Novel Data Summaries for Join Query Optimization

Walter Cai

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

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Dan Suciu, Chair

Magdalena Balazinska

Paul Beame

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

Novel Data Summaries for Join Query Optimization

Walter Cai

Chair of the Supervisory Committee:
Professor Dan Suciu
The Paul G. Allen School for Computer Science and Engineering

As the demand for data intensive pipelines has grown and the diversity of settings has expanded, generalizability of database management systems has suffered. The execution of join queries, especially multi-join queries, remains one of the field's greatest challenges. We propose novel approaches to such queries using data sketching.

Traditional cost-based query optimizers have existed for decades, becoming the de facto method for designing performant analytical systems. Nevertheless, these systems are still hampered by the cost estimation stage. In particular, modern systems fall back on strong assumptions about the underlying data when confronted with multijoin queries. In lieu of chasing perfect estimates over multi table queries, we propose the application of theoretically guaranteed cardinality upper bounds. These have the benefit that they force the optimizer to act conservatively and deliver fewer high risk plans to the executor. We demonstrate that the use of bounds leads to fewer disastrous plans than traditional cost estimation techniques but is still on par with 'easy' queries where traditional query optimization techniques already perform well. We also preview how this technique may be generalized to large scale distributed data scenarios.

Streaming query optimization introduces fresh challenges on top of post hoc analytic pipelines. While queries are often semantically simpler, the introduction of temporal semantics, required immediacy of output results, and less reliable hardware puts a strain on the execution layer. In particular, the natural method of combining separate data streams –a temporal join– places the onus of adapting to changing stream characteristics on an integrated optimizer-executor. We propose a novel state management algorithm applicable to the threshold-function-over-joins-scenario; a common setting in streaming data management. We demonstrate significant state savings and prove that our method is optimal while still guaranteeing no false positives; no threshold function triggers will be lost.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

While a doctorate is technically an individual degree, it is abundantly clear that this dissertation is the product of a long line of people who have helped and supported me on this path. To my advisors Dan Suciu and Magdalena Balazinska, I extend the greatest gratitude. Your expertise and enthusiasm are undoubtably the driving force behind our research and none of this would be possible if it weren't for your astounding patience. I would also like to acknowledge other professors/advisors from the University of Washington and elsewhere who have been instrumental in my development. Ravi Ramakrishna, Camil Muscalu, Keith Dennis, and Bob Strichartz for guiding me through undergrad and lending me just a hair of mathematical maturity. Bobby Kleinberg and Steve Wright for showing me the bridge between mathematics and computer science; AnHai Doan and Jeff Naughton for mentoring me when I first entered the database field; Philip Bernstein and Wentao Wu for broadening my view to the streaming side of databases; Daniel Ting for introducing me to data sketching; and finally Paul Beame and Aleksandr Aravkin for being willing to sit on my committee on short notice and still contribute insightful questions.

I was equally fortunate to be surrounded by an unparalleled collection of fellow graduate students and postdocs both in the database group and outside. I hesitate to list them all but I'm going to do my best. I will surely miss some names but please don't feel slighted; I'm very tired. Adam Elder, Adel Ardalan, Babak Salimi, Batya Kenig, Brandon Haynes, Cong Yan, Dominik Moritz Dong He, Eunice Jun, Guna Prasaad, Iain Carmichael, Jason Hoffman Jenny Ortiz, Jingjing Wang, Jonathan Leang, Krishna Pillutla, Kyle Deeds, Laurel Orr, Maaz Ahmad, Maureen Daum, Max Schleich, Meghan Cowan, Moe Kayali, Morelle Arian, Parmita Mehta, Remy Wang, Robert Baraldi, Ryan Maas, Shana Hutchinson, Shrainik Jain, Shumo Chu, Srini Iyer, Tomer Kaftan, and Trevor Perrier.

There are of course others who were instrumental for me and I close by expressing my gratitude to some of them. To the many paper reviewers who have assessed my submissions, I'm so sorry, and also thank you. To Elise deGoede Dorough, without whom the Allen school could not function. To Iain, we sold out together so hopefully this pans out. To my parents and Elisabeth, for their unending postitivity, whether or not I was in the mood. To Luna (a cat), she's new but she's pretty cool. And finally to Anna, for her patience, strength, and kindness; I promise I'll adopt more regular sleeping patterns.

DEDICATION

To Mama, Baba, Elisabeth, and Anna

Chapter 1

INTRODUCTION

As more and more industries switch on to the idea of relying on data to drive their most important decisions, data analytics has emerged as one of the most demanding and diverse fields of computer science. These businesses require tools to address their ever growing, ever more complex workloads. Database management systems (DBMS's) have long been the tool of choice for analyzing data, offering an elegant balance of expressivity, efficiency, and general ease of use. Often one of the central requirements for a DBMS is to allow users to combine different datasets. This combination creates richer datasets ready to be analyzed by the user. We trust the DBMS to make smart decisions on how best to perform this task. In particular, the user relies on the *optimizer* component of the DBMS to decide what the precise order of operations should be to accomplish these combinations and to dictate what information to store along the way. I.e., we trust the optimizer to come up with how best to compute this efficiently. This is a significant responsibility: one plan may take days, while another may take seconds. These challenges are often exacerbated by the specific distributions, ordering, and available indexes of one dataset versus another or even the raw form factors of the data. These challenges are well known and well studied, but despite decades of research choosing and executing a good plan remains a fundamental bottleneck to modern data analytics pipelines. As the magnitude and complexity of data analytics tasks grows, the need to address this bottleneck is of utmost importance.

One of the most difficult operations for database systems to optimize and execute is a *join*. A join operation combines two datasets together and is represented mathematically with a $\bowtie$ symbol. Generally, these datasets are tabular; lists of tuples where each tuple shares some common structure and describes some instance of data. We refer to such datasets as

name	id
walter	0
dan	1
magda	2
jeff bezos	3
archimedes	4
jay inslee	5

(a) Employee (E)

person_id	boss_id
0	1
0	2
1	5
2	5
3	0
4	0
5	3

(b) Reports-To (RT)

person_name	boss_name
walter	dan
walter	magda
dan	jay inslee
magda	jay inslee
jeff bezos	walter
archimedes	walter
jay inslee	jeff bezos

(c) (E $\bowtie_{\texttt{p\_id}}$ RT) $\bowtie_{\texttt{b\_id}}$ E

Figure 1.1: Example relations.

tables or *relations*. We use these terms interchangeably. Note that the join of two relations is another relation. The join *predicate* is the boolean condition on which two individual tuples from different tables (or possibly the same table) are linked. If a pair of tuples from two input relations satisfies the join predicate, their concatenation will appear in the join output [124].

To illustrate a join, we include Figure 1.1 which contains a simple example describing a company's employees along with the company's reporting structure. The figure contains two *base* relations; Table 1.1a associates an employee's name with their ID, and Table 1.1b associates an employee's ID with the ID of someone whom they report to. Base relations are stored explicitly by the DBMS and are used as the inputs to joins. Queries generally reference base relations directly. Consider the following query: we wish to generate the set of name pairs between employees and their managers. The DBMS must combine the relations in a chaining structure. The DBMS can start by joining the `id` column in the employee table with `person_id` in the reports-to table. This produces an *intermediate* relation or product. The DBMS may then continue by joining the `boss_id` column from the intermediate relation again with the `id` column in the employee table. In this example, the join predicates are equivalence predicates on the different id columns. After projecting to only names (dropping the now superfluous id columns), the desired name pairs can be found in Table 1.1c. Note that this query involved more than two relations. This is an example of a *multijoin.* Multijoins will be a heavy emphasis of this dissertation.

As in the above example, joins are generally chained together, one relation at a time. Importantly, the optimizer is free to choose the order in which relations are included in growing intermediate relations. While the above example is straightforward (there are really only two choices for join orders), joining relations in a poor order can lead to unacceptable wait times for results [91, 97]. Ideally, the optimizer will avoid poor orders. However, the optimizer can be tripped up and pick a poor order when the join structure or underlying data becomes more complex. How can we address these challenges? In this work we propose the use of sophisticated data sketching techniques addressing key problems that arise in the presence of joins.

We place particular emphasis on two specific areas. The first is *cardinality estimation* in traditional static analytical query optimization [64, 91]. Cardinality estimation is the subtask of estimating the number of rows of some join output. Deviating from the norm, we do not target "accurate" estimates but instead target theoretically guaranteed upper bounds for intermediate join cardinalities. In particular we demonstrate how to improve existing bounding techniques, making them more practical. In this sense, our contributions may be viewed as practical and mathematically driven modifications to the traditional query optimizer structure. While we defer a more detailed description of the traditional DBMS to Chapter 2, we preview our primary research contributions below.

The second area that we target is streaming joins. Streaming data refers to data that arrives live to the database versus static data that the DBMS has full access to when the query is placed [1, 46, 55]. Since the system cannot control when data arrives, joins are often executed using approximate predicates. An example of approximate predicates is the *interval join* where two tuples join if the difference between their timestamps is less than some δ instead of exactly equal. Querying streaming data generates a whole new class of challenges over its static counterpart. We focus on query execution *state* management; the handling of metadata during query execution. We provide an algorithm for state reduction in common streaming scenarios and go on to prove that our algorithm is optimal.

1.1 Research Contributions

In this section we categorize and preview the key contributions in our research. As stated earlier, these contributions approximately fall under two general areas. The first is the classical cardinality estimation sub-task that characterizes static data analytics and is the key problem during analytic query optimization. The second addresses join runtime execution optimization in the streaming data scenario.

1.1.1 Cardinality Estimation and Bounds

The first area of emphasis is rooted in traditional SQL analytical workloads where we target the problem of cardinality estimation in multijoin queries. Cardinality estimation refers to the problem of estimating the size of join relations. In the earlier example this might refer to the size of a two table subquery such as

$$\text{Employee} \bowtie \text{Reports-To}$$

or even the size of the final product

$$\text{Employee} \bowtie \text{Reports-To} \bowtie \text{Employee}$$

While this may seem trivial for smaller queries, the number of subqueries that must be estimated grows exponentially with the number of relations in the join. This is due to an exponential number of distinct subsets of the set of all base relations. While in most cases not every possible subset (subquery) must be estimated, generally the number subsets that do require estimation still grows exponentially. Although the explicit enumeration of subqueries is not challenging, the exponential factor forces the optimizer to produce each cardinality estimate relatively quickly. Thus, when we claim that cardinality estimation is challenging, the true challenge lies in being able to do so fast. The time constraint makes estimating the interaction between base relations very difficult; there simply isn't enough time to track all the intertwining and compounding interactions found in complex multijoin queries. Since the interaction of joining relations is purely based on their attribute value distributions (the frequencies with which different values appear in their joining columns) the problem can be re-framed as the concise encoding of attribute value distributions [88]. In particular, we need to keep track of *correlation* and *skew*.

Skew occurs when values in a joining attribute occur with disproportionate frequencies. For example, the string `Walter Cai` might only appear in a handful of tweets while `Barack Obama` will occur in billions. In this scenario we say `Obama` is a *skewed* value and we often describe distributions that contain attribute values that display this behavior as skewed. We also use the descriptors *frequent* and *heavy* interchangeably with skewed. *Correlation* piggybacks on top of skewed attribute value distributions. Suppose there is an accompanying dataset of Facebook posts which again are affiliated with the person they mention. Again, `Obama` will appear far more often than `Cai`. Were we to join the two relations, the skewed values will often coincide. As this example demonstrates, the nature of real world data is that skewed values in one relation are more likely to be skewed in the other relation as well. More technically, we say the two datasets' attribute value distributions are *correlated*. The challenge of accurately tracking correlation and skew across the join columns of multiple base relations makes achieving precise estimates for even a single subquery very difficult, not to mention for millions of sibling subqueries.

