial applications rely on pre-compiled parameterized procedures to interact with a database. Unfortunately, executing a procedure with a set of parameters different from those used at compilation may be arbitrarily sub-optimal. Parametric query optimization (PQO) attempts to solve this problem by exhaustively determining the optimal plans in each point of the parameter space at compile time. However, PQO is likely not cost-effective if the query is executed infrequently or if it is executed with values only within a subset of the parameter space. In this chapter we propose instead to progressively explore the parameter space and build a parametric plan during several executions of the same query. We introduce algorithms that, as parametric plans are populated, are able to frequently bypass the optimizer but still execute optimal or near-optimal plans.

5.1. Introduction

There are two trivial alternatives to deal with the optimization and execution of parameterized queries. One approach, termed Optimize-Always, is to call the optimizer and generate a new execution plan every time the query is invoked. Another trivial approach, termed Optimize-Once, is to optimize the query just once with some set of parameter values and reuse the resulting physical plan for any other set of parameters. Both approaches have disadvantages. Optimize-Always requires an optimization call for each execution. The optimization call may be a significant part of the total execution time especially for simple queries. In addition, Optimize-Always may limit the number of concurrent queries in the system, as the optimization process itself may consume too much memory and may limit throughput. On the other hand, Optimize-Once returns a single plan that is used for all points in the parameter space. The problem is that

the chosen plan may be arbitrarily sub-optimal in all points of the parameter space other then the point for which the query was optimized for.

5.1.1. Parametric Query Optimization

An alternative to Optimize-Always and Optimize-Once is *Parametric Query Optimization* (PQO). At optimization time, PQO determines a set of plans such that, for each point in the parameter space, there is at least one plan in the set that it is optimal. The regions of optimality of each plan are also computed. PQO proposals often assume that the cost formulas of physical plans are linear or piece-wise linear with respect to the cost parameters or that the regions of optimality are connected and convex. However, in reality, the cost functions of physical plans are not necessarily linear or piece-wise linear and the regions of optimality are not necessarily connected nor convex. In addition, PQO has a much higher cost than optimizing a query a single time (e.g., PQO may require multiple invocations of the optimizer with different parameters [48, 49]). Thus, from the database perspective, when a parametric query execution request arrives, it is not clear if PQO should be used or not: it may not be cost-effective to solve the PQO problem if the procedure is not executed frequently or if it is executed with values only within a sub-space of the entire parameter space.

5.1.2. Contributions

The main contributions of this chapter are:

- In Section 5.2 we propose ***Progressive Parametric Query Optimization*** (PPQO), a new technique to improve the performance of processing parameterized queries. We also propose the Parametric Plan Interface as a way to incorporate PPQO in a DBMS with minimal changes to query processing.

- In Section 5.3 we propose ***Bounded***, an implementation of PPQO with proven guarantees of optimality.

- In Section 5.4 we propose *Ellipse*, another implementation of PPQO with higher hit rates and better scalability than Bounded.

- Finally, in Section 5.5 we present an extensive performance evaluation of PPQO using a prototype implementation and Microsoft's SQL Server 2005 DBMS.

5.2. Progressive Parametric Query Optimization

We propose a new technique called *Progressive Parametric Query Optimization* (PPQO) that addresses the shortcomings of PQO listed in Section 5.1.1. In essence, we want to progressively solve or approximate the solution to the PQO problem (formalized in Section 5.2.1) as successive query execution calls, with potentially different input parameters, are submitted. Given a query and its parameter values, an optimization call returns the optimal physical plan and the estimated cost of executing it. PPQO intercepts the inputs and outputs to and from the optimizer and registers which plans are estimated to be optimal for which points in the parameter space in a structure called *Parametric Plan* (PP), as described in Section 5.2.2.

Eventually, as parametric plans are populated, PPQO may be able to bypass the optimization process. Instead, when a query execution request arrives, PPQO uses its parametric plan to infer which plan to use for a particular set of parameter values. If it is able to find a plan, then optimization is avoided. Otherwise, an optimization call is made and its estimated optimal plan and cost is added to the query's parametric plan for future use. Due to the size of the parameter space, parametric plans cannot be implemented simply as an exact lookup against a cache of plans as there would be too many cache misses. Also, due to the non-linear and discontinuous nature of cost functions, parametric plans should not be implemented as nearest neighbor lookup structures as there will be no guarantee that the optimal plan of the nearest neighbor is optimal or close to optimal for the parameter point being considered.

5.2.1. The PQO Problem

A formal description of the PQO problem (adapted from other work [35, 48]) is presented below:

- A (parametric) query Q is a text representation of a relational query with placeholders for m values $vp = (v_1, \ldots, v_m)$. Vector vp is called a ***ValuePoint***.

- Let plan p be some execution plan that evaluates query Q for vp. The cost function of p, $p(cp)$, is a function of n cost parameters, $cp = (s_1, \ldots, s_n)$. Vector cp is called a ***CostPoint*** and each s_i is a cost parameter with an ordered domain.

- For every legal value of the parameters, there is some plan that is optimal. Given a parametric query Q, the ***maximum parametric set of plans*** (MPSP) is the set of plans, each of which is optimal for some point in the n-dimensional cost-based parameter space. MPSP = $\{p \mid p$ is optimal for some point in the cost-based parameter space$\}$.

- The ***region of optimality*** for plan p is denoted $r(p)$, $r(p) = \{(t_1, \ldots, t_n) \mid p$ is optimal at $(c_1 = t_1, \ldots, c_n = t_n)\}$.

- A ***parametric optimal set of plans*** (POSP) is a minimal subset of MPSP that includes at least one optimal plan for each point in the parameter space.

- The ***parametric query optimization*** (PQO) problem is to find a POSP and the region of optimality for each plan in POSP.

5.2.2. The Parametric Plan Interface

The Parametric Plan (PP) interface has two operations, addPlan and getPlan described below. PP is used during query processing as shown in Figure 57.

- **addPlan**(Q, cp, p, $cost$) – registers that plan p, with estimated cost $cost$, is optimal for query Q at CostPoint cp.

- **getPlan**(Q, cp) – returns the plan that should be used for query Q and CostPoint cp or returns null.

```
processQuery (   inputs: Query Q, ValuePoint vp
                 inputs/outputs: PP p  ) {
   CostPoint cp ←φ(Q, vp );              // Convert ValuePoint to CostPoint
   Plan p←pp.getPlan(Q, cp );            // what plan to use?
   if (p == NULL) {
      Cost cost;                          // cost is output parameter in call below
      p←optimize(Q, vp , cost);          // calls  optimizer
      p .addPlan(Q, cp , p, cost);       // stores info in PP
   };
   execute(p);
};
```

Figure 57 – Using Parametric Plans

Function φ consults the database catalog and query Q, and transforms ValuePoint vp into CostPoint cp . Function φ is optimizer specific. Section 5.2.4 justifies why φ is needed.

Besides PPQO, strategies Optimize-Always and Optimize-Once can also be coded with simple implementations of the PP interface. For Optimize-Always, addPlan is an empty method and getPlan simply returns null, forcing an optimization for every query, as shown in Figure 58. For Optimize-Once, as shown in Figure 59, addPlan saves the plan it is given as input the first time it is called and getPlan returns that plan in all calls.

```
01: class Optimize-Always implements PP         // Implements PP interface
02: begin-class
03:    addPlan(inputs: Query Q, CostPoint cp , Plan p, Cost cost) { };
04:    getPlan(inputs: Query Q, CostPoint cp ; outputs: Plan p) {
05:       return null;
06:    };
07: end-class;
```

Figure 58 – Implementation of Optimize-Always

```
01: class Optimize-Once implements PP          // Implements PP interface
02: begin-class
03:   Plan p;
04:   Optimize-Once() {p==null;}               // constructor
05:   addPlan(inputs: Query Q, CostPoint cp , Plan p, Cost cost) {
06:     if (!this.p) {this.p=p;}                // saves first plan it gets
07:   };
08:   getPlan(inputs: Query Q, Cost-Point cp ; outputs: Plan p) {
09:     return this.p;                          // returns first plan
10:   }
11: end-class;
```

Figure 59 – Implementation of Optimize-Once

5.2.3. Requirements and Goals

With PPQO we want to avoid as many optimization calls as possible and we are willing to execute sub-optimal plans if they have costs close to the cost of the optimal plan. Thus, PP implementations must obey the *Inference Requirement* below.

INFERENCE REQUERIMENT: After a number of addPlan calls, there must be cases where PP.getPlan(Q, cp) returns a (near-)optimal plan p for query Q and CostPoint cp , even if PP.addPlan(Q, cp , p, $cost$) was never called.

Given a sequence of execution requests of the same query with potentially different input parameters, PPQO has two conflicting goals:

- **GOAL 1**: Minimize the number of optimization calls; and

- **GOAL 2**: Execute plans with costs as close to the cost of the optimal plan as possible.

Note that a cache implementation of the PP interface–storing (Q, cp) pairs as the lookup key and (p, $cost$) as the inserted value–cannot fulfill the inference requirement because it would returns hits only for previously inserted (Q, cp) pairs. Instead, we propose two PPQO implementations, each giving priority

to one of the above goals: **Bounded**–described in Section 5.3–gives priority to Goal 2; **Ellipse**–described in Section 5.4–gives priority to Goal 1.

### 5.2.4. The Parameter Transformation Function φ

This section justifies why φ is needed and how is it implemented. A *value parameter* refers to an input parameter of a parametric SQL query to be executed. A *cost parameter* is an input parameter in formulas used by the optimizer to estimate the cost of a query plan. Cost parameters are estimated during query optimization from value parameters and from information in the database catalog. (Physical characteristics that affect the cost of query plans but do not depend on the query parameters–e.g., the average size of tuples in a table or the cost of a random I/O–are considered physical constants, not cost parameters.)

An important type of cost parameter used during optimization is the estimated number of tuples in (intermediate) relations processed by the query plan: most query plans have cost formulas that are monotonic with the number of tuples processed by the query. On the other hand, there is no obvious relationship between the value parameters and the cost of the query plans. Thus, it becomes much easier to characterize the regions of optimality using a cost-based parameter space than using a value-based parameter space. In Example 5.1, below, and in what follows, we use a cost-based parameter space whose dimensions are (predicate or join) selectivities. (The estimated number of tuples of each relation processed by a query is typically derived from selectivities of sub-expressions computed during query optimization.)

Example 5.1: Relation FRESHMEN(NAME, AGE) describes 1st-year graduate students. The age distribution of students is showed in Figure 60. Consider different queries of the form SELECT * FROM FRESHMEN WHERE AGE=X OR AGE=Y. Assume that the optimal plan for queries that retrieve less than 5% of FRESHMEN tuples is P_{IDX}, a plan using an index on column AGE. For all other queries, the optimal plan is

P$_{FS}$, a full-table scan on FRESHMEN. The parameters of this query can be represented as the absolute values used for parameters X and Y or as the selectivities of predicate AGE=X and predicate AGE=Y. Accordingly, the costs of physical P$_{IDX}$ and P$_{FS}$ can be represented in value-based parameter spaces or in selectivity-based parameter spaces as seen in Figure 61. ∎

In our implementation, function φ takes query Q and its SQL parameters–the ValuePoint vp –and returns cp as a vector of selectivities. Computing the selectivities in cp corresponds to *selectivity estimation*, a sub-task of query optimization. Other components of query optimization–e.g., plan enumeration, rule transformation, plan costing, and plan pruning–are not executed by function φ. (Note that the arity of the value-based parameter space and of the selectivity-based parameter space are not necessarily the same.) For range predicates and equality predicates, computing selectivity values from actual values–the task of φ–can be done efficiently by lookups on cumulative histograms.
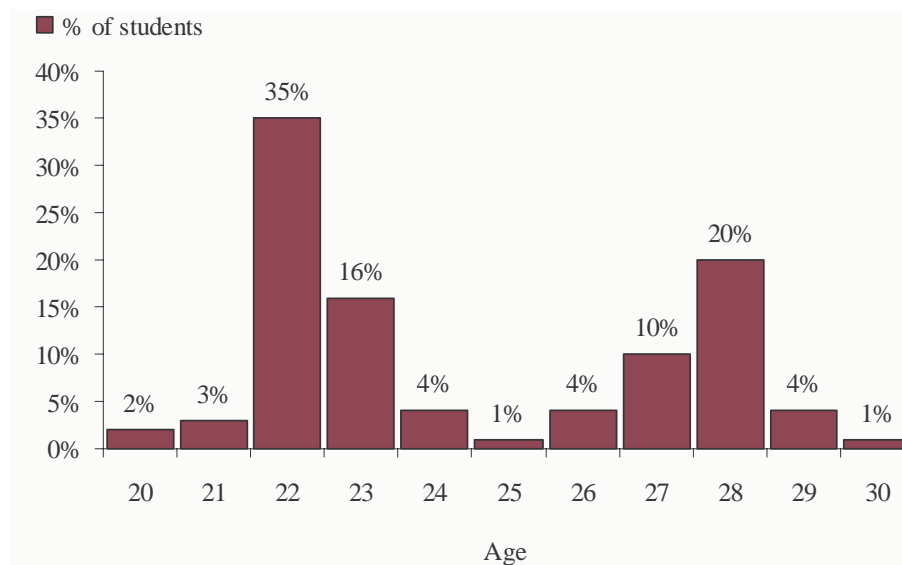
Figure 60 – Age distribution in table FRESHMEN
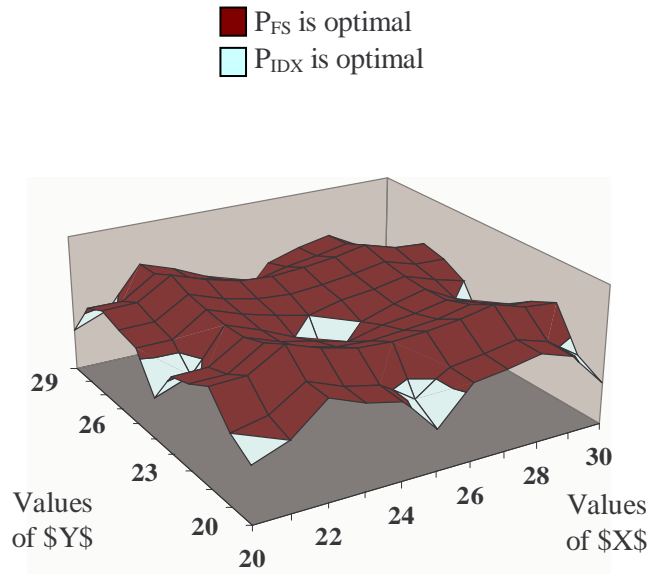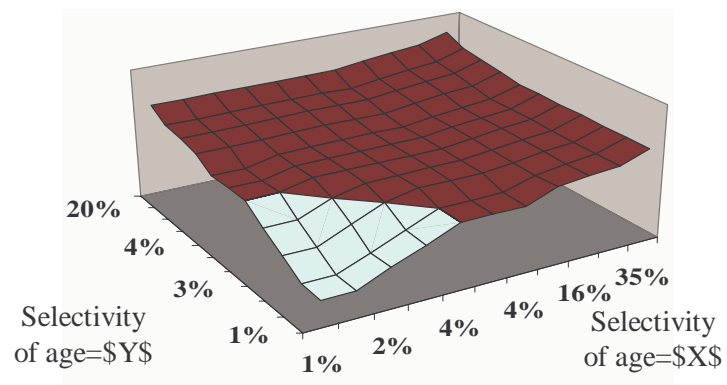
Figure 61 – Value-based and selectivity-based parameter space