This tracking is made more difficult by *filter* predicates. Operators that restrict which tuples from a relation may be considered in some query. For example, in the above example, the user may specify that they only wish to consider tuples including `cai` in the Facebook dataset. This drastically changes the distribution of the data and can contribute to greater innacuracy during cardinality estimation.

Consider how the errors from not properly handling skew and correlation might affect an

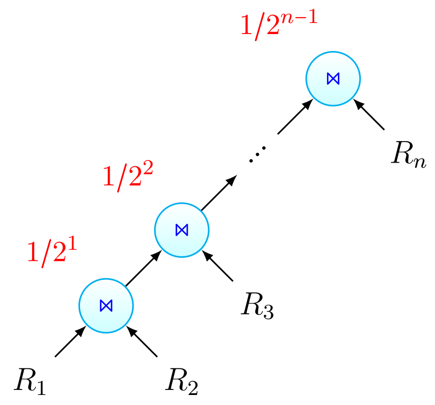


Figure 1.2: Left-deep join tree.

optimizer's cardinality estimate. Take the following n-table chain join.

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \tag{1.1}$$

For simplicity, consider one join ordering where relations are joined into the growing intermediate product from left to right.

$$\Bigg(\underbrace{\bigg(\underbrace{\big(R_1 \bowtie R_2\big)}_{\substack{\text{intermediate}\\\text{product}}} \bowtie R_3\bigg)}_{\substack{\text{intermediate}\\\text{product}}} \bowtie \cdots\Bigg) \bowtie R_n$$

This is often referred to as a *deep* tree. Although joins are logically commutative ($A \bowtie B$ is the same as $B \bowtie A$) physical join algorithms often require that one relation is the *inner* relation and one relation is the *outer* relation. The join output is equivalent but the ordering in the binary operator still matters. If the intermediate product is the outer relation in each join operation, this is further an example of a *left-deep* join tree. The tree structure is evident from the illustration in Figure 1.2.

What is often observed is that modern cardinality estimation formulas underestimate join size. Suppose that this occurs at each successive step in the chain join above and that each join underestimates by a factor of 2. Since each join is estimated based on the size/size estimates of its inputs, underestimation from a subquery will affect the accuracy of cardinality estimates higher up the join tree. This manifests as underestimation that compounds with each successive join. Thus, error is worse higher up the join tree and the final product will be underestimated by a factor of

$$\underbrace{2 \times 2 \times \cdots \times 2}_{(n-1)\text{–many}} = 2^{n-1}$$

where n is the number of relations in the join. Figure 1.2 illustrates this error in red above each join operator. This error can be worsened by filter predicates which drastically change the profile of a relation and render most table level statistics unrepresentative and/or insufficiently detailed. We will argue that underestimation is the riskier of underestimation versus overestimation and develop methods to address this.

Current state of the art cardinality estimation algorithms can generally be fit within a trade-off space between latency and accuracy. That is, the amount of time it takes to generate a cardinality estimate versus the accuracy of the result. This is not a surprising trade-off as more time to process data would logically lead to a more accurate estimate. At one end of the spectrum, traditional cardinality estimation methods such as histograms and strong assumptions prioritize latency but can lead to disastrous results in the presence of correlated and skewed data [124, 119, 110]. Next, static sampling methods lead to provably unbiased estimators although they are significantly slower than traditional methods and break when the static sample fails to capture sufficient base relation tuples to represent the filter and join predicates [36, 40, 115]. At the far end of the spectrum is dynamic sampling. Dynamic sampling methods are less brittle than their static counterparts but incur massive latency as well as often restrict the space of join orderings [94, 95]. The newest trend in cardinality estimation is to use machine learning models to assist with the estimation task [88, 87, 101]. However, these methods have yet to be shown to be generalizable and are exceedingly difficult to debug.

All above methods are designed to generate accurate and unbiased estimates on a per-query basis. In contrast, we take a holistic view of a workload of several queries. We observe that in many instances it is a minority of very slow queries that constitute the majority of poor workload performance. From a statistical standpoint the distribution of workload query runtimes has a long tail consisting of slow queries. These tail end queries are often bogged down by massively underestimated intermediate products [91]. This underestimation leads the optimizer to choosing aggressive plans forcing the DBMS to process these large intermediate products. We target these heavy tail queries by pivoting away from accuracy as a metric for success and instead towards safety. We propose replacing estimates with sufficiently tight provable upper bounds [18]. In this manner we avoid the heavy tail and remain on par with the majority of queries that traditional cardinality estimation already succeeds in finding an efficient plan for.

In order to achieve this result, we demonstrate the first practical implementation and integration of *entropic bounds* into a modern database management system; Postgres [122, 26]. Entropic bounds are a class of existing cardinality bounding formulas. We describe entropic bounds more thoroughly in Chapter 2. This initial implementation builds on the concepts of entropic bounds by combining the worst case optimal performance inherent to entropic bounds with more fine-grained data sketches for the underlying data. While our method again explores the same trade-off space between cardinality estimation latency and accuracy, it limits the risk of poor estimates leading to disastrous plans. The result is a theoretically strong foundation that partially bypasses cardinality estimation; arguably the hardest problem in query optimization.

We continue this line of research by previewing the incorporation of the cardinality bound principle in a modern big data management system: spark. Big data management systems (and distributed DBMS's) differ from traditional local single node systems in that they are predicated on the collective data processing power of a flexible fleet of commodity hardware [148, 47, 143]. While this infrastructure allows for fluid scaling and the ability to process vast quantities of data in parallel, the distributed setting introduces new challenges to the cardinality estimation process and hence joins remain a bottleneck. In particular, the need to reshuffle data between many data operators places a higher priority on robust query plans and join orders. While bounds can serve to mitigate these risks and ensure more robust plans, the distributed setting poses additional challenges for the collection and maintenance of the data sketches necessary to calculating cardinality bounds. We address these issues and demonstrate the practicality of our solutions.

Our key contributions in the area of static multijoin query cardinality estimation are enumerated below. These contributions comprise and extend our SIGMOD 2019 publication [26].

- We develop pessimistic cardinality estimation: a framework for improving entropic bounds via finer grained statistics.

- We provide a prototype implementation of the pessimistic cardinality estimation framework.

- We demonstrate the practicality of our approach based on challenging real world data.

- We preview a framework for the generalization of pessimistic cardinality estimation approach that addresses key challenges in scaling our approach.

1.1.2 Stream Query Optimization

The second area of focus for this thesis is stream join execution. Data Stream Management Systems (DSMS's) are the tool of choice to wrangle streaming data rather than traditional DBMS's which are predicated on static relations [105, 10, 55, 39, 28, 7, 150]. One can argue that DSMS's are just specialized versions of DBMS's that are predicated on the idea of streaming tuples rather than static relations. However, this form factor introduces another layer of complexity over traditional SQL engines. The requirements are significant enough that we treat them as a separate architecture in this dissertation. In particular, query output must be able to reflect changes based on these new tuples streaming in. Thus, in the streaming scenario, queries no longer have fixed final answers. These are often refered to as *standing* or *continuous* queries[39, 33]. Instead, the lifetime of a query can be infinite, depending only on the existence of new stream tuples. Practical scenarios that illustrate the streaming space include the constant accumulation and analysis of computing log files or data from Internet of Things (IoT) devices such as phones, cars, or traffic cameras.

In focusing on stream joins we shift away from query optimization that occurs ahead of query execution and instead explore optimal state management during execution. Again, state consists of the data/metadata that needs to be maintained by certain operators during execution. While some operators are *stateless* such as the filters that may be applied on a per tuple basis and that need not worry about remembering data dependent information during execution, stateful operators must store some data-dependent information. A simple example of a stateful operator is a summation. The summation operator must keep a running sum while iterating over tuples. A more complex example is a symmetric hash join where hash tables on both relations are built in parallel. Tuples from each relations are used to probe the neighbor's hash table and then inserted into their own relation's hash table. In this way the join algorithm guarantees that no join outputs are missed. In this scenario, both hash tables may be considered state associated with the symmetric hash join operator. The presence of such stateful operators is particularly tricky during the execution of streaming joins. As such, join state management becomes a bottleneck, especially when resources are scarce.

A natural extension of streaming query workloads is the temporal join. A temporal join is semantically equivalent to a standard join but operates on a special timestamp attribute that is packaged with every streaming tuple. For example, given two streams R and S, tuple $r \in R$ joins with tuple $s \in S$ if their associated timestamps are within some length of time $\omega/2$ from one another. This is in contrast to the standard equijoin where tuples join if their joining column values are equal. Thus the difference comes down to precise nature of the

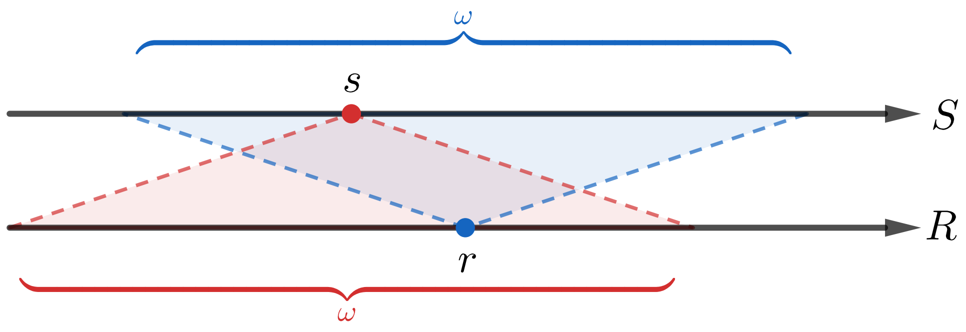


Figure 1.3: Simple stream join example.

join predicate. An illustration of the above query is included in Figure 1.3. Note that both r and s project symmetric and semantically equivalent windows onto S and R respectively. Any prospective join partner must live inside the associated window.

Often, these streams are transmitted over unreliable network and with variable latency. This is particularly common in edge computing and IoT scenarios where the edge device is a rudimentary sensor or has spotty connection to the central processor node in the system [126]. Whether due to variable latency or outright network failure, the system is forced to cache results from the faster or more reliable network and wait for the corresponding tuples from the unreliable stream. If a system lacks sufficient resources to handle this caching then the system will either fail or be forced to drop tuples.

We explore one scenario where omitting tuples is highly beneficial. Often, the system is tasked with applying a threshold function over the output of these stream joins [128, 99]. An alarm is triggered if a join tuple exceeds the threshold value.

Consider the example of a manufacturing facility with several machines. If condensation forms on the surface of the machines, it is likely that the product will be tainted or that the machines will begin to corrode. In order to avoid this problem, the plant supervisor is tasked with keeping the machines dry. The supervisor can employ an array of sensors that measure three statistics about the facility environment: air temperature, surface temperature, and relative humidity. These sensors stream readings on an IoT network to a rudimentary central processing node that will raise an alarm if it detects the possibility of condensation forming on the surface of one or more machines. That is, if some threshold function that takes the sensor readings as input crosses some threshold. Depending on the sophistication of the sensors, network, and central processing node the above system may break down and cause a catastrophic cache overflow. This may result in manufacturing delays, wasted materials, and damaged manufacturing equipment.

In order to combat such a scenario, we design an efficient state reduction algorithm and complimentary data sketch that is applicable to a broad collection of threshold functions. In particular, we develop an optimal omission policy for quasiconvex threshold functions where the state savings performance is agnostic of true throughput except in very low throughput situations. Instead, it depends only on the join interval size. We prove that our omission policy is optimal while guaranteeing equivalent alarm behavior sans tuple omission; no false positives, and no false negatives. Furthermore, we demonstrate how to generalize the omission policy to more complex multi-stream joins based on join topology as well as address tangential challenges in applying our omission policy [27].

Our key contributions in the area of stream join query optimization first appeared at VLDB 2021 and are as follows [27]:

- We develop an algorithm for theoretically optimal state reduction in the case of threshold functions over streaming joins.
- We generalize our technique to multi-stream joins.
- We run experiments that validate the technique's substantial state savings.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 is a background chapter. We review the architecture of typical DBMS's and go on to discuss the current state of the art in modern cardinality estimation including a review of the mathematical background to our cardinality bounds. We continue by covering the fundamentals of stream join semantics and how systems built for streams differ from traditional DBMS's. In Chapter 3 we describe our novel bounding technique as well as preview the results from our prototype local DBMS implementation incorporating bounds. In chapter 4 we generalizing our cardinality bounds to the distributed setting and discuss the new challenges that arise in this scenario. In Chapter 5 we change gear and present our optimal streaming state management policy. Finally, we conclude in Chapter 6.

Chapter 2
BACKGROUND AND RELATED WORK

In this chapter we will preview the current state of the art in query optimization, focusing on the areas that constitute our major contributions. We begin with a preview of the traditional declarative query optimization workflow. We place particular emphasis on the cardinality estimation subtask, characterizing it as a linchpin and discuss the strengths, and weaknesses of current state of the art techniques. We also discuss entropic bounds as they represent the foundation of the pessimistic cardinality bounding framework. We then branch into stream query optimization placing emphasis on new constraints and priorities placed on streaming systems as versus traditional analytic systems or even transactional systems.

2.1 The Traditional DBMS Query Workflow

As data becomes a more and more vital building block of everyday decisions, the need for scalable, efficient, and easy to use data processing systems grows as well. The traditional analytic query processor has been developed for decades, starting with hierarchical systems such as IBM's IMS, then network models such as CODASYL, and finally to the more familiar Relational model [42] that has stabilized as the standard [66].

Although the idea of records and tables didn't differ significantly between these foundational data models, the relational model pushed new ideas that have become the bedrock of modern analytical systems. In particular, the relational model emphasized a declarative querying interface that took the responsibility of optimizing queries out of the hands of application developers [98]. Not only did this open the door for advances in automated query optimization, it made DBMS's more usable to non-expert users.

We describe the general query processing system as composed of 3 primary components;

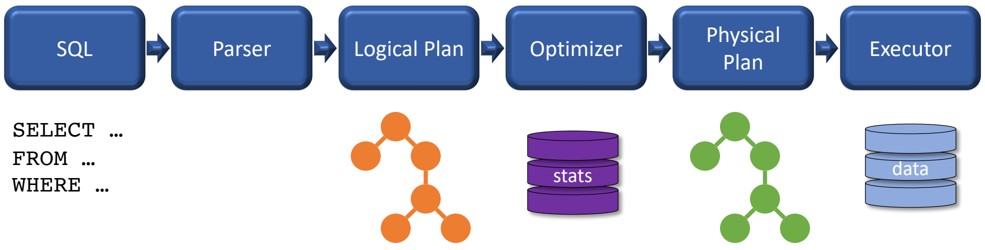


Figure 2.1: Typical analytic query workflow in a DBMS.

the Parser, Optimizer, and Executor. While the borders that separate these components are certainly blurry from one system to the next, the overall workflow is as follows. The system accepts a declarative query (e.g. SQL) and parses it to create a base logical plan; a mathematical structure that describes all the logical components of the desired query. The logical plan is fed to the optimizer which (as the name would suggest) optimizes it. That is, the optimizer transforms this logical plan into a physical plan; an actual algorithm that achieves the desired output as encoded by the logical plan. In particular, the optimizer decides on a join order; the order in which base/intermediate relations are joined together to produce the final output. A well-known approach, pioneered by the System R project, selects the optimal order using dynamic programming [127]. The optimizer starts by building an "optimal" plan for all two table subqueries. It then constructs what is an optimal plan for all three table subqueries built on top of the different plans that already exist for two table subqueries. Then four table subquery plans are built on top of the two and three table plans and so on. At the end, there will be a single plan for the desired n-table query that is provably optimal given the assumptions made by the optimizer along the way are true. These assumptions are the stumbling block that we seek to fix. Finally, the physical plan is performed by the executor which routes the resulting data to the user. Figure 2.1 illustrates this typical analytic query workflow.

For now we focus on the optimization layer. In later sections we will shift partial focus

`name`	`id`
walter	0
dan	1
magda	2
jeff bezos	3
archimedes	4
jay inslee	5

(a) Employee Table

`person_id`	`boss_id`
0	1
0	2
1	5
2	5
3	0
4	0
5	3

(b) Reports-To Table

Figure 2.2: Cross product of example base relations from Figure 1.1.

to the execution layer as the rolls of optimization and execution become more intertwined in the context of streaming data. In particular, we are interested in how the optimizer handles joins.

2.2 *Joins*

Recall from Chapter 1 that joins are fundamentally the act of linking tuples from different data sources. We now give joins a more formal treatment. A join operator takes three inputs, a left relation R, a right relation S, and a join condition. Semantically, a join is equivalent to a filter predicate applied to the cross product of the two relations. Take the two base relations given in Figure 1.1: Employee, and Reports-To. For convenience, we reproduce these tables in Figures 2.2a and 2.2b. Their cross product is a relation with four columns (the union of the two pairs of columns from the base relations) and would contain 42 rows: One for each pair of rows; one from the Employee and one from Reports-To.

$$|\text{Employee} \times \text{Reports-To}| = |\text{Employee}| \cdot |\text{Reports-To}| = 6 \cdot 7 = 42$$

The cross product is displayed in Figure 2.3a. The join condition then reduces this cross product to only those rows that satisfy that condition. Equijoins are the most common join

E_name	E_id	RT_pid	RT_bid
walter	0	0	1
walter	0	0	2
walter	0	1	5
walter	0	2	5
walter	0	3	0
walter	0	4	0
walter	0	5	3
dan	1	0	1
dan	1	0	2
⋮	⋮	⋮	⋮
jay inslee	5	4	0
jay inslee	5	5	3

(a) Employee × Reports-To

E_name	E_id	RT_pid	RT_bid
walter	0	0	1
walter	0	0	2
dan	1	1	5
magda	2	2	5
jeff bezos	3	3	0
archimedes	4	4	0
jay inslee	5	5	3

(b) $\sigma_{\texttt{E\_id=RT\_pid}}(\text{Employee} \times \text{Reports-To})$ $\equiv \text{Employee} \bowtie_{\texttt{E\_id=RT\_pid}} \text{Reports-To}$

Figure 2.3: Joins are semantically equivalent to the composition of a cross product and filter predicate

predicate: a cross product tuple survives filtering only if a pair of columns share the same value. In the example above, the most logical join condition would be that columns E_id (the id column from the Employee relation) and RT_pid (the person_id column from the Reports-To relation) are equal. The result is shown in figure 2.3b

Thus the join

$$\big(\text{Employee}\big) \bowtie_{\text{Employee}.\texttt{id}=\text{Reports-To}.\texttt{person\_id}} \big(\text{Reports-To}\big)$$

is semantically equivalent to

$$\sigma_{\texttt{E\_id=RT\_pid}}\big(\text{Employee} \times \text{Reports-To}\big)$$

From an algorithmic standpoint, executing a cross product first and then filtering is a highly inefficient procedure in almost all cases. Instead we will combine the two operators to execute them simultaneously. Algorithms that implement this behavior are physical join algorithms such as nest-loop, hash, and merge joins [124]. However, the ability to reduce a join to general filter and cross product operators is important, since instances of both operators are commutative (provided the filter operates on columns that are present; one cannot apply a filter before a necessary column appears). The net result is that joins are also commutative. Thus, if one defines a multijoin (a join with greater than two base relations), the order in which one incorporates new relations into intermediate relations doesn't matter; we can execute joins in whatever order we wish and the end result will be equivalent. This is powerful in that it provides an exponential number of join orders from which to choose from but problematic in that we must avoid poor join orderings.

While direct multijoin algorithms (algorithms that ingest multiple relations at a time) exist [140, 41, 114], most systems are built on the binary join paradigm. That is, most join algorithms consume two relations at a time. When a query involves greater than two relations, this necessitates the processing of intermediate products relations. These intermediate products are logical relations and each tuple that they contain must be processed during query execution. Moreover, different join orders will feature different intermediate relations. For this reason, the size of these intermediate relations is a key factor in the performance of one join ordering over another.

We now arrive at the subtask of cardinality estimation; the problem of estimating how large these intermediate relations will be. This problem has long plagued DBMS designers and we review the current state of the art in the following Section.

2.3 Cardinality Estimation Methods

Consider two relations R and S with sizes $|R|$ and $|S|$. Let the domain of the joining columns in each relation be D and let R_x be the set of tuples in R that take value x. Similarly for S_x. Define the value distributions for R and S to be $\{r_x\}_{x \in D}$ and $\{s_x\}_{x \in D}$ respectively. That is $r_x = |R_x|$ and $s_x = |S_x|$. For any relation, we call the subset of the domain which actually appears in the relation to be the *active domain.* More formally, the active domain of R is

$$\{x : r_x > 0\} \subseteq D$$

For now we restrict our view to the equijoin case. The size of the join in this scenario is as follows;

$$|R \bowtie_x S| = \sum_{x \in D} r_x \cdot s_x$$

The objective of cardinality estimation is to return an accurate estimator for this sum.

2.3.1 Classical Cardinality Estimation

Traditional optimizers rely on strong assumptions about the underlying data [141, 122]. In particular, optimizers –absent of other statistics– have assumed that attribute value distributions are *uniform.* I.e., that all attribute values within a relation appear with the same frequency. Given the common distinct values statistic, these systems may construct a simpe formula for cardinality estimates. For example, given two relations R and S with sizes $|R|$ and $|S|$ and distinct value counts $dv_R = |\{x : r_x \neq 0\}|$ and $dv_S = |\{x : s_x \neq 0\}|$, the estimated cardinality of $R \bowtie S$ is as follows [124]:

$$\frac{|R||S|}{\max(dv_R, dv_S)} \tag{2.1}$$

Note that the distinct value statistic is the size of the active domain. The explanation for why this formula fails when confronted with skewed and correlated data can be gleaned from a simple rewrite:

$$\frac{|R||S|}{\max(dv_R, dv_S)} = \frac{|R|}{dv_R}\frac{|S|}{dv_S}\min(dv_R, dv_S) \tag{2.2}$$

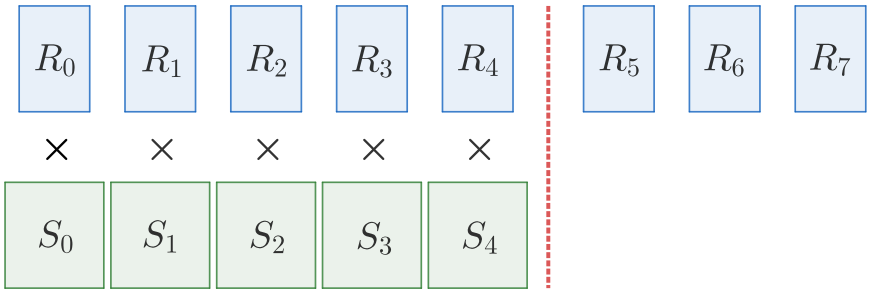


Figure 2.4: Traditional cardinality estimation relies on strong assumptions about the underlying data.