5.3. The Bounded PPQO Implementation

The first of the two proposed PPQO implementations, termed Bounded, is described in this section. This implementation provides guarantees on the quality of the plans returned by getPlan(Q, cp), thus focusing on Goal 2 of PPQO (see Section 5.2.3). Either the returned plan p is null–meaning that an optimization call cannot be avoided–or plan p has a cost guaranteed to be within a bound (specified by the user) of the cost of the optimal plan.

5.3.1. Preliminaries and Definitions

- **Relationship equal ($\equiv$):** Given $cp_1=(c_{1,1}, \ldots, c_{1,n})$ and $cp_2=(c_{2,1}, \ldots, c_{2,n})$, $cp_1 \equiv cp_2$ iff $c_{1,i}=c_{2,i}$, $\forall i$.

- **Relationships below ($\triangleleft$) and above ($\triangleright$):** Given $cp_1=(c_{1,1}, \ldots, c_{1,n})$ and $cp_2=(c_{2,1}, \ldots, c_{2,n})$, $cp_1 \triangleleft cp_2$ ($cp_1 \triangleright cp_2$) iff $c_{1,i} \leq c_{2,i}$ ($c_{1,i} \geq c_{2,i}$), $\forall i$ and $\exists i$, $c_{1,i} \neq c_{2,i}$.

- **Transitive property of $\triangleleft$ and $\triangleright$.** From the definitions it follows that if $cp_1 \triangleleft cp_2$ ($cp_1 \triangleright cp_2$) and $cp_2 \triangleleft cp_3$ ($cp_2 \triangleright cp_3$) then $cp_1 \triangleleft cp_3$ ($cp_1 \triangleright cp_3$).

- **Monotonic Assumption (MA):** Given plan p and CostPoints cp_1 and cp_2, if $cp_1 \triangleleft cp_2$ then $p(cp_1) \leq p(cp_2)$.[18]

- **$\emptyset t$ (cpt):** It is the cost of an optimal plan at cp .

---

[18] All cost parameters we use are selectivities. Since higher selectivities imply more tuples to process, the monotonic assumption follows the intuition that plans that process more tuples likely cost more than plans that process less tuples. Although not true for all queries–e.g., queries using SQL clause NOT EXISTS may have non-monotonic costs–plans with non-monotonic costs are less common than plans with costs monotonic with the number of processed tuples.

Theorem 5.1: If $\exists t_i = (cp_i, plan_i, cost_i)$, $\exists t_j = (cp_j, plan_j, cost_j)$, such that plan $plan_i$ ($plan_j$) is an optimal plan at cp_i (cp_j) with cost $cost_i$ ($cost_j$), $cp_i \prec cp \prec cp_j$ and $cost_j \in [cost_i, cost_i*M+A]$, then $plan_j(cp) \in [Op(cp), Op(cp)*M+A]$.[19]

Proof: See Section 5.8

Theorem 5.1 states that the difference between the cost of $plan_j$ at cp_j and the cost of $plan_i$ at cp_i can be used to bound the cost of $plan_j$ at cp, as long as costs are monotonic and $cp_i \prec cp \prec cp_j$.

5.3.2. Implementation of addPlan

Function addPlan(Q, cp, p, $cost$)–shown in –associates with each parametric query Q a list, T_Q, of triples $(cp, p, cost)$ ordered by $cost$, where p is an optimal plan at CostPoint cp with an estimated execution cost of $cost$.

```
addPlan (inputs: Query Q, CostPoint cp, Plan p, Cost cost) {
   List T_Q←getList(Q);              // Get the list of triples for this query
   if (T_Q ==null) {
      T_Q = new List();              // If there is no list, create one
   }
   T_Q.insert(cp, p, cost);          // Inserts triple in cost order
   setList(Q, T_Q)                   // adds or replaces list T_Q into catalog
}
```
Figure 62 – Bounded's addPlan

5.3.3. Quality Guarantees of getPlan

Bounded's getPlan(Q, cp) is guaranteed to either return null or to return a plan with an estimated cost as close to the estimated optimal cost as desired. Specifically, for any constants $M \geq 1$ and $A \geq 0$, Bounded's getPlan guarantees that, after calling p=getPlan(Q, cp) one of the following holds:

[19] M is the multiplicative factor and A is the additive factor specified by the user.

- p is null or

- $p(cp) \in [Op(cp), Op(cp) *M+A]$.

Definition of bounding pair and bounded plan: Given two triples $t_1=(cp_1, p_1, cost_1)$ and $t_2=(cp_2, p_2, cost_2)$, where cp_1 and cp_2 are CostPoints, $cost_1$ ($cost_2$) is the positive cost of optimal plan p_1 (p_2) at cp_1 (cp_2), and any constants $M \geq 1$ and $A \geq 0$. If $cp_1 < cp < cp_2$ and $cost_2 \in [cost_1, cost_1*M+A]$ then we say that:

- (t_1, t_2) **bound** cp.

- Plan p_2 **is bounded** at cp.

Note that, by Theorem 5.1, if p_2 is bounded at cp then $p_2(cp) \in [Op(cp), Op(cp)*M+A]$. Given $M \geq 1$, $A \geq 0$, query Q and CostPoint cp, Bounded's getPlan (see Section 5.3.4) searches for a (t_1, t_2) pair that bounds cp and returns p_2, a bounded plan at cp, fulfilling point ii) above. If no (t_1, t_2) bounding pair for cp exists, getPlan returns null, fulfilling point i) above.

Example 5.2: For some query Q, assume that Bounded.addPlan was already called for the triples showed in Figure 63 (i.e., $T_Q=(t_1, t_2, t_3, t_4, t_5, t_6, t_7)$).
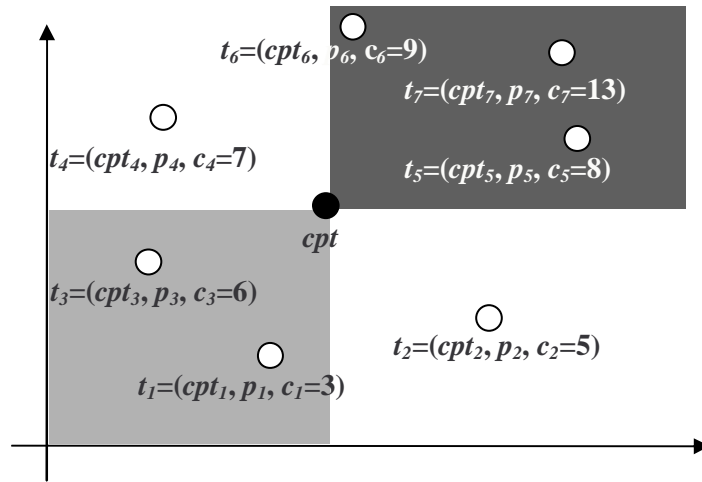
Figure 63 – Triples stored in Bounded

Given, cp –showed as a black circle–in the cost-based parameter space, $M=1.5$, and $A=0$, what plan will Bounded.getPlan(Q, cp) return? There are six pairs (cp_i, cp_j) such that $cp_i ◄ cp ◄ cp_j$: (cp_1, cp_5), (cp_1, cp_6), (cp_1, cp_7), (cp_3, cp_5), (cp_3, cp_6), and (cp_3, cp_7). From those pairs, only two triples bound cp: pair (t_3, t_5), because $c_5 \in [c_3, c_3*1.5+0] \Leftrightarrow 8 \in [6, 9]$, and pair ($t_3$, t_6), because $c_6 \in [c_3, c_3*1.5+0] \Leftrightarrow 9 \in [6, 9]$. Thus, both plan p_5 and plan p_6 are bounded at cp and either of them can be returned by getPlan. ∎

5.3.4. Implementation of getPlan

Consider T_Q, the list containing k triples (cp_i, p_i, $cost_i$) maintained by method addPlan. A naïve implementation of getPlan enumerates all pairs of tuples (t_i, t_j), $t_i \in T_Q$, $t_j \in T_Q$, $t_i \neq t_j$ and tests if any pair bounds cp. If some pair (t_i, t_j) bounds cp, then plan p_j can be returned as the answer to getPlan.

To avoid the enumeration of all of pairs of triples that have to be checked, getPlan divides T_Q into two lists. Then, given the properties of the two lists (described below), it is possible to trivially select a single triple, t_1, from one list and a single triple, t_2, from the other list such that only pair (t_1, t_2) needs to be checked.

Definition of ◄ (below) operator and ▶ (above) operator: Consider a list, T_Q, containing k triples $(cp_i, p_i, cost_i)$ ordered by $cost_i$, with $i=0...k-1$, where cp_i is a CostPoint and $cost_i$ represents the cost of executing the optimal plan p_i at cp_i. Given cp, another CostPoint, $T_{Q◄cp}$ is the list of triples $(cp_i, p_i, cost_i)$ from T_Q, ordered by $cost_i$, such that $cp_i ◄ cp$. Similarly, $T_{Q▶cp}$ is the list of triples $(cp_i, p_i, cost_i)$ from T_Q ordered by $cost_i$, such that $cp_i ▶ cp$. $T_{Q◄cp}$ and $T_{Q▶cp}$ are trivially constructed from a single pass over T_Q. Note that, by definition, $cp_b ◄ cp ◄ cp_a$, $\forall cp_b : t_b = (cp_b, p_b, cost_b) \in T_{Q◄cp}$, $\forall cp_a : t_a = (cp_a, p_a, cost_{ba}) \in T_{Q▶cp}$.

Example 5.3: Let $T_Q = (t_1, t_2, t_3, t_4, t_5, t_6,)$, where the t_i are the triples shown in Figure 63. Then $T_{Q◄cp} = (t_1, t_3)$ (the triples in the light gray area) and $T_{Q▶cp} = (t_5, t_6, t_7)$ (the triples in the dark gray area). ∎

Theorem 5.2: If $\exists cp_b : t_b = (cp_b, p_b, cost_b)$, $t_b \in T_{◄cp}$, $\exists cp_a : t_a = (cp_a, p_a, cost_a)$, $t_a \in T_{▶cp}$, such that $cost_a \in [cost_b, cost_b * M + A]$, then $cost_{first} \in [cost_{last}, cost_{last} * M + A]$, where $cost_{first}$ is the cost of the first triple in $T_{▶cp}$ and $cost_{last}$ is the cost of the last triple in $T_{◄cp}$.

Proof: See Section 5.8

Theorem 5.2 states that when searching list T for a pair of triples that bound cp, we only need to test the pair composed by the last triple from $T_{◄cp}$ and the first triple from $T_{▶cp}$.

As shown in Example 5.2 in Section 5.3.3, there is potentially more than one possible solution to getPlan(Q, cp). However, if there is a solution, by Theorem 5.2, we need only to check if $cost_{first} \in [cost_{last}, cost_{last} * M + A]$, where c_{first} is the cost of the first triple in $T_{Q▶cp}$ and c_{last} is the cost of the last triple in $T_{Q◄cp}$. If $cost_{first} \in [cost_{last}, cost_{last} * M + A]$, then p_{first}, the plan in the first triple of $T_{Q▶cp}$, is returned.

Before addPlan is called the first time, any getPlan call returns null. As new triples are added, the hit rate of getPlan is expected to increase. Intuitively, as more triples are added, the more likely it is that getPlan returns a plan because it is more likely that any two triples fulfill the requirements of the Theorem 5.2. Note also that the lower the values of M and A, the less likely it is to find pairs of triples that fulfill the requirements of Theorem 5.2, and thus, more added triples are needed to obtain higher hit rates.

```
getPlan (inputs: Query Q, CostPoint cp ; outputs: Plan p) {
  List T_Q ←getList(Q);              // gets list of triples for Q
  if (T_Q ==null) {
    return null;
  }
  Triple last=null;                  // last triple of T_Q‹cp
  for Triple t in T_Q {              // in cost order
    if (t.cp ≡ cp ) {return t.p;}    // exact match?
    if (t.cp ‹ cp ) {last = t;}      // keep track of last triple of T_Q‹cp
    if (t.cp › cp ) {                // first triple of T_Q›cp
      if (last == null) {
        return null;
      }
      if (t.c∈ [last.c, last.c*M+A]) {
        return t.p;
      }
    }
  }
}
```

Figure 64 – Bounded's getPlan

Note that getPlan, shown in Figure 64, makes at most a single pass over T_Q; thus, it has $O(|T_Q|)$ time complexity, where $|T_Q|$ is the number of elements in T_Q.

5.4. The Ellipse PPQO Implementation

Bounded's getPlan provides strong guarantees on the cost of plans returned. However, we expect low hit rates of Bounded's getPlan for small values of M and A or before Bounded's T_Q has been populated. In this section we propose Ellipse, another PPQO implementation of the PP interface, designed to address PPQO's Goal 1: to have higher hit rates.

To have higher hit rates, Ellipse drops the guarantee of only returning plans with near-optimal costs. Instead, Ellipse's getPlan returns Δ-*acceptable plans*.

Definition of Δ-Acceptable Plans: For $\Delta \in [0, 1]$, if plan p is optimal at points cp_1 and cp_2 in the cost-based parameter space, then plan p is Δ-acceptable at point cp in the cost-based parameter space iff distance(cp_1, cp_2)/(distance(cp, cp_1) + distance(cp, cp_2)) $\geq \Delta$, where the function distance returns the Euclidian distance between two points in an n-dimensional space.

It follows from the definition of Δ-acceptable that if p is optimal at cp_1 and cp_2, then p is 1-acceptable only on points between cp_1 and cp_2 and p is 0-acceptable at all points. Note that in a 2-dimentional space, the area where p is Δ-acceptable is equivalent to the definition of an ellipse; if p is optimal for cp_1 and cp_2, then p is Δ-acceptable at cp if cp is on or inside an ellipse of foci cp_1 and cp_2 such that the distance between the foci, distance(cp_1, cp_2), over the sum of the distances between cp and the foci, distance(cp, cp_1) + distance(cp, cp_2), is Δ. Figure 65 shows the areas where p is 0.5-acceptable, 0.8-acceptable, and 1-acceptable if p is optimal at cp_1 and cp_2.
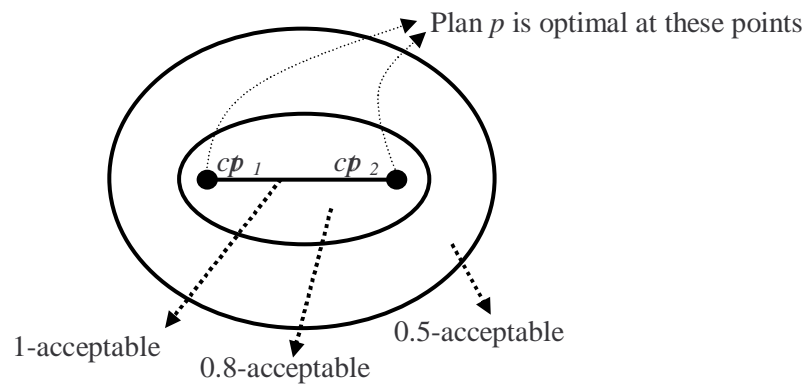
Figure 65 – Examples of Δ-acceptable plans

5.4.1. Implementation of addPlan

For each query Q and for each plan p that is optimal in some point of the parameter space, Ellipse's addPlan(Q, cp, p, *cost*) function–shown in Figure 66–maintains a list of (cp, *cost*) pairs where p is optimal for Q.

5.4.2. Implementation of getPlan

The implementation of Ellipse.getPlan consists of, for each optimal plan *plan*, iterating over pairs of points where *plan* is optimal for the given query, Q. For each pair of points (cp_1, cp_2), we test if *plan* is Δ-acceptable at the given point cp. If it is, getPlan returns *plan*, otherwise getPlan continues trying other points and other plans. If all pairs of points of all plans for Q are exhausted without an Δ-acceptable plan being found, Ellipse.getPlan returns null. The algorithm is shown in Figure 67.

```
addPlan(inputs: Query Q, CostPoint cp , Plan p, Cost cost) {
  PointList L←getPointList(Q, p);        // where is p optimal?
  if (L==null) {                          // if no points were p is optimal for Q…
    L = new PointList();                  // … create new PointList
    PlanList P←getPlanList(Q);            // optimal plans for Q
    if (P==null) {
      P=new PlanList(p);
    } else {
      P.insert(p);                        // add new optimal plan to list
    }
    setPlanList(Q, P);                    // adds or replaces list P in catalog
  }
  L.insert(cp , cost);                    // Adds new information about p to L.
  setPointList(Q, p, L)                   // adds or replaces list L in catalog
}
```

Figure 66 – Ellipse's addPlan

```
getPlan (inputs: Query Q, CostPoint cp ; outputs: Plan p) {
  PlanList P ←getPlanList(Q);            // gets optimal plans
  if (P ==null) {return null;}            // tests for empty list
  for Plan plan  in P {
    PointList L←getPointList(Q, plan );  // gets list of points
    for PointPair (cp 1, cp 2) in L {     // enumerates point pairs
      if (dist(cp1 , cp2 ) / (dist(cp , cp1 ) + dist(cp , cp2 ))≥Δ) {
        return plan ;                     // found an Δ-acceptable plan
      }
    }
  }
  return null;
}
```

Figure 67 – Ellipse's getPlan

5.5. Experimental Evaluation

In this section we describe an experimental evaluation of PPQO using Microsoft's SQL Server 2005. The client application implements the pseudo-code described in Sections 5.2, 5.3, and 5.4; SQL Server is used only to obtain estimated optimal plans and estimated costs of plans and to implement function φ (lines 7 and 3 in Figure 57).

5.5.1. Dataset, Metrics, and Setup

The TPC-H benchmark [85] was used to evaluate the PPQO implementations. Table 8, below, shows which tables are joined by each query. (The queries' full SQL text, too large to show here, is shown in section 5.9.) The tables are lineitem (L), orders (O), customer (C), supplier (S), part (P), partsupp (T), nation (N), and region (R).

As in Reddy and Haritsa [76], and unless otherwise noted, we added two extra selections to the TPC-H queries to more easily explore the parameter space. The two selections are of the form $col_i \leq val_i$, $i=1,2$, where, for each query, col_i is one of the two columns shown in Table 8 and val_i is a random value from the domain of the column.

Table 8 – Description of TPC-H queries used

Query	Tables Joined	Column 1	Column 2
7	LOCSNN	c_acctbal	o_totalprice
8	LOCPSNNR	s_acctbal	l_extendedprice
9	LOTPSN	s_acctbal	l_extendedprice
18	LLOC	c_acctbal	l_extendedprice
21	LLLOSN	s_acctbal	l_extendedprice

For each query tested, we generated 10,000 random val_1 and val_2 values. (A (val_1, val_2) pair is a ValuePoint.) To guarantee that random parameter values uniformly explore the parameter space, we altered the values in the columns subject to the extra selections to enforce uniform distributions.

For each query and each ValuePoint vp we make a PP.getPlan lookup call (see Figure 57 in page 126), where PP is an Optimize-Once, Optimize-Always, Bounded, or Ellipse object. If getPlan returns a plan we call it a hit and check if the plan is optimal; if it is not optimal we check how its estimated cost compares with the estimated optimal cost. These give rise to the following metrics:

- **HitRate**: The percentage of PP.getPlan(Q, cp) calls that return a plan.

- **OptRate**: The percentage of such plans that are optimal.

- **SO**: How sub-optimal a returned plan is: $p_{hit}(cp)/Op(cp)$, with p_{hit}=PP.getPlan(Q, cp). SO$\geq$1.

- **AvgSO**: The average of all SO

- **MaxSO**: The maximum of all SO; reflects how risky a PP implementation can be.

- **Number of points**: Number of (cp, $plan$, $cost$) triples stored in a ParametricPlan. Equal to the number of misses.

- **Number of plans**: Number of distinct optimal plans observed.

- **QP**: Number of queries processed.

The experiments were run on a lightly loaded Pentium M at 1.73GHz with 1GB of RAM and using TPC-H scale factor 1. Indexes and statistics were built on all columns subject to selections and on all primary and foreign key columns. The optimizer cache was emptied before each optimization call. To compute some of the metrics above, the cost of sub-optimal plans (returned by PPQO) also had to be estimated. To estimate those costs, each sub-optimal plan was forcibly costed by SQL Server [64].

Unless stated otherwise, Bounded was run with M=1.1, A=0 and Ellipse was run with Δ=0.95.

5.5.2. Variation on HitRate and OptRate

The first experiment consisted of processing 10,000 queries using different random ValuePoints (i.e., 10,000 different random sets of SQL parameter values) for each query and observing how HitRate and

OptRate varied for Bounded and Ellipse. This experiment was performed for five TPC-H queries and the results are shown in Figures 68-72. Several trends can be observed:

- Ellipse always has a higher HitRate than Bounded.

- Except for Query 8 (more on this below), Bounded always has a higher OptRate than Ellipse.

- HitRate converges quickly, but OptRate converges slightly faster.

- HitRate monotonically increases as a function of QP because more queries processed imply a monotonically increasing number of misses and each miss adds more information to the ParametricPlan, therefore increasing the likelihood of future hits.

- OptRate naturally varies up and down, as the initial random (cp, pan, $cost$) triples are added to the ParametricPlan object, until it converges.
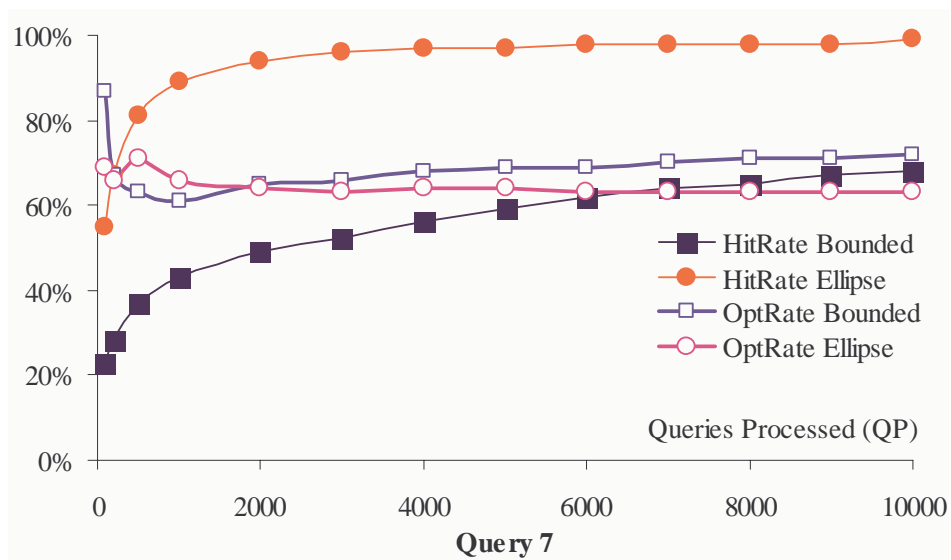
Figure 68 – Variation of HitRate, OptRate for 10,000 QP; Query 7
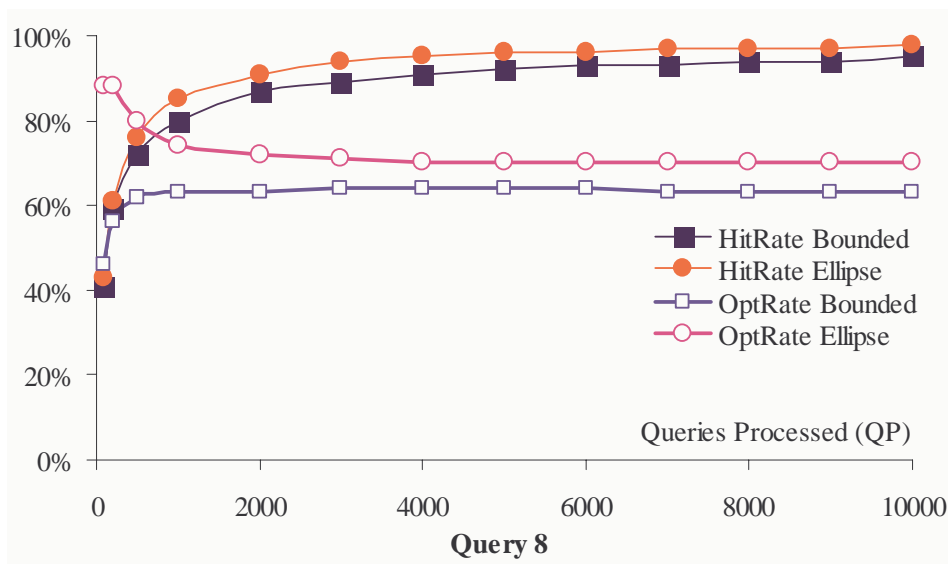
Figure 69 – Variation of HitRate, OptRate for 10,000 QP; Query 8
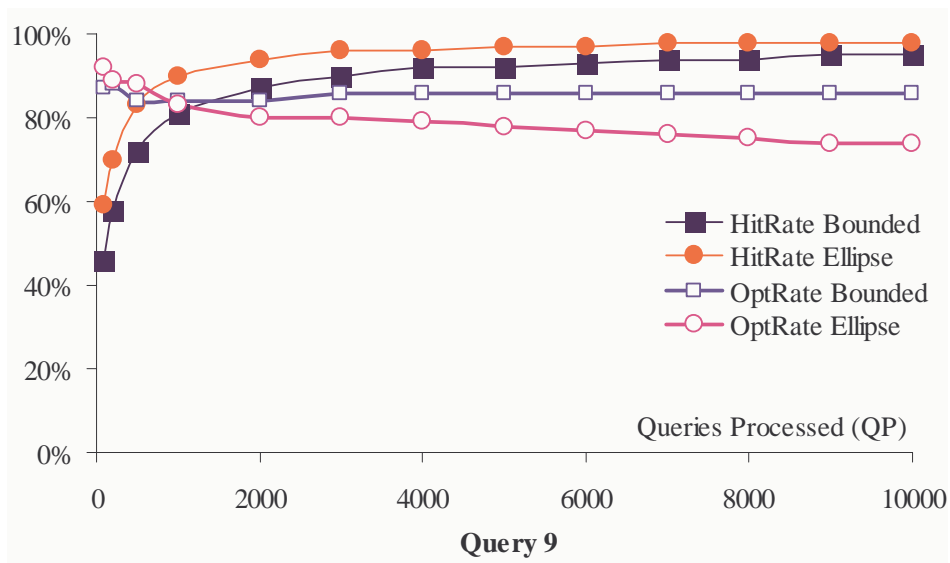