The first two terms assume that the the join columns are uniformly distributed. That is, for any attribute value $x \in D$ that appears in R (S), we assume $r_x = |R|/dv_R$ ($s_x = |S|/dv_S$). Under this assumption, if value x appears in both R and S, then the set tuples in $R \bowtie S$ that are the result of joining on value x will be the cross product of R_x and S_x:

$$|R_x \bowtie S_x| = |R_x \times S_x| = r_x s_x = \frac{|R|}{dv_R}\frac{|S|}{dv_S}$$

Finally, the optimizer assumes that the smaller of the distinct value sets derived from R, and the distinct value sets derived from S, is a subset of the larger. Each value x in the smaller active domain will contribute a total of $|R||S|/dv_R dv_S$ tuples to the join output. This produces the factor of $\min(dv_R, dv_S)$ appearing as the third multiplicative term in the righthand side of Equation 2.2. These assumptions are depicted in Figure 2.4.

While the assumption of maximal overlap between active domains should push the estimate higher, the uniformity and independence assumption is a best case scenario assumption and should push the estimate lower. As is with most things, two wrongs do not make a right and these assumptions do not cancel each other out. While the active domain overlap assumption often holds true, or at least close to true, the uniformity assumption often breaks. This leads to massive underestimation which compounds with every subsequent join in a multijoin query.

Underestimation from the uniformity model is worsed by a secondary assumption often made my optimizers; *independence*. Independence assumes that filters predicates across the same and different relations are independent (aka uncorrelated). The way this manifests in cardinality estimation formula is that the selectivity of the conjunction of two filters is assumed to be the product of the selectivities of each of the filters independently:

$$|\sigma_{x=\texttt{a}}(R) \bowtie_z \sigma_{y=\texttt{b}}(S)| = \text{selectivity}_{x=\texttt{a},y=\texttt{b}} \cdot |R \bowtie_z S| \approx \text{selectivity}_{x=\texttt{a}} \cdot \text{selectivity}_{y=\texttt{b}} \cdot |R \bowtie_z S|$$

Real world data and workloads generally break this assumption since pairs of filter predicates are often written to capture related attribute values. The product of each filters' selectivity thus underestimates the selectivity of the conjunction of the filters thus leading to further underestimation.

The upside to this estimation method is that it is exceedingly fast in delivering estimates, leading to a very short optimization time. However, systematic underestimation leads to more aggressive query plans; plans that are marginally faster if the relations adhere to the above idealistic assumptions but which can lead to disastrous execution times when the size of intermediate relations blows up.

2.3.2 Sampling based Methods

The above formulas lead to fundamentally biased estimators. Researchers have instead pivoted to achieve unbiasedness. Sampling for cardinality estimation is attractive since it inherently handles all single relation filter predicates while also delivering unbiased estimates. A significant portion of recent work on cardinality estimation focuses on the generalization and optimization of sampling methods [92, 40, 141, 78, 50]. We describe some of the more important variations below.

Static uniform Bernoulli sampling has existed for decades in database systems offering a per-table precomputed sample. Let $R(x, y)$ and $T(y, z)$ be two relations with precomputed samples $\mathcal{S}_R(x, y)$ and $\mathcal{S}_T(y, z)$. If an optimizer needs an estimate for a simple join: $R(x, y) \bowtie$

$T(y,z)$, we may use $\mathcal{S}_R$ and $\mathcal{S}_T$ to produce an unbiased estimate:

$$\left|R(x,y) \bowtie T(y,z)\right| \approx \left|\mathcal{S}_R(x,y) \bowtie \mathcal{S}_T(y,z)\right| \cdot \frac{|R|}{|\mathcal{S}_R|} \cdot \frac{|T|}{|\mathcal{S}_T|}$$

This approach can be generalized to queries with arbitrary number of relations and join predicates between them:

$$\left|\bowtie_{j\in[m]} R_j\right| \approx \left|\bowtie_{j\in[m]} \mathcal{S}_{R_j}\right| \cdot \prod_{j\in[m]} \frac{|R_j|}{|\mathcal{S}_{R_j}|} \tag{2.3}$$

However, in practice this method suffers in the presence of highly selective filters, or many tables appearing in the join. In the case of highly selective filters, the precomputed uniform samples might not have sufficient size (possibly size zero following filter application) and won't be able to capture the true selectivity. In the case of many tables, it is likely that uniform samples won't have sufficient intersection and might lead to empty sample-joins. This is the primary drawback of Bernoulli sampling where independent sampling from different tables often yields near empty or empty joins of the samples.

Correlated sampling first proposed by Vengerov et al. [141] is a clever twist for generating samples from multiple tables which successfully join. Consider the same running example: we wish to estimate the cardinality of $R(x,y) \bowtie T(y,z)$. We fix some sampling probability threshold p and random hash function h mapping the domain of the join attribute to $[0,1]$ and define sample sets $\mathcal{S}_R$, and $\mathcal{S}_T$ as follows:

$$\mathcal{S}_R = \{r \in R : h(r[y]) < p\}$$
$$\mathcal{S}_T = \{t \in T : h(t[y]) < p\}$$

Note that for relations with multiple join attributes, it suffices to use one hash function per join attribute. Furthermore, the sample for such a relation will be dependent on all join attributes present in that relation:

$$\mathcal{S}_R = \Big\{ r \in R : (h_y(r[y]) < p) \wedge (h_z(r[z]) < p) \Big\} \tag{2.4}$$

We call this conjunctive inclusion. Because we hold the hash function h constant over the two relations, the samples are correlated and Equation 2.3 would generate a biased estimator. We may unbias the sample join as follows:

$$\begin{aligned} \Big| R \bowtie T \Big| &\approx \Big| \mathcal{S}_R \bowtie \mathcal{S}_T \Big| \cdot \Big(\mathbb{P}\left(r \bowtie t \in \mathcal{S}_R \bowtie \mathcal{S}_T \right) \Big)^{-1} \\ &= \Big| \mathcal{S}_R \bowtie \mathcal{S}_T \Big| \cdot \Big(\mathbb{P}(r \in \mathcal{S}_R \wedge t \in \mathcal{S}_T) \Big)^{-1} \\ &= \Big| \mathcal{S}_R \bowtie \mathcal{S}_T \Big| \cdot p^{-1} \end{aligned}$$

Generalizing to multijoin tables: let the join $\bowtie_{j \in [m]} R_j$ feature n join variables. In the case of a simple chain join, we may write $n = m - 1$. In the general case our bounding formula have:

$$\Big| \bowtie_{j \in [m]} R_j \Big| \approx \Big| \bowtie_{j \in [m]} \mathcal{S}_{R_j} \Big| \cdot p^{-n}$$

Correlated Sampling addresses the shortcomings of Bernoulli sampling and performs better on relations with larger active domains but instead suffers when datasets are very skewed, in which case the static sample size can be far too large or alternatively far too small. Both options are poor. Overly large sample sizes defeats the purpose of optimization as one will be performing too much of the query ahead of the execution phase. Overly small sample sizes fail to provide sufficient signal to generate a decent execution plan.

A variant of correlated sampling is designed to handle the problem that tuples from tables with multiple join columns would need to satisfy the hash value threshold for all columns independently of one another. This conjunctive inclusion policy (defined in Equation 2.4) makes it difficult to tune the threshold parameter p across tables since tables with multiple join attributes might yield overly sparse samples, while tables with fewer join attributes produce samples that are too large. As a solution, it is instead possible to use a disjunctive policy: a tuple is included in the sample if any one of the multiple join columns is included in the sample.

$$\mathcal{S}_R = \left\{ r \in R : (h_y(r[y]) < p) \vee (h_z(r[z]) < p) \right\} \tag{2.5}$$

For small values of p, this makes the sampling rate approximately equal to $J \cdot p$ where J is the number of join variables present in a relation. Thus, the constant factor growth is more easily tuned ahead of time than exponential shrinkage in the conjunctive case. This is particularly attractive where different queries in a workload join on different subsets of attributes. The same static sample can be used in all queries, as the join columns in different tables will be sampled in a correlated fashion and the sample sizes correspond linearly to the sample rate rather than exponentially. However, this variant also makes debiasing the estimate more complex. Let J_j be the number join variables appearing in relation R_j:

$$\left| \Join_{j \in [m]} R_m \right| \approx \left| \Join_{j \in [m]} \mathcal{S}_{R_j} \right| \cdot \prod_{j \in [m]} 1 - (1 - p)^{J_j}$$

We further note that the threshold probability p need not be held constant across the entire query and can instead be independently set for each join attribute.

Two level join sampling is an approach proposed by Chen et al [40]. The authors categorize the strength and benefits of Bernoulli and Correlated sampling algorithms and seek to combine them with a single "2-level" sampling algorithm. The authors claim that correlated sampling is powerful for capturing correlation between joining columns in separate tables, whereas Bernoulli sampling is useful for capturing correlation between columns within the same table. Correlated and Bernoulli sampling correspond naturally to levels 1 and 2, respectively, in this sample structure. Given level-1 sampling probability p_1, and level-2 sampling probability p_2, Chen et al. describe a per-table single pass algorithm where the end result will be a sample set with the following characteristics. For any join value v, there is probability

p_1 that tuples with join value v will be included in the sample. If tuples with join value v are included, then there will be a single "sentry" tuple drawn uniformly from all tuples including v. All other such tuples will be included with probability p_2.

The weakness with static sampling is failure in the face of highly selective filter predicates. If no tuple in a sample satisfies the filter, then only an estimate of zero will be produced. Index based join sampling, introduced by Leis et. al. [92], seeks to combat the sparse sample problem by replacing the precomputed samples with an iterative process that leverages existing indexes on the full relations to build new samples for intermediate relations at runtime similar to Wander Join [94]. Specifically, they start with a uniform sample from some base relation in the query and iteratively build outwards along join predicates. In this manner, the method may generate a set of samples for each necessary intermediate result. The algorithm is recursive and ingests some sample $\mathcal{S}$ from an intermediate join or base relation T. The algorithm also ingests a base relation A to be logically joined in to T and which has a precomputed index. The output is a sample $\mathcal{S}'$ from the join $T \bowtie A$. Sample tuples in $\mathcal{S}'$ are generated by using tuples from $\mathcal{S}$ to probe the index on A, returning tuples that will successfully appear in $T \bowtie A$. In order to avoid blowup or starvation inside the intermediate join samples, the authors describe a budgeting and doubling back scheme which is designed to maintain a relatively constant sample size as more tables are logically included. While these dynamic sampling schemes are more robust to diverse workloads than static sampling, the tradeoff is that optimization time takes significantly longer.

Improved selectivity estimation [109] combines data sketching (referred to as data synopses in the text), and sampling based methods to improve cardinality estimation. However, the authors do not extend their methodology to include joins of any kind but instead focus only on multiple filters on a single relation.