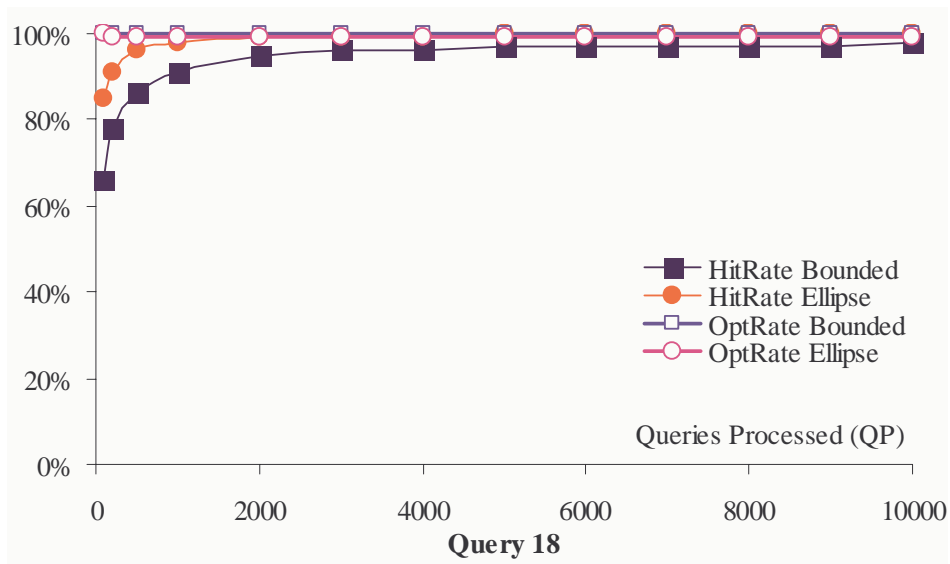**Figure 70 – Variation of HitRate, OptRate for 10,000 QP; Query 9**

Figure 71 – Variation of HitRate, OptRate for 10,000 QP; Query 18
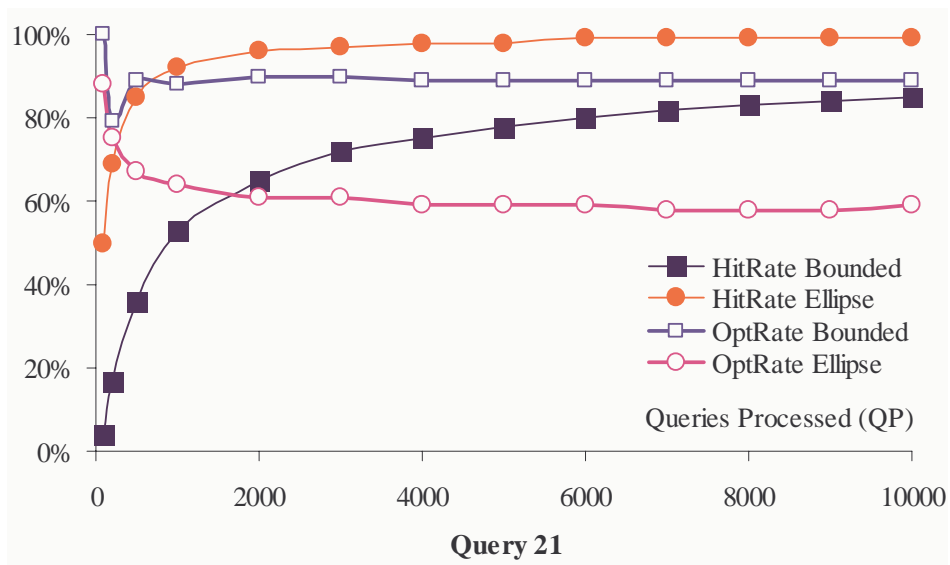

Figure 72 – Variation of HitRate, OptRate for 10,000 QP; Query 21

5.5.3. Number of Plans, Number of Points, Space, and Time

Figures 73 and 74 show the total number of plans and total number of points stored in the ParametricPlan objects at the end of the experiments of the previous section. Bounded has both a higher number of plans and a higher number of points because it has a lower HitRate; for every miss there will be a new point stored in the ParametricPlan object. Figure 75 shows that space consumed to store the plans and the points within the ParametricPlan objects varied between ~600Kbytes to ~1300Kbytes. We used the original uncompressed XML plan representations provided by SQL Server. Storing zip-compressed XML plans instead would decrease the size of the plan representation by a factor of 10.

Figure 76 reports the time taken by the Bounded and Ellipse. Time (in seconds) includes time elapsed during optimization (if there is a miss), during addPlan, and during getPlan, but not query execution time. For comparison purposes, the time taken for Optimize-Once and Optimize-Always is also included. After 10,000 queries have been processed, Optimize-Always took between 5.2 and 13.6 times longer than Bounded and between 10.7 and 18.5 times longer than Ellipse. Ellipse was always faster than Bounded because it had fewer optimize and addPlan calls (due to higher HitRates) and faster getPlan calls (because it has less information stored in its parametric plans).
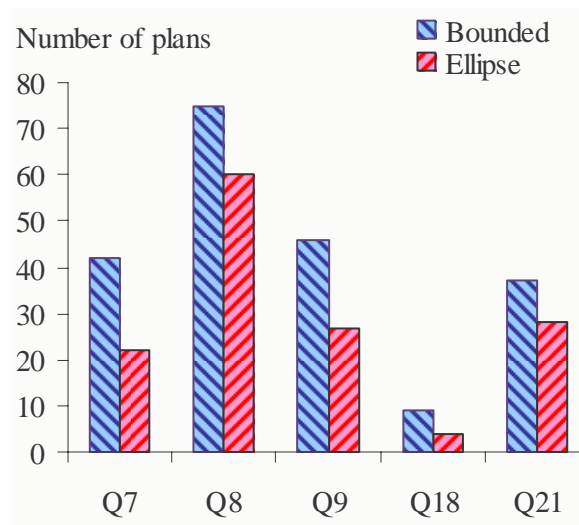
Figure 73 – Number of plans after 10,000 queries processed
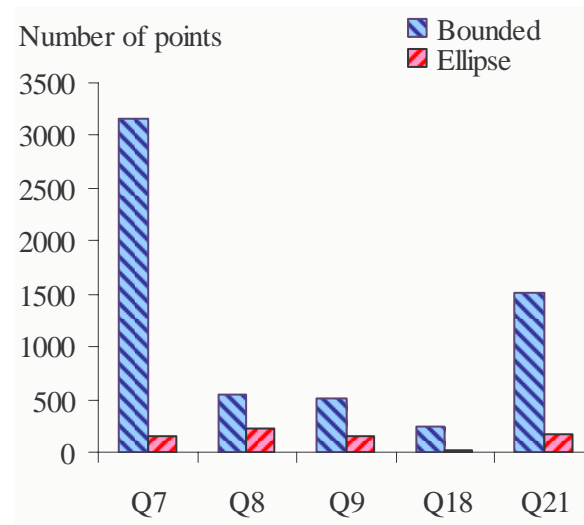
Figure 74 – Number of points after 10,000 queries processed
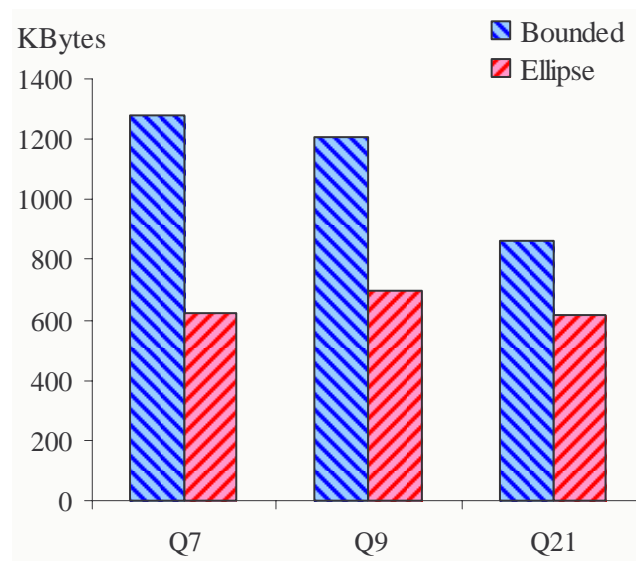
Figure 75 – Space consumed after 10,000 queries
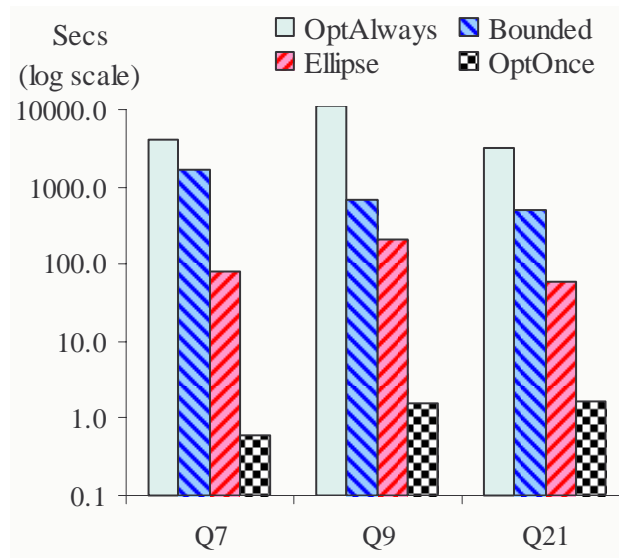
Figure 76 – Optimization and ParametricPlan time to process 10,000 queries

5.5.4. MaxSO and AvgSO

Figures 77 and 78 show, respectively, the MaxSO and AvgSO for Bounded, Ellipse, and Optimize-Once (OptOnce in the graphs) for the same experiments as in the previous two sections.
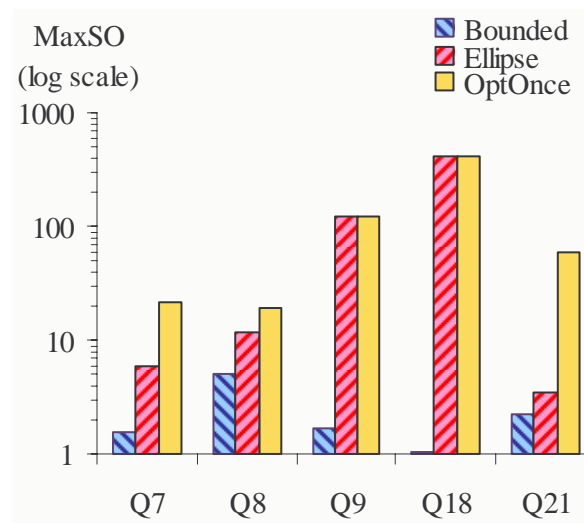
Figure 77 – MaxSO
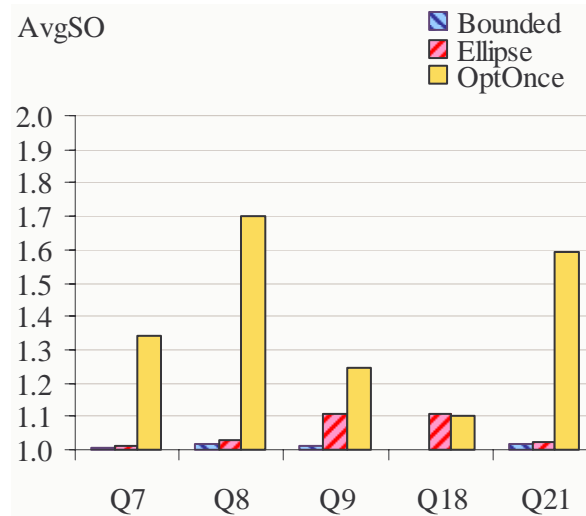
Figure 78 – AvgSO

Another surprise was how well Optimize-Once did in the AvgSO metric. On average, across all queries, Optimize-Once returned plans with costs ~140% the cost of optimal (the same average was ~101% for Bounded and ~106% for Ellipse). One possible explanation is the following. Optimize-Once obtains the optimal plan for the first of the 10,000 random parameter values and reuses that plan for all other values. If that first plan happens to be the plan with the minimal cost variation in the plan space, then there is a significant chance that that plan will do well in many other points in the space. Consider Figure 79, which shows a conceptual representation of the costs of four different plans, each optimal in different regions of the parametric space.
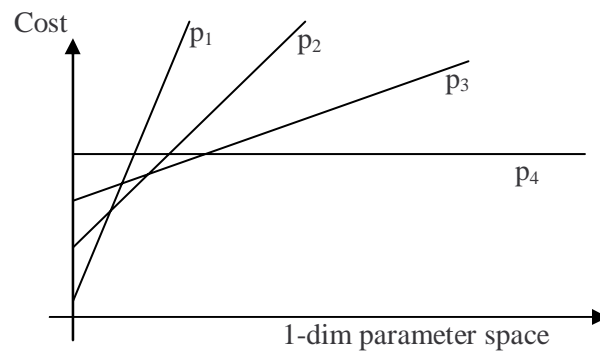
Figure 79 – Typical costs of optimal plans

Executing either plan p_3 or plan p_4 for all points of the parameter space would yield costs, on average, not much higher than the cost of optimal. Coincidently, the likelihood that any given point lies in the space where either p_3 or p_4 are optimal is very high, and thus, by random chance, Optimize-Once is likely to use a plan that is not catastrophic. We will explore this issue further in Section 5.5.6.

5.5.5. Vary Bounded's M and Vary Ellipse's Δ

In this experiment the value *M* of Bounded was varied from 1.1 to 4, for query 21 (to avoid clutter, and because its line is similar to the line of *M*=3, *M*=4 is not shown). The values of HitRate and OptRate are shown in Figures 80 and 81. As expected, a lower value for *M* (tighter optimality bound) results in a higher OptRate but a lower HitRate.

The same query 21 with the same random parameter values was run using Ellipse while varying Δ from 0.85 to 0.99 (Δ=0.85 not shown). As expected, a higher Δ results in a lower HitRate but a higher OptRate. These results appear in Figures 82 and 83.
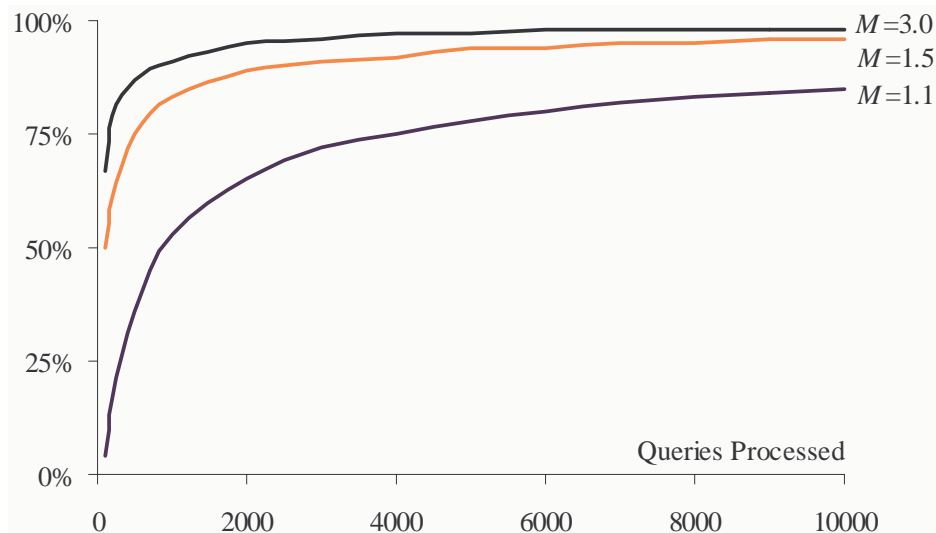
Figure 80 – HitRate for Bounded; Query 21
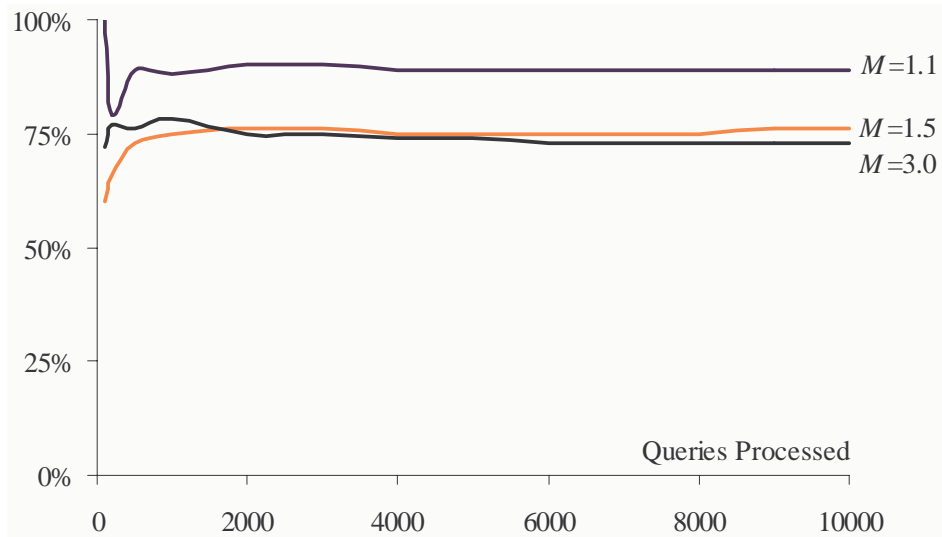
Figure 81 – OptRate for Bounded; Query 21
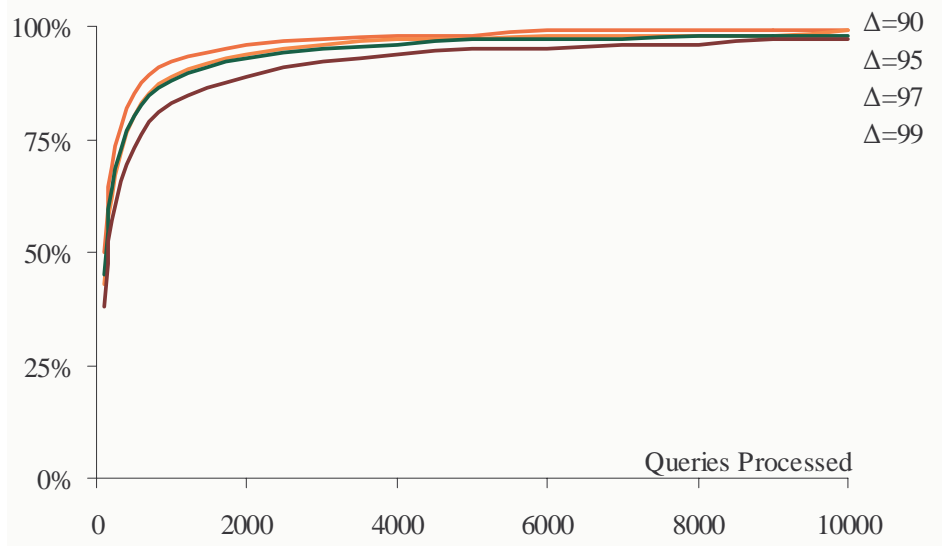
Figure 82 – HitRate for Ellipse; Query 21
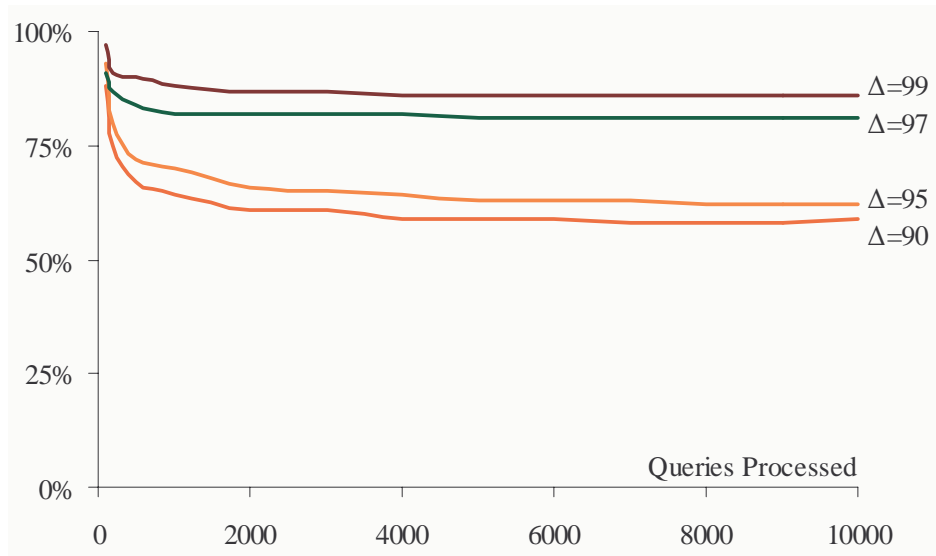
Figure 83 – OptRate for Ellipse; Query 21

5.5.6. Vary Query Order

This experiment assessed the impact of the order of the incoming queries on the performance of the algorithms. The same 10,000 random values used for Query 21 were used again, but the order in which those 10,000 queries were processed was chosen randomly. Six random orders were generated and processed with Bounded (M=1.1, A=0), Ellipse (Δ=0.9), and Optimize-Once. The results are shown in Figures 84–87 and summarized in Table 9.

Table 9 – Effects of different query orders

	OptRate			HitRate		
	Max	**Min**	**Avg**	**Max**	**Min**	**Avg**
Bounded	89.0%	86.0%	87.8%	86.0%	85.0%	85.8%
Ellipse	71.0%	59.0%	65.7%	99.0%	99.0%	99.0%
OptOnce	48.0%	3.0%	35.2%	-	-	-

Query order had essentially no effect on the final values of Bounded's OptRate, Bounded's HitRate, and Ellipse's HitRate but it had a medium impact on the final value of Ellipse's OptRate.

On the other hand, for Optimize-Once, query order had a very significant impact on OptRate, with values between 3% and 48%. An interesting observation is that the performance of Optimize-Once was

exactly the same for four out of those six random orders. Further analysis showed that, although the very first value of each of the six random orders were all different, for four of them, the corresponding optimal plan was the same. This follows the observation (Section 5.5.4, Figure 79, and [76]) that some plans have very large optimality areas.
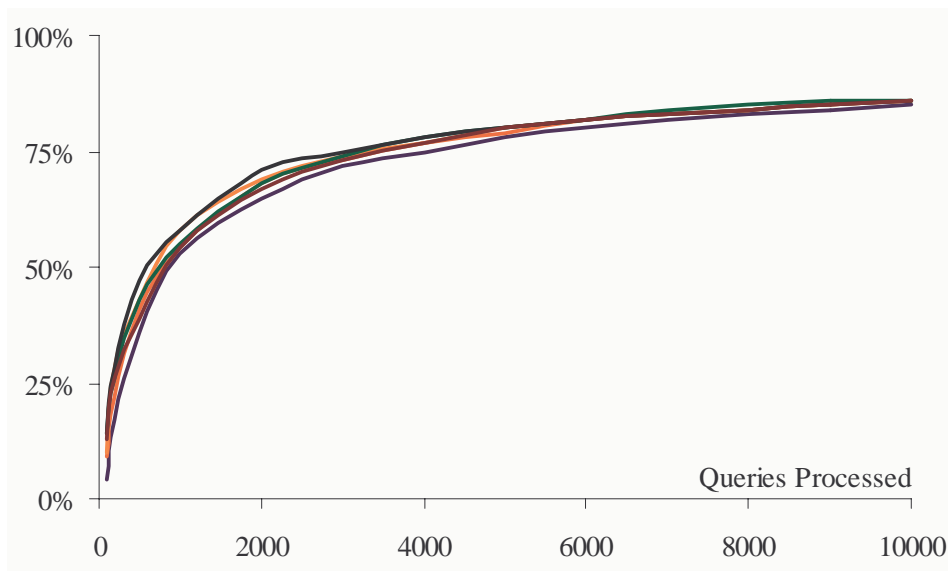
Figure 84 – HitRate for Bounded, 6 random query orders; Query 21
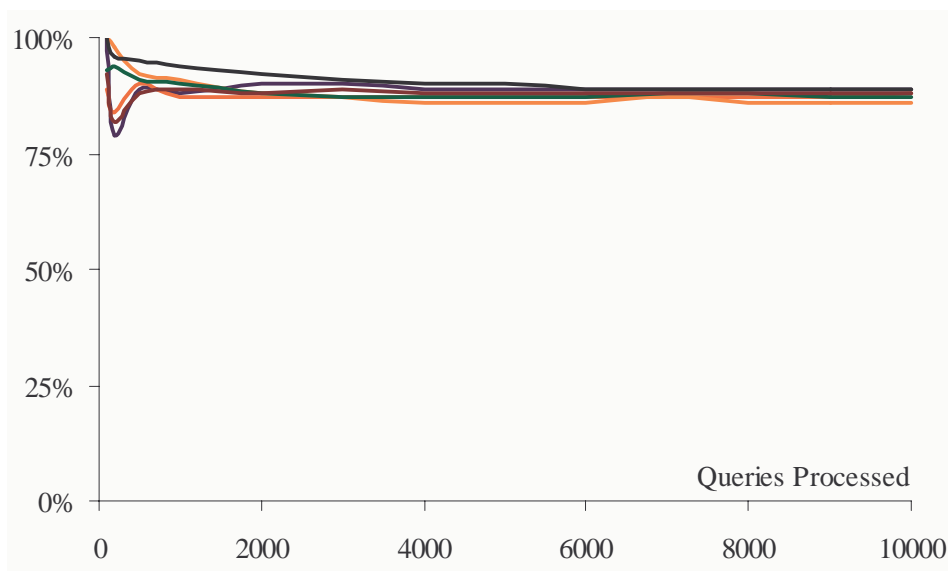
Figure 85 – OptRate for Bounded, 6 random query orders; Query 21
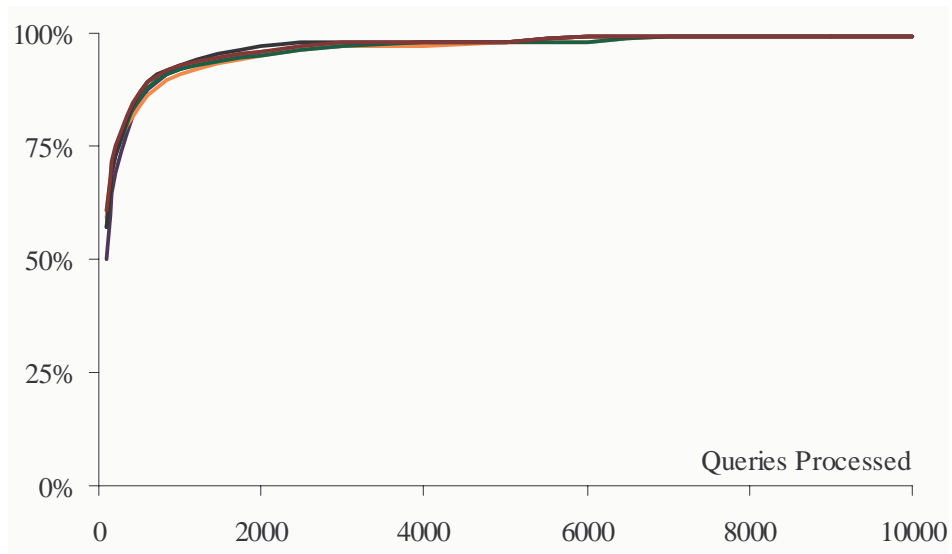
Figure 86 – HitRate for Ellipse, 6 random query orders; Query 21
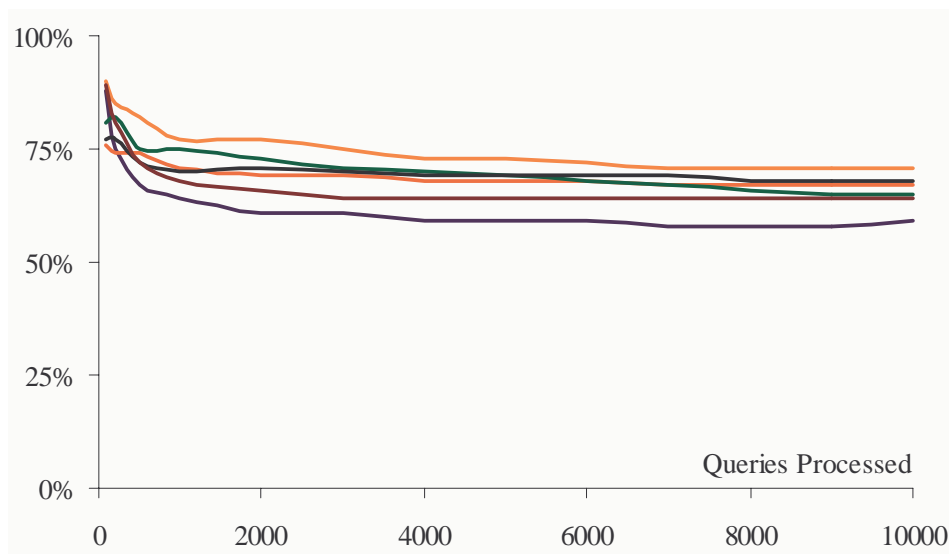
Figure 87 – OptRate for Ellipse, 6 random query orders; Query 21