2.3.3 Robust Query Optimization

Robust query optimization and bounded query execution is an area focusing not necessarily on picking the fastest plan, but instead places greater value on avoiding poor plans, even at the cost of slightly suboptimal planning and execution time [13]. Babcock et al. propose generating a probability distribution view of a cardinality estimate. The authors allow users to submit a confidence threshold at runtime at which point the query optimizer combines the threshold and cardinality estimate distribution to return a cardinality point estimate with varying aggressiveness [21]. In the context of query re-optimization, Babu et al. propose a confidence interval view of the cardinality estimates, which reduces the need for later query re-optimization [23].

2.3.4 Machine Learning based Optimization

The newest trend in query optimization replaces one or several components in an optimizer with Machine Learning (ML) models. The most directly comparable method converts the cardinality estimation functionality with a regressor [82, 83]. As with most ML applications, the greatest challenges come from training. Because of the diversity of datasets, training on one dataset will likely not lead to an optimizer that can easily be deployed on a different dataset or even different workload on the same dataset. For this reason, ML models that explicitly target cardinality or selectivity will often need to be generated on a per-dataset level at least partially. If a model trains directly on a dataset, one can argue that the model is simply learning the attribute value distributions of the underlying data. An additional signal source is past queries (real or synthetic). This optimizer-execution feedback loop was introduced in the 1990's [2, 3] and was popularized by Stillger et al. in their work on IBM's DB2 [131]. Wu et al. rely on past queries in the workload to tune estimates, leveraging the fact that common subqueries are often repeated in production workloads [147]. Woltmann et al. target the training phase directly by enabling training on pre-aggregated data [146]. Others restrict the scope of the problem to achieve tractability. For instance, several authors focus on the problem of estimating the effect of multiple filters but only on single relation queries [65, 49]. The Alex system focuses on the problem of needing to update learned index structures on nonstatic data [48]. Other approaches target tangential problems in the optimizer such as the plan enumeration stage [90].

Some systems take a holistic view of the optimizer and replace multiple components –if not the entire optimizer– with ML models [132, 102, 116]. SageDB presents such a vision but still acknowledges the difficulty of vertically integrating a system as complex as a traditional optimizer into a single model [87]. Many of the same authors also published a paper on learned index structures which promoted the concept of replacing common index structures such as b-trees with learned models [88]. The paper can be credited with kicking off this new trend of utilizing ML models within key components of the system and its impact on research is undeniable. Cuttlefish also took a more aggressive approach to query optimization by framing plan selection as a multi-armed bandit problem [77]. The system pivoted between different physical operators to test their efficiency with the goal of eventually converging on the fastest operator.

While ML models show some promise, their greatest criticism up to this point is that their application in systems treats the models as black boxes. The complexity of underlying data forces complex models, which in turn lack transparency. While a technique might show promise in a narrow application –one domain, one dataset, one query– practitioners have little guidance and little reason to believe that those benefits would carry over to other settings. It is partially for these reasons that in later sections we lay out our reasoning for simplicity and conservatism in query optimization.

2.4 Entropic Bounds

Our cardinality bounding approach is based on the seminal work by Atserias, Grohe, and Marx and their introduction of entropic bounds [18]. These bounds are derived from analyzing the deep connection between information theory and relational databases. For convenience we first review the definition of entropy.

2.4.1 Entropy

Informally, entropy is a mathematical measure of randomness. For the purposes of this document, we refer to the information theoretic definition of entropy of discrete random variables.

Formally, given some discrete random variable X that takes values x_i with probability p_i we define the entropy of X to be

$$h(X) = -\sum_i p_i \log(p_i)$$

Intuitively, the uniformly distributed random variables where each p_i is equal are the "most" random and take the value $-\log(p_i)$ whereas more skewed variables will have smaller entropy values.

2.4.2 The AGM Bound

The first step towards understanding these bounds comes from analyzing the connection between information theory and relational databases. Framing relational joins as hypergraphs, Atserias et al. generate upper bounds based on fractional edge covers [18]. More formally, consider a conjunctive query on relations $R_1, \ldots, R_m$ with attribute collections $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$. Using Datalog, the query can be expressed as:

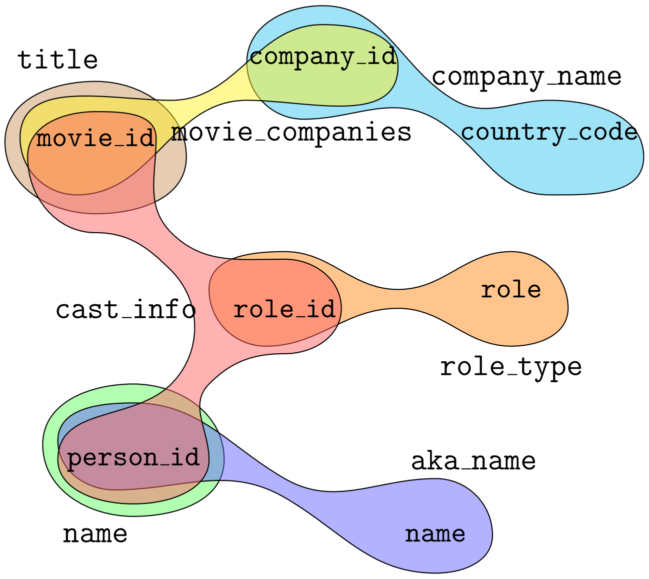
$$Q(\boldsymbol{a}) \text{:-} R_1(\boldsymbol{a}_1), \ldots, R_m(\boldsymbol{a}_m)$$

where $\boldsymbol{a} = \cup_j \boldsymbol{a}_j$. We define a *hypergraph* $\mathcal{H}$ which models the schema of the query. A hypergraph is a generalization of an undirected graph where the *hyperedges* of the hypergraph are defined as arbitrary nonempty subsets of the ve, instead of only strictly size 2 subsets. Let the vertices of $\mathcal{H}$ corresponding to the attributes $a \in \boldsymbol{a}$. For each relation R_j include a hyperedge corresponding to the attributes-set $\boldsymbol{a}_j$. Figure 2.5 describes query `08c` from the popular Join Order Benchmark (JOB) [91]. For the SQL code given in Figure 2.5a, we provide the corresponding hypergraph in Figure 2.5b.

A *fractional edge cover* $(u_1, \ldots, u_m)$ of $\boldsymbol{a}$ on $\mathcal{H}$ is a set of values $u_j \in \mathbb{R}_{\geq 0}$ corresponding to the hyperedges $\boldsymbol{a}_j$ where for all vertices a of the hypergraph, the sum of the u_j values

```
SELECT
    MIN(a1.name) AS writer_pseudo_name,
    MIN(t.title) AS movie_title
FROM
    aka_name AS a1,
    cast_info AS ci,
    company_name AS cn,
    movie_companies AS mc,
    name AS n1,
    role_type AS rt,
    title AS t
WHERE
    cn.country_code = '[us]'
    AND rt.role = 'writer'
    AND a1.person_id = n1.id
    AND n1.id = ci.person_id
    AND ci.movie_id = t.id
    AND t.id = mc.movie_id
    AND mc.company_id = cn.id
    AND ci.role_id = rt.id
    AND a1.person_id = ci.person_id
    AND ci.movie_id = mc.movie_id;
```

(a) SQL query.



(b) Corresponding hypergraph $\mathcal{H}$.

Figure 2.5: JOB query `08c`.

corresponding to hyperedges containing a is at least 1. That is, attribute a is *covered*:

$$\forall a \in \boldsymbol{a}: \quad \sum_{j: a \in \boldsymbol{a}_j} u_j \geq 1$$

Asterias et al. proved that we may bound the join cardinality as follows [18]:

$$|Q| \leq \prod_{j=1}^{m} |R_j|^{u_j}$$

This class of bounds is referred to as the *AGM bound* after being originally developed by Atserias et al [18]. A proof depends on Shearer's Lemma and may be found in Appendix

A. Note that these formulas assume uniqueness of full tuples within each relation. This restriction is a fact that must be enforced by the underlying database system. Note that while some bounding formulas are still valid even in the presence of repeated rows, this is not generally the case. One method to avoid the pitfalls of non-unique rows is to introduce a synthetic dummy variable that is unique to every row in a relation.

Khamis et al. extend the AGM bound to include degree parameters [81]. Their contributions generalize the information theory versus relational join analogy to include conditional entropic formulations. We refer to this broader class of bounds as the *KNS bound*. While the KNS bound is broader, the actual calculation of all formulas within KNS can be highly complex. For this reason, we will demonstrate how to produce a pared down but still effective subset in Subsection 3.1.

2.5 Streaming Query Optimization

We now pivot to stream query optimization. Stream query management differs from traditional analytic query management in that data is not assumed to be complete at time of querying. Instead, data is continuously streamed in from different data sources and the Data Stream Management System (DSMS) is charged with delivering new query results based on the new fresh stream tuples. In this scenario, stream query output may not be finite, assuming the streams are not finite either. Examples of data streams include mobile network usage data, log files from data warehouses, and even social network activity.

This pushes the demands of DSMS away from the architecture of traditional analytic systems. In particular, the continuous nature of the analytic task means that a single optimization stage as depicted in Figure 2.1 is impractical. We may no longer rely on a continuous flow of data from some local data source whose access the DBMS controls. Instead, the DSMS must contend with an uncontrolled stream of data and adapt to unforeseen circumstances as necessary. This blurs the line between the optimization and execution phases of our standard analytical workflow. For this reason the optimizations that we describe in Chapter 5 can be viewed as execution optimization as versus traditional query optimization.

However, given the unique challenges presented by streams, we argue that the contributions still fall under the umbrella of join query optimization.

The observant reader will note that the domain of DSMS's might be more closely related to transactional DBMSs rather than an analytic DBMS. Transactional systems are a class of DBMS that focus less on the analysis of static relations and more on the scalable, concurrent, and correct processing of transactions; small changes to underlying data that may not be complicated but are numerous. Both areas contend with flucuating flow rate of data and the processing of many smaller queries/transactions instead of a handful of large analytical queries. Furthermore, both areas prioritize throughput. However, DSMSs differ from the transactional paradigm in that they still emphasize complex analysis whereas transactional systems are more concerned with serializability and scheduling.

DSMSs also introduce a new spin on joins. Temporal joins are joins whose predicate involves a special attribute that is packaged with each tuple. This attribute is a timestamp, most often associated with the creation, transmission, or arrival of the tuple. Semantically, we can think of the timestamp as some numeric value and a temporal join as simply acting on this numeric value. However, there are inherent challenges given the connection between the manifestation of the tuple and the timestamp value. In the following section, we explore this concept further.