5.5.7. Vary Number of Dimensions

In all the experiments so far, the parameter space was 2-dimensional. The next experiment varies the number of dimensions, from 1 to 4. Query 8 is used (with extra parametric selections as needed) because it was the one with the largest number of plans and thus, more likely to suffer from the "curse of dimensionality": an exponential growth of complexity with a linear increase in the number of dimensions. The query was then run for 10,000 random values for Bounded (M=1.1, A=0) and Ellipse (Δ=0.95). The results, showed in Figures 88-91 are summarized in Table 10 below.

Table 10 – Variation of number of dimensions

	OptRate				HitRate			
	1-D	2-D	3-D	4-D	1-D	2-D	3-D	4-D
Bounded	77%	65%	65%	56%	100%	94%	88%	49%
Ellipse	99%	74%	62%	58%	100%	98%	96%	88%

The results clearly indicate that as the number of dimensions in the parameter space increases, the lower the OptRate and HitRate. Some of the reasons that contribute to this effect are:

- Given a point cp centered in the middle of the parameter space, the percentage of space ‹ cp (or › cp) decreases exponentially with the number of dimensions (affects Bounded).

- The number of unique optimal plans increases exponentially (affects Ellipse).

Even though the number of plans and number of points increase exponentially for both Bounded and Ellipse, they increase slower for Ellipse; see Figures 92 and 93.
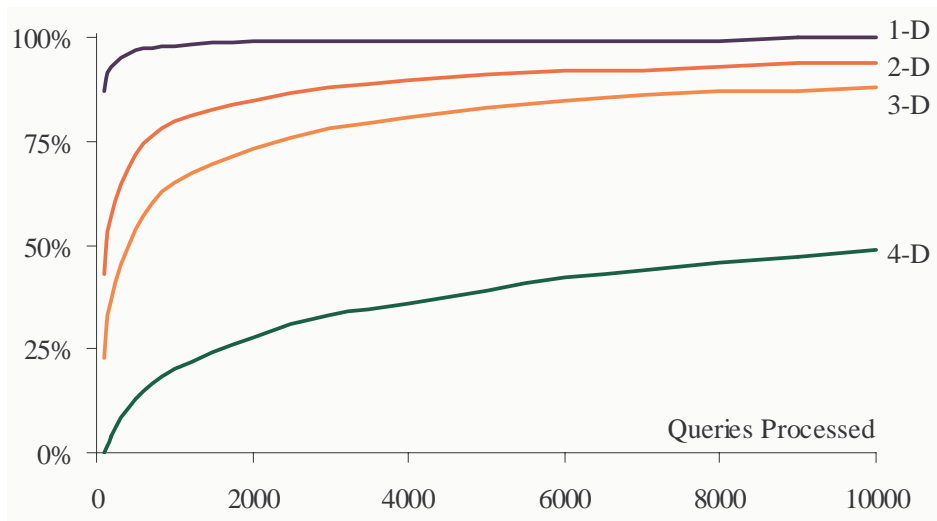
Figure 88 – Vary number of dimensions; HitRate for Bounded; Query 8
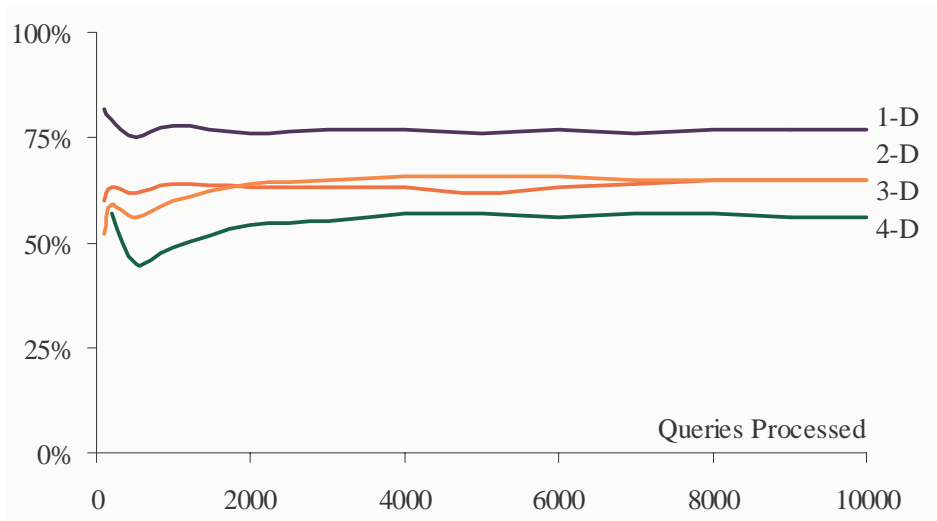
Figure 89 – Vary number of dimensions; OptRate for Bounded; Query 8
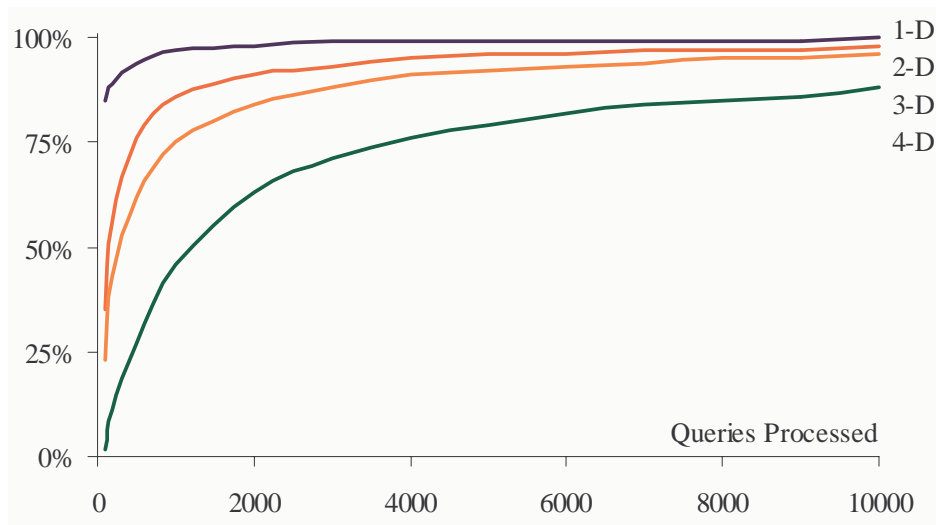
Figure 90 – Vary number of dimensions; HitRate for Ellipse; Query 8
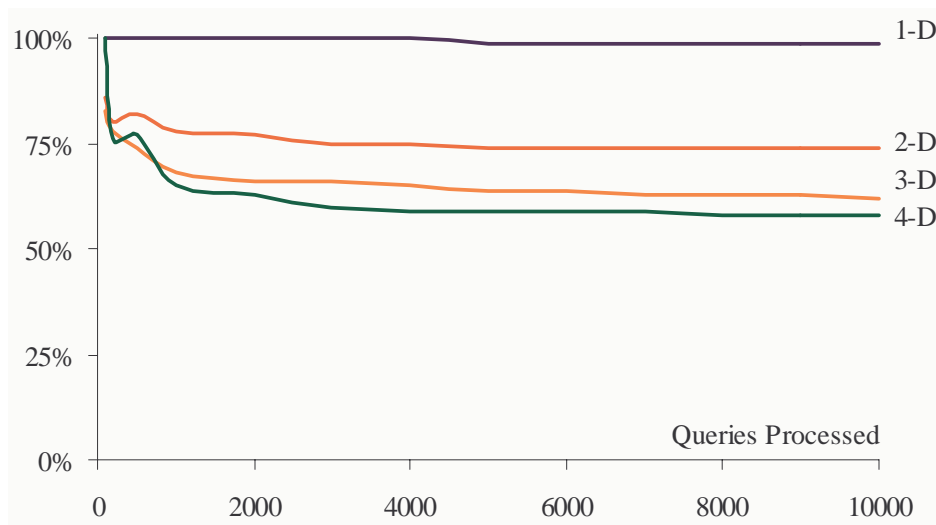
Figure 91 – Vary number of dimensions; OptRate for Ellipse; Query 8
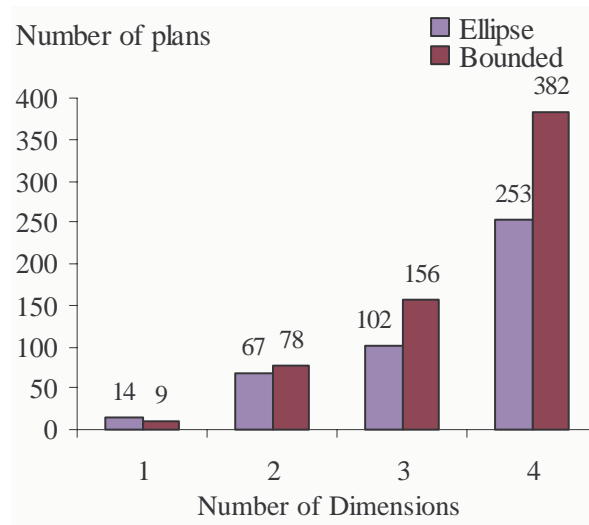
Figure 92 – Number of plans with varying number of dimensions; Query 8
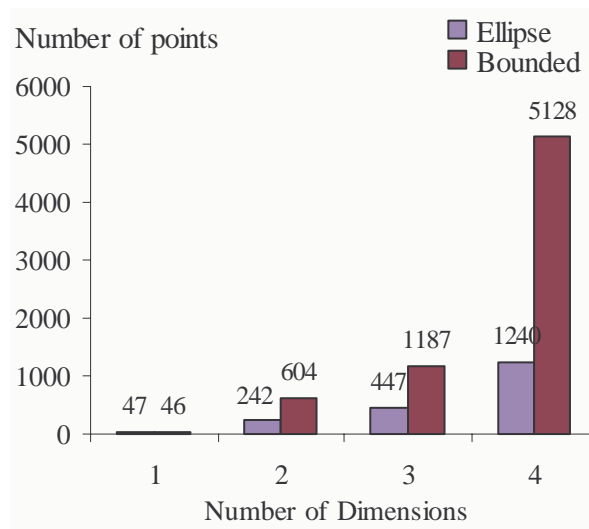
Figure 93 – Number of points with varying number of dimensions; Query 8

5.6. Related Work

Parametric query optimization was first mentioned by Graefe [40] and Lohman [61]. This pioneering early work also proposed *dynamic query plans* and a new meta-operator, the *choose-plan* [40]. Dynamic query plans include more than one physical plan choice. The plan to use is determined at run-time by the choose-plan operator after it costs the multiple alternatives given the now known parameter values. How to enumerate dynamic query plans was proposed only later [26] with the concept of *incomparability of costs*: in the presence of unbound parameters at optimization-time, plan costs are represented as intervals; if intervals of alternative plans overlap, none is pruned. At run-time, when parameters are bound to values, the choose-plan selects the right plan to use. This approach may enumerate a very large number of plans, as shown by [77], and all those plans may have to be re-cost at run-time by the choose-plan operator.

Ioannidis et al [52] coined the term Parametric Query Optimization and proposed using randomized algorithms to optimize in parallel the parametric query for all possible values of unknown variables. This approach is unfeasible for continuous parameters, gives no guarantees on finding the optimal plan for a query, and places no bounds on the optimality of the plans produced.

Ganguly [35] uses a geometric approach to solve the PQO problem for one and two parameters under the assumption that cost functions are linear and that regions of optimality of plans are convex. Ganguly also solved PQO for restricted forms of non-linear, one-parameter, cost functions. Prasad [71] extended the geometric approach to solve PQO for ternary linear cost functions and binary non-linear functions. Hulgeri and Sudarshan [48] propose a solution to PQO that handles piecewise linear cost functions for an arbitrarily number of parameters but requires substantial changes to the query optimizer. AniPQO [49] is a recent technique that approximates the solution to PQO for non-linear functions and for an arbitrary number of parameters. AniPQO approximates optimality regions to n-dimensional convex polytopes and finds its solution to PQO by calling the optimizer multiple times and evaluating plan costs up to

thousands of times. Unlike AniPQO, PPQO never calls the optimizer or costs plans more often than what a traditional non-PQO approach would.

5.7. Conclusions

Progressive Parametric Query Optimization (PPQO) improves the performance of processing parameterized queries by combining the benefits of competing strategies. Like Optimize-Always and PQO, most of the times, PPQO selects plans that are estimated to be optimal or near-optimal. Like Optimize-Once, PPQO is able to avoid optimization calls in up to 99% of the queries. Like other PQO proposals, PPQO discovers most optimal plans and approximated optimality areas. In addition, unlike PQO, PPQO does not perform extra optimizer calls or extra plan-cost evaluation calls. At execution time, PPQO can select which plan to execute by using only the input cost parameters; there is no need to re-cost any plan. Finally, recent work [76] shows that assumptions commonly held by PQO (plan convexity, plan uniqueness, and plan homogeneity) do not hold. These discoveries do not affect PPQO. The only assumption taken by PPQO is the monotonicity of plan costs.

PPQO is also amenable to be implemented in a complex commercial database system as it requires minimal changes to the optimization or execution processes.

PPQO was evaluated in a variety of settings, with queries joining up to eight tables, with multiple sub-queries, up to four parameters, and in plan spaces with close to 400 different optimal plans. PPQO yielded good results in all scenarios except for the Bounded algorithm in complex queries using a 4-D parameter space. However, even in this challenging scenario, Ellipse was on average executing plans just 3% more costly than the optimal, while avoiding 87% of all optimization calls.

5.8. Proofs of Theorem 5.1 and Theorem 5.2

This section uses the three Lemmas below to prove Theorem 5.1 and Theorem 5.2. Lemma 1 states that if the Monotonic Assumption holds for every plan considered, than the cost of the optimal plan at any point (regardless of what the optimal plan is at any single point) also increases monotonically with the parameters. This result is used later to bound the cost of some plan p in points where plan p was never executed.

Lemma 1: If $cp_1 \triangleleft cp_2$, $cost_1 = p_1(cp_1) = Op(cp_1)$, and $cost_2 = p_2(cp_2) = Op(cp_2)$ then $cost_1 \leq cost_2$. (1)

Proof:

There are only two cases: either p_2 is optimal at cp_1 or p_2 is not optimal at cp_1.

- If p_2 is optimal at cp_1, then $cost_1 = p_2(cp_1)$. 　　　　　　　　　　　　　　　　(2)

- If p_2 is not optimal at cp_1, then $cost_1 < p_2(cp_1)$. 　　　　　　　　　　　　　　(3)

- By (2) and (3), $cost_1 \leq p_2(cp_1)$. 　　　　　　　　　　　　　　　　　　　　　(4)

- The Monotonic Assumption and $cp_1 \triangleleft cp_2$ imply: $p_2(cp_1) \leq p_2(cp_2) = cost_2$. 　　　(5)

- By (4), (5), $cost_1 \leq cost_2$. 　　　　　　　　　　　　　　　　　　　　　　■

Lemma 2 and Lemma 3 together state that if $M \geq 1$, $cost_z \in [cost_x, cost_x*M+A]$ and $cost_x \leq cost_y \leq cost_z$, then both $cost_z \in [cost_y, cost_y*M+A]$ and $cost_y \in [cost_x, cost_x*M+A]$.

Lemma 2: If $cost_z \in [cost_x, cost_x*M+A]$ and $cost_x \leq cost_y \leq cost_z$, then $cost_y \in [cost_x, cost_x*M+A]$. 　(6)

Proof:

- $cost_z \in [cost_x, cost_x*M+A] \land cost_x \leq cost_y \leq cost_z \Rightarrow$

  $\Rightarrow cost_x \leq cost_z \leq cost_x*M+A \land cost_x \leq cost_y \leq cost_z \Rightarrow cost_x \leq cost_y \leq cost_x*M+A$

  $\Rightarrow cost_y \in [cost_x, cost_x*M+A]$ 　　　　　　　　　　　　　　　　■

Lemma 3: If $M \geq 1$, $cost_z \in [cost_x, cost_x*M+A]$, and $cost_x \leq cost_y \leq cost_z$, then $cost_z \in [cost_y, cost_y*M+A]$.　　(7)

Proof:

- Since $M \geq 1$, it follows that $cost_x \leq cost_y \Rightarrow cost_x*M+A \leq cost_y*M+A$　　　　　(8)

- By $cost_z \in [cost_x, cost_x*M+A]$ and (8) it follows that $cost_z \leq cost_x*M+A \leq cost_y*M+A$　　(9)

- By $cost_x \leq cost_y \leq cost_z$ and (9) it follows that $cost_y \leq cost_z \leq cost_y*M+A$　　　(10)

- (10) is equivalent to $cost_z \in [cost_y, cost_y*M+A]$.　　　　　　■

Theorem 5.1: If $\exists t_i = (cp_i, plan_i, cost_i)$, $\exists t_j = (cp_j, plan_j, cost_j)$, such that plan $plan_i$ ($plan_j$) is an optimal plan at cp_i (cp_j) with cost $cost_i$ ($cost_j$), $cp_i \triangleleft cp \triangleleft cp_j$ and $cost_j \in [cost_i, cost_i*M+A]$, then $plan_j(cp) \in [Op(cp), Op(cp)*M+A]$.

Proof:

- By Lemma 1 and $cp_i \triangleleft cp \triangleleft cp_j$ it follows that $cost_i \leq Op(cp) \leq cost_j$.　　　(11)

- By (11), Lemma 3, and $cost_j \in [cost_i, cost_i*M+A] \Rightarrow cost_j \in [Op(cp), Op(cp)*M+A]$.　　■

Definition of ◄ (below) operator and ► (above) operator [Reprint from page 135]: Given a list, T, of k triples $(cp_i, p_i, cost_i)$ ordered by $cost_i$, with $i=0...k-1$, where cp_i is a CostPoint and $cost_i$ represents the cost of executing the optimal plan p_i at cp_i and given cp, another CostPoint we define the following two operations:

- $T_{\triangleleft cp}$ is the list of triples $(cp_i, p_i, cost_i)$ from T, ordered by $cost_i$, such that $cp_i \triangleleft cp$.

- $T_{\triangleright cp}$ is the list of triples $(cp_i, p_i, cost_i)$ from T ordered by $cost_i$, such that $cp_i \triangleright cp$.

Note that, by definition, $cp_b \triangleleft cp \triangleleft cp_a$, $\forall cp_b : t_b = (cp_b, p_b, cost_b) \in T_{\triangleleft cp}$, $\forall cp_a : t_a = (cp_a, p_a, cost_{ba}) \in T_{\triangleright cp}$.

Theorem 5.2: If $\exists cp_b : t_b = (cp_b,\ p_b,\ cost_b)$, $t_b \in T_{\triangleleft}cp$, $\exists cp_a : t_a = (cp_a,\ p_a,\ cost_a)$, $t_a \in T_{\triangleright}cp$, such that $cost_a \in [cost_b,\ cost_b{*}M{+}A]$, then $cost_{first} \in [cost_{last},\ cost_{last}{*}M{+}A]$, where $cost_{first}$ is the cost of the first triple in $T_{\triangleright}cp$ and $cost_{last}$ is the cost of the last triple in $T_{\triangleleft}cp$.

Proof:

- By the definitions of $T_{\triangleleft}cp$ and $T_{\triangleright}cp$, and Lemma 1: $cost_b \leq cost_{last} \leq Op(cp) \leq cost_{first} \leq cost_a$.   (12)

- By $cost_a \in [cost_b,\ cost_b{*}M{+}A]$, (12) and Lemma 3, it follows that $cost_a \in [cost_{last},\ cost_{last}{*}M{+}A]$ (13)

- Finally, by (12), (13), and Lemma 2, it follows that $cost_{first} \in [cost_{last},\ cost_{last}{*}M{+}A]$.   ■

5.9. Queries

This section contains the 5 queries used in the experiments of Section 5.5. The queries, originally from the TPC-H benchmark, were altered with additional selection predicates to allow the exploration of a 2-D parameter space. The queries in this format were first used by Reddy and Haritsa [76] and were shared to us by Haritsa. Query 8 was subsequently altered with additional selection predicates to allow the exploration of 3-D and 4-D parameter spaces as well.

The additional selection predicates are of the form "column $\leq$ \$Vx\$", with x taking a value between 0 and 3. In the experiments, the values for the \$Vx\$ parameters were randomly generated within the domain of column. For each column subject to these additional selection predicates, the values in the relations were changed to force a uniform distribution.

5.9.1. Query 7

```
select
    supp_nation,
    cust_nation,
    l_year,
    sum(volume) as revenue
from
    (
        select
            n1.n_name as supp_nation,
            n2.n_name as cust_nation,
            YEAR (l_shipdate) as l_year,
            l_extendedprice * (1 - l_discount) as volume
        from
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2
        where
            s_suppkey = l_suppkey
            and o_orderkey = l_orderkey
            and c_custkey = o_custkey
            and s_nationkey = n1.n_nationkey
            and c_nationkey = n2.n_nationkey
            and (
                (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
                or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
            )
            and l_shipdate between '1995-01-01' and '1996-12-31'
            and orders.o_totalprice <= $V0$
            and customer.c_acctbal <= $V1$
    ) as shipping
group by
    supp_nation,
    cust_nation,
    l_year
order by
    supp_nation,
    cust_nation,
    l_year;
```

5.9.2. Query 8

```
select
    o_year,
    sum(case
        when nation = 'BRAZIL' then volume
        else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            YEAR(o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part,
            supplier,
            lineitem2,
            orders,
            customer,
            nation n1,
            nation n2,
            region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and o_orderdate between '1995-01-01' and '1996-12-31'
            and p_type = 'ECONOMY ANODIZED STEEL'
            and lineitem2.l_extendedprice <= $V0$
            and supplier.s_acctbal <= $V1$
            and orders.o_totalprice <= $V2$         // Used only in 3-D and 4-D spaces
            and customer.c_acctbal <= $V3$          // Used only in 4-D spaces
    ) as all_nations
group by o_year
order by o_year;
```

5.9.3. Query 9

```
select
    n_name,
    o_year,
    sum(amount) as sum_profit
from
    (
        select
            n_name,
            YEAR(o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
        from
            part,
            supplier,
            lineitem,
            partsupp,
            orders,
            nation
        where
            s_suppkey = l_suppkey
            and ps_suppkey = l_suppkey
            and ps_partkey = l_partkey
            and p_partkey = l_partkey
            and o_orderkey = l_orderkey
            and s_nationkey = n_nationkey
            and p_name like '%green%'
            and supplier.s_acctbal <= $V0$
            and partsupp.ps_supplycost <= $V1$
    ) as profit
group by
    n_name,
    o_year
order by
    n_name,
    o_year desc;
```

5.9.4. Query 18

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem2
where
    o_orderkey in (
        select
            l_orderkey
        from
            lineitem2
        where
            lineitem2.l_extendedprice <= $V0$
        group by
            l_orderkey
        having
            sum(l_quantity) > 300
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
    and customer.c_acctbal <= $V1$
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate
```

5.9.5. Query 21