2.5.1 Temporal Joins

There are two semantic considerations when defining a time-based join between two or more streams. The first is how to assign timestamps to tuples. Using *event time* semantics, the event source assigns a timestamp to each base stream tuple that remains with the tuple as it passes through the DSMS. Using *processing time* semantics, the DSMS assigns a timestamp to each stream tuple when it arrives, that is, when it is "processed". Event time semantics are preferred as event-time timestamps are not susceptible to network failures or latency concerns. The timestamps are also a more accurate representation of the base tuple data. Processing time semantics are generally easier to handle as tuples can never arrive out of

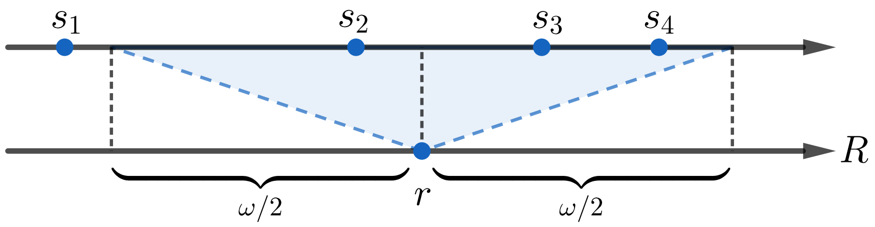


Figure 2.6: Sliding join between R and S. Tuple r defines time interval $[t_r - \omega/2, t_r + \omega/2]$. Tuples s_2, s_3, s_4 join with r since they live inside this interval. Tuple s_1 falls outside the interval and is therefore not a joining partner.

order since the clock at the DSMS is the ground truth. In this work we use event time semantics as it is more representative of current systems.

The second consideration is how we define the join predicate. Again there are generally two approaches: *sliding* window and *hopping* window. In sliding window semantics (a.k.a. interval join), tuples r and s join if their event times fall within $\omega/2$ of one another. See Figure 2.6. In hopping window join semantics, the windows have a specific length ω and move forward in steps of size δ. That is, if the first window is $[0, \omega]$, then the second is $[\delta, \delta + \omega]$, the third is $[2\delta, 2\delta + \omega]$, and so on. Tuples r and s join if there exists a window W such that $t_r, t_s \in W$. Note that tuples may fall within multiple windows if $\delta < \omega$.

Both sliding and hopping semantics define a collection of time windows where tuples join if there exists a window in that collection that contains both. Sliding windows define a superset of the windows defined by hopping windows. In fact, assuming some nontrivial and bounded event time space, hopping defines a finite number of windows whereas sliding defines an infinite number. Thus, while we focus on sliding semantics, it should be clear that our methods are easily applicable to hopping semantics as well. However, the discrete nature of hopping yields more obvious and straightforward omission techniques. For the rest of this thesis we use ‘interval join’ to mean a ‘sliding window’ temporal join.

2.5.2 Streaming Queries Related Work

Although we are unaware of any existing work on our specific problem of threshold queries over streaming joins, we have found similar scenarios in the literature where our proposed techniques may have potential applications.

Load Shedding

One closely related area of research is load shedding, where tuples are dropped (sometimes at random) to prevent overloading and hence increased latency [46, 134, 22]. One can view our omission policy as a theoretically optimal form of load shedding.

Dropping tuples randomly leads to an obvious trade-off in query answer degradation. Past work has often focused on how to implement this degradation gracefully, generally in response to “bursty” stream behavior overwhelming the system.

Local Geometric Constraint Thresholding

Another line of related research focuses on multiparameter threshold functions over distributed data [125]. Unlike our work, it preaggregates the data with respect to each object and compute node and does not explicitly join on timestamps. For instance, vector $x_{j,i}$ corresponds to some subset of the data corresponding to object j and living at node i. The full vector describing object j is

$$x_j = \sum_i x_{j,i}$$

The threshold function is applied to x_j. In order to avoid a full distributed aggregation to find objects that trigger this threshold, the authors describe an algorithm for generating bounds local to each node using geometric constraints of the parameter vector space. These local geometric constraints were first highlighted by Sharfman et al. [129, 128]. They generalize this method from exclusively monotonic functions to functions that may be expressed as the difference of monotonic functions—a class of functions that subsumes the set of globally quasiconvex functions. Giatrakos et al. expand on this local geometric constraint theme by incorporating predictors to further reduce the number of system synchronizations that need to be performed [54]. Their approach relies on individual nodes reliably predicting the “drift” in vector values in neighboring nodes since the most recent synchronization. This introduces a trade-off space between system load and query answer quality similar to many load shedding algorithms.

Analytic Functions over Data Streams

Most commercial streaming and time-series database systems now support *analytic functions* over data streams or time series. For example, TimescaleDB [135], a time-series database built on top of PostgreSQL, offers simple analytic functions such as `first()`, `last()`, and so on. The recently launched Amazon Timestream [10], a serverless time-series database hosted on Amazon AWS, supports more advanced functions such as computing the cosine similarity of two vectors. The query language of InfluxDB [72] provides an even richer set of functions for time-series analysis, such as `holt_winters()` [58, 70]. Although the query language reference manuals of these systems do not provide specific examples, it is straightforward to write queries that apply thresholds on top of such analytic functions.

Time-Series Analysis

Thresholding has been a common technique in time-series analysis for various applications. For example, *threshold models*, a popular class of nonlinear models in time-series analysis, have been around for decades [137]. The basic idea is to use different models for different parts of a time series that are above or below a threshold. Another example is *threshold-based data mining* [16, 17], which uses threshold queries for similarity search over time-series data. The idea is to truncate time series using a threshold, and then measure similarity between (two) time series using a distance function that only considers the intervals above the threshold. It is possible to express such distance computation using timestamp-based, streaming joins.

Streaming Joins

Joins over two or more streams have become increasingly popular in real-world applications. As a result, stream processing systems, such as Apache Spark Structured Streaming [14], Apache Flink [28], and Microsoft's Azure Stream Analytics [105], recently started supporting streaming joins. One common technique implemented by existing systems is the *symmetric hash join* algorithm (and an analogous symmetric nested-loop join algorithm) [55, 56, 79, 145], which has to buffer join states in main memory. This raises challenges in applications where the size of join states can be extremely large. Specialized systems, such as Google's Photon [11], have been built to deal with such cases. Our technique provides another novel perspective on reducing the amount of join state to be kept.

Stream Query Optimization

There has also been quite a bit of work on the query optimization side of stream join processing. For example, Viglas and Naughton [142] proposed cost models for both nested-loop join and symmetric hash join in the streaming context with the goal of maximizing the query output rate, which can be easily integrated into classic query optimization frameworks such as ones that are based on dynamic programming. Ayad and Naughton [20] further proposed a query optimization framework for conjunctive queries over data streams that considers resource constraints. There are various other optimization techniques for stream query processing in general, such as operator separation, fusion, and reordering (see [69] for a survey). It would be interesting future work to consider the interaction between our technique and existing query optimization techniques. For instance, if an input stream of a join is the output stream of a subquery, then we can perhaps push down our tuple omission strategy into input streams of that subquery.

Stream Memory Management

There has been work on systems-oriented techniques for memory management, for example, offloading operator state in SEEP [29] and Google Cloud Dataflow [30]. These techniques are orthogonal to our approach, which exploits query semantics for reducing memory. Memory reduction techniques have been applied in the context of specific operators such as aggregates over sliding windows [138, 133]; our work uses a similar flavor in the context of state reduction for join queries.

2.5.3 Quasiconvex Functions

One remaining piece of background is the concept of emphquasiconvex functions. We focus on this class of functions as they will allow for our omission policy which we describe in Chapter 5.

We say a function $f : \mathbb{R} \to \mathbb{R}$ is quasiconvex iff for all $x_1, x_2 \in \mathbb{R}$, for all $\lambda \in [0, 1]$

$$f\big(\lambda x_1 + (1 - \lambda)x_2\big) \le \max\big\{f(x_1), f(x_2)\big\}.$$

That is, for all pairs of points x_1, x_2, when f is evaluated on a point x in between x_1 and x_2, $f(x)$ cannot exceed both $f(x_1)$ and $f(x_2)$. We provide an example in Figure 2.7. We can generalize the notion of quasiconvexity to a function over multiple variables, $f : \mathbb{R}^n \to \mathbb{R}$, where values $x_1, x_2 \in \mathbb{R}^n$ are vector valued.

We do not require that a function be globally quasiconvex. We only require it to be quasiconvex with respect to any streaming input on which we apply our omission policy. A multivariate function $f : (x_1, x_2, \ldots, x_n) \to \mathbb{R}$ is quasiconvex with respect to x_1 if for any values $\bar{x}_{i \neq 1}$ the univariate function $g(x_1) = f(x_1, \bar{x}_2, \ldots, \bar{x}_n)$ is quasiconvex.

Many classes of functions are quasiconvex. Examples include convex functions, linear functions, monotonic functions (including step functions, such as floor and ceiling), positive quadratic, logarithmic, exponential, simple inequality boolean triggers, and the application of trained linear or logistic regressors. The question of how to automatically determine if a

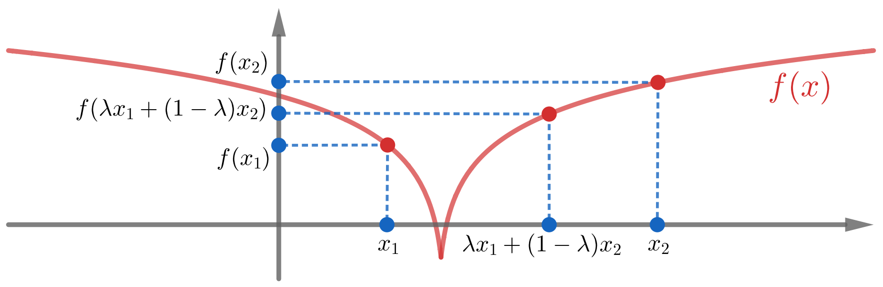


Figure 2.7: Example quasiconvex function.

function is quasiconvex is addressed in Section 5.3.

Chapter 3

TIGHTER CARDINALITY BOUNDS

Recall from Chapter 2.1 that DBMSs parse declarative queries, construct plans, and then execute. Often, the declarative query necessitates one or more joins be performed. Joins are a fundamental data operation and are a near requirement for any production database management system (DBMS). The existence of multiple data access methods, physical join algorithms, and in particular *join orderings* means the DBMS has a plethora of options from which to choose. Recall join orderings refer to the specific orderings in which base (and intermediate) tables are joined together in order to produce the desired output. Recall further that base tables are relations that already exist as versus intermediate relations which are computed during plan execution. This complexity is compounded by special orderings that are either required or a byproduct of different join algorithms and access methods. For general queries the number of valid join trees is factorial in the number of relations (in fact a product of Catalan numbers on top of factorial expressions [111]). In order to make sense of all the available plans, a query optimizer enumerates some or all choices and proceeds to pick one to execute. The choice of which plan to follow is important. Just because the output of the different plans is the same doesn't mean that the efficiency of each plan is the same (or even close). There can exist many orders of magnitude between the runtime of one plan versus the runtime of a semantically equivalent alternative. To make this choice, the optimizer tabulates an estimated cost for each plan and proceeds to pick the cheapest. This tabulation is parameterized with approximate costs that represent the amount of time it takes to perform operations in the plan; one disk read, one index lookup, one boolean filter evaluation, etc. However, one aspect of these large parameterized summations is highly data dependent; the cardinality (size) of intermediate relations. Since most physical join algorithms are binary, any query with greater than two base tables will necessarily produce intermediate relations than can then be joined with base or other intermediate relations to work towards the desired final query output. Leis et al. demonstrate that given a perfect black box cardinality estimator, systems can estimate plan cost almost perfectly (up to a constant multiplicative factor) [91]. This implies that the parameterized formulas work but are crippled by poor cardinality estimates. For this reason, cardinality estimation is often referred to as the Achilles heel of query optimization [97].

Consider a simple four relation chain query:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$$

In Figure 3.1 we depict every such join tree for the above query.

Luckily, not every intermediate relation is unique; we group the five unique shared subqueries by color. In total, valid join plans will cover an exponential number of subqueries, each of which will require an estimate from the optimizer. While this enumeration may seem daunting, the greatest challenge is one of data dependency. Small errors estimating selectivity of filter predicates on base relations compound with an inability to efficiently estimate correlation between joining columns. Recall that selectivity refers to the fraction of tuples that satisfy a boolean filter predicate. The result is that the higher one climbs in the join tree, the greater variance is experienced in intermediate join cardinality estimation. The more relations the query has, the more complex the family of valid join trees are, which in turn leads to poorer plans.

As argued in Chapter 2.3, current state of art methods fail to accurately estimate the cardinality of intermediate relations. In fact, the prevailing over-reliance on strong assumptions about the underlying data has led to systematic underestimation across nearly all major production systems. We propose the use of cardinality bounds in lieu of cardinality estimates and demonstrate how to tighten these bounds resulting in more robust query optimization.

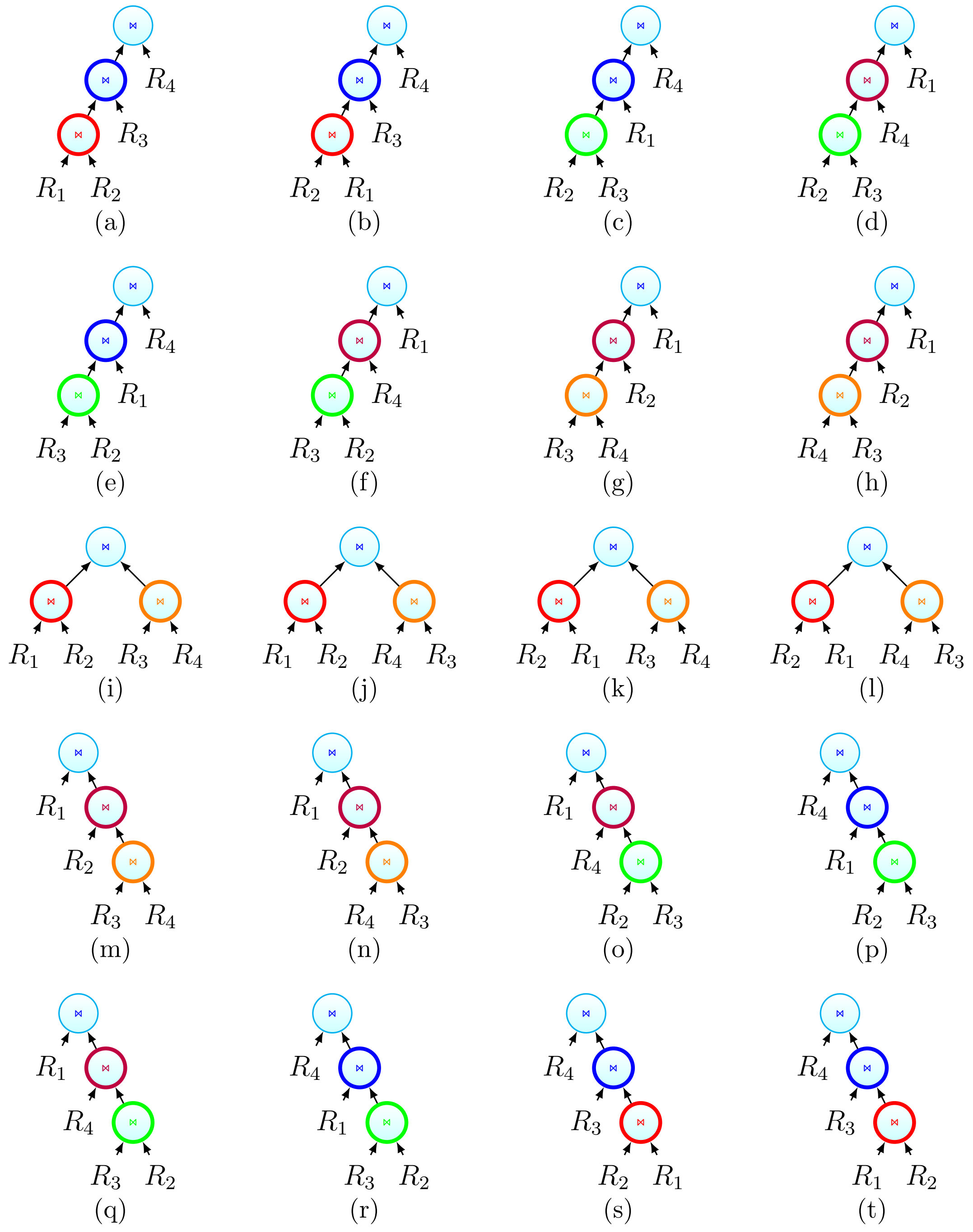

R_4 R_3 R_1 R_2 (a)
R_4 R_3 R_2 R_1 (b)
R_4 R_1 R_2 R_3 (c)
R_1 R_4 R_2 R_3 (d)
R_4 R_1 R_3 R_2 (e)
R_1 R_4 R_3 R_2 (f)
R_1 R_2 R_3 R_4 (g)
R_1 R_2 R_4 R_3 (h)
R_1 R_2 R_3 R_4 (i)
R_1 R_2 R_4 R_3 (j)
R_2 R_1 R_3 R_4 (k)
R_2 R_1 R_4 R_3 (l)
R_1 R_2 R_3 R_4 (m)
R_1 R_2 R_4 R_3 (n)
R_1 R_4 R_2 R_3 (o)
R_4 R_1 R_2 R_3 (p)
R_1 R_4 R_3 R_2 (q)
R_4 R_1 R_3 R_2 (r)
R_4 R_3 R_2 R_1 (s)
R_4 R_3 R_1 R_2 (t)

Figure 3.1: Non-cross product valid join trees for chain query $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$. Equivalent subqueries highlighted in the same color.

3.1 Deploying Cardinality Bounds

In this section we present the first version of our approach. The key idea is to use data sketches to tighten the bounds previewed in Section 2.4.2. The remainder of this Section is divided up as follows. In Section 3.1.1, we define our core data structure; the *Bound Sketch* (BS). In Section 3.1.2, we describe how the BS is used to generate and tighten theoretical join cardinality upper bounds. We follow up this discussion with two performance optimizations which make bounding more feasible. In Section 3.1.3, we describe our selection predicate propogation technique as well as the reasoning behind using it. In Section 3.1.4, we introduce our hash bucketization budgeting scheme. Finally, in Section 3.1.5, we describe the process of automatically generating the bounding formula given the query's hypergraph structure.

3.1.1 The Bound Sketch

We first describe the structure of the BS. Take a relation $T(\boldsymbol{a})$ with attributes $a \in \boldsymbol{a}$ and random hash function $H : W \mapsto \{1, \dots, M\}$ where W is the domain of the attributes. Let $[M]$ denote integer values $\{1, \dots, M\}$ which is the image of the hash function. Tuples in T take values in the domain $W^{|\boldsymbol{a}|}$. For any tuple t, let $t[a]$ refer to the attribute value of a in t. Thus, tuples t may be mapped to index value arrays $I \in [M]^{|\boldsymbol{a}|}$ via the hash function. Note that multiple distinct tuples may map to the same index array. Given such an index I, let the $I[a]$ refer to the position in I corresponding to attribute a. For each index array I, we define the subset T^I of relational instance T as those tuples $t \in T$ that hash to the values in I.

$$T^I = \Big\{ t \in T : H(t[a]) = I[a], \quad \forall a \in \boldsymbol{a} \Big\}$$

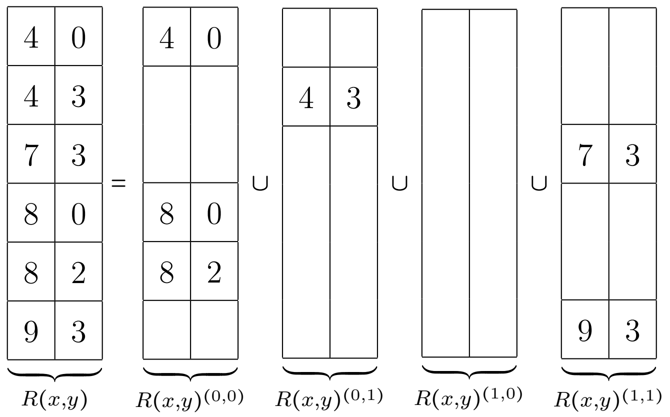
The BS on T is a collection of $|\boldsymbol{a}| + 1$-many $|\boldsymbol{a}|$-dimensional tensors of size $M \times \cdots \times M$ Each cell of the first tensor contains a *count* value c. We define the I-th entry of this tensor as $c_{T^I} = |T^I|$. Each cell of the remaining $|\boldsymbol{a}|$ tensors contains a *degree* value d. In this thesis, we use the term *fanout* interchangeably with *degree*. Each of these tensors corresponds to a conditional attribute in $\boldsymbol{a}$. We define the degree parameter corresponding to variable $a \in \boldsymbol{a}$ as the maximum degree for attribute a from the partition T^I. I.e. the frequency of the most frequent a attribute value from the tuples in T^I. More formally:

$$d_{T^I}[a] = \max_{\left\{\substack{w \in W: \\ H(w)=I[a]}\right\}} \Big| \underbrace{\{t : t[a] = w\}}_{\subseteq T^I} \Big|$$

Finally, we describe the BS as:

$$\Big(\underbrace{[c_{T^I}]}_{\text{count tensor}}, \underbrace{[d_{T^I}[a_1]], \ldots, [d_{T^I}[a_{|\boldsymbol{a}|}]]}_{|\boldsymbol{a}|\text{-many degree tensors}} \Big) \in \left([M]^{|\boldsymbol{a}|}\right)^{|\boldsymbol{a}|+1}$$

As an illustrative example, consider the following relation $R(x, y)$. The hash function is an indicator for even values: $h(x) = x\%2$. We first logically partition R into $2 \cdot 2 = 4$ subsets:



This data generates the following tensor components in the BS:

$$\left(c_R = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix},\ d_R[x] = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix},\ d_R[y] = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \right)$$

Each statistic may be computed by hand: Count sketch c_R is computed by counting the number of tuples in each position; three, one, zero, and two. Degree sketch $d_R[x]$ captures the frequency of the most frequent x value in each partition: two (for attribute value 8), one (for attribute value 4), zero, and 1 (for either attribute values 7, or 9). Similarly for $d_R[y]$ Note that many BS's are constructed off-line and may be called at runtime for any joins involving the relevant relation. Those BS's that are constructed at runtime are done so to satisfy query-specific filter constraints and we will later argue that these tables would likely have had to be scanned/probed at runtime, anyway. We further note it is possible to create the BS from a single pass over the relation. For more details, we refer the reader to Appendix B.

3.1.2 Our Cardinality Bounds

Next, we describe how the BS is used to tighten guaranteed cardinality upper bounds compared to when applying the bounding formula without use of the BS. Consider schema

$$R_1(\boldsymbol{a}_1), \ldots, R_n(\boldsymbol{a}_n)$$

and database instance D on relations $R_1, \ldots, R_n$. Let $\boldsymbol{a} = \cup_j \boldsymbol{a}_j$ be the set of all attributes. In this context if two attributes are equijoined in the given query, then they are considered the same attribute. Define index array $I \in [M]^{|\boldsymbol{a}|}$ as before, and let $I[\boldsymbol{a}_j]$ be the sub-array of I whose values correspond to the attributes present in $\boldsymbol{a}_j$. Define a database instance D^I as a subset of database instance D where for each relational instance $R_j(D)$, we take the subset

$$R_j\left(D^I\right) = \left\{t \in R_j(D) : \forall a \in \boldsymbol{a}_j, \quad h\left(t\left[a\right]\right) = I[a]\right\}$$

That is, the set of all tuples in $R_j(D)$ that hash to the correct index values for each attribute in $\boldsymbol{a}_j$. Define database instance subset $D^I = \{R_1\left(D^I\right), \ldots, R_m\left(D^I\right)\}$. Given a conjunctive query $Q(D)$, observe that we may reconstruct the full conjunctive query using only these

D^I:

$$Q(D) = \bigcup_{I \in [M]^{|\boldsymbol{a}|}} Q\left(D^I\right) \tag{3.1}$$

Note that this is a necessarily disjoint union. We now invoke the primary results of the AGM and KNS bounds (See Chapter 2.4.2). As an illustrative example we describe a triangle query:

$$Q\left(x, y, z\right) \text{:-} R\left(x, y\right), S\left(y, z\right), T\left(z, x\right) \tag{3.2}$$

Take a triple of random variables (X, Y, Z) corresponding to attributes x, y, z, respectively, and ranging uniformly over the collection of all tuples in the true output. The size of the query is tied to the joint entropy of all three variables. Specifically, $h(X, Y, Z) = \log|Q(x, y, z)|$. By construction on the domain space of the triple (X, Y, Z), and since our query only includes equivalence join predicates, for any subset of $\{X, Y, Z\}$ which happens to correspond exactly to the attributes of some relation in our schema we may bound the entropy of that subset of variables. Specifically:

$$h(X, Y) \leq \log(c_R), \quad h(Y, Z) \leq \log(c_S), \quad h(Z, X) \leq \log(c_T)$$

Similarly, we may relate conditional entropic formulas to our degree statistics:

$$\begin{aligned} h(X|Y) &\leq \log(d_R^y), & h(Y|X) &\leq \log(d_R^x) \\ h(Y|Z) &\leq \log(d_S^z), & h(Z|Y) &\leq \log(d_S^y) \\ h(Z|X) &\leq \log(d_T^x), & h(X|Z) &\leq \log(d_T^z) \end{aligned}$$

We may exploit conditional subadditivity, as well as Shearer's lemma (Lemma A.0.2) to generate entropic bounds for $h(X,Y,Z)$:

$$h(X,Y,Z) \leq \begin{cases} h(X,Y) + h(Z|Y),\ h(Y,Z) + h(X|Z),\ h(Z,X) + h(Y|X) \\ h(X,Y) + h(Z|X),\ h(Y,Z) + h(X|Y),\ h(Z,X) + h(Y|Z) \\ \frac{1}{2}h(X,Y) + \frac{1}{2}h(Y,Z) + \frac{1}{2}h(Z,X) \end{cases} \tag{3.3}$$

Each of these entropic bounding expressions corresponds to a bounding formula for $Q(x,y,z)$. We enumerate the corresponding query cardinality bounding formulas for the entropic bounding expressions in Equation 3.3:

$$|Q(D)| \leq \begin{cases} c_R d_S^y, \quad c_S d_T^z, \quad c_T d_R^x \\ c_R d_T^x, \quad c_S d_R^y, \quad c_T d_S^z \\ (c_R)^{\frac{1}{2}} (c_S)^{\frac{1}{2}} (c_T)^{\frac{1}{2}} \end{cases} \tag{3.4}$$

While these formulas represent the complete KNS bound, we often will only generate a subset. Specifically, those formulas expressible with the BS and also not strictly dominated by other bounding formulas. For a full justification, we refer the reader to Appendix A as well as the original papers: [81, 18]. Finally, we may combine the bounds over the database partitions defined in Equation 3.1. Summing over all index combinations provides an upper bound on D.

$$|Q(D)| \leq \sum_{I \in [M]^{|\boldsymbol{x}|}} \min \begin{cases} c_{R^I} d_{S^I}^y,\ c_{S^I} d_{T^I}^z,\ c_{T^I} d_{R^I}^x \\ c_{R^I} d_{T^I}^x,\ c_{S^I} d_{R^I}^y,\ c_{T^I} d_{S^I}^z \\ (c_{R^I})^{\frac{1}{2}} (c_{S^I})^{\frac{1}{2}} (c_{T^I})^{\frac{1}{2}} \end{cases} \tag{3.5}$$

The bound derived from 3.5 should be tighter than the bound derived from Equation 3.4 since it has access to more information about the underlying data. In Section 3.2 we demonstrate that this behavior is exhibited on real world data.

3.1.3 Filter Propagation and Preprocessing

We now describe an optimization to make calculation of these bounds more tractable. Specifically, we propagate filters through foreign key joins to simplify the join topology. We again consider the JOB query featured in Figure 2.5. We provide an alternative relation-centric graphical representation of this query in Figure 3.2. In this illustration the nodes represent relations and edges between the nodes represent join predicates. Note that there exist 4 distinct join attribute equivalence classes across the seven tables. Specifically, these are the ID's for individual films, people, companies, and role types represented by edge colors blue, red, orange, and violet, respectively. Primary-key-foreign-key (PK-FK) constraints are represented by arrows pointing in the direction of the relation where the attribute is a primary key. Dashed lines represent foreign-key-foreign-key (FK-FK) joins. To increase efficiency at the planning stage, it is best to eliminate as many join attributes as early as possible.

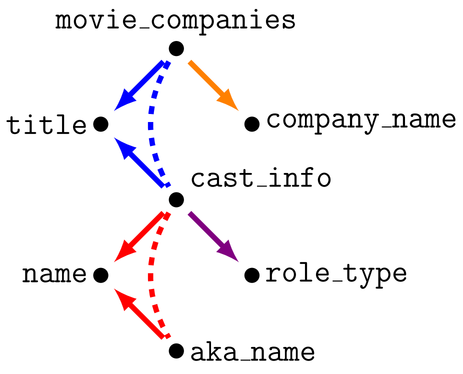


Figure 3.2: JOB query `08c` join attribute graph.

This is easily done by first analyzing the query topology and then eliminating join attributes (and hence also some relations) which will not increase the output cardinality. For instance, JOB query `08c` features the selection:

```
role_type.role = 'writer'
```

This particular string selection corresponds to a set of role type ID's. Often this is a highly selective predicate and in this case only corresponds to a single ID: $\{4\}$. It is therefore possible to preprocess the relation `cast_info` and generate a fresh BS on $\sigma_{\texttt{role\_type\_id=4}}(\texttt{cast\_info})$.

This is not practical (or beneficial) in all cases. For instance, JOB query `08c` also features the following selection:

```
company_name.country_code = '[us]'
```

In contrast to the previous example, this selection will correspond to several thousand company ID's[1]. Our system detects the large number of returned IDs while preprocessing `company_name` and will instead default to generating an updated BS for

$$\sigma_{\texttt{country\_code='[us]'}}(\texttt{company\_name})$$

In this manner, we decompose a join into only those predicates which threaten to cause blowup in intermediate products.

Preprocessing also helps us broaden the scope of our method. Although we are currently only able to handle equivalence predicates during our bound generation, propagation of selections allows us to handle `LIKE`, and inequality $(\geq, >, \neq, \leq, <)$ filter predicates. This method requires scanning relevant base relations at runtime. However, only a single scan is required and we argue that for queries featuring FK-FK joins between large relations, the preprocessing time is insignificant compared to the risk of poor plan selection. In the case of our running example (JOB `08c`), these "dangerous" attributes featuring FK-FK join predicates

[1] 132,917 individual companies

are people, and films and are highlighted as dashed lines in Figure 3.2.

We emphasize that some sketches can be cached ahead of time, while others must be populated at runtime. If a relation is subject to any filter predicates following filter selection propagation, then the BS associated with that relation must be populated at runtime. Otherwise, the BS may be drawn from cache and will not require a scan of the table.

3.1.4 Hash Partition Budgeting

The hash partitioning approach on join attributes leads to the question: how many buckets should be used for each join attribute? If we set a fixed partition size for each attribute, the set of all hash value combinations can grow exponentially with the number of join attributes even after selection propagation. Generating a bound for each combination of hash values can quickly become computationally impractical as the number of join attributes grows. Furthermore, it is in fact possible for the sum of these bounds to increase as we increase hash size. This is due to hash collisions between non-joining tuples, and the exponential in hash size number of terms (hash value combinations) in the bound summation. We have included a simple example in Appendix C demonstrating the potentially non-monotonic behavior of our bound.

We simultaneously address both these problems by introducing a hash partition budgeting scheme. The underlying idea is simple: for any bounding formula, and bucket threshold B, we only calculate a bound for at most B hash value combinations. We distribute this budget to those attributes that are covered *unconditionally* by a relation. That is, the variable associated with the attribute appears in an unconditional entropic term in the entropic bounding formula associated with the cardinality bounding formula. By only partitioning unconditionally covered variables, we get the desired behavior that increasing partition budget should tighten our bound. In fact, if we only increase the partitioning budget by integer factors from B to $B' = B \cdot r$ and thereby only allow subpartitioning of existing buckets, this monotonic behavior is guaranteed. Monotonic behavior is not guaranteed when partitioning join attributes that are covered conditionally. For example, consider the following query:

$$Q(x, y, z, w) \text{:-} \texttt{aka}(x, y), \texttt{ci}(y, z), \texttt{mc}(z, w), \texttt{cn}(w) \tag{3.6}$$

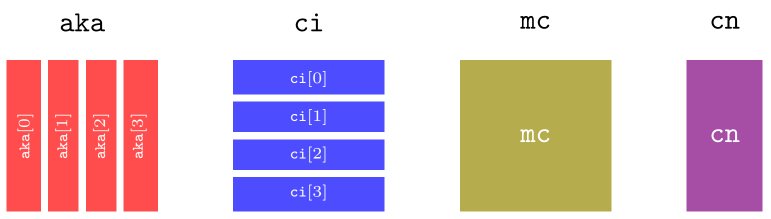
This query yields multiple bounding formulas including the following two examples and their corresponding entropic formulas:

$$c_{\texttt{aka}} \cdot d^{y}_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}} : \qquad h(X, Y) + h(Z|Y) + h(W|Z) \tag{3.7}$$

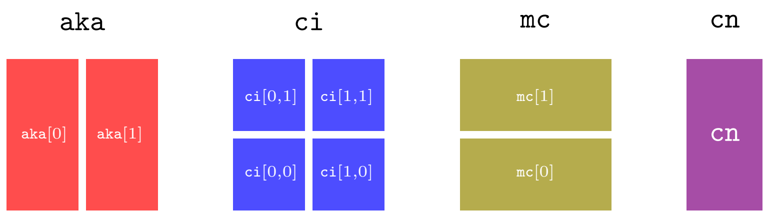
$$d^{y}_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}} : \qquad h(X|Y) + h(Y, Z) + h(W|Z) \tag{3.8}$$

For the bound formula in Equation 3.7, we dedicate all of our B budget to the only unconditionally covered join attribute: y (covered unconditionally by `aka`). Note that x is not a join attribute; we would not benefit from partitioning x. In this scenario, the number of hash buckets for y, z, and