```
select
    s_name,
    count(*) as numwait
from
    supplier,
    lineitem2,
    orders,
    nation
where
    s_suppkey = lineitem2.l_suppkey
    and o_orderkey = lineitem2.l_orderkey
    and o_orderstatus = 'F'
    and exists (
        select
            *
        from
            lineitem2 l2
        where
            l2.l_orderkey = lineitem2.l_orderkey
            and l2.l_suppkey <> lineitem2.l_suppkey
    )
    and not exists (
        select
            *
        from
            lineitem2 l3
        where
            l3.l_orderkey = lineitem2.l_orderkey
            and l3.l_suppkey <> lineitem2.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and supplier.s_acctbal <= $V0$
    and lineitem2.l_extendedprice <= $V1$
    and n_name = 'SAUDI ARABIA'
group by
    s_name
order by
    numwait desc,
    s_name;
```

CHAPTER 6

CONCLUSIONS

Most commercial DBMSs use a sequential plan-first execute-next query processing strategy: the optimizer enumerates and costs plans and passes the plan with the lowest estimated cost to the executor; then the executor processes the plan to completion. The effectiveness of this non-adaptive approach relies heavily on the quality of the plans produced by the optimizer. The quality of those plans depends on the information available to the optimizer: if the statistics used to cost plans are missing or incorrect, the optimizer is likely to select a sub-optimal plan. Several hardware and software trends are making this hard problem harder. For example, the optimization space is increasing exponentially because there are more operators to considerer, larger datasets to manage, and more complex queries to optimize. Thus, the optimizer is increasingly more likely of selecting a sub-optimal plan.

6.1. Contributions

Given the trends above, instead of focusing on providing more information to the optimizer, we proposed a series of query processing techniques that correct optimizer mistakes or execute robust plans (plans with good performance that are insensitive to optimizer mistakes).

Content-Based Routing, described in   , is a strategy that assigns different execution plans for subsets of data with different statistical properties. The different subsets of data and their respective plans are adaptively determined at run-time with no intervention from the optimizer. Our most important contribution was to show that content-based learning and routing can be simultaneously inexpensive and adaptive while still achieving significant performance improvements.

Rio, described in Chapter 3, is a second-generation AQP system that improves on previous AQP proposals in several ways. Previous AQP systems, although able to correct some optimizer mistakes, still suffered from the optimizer mistakes in an indirect way. By using a traditional optimizer, those systems were more likely to start execution with a sub-optimal plan. Although the sub-optimal plan could frequently be detected and replaced by an optimal plan, the process could be inefficient. The sub-optimal plan was not quickly detected, and when it was, query processing work could have to be thrown away and repeated. In addition, previous AQP approaches would sometimes only find an optimal plan after multiple re-optimization steps, each with its own potential inefficiencies. With Rio, we extensively re-engineered query optimization and query execution to make the system as insensitive as possible to optimizer mistakes. The optimization module was changed to give priority to robust plans, i.e., plans insensitive to incorrect estimates. This was partially done by assigning levels of uncertainty to estimated statistics based on the way they are estimated. This, in turn, revealed which plans were robust and which plans were risky. In addition, new switch operators were inserted in the plan tree to minimize the work lost if a change of plan was needed. Finally, query execution was changed to allow faster detection of sub-optimal plans and faster convergence to the optimal plan.

SHARP, described in Chapter 4, is an AQP strategy fundamentally different from Rio. Rio required very substantial changes to the query optimization and query execution modules including small modifications in all operators. Rio also allowed for multiple inter-leavings of optimization and execution. By contrast, the proposed new SHARP operator encapsulates almost all AQP changes needed; the remaining query processing engine is largely unaffected. SHARP also explores a new trade-off: instead of executing arbitrarily query plans, and being able to preempt execution and re-invoke optimization, SHARP adopts a two-step adaptive approach. First, run-time late-binding decisions determine the driving relation. Second, tuple routing continuously potentially changes the join order–probing sequence of tuples from the build sources–within the orders available after the driving relation was fixed. This two-step adaptive process yields two benefits: i) it requires less memory than previous adaptive operators and ii)

simplifies the design of second-stage processing. In addition, the second-stage was designed to be insensitive to the join order determined by the optimizer, thus, insensitive to optimizer mistakes, i.e., the second-stage executed efficient plans regardless of what the optimizer specifies. On the other hand, SHARP is not a general AQP solution. SHARP is a multi-join, relational operator that joins three or more relations of a star-join. Nevertheless, SHARP shows that it is possible to implement adaptive query processing strategies in a DBMS with minimal changes to the system but with great positive impact on performance.

Progressive Parametric Query Optimization, described in Chapter 5, is an AQP strategy designed to deal with the lack of information about values in parameterized queries at optimization time. The trivial solutions are to ignore the problem (Optimize-Once) or to defer optimization until the values are known (Optimize-Always). Optimize-Once returns a plan which may be arbitrarily sub-optimal. Optimize-Always unnecessarily consumes too many resources in optimization calls. A more elaborated approach, Parametric Query Optimization, may also produce too many optimization calls. PQO may also be not applicable because its requirements (plan convexity, plan uniqueness, and plan homogeneity) do not hold in real systems. Our proposal, PPQO, improves the performance of processing parameterized queries by combining the benefits of competing strategies: most of the times, PPQO selects plans that are optimal or near-optimal, while avoiding optimization calls in up to 99% of the queries, and being indifferent to plan convexity, plan uniqueness, and plan homogeneity.

6.2. Final Words

We do not claim that any one of the techniques presented in this thesis completely solves the current problems faced by query optimizers. On the contrary, we think that the complexity of the optimization problem together with the unavailability of information to cost the alternatives will require many different complementary solutions, some of which are adaptive query processing strategies, some not. We think that obtaining information not stored in the catalog and providing it just-in-time to the optimizer [2, 6, 73,

80], learning from past queries and mistakes [83], and using techniques from data stream systems are other interesting avenues to address the issues discussed in this thesis. In future work we plan to integrate some of those other approaches with the AQP techniques proposed here.

BIBLIOGRAPHY

[1] S. Acharya et al. Join Synopses for Approximate Query Answering. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1999)*, June 1999.

[2] M. J. Anderson, R. L. Cole, W. S. Davidson, W. D. Lee, P. B. Passe, G. R. Ricard, L. W. Youngren. Index key range estimator. U. S. Patent 4,774,657, September 27, 1988.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull*, 26(1): 19-26 (2003).

[4] R. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computer Systems*, 21(1):36–86, 2003.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2000)*, May 2000.

[6] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2005)*, June 2005.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and Issues in Data Stream Systems. *PODS* 2002: 1-16.

[8] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. *In Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004.

[9] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004. Demonstration proposal.

[10] S. Babu and P. Bizarro. Adaptive Query Processing in the Looking Glass. In *Proc. of Second Biennial Conf. on Innovative Data Systems Research (CIDR'2005)*, January 2005.

[11] S. Babu, P. Bizarro, and D. J. DeWitt. Proactive Re-optimization. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2005)*, June 2005.

[12] S. Babu. Adaptive Query Processing in Data Stream Management Systems. *Ph.D. Thesis*, Stanford University, September 2005. Available at http://dbpubs.stanford.edu:8090/pub/2005-24. Accessed July 2006.

[13] J. Beale. Snort 2.1 Intrusion Detection. Syngress Publishing, 2004.

[14] D. Bitton, D. J. DeWitt, and C. Turbyfil. Benchmarking Database Systems: A Systematic Approach. In *Proceedings of the 1983 Very Large Database Conference (VLDB'1983)*, October 1983.

[15] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-Based Routing: Different Plans for Different Data. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2005)*, September 2005.

[16] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13(7): 422-426 (1970).

[17] L. Bouganim, F. Fabret, C. Mohan, P. Valduriez. A Dynamic Query Processing Architecture for Data Integration Systems. *IEEE Data Engineering Bulletin*, 23(2): 42-48 (2000).

[18] H. Brönnimann et al. Efficient data reduction with EASE. In *Proc. of the Ninth ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD'2003)*, August 2003.

[19] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2002)*, June 2002.

[20] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2002)*, August 2002

[21] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR'2003)*, January 2003.

[22] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1999)*, June 1999.

[23] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004.

[24] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. ACM Trans. on Database Systems, 9(2): 163-186, 1984.

[25] F. Chu, J. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. of the 1999 ACM Sympon Principles of Database Systems (PODS '1999)*, June 1999.

[26] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1994)*, June 1994.

[27] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2003)*, June 2003.

[28] B. Dageville and M. Zait. SQL memory management in Oracle9i. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2002)*, August 2002.

[29] A. Deshpande. An initial study of overheads of eddies. SIGMOD Record 33(1): 44-49 (2004)

[30] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2001)*, May 2001.

[31] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-Driven Data Acquisition in Sensor Networks. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2004)*, September 2004.

[32] A. Deshpande and J. M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2004)*, September 2004.

[33] A. Deshpande, C. Guestrin, S. Madden, W. Hong. Exploiting Correlated Attributes in Acquisitional Query Processing. In *Proc. of the Intl. Conf. on Data Engineering (ICDE '2005)*, April 2005.

[34] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the Sixth ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD'2000)*, August 2000.

[35] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'1998)*, August 1998.

[36] L. Getoor, B. Taskar, D. Koller. Selectivity Estimation using Probabilistic Models. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2001)*, May 2001.

[37] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2001)*, May 2001.

[38] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. on Database Systems*, 27(3): 261-298, 2002.

[39] L. Golab and T. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[40] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1989)*, June 1989.

[41] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computer Surveys* 25(2): 73-170. 1993.

[42] J. Gray. A "Measure of Transaction Processing" 20 Years Later. *Microsoft Research Technical Reprt TR-2005-57* , April 2005. Available at http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2005-57. Accessed on July 2006.

[43] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 33rd Annual ACM Symp on Theory of Computing* , July 2001.

[44] P. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1999)*, June 1999.

[45] J. M. Hellerstein, P. Haas and H. J. Wang. Online Aggregation. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1997)*, June 1997.

[46] J. M. Hellerstein, M. J. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[47] J. Hellerstein, R. Avnur, and V. Raman. Informix Under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery Journal*, 4(4), October 2000.

[48] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2002)*, August 2002.

[49] A. Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2003)*, June 2003.

[50] I. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004.

[51] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1991)*, May 1991.

[52] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'1992)*, August 1992.

[53] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1999)*, June 1999.

[54] Z. Ives, A. Levy, et al. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, June 2000.

[55] Z. Ives. Efficient Query Processing for Data Integration. *Ph.D. thesis*, University of Washington–Seattle, WA, USA, August 2002.

[56] Z. Ives, A. Halevy, D. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004.

[57] C. Jermaine, A. Dobra, A. Pol, S. Joshi. Online Estimation For Subset-Based SQL Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2005)*, August 2005.

[58] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1998)*, June 1998.

[59] M. Kitsuregawa, M. Nakayama, and M. Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'1989)*, August 1989.

[60] R. Lawrence. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2005)*, September 2005.

[61] G. M. Lohman. Is Query Optimization a 'Solved' Problem? Workshop on Database Query Optimization. *Oregon Graduate Center CompSci. Tech. Rep89-005* , May 1989.

[62] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2003)*, June 2003.

[63] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, H. Pirahesh. Robust Query Processing through Progressive Optimization. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'2004)*, June 2004.

[64] Microsoft Corporation. Plan Forcing Scenario: Create a Plan Guide That Uses a USE PLAN Query Hint. *SQL Server 2005 Books Online.* Available at http://msdn2.microsoft.com/en-us/library/ms190454.aspx. Accessed July 2006.

[65] Microsoft Corporation. Logical and Physical Operators Reference. *SQL Server 2005 Books Online*. Available at http://msdn2.microsoft.com/en-us/library/ms345824.aspx. Accessed July 2006.

[66] T. Mitchell. Machine Learning. McGraw-Hill, 1997.

[67] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of Second Biennial Conf. on Innovative Data Systems Research (CIDR'2003)*, January 2003.

[68] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'1998)*, September 1998.

[69] H. Paques, L. Liu, and C. Pu. Distributed Query Adaptation and Its Trade-offs. In *Proc. of ACM Symp on Apied Computing* , March 2003.

[70] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1996)*, June 1996.

[71] V. G. V. Prasad. Parametric Query Optimization: A Geometric Approach. *Master Thesis.* IIT, Kampur, India, 1999.

[72] R. Ramakrishnan, and J. Gehrke. *Database Management Systems*, 3rd edition. McGraw-Hill Science/Engineering/Math, August 14, 2002.

[73] R. Ramamurthy and D. J. DeWitt. Buffer-pool Aware Query Optimization. In *Proc. of Second Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2005.

[74] V. Raman, A. Deshpande, J. M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE2003)*, April 2003.

[75] R. Ramey. 2005 Performance/Price Sort and PennySort. Available at http://research.micro-soft.com/barc/SortBenchmark/2005_PostMansSort.pdf. Accessed July 2006.

[76] N. Reddy and J. R. Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2005)*, September 2005.

[77] S. V. U. Maheswara Rao. Parametric Query Optimization: A Non-Geometric Approach. *Master Thesis*, IIT, Kampur, India, 1999.

[78] M. A. Shah. Flux: A Mechnism for Building Robust, Scalable Dataflows. *Ph.D. Thesis*, University of California–Berkeley, CA, USA, 2004.

[79] P. Selinger, M. M. Astrahan, D.D. Chamberlin, R. A. Lorie, T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1979)*, May 1979.

[80] K. D. Seppi, J. W. Barnes, and C. N. Morris. A Bayesian approach to database query optimization. *ORSA Journal on Computing*, 5(4):410–419, 1993.

[81] P. Seshadri. Predator: A Resource for Database Research. *SIGMOD Record,* 27(1): 16-20, 1998.

[82] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org.

[83] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO -DB2's LEarning Optimizer. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2001)*, September 2001.

[84] F. Tian, D. J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2003)*, September 2003.

[85] Transaction Processing Performance Council. *The TPC-H Benchmark*. Available at http://www.tpc.org/tpch/. Accessed March 2006.

[86] P. Tucker et al. Exploiting Punctuation Semantics in Continuous Data Streams. *Transactions on Knowledge and Data Engineering*, 15(3): 555-568, 2003.

[87] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD'1998)*, June 1998.

[88] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2): 27-33 (2000).

[89] S. Viglas. Novel Query Optimization and Evaluation Techniques, *Ph.D. Thesis*, Department of Computer Sciences, University of Wisconsin-Madison, WI, USA, June 2003.

[90] S. Viglas, J. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB'2003)*, September 2003.

[91] A. N. Wilschut and P. M. G. Apers. Pipelining in Query Execution. In *Proc. of the First Intl. Conf. on Parallel and Distributed Information Systems (PDIS 1991)*, December 1991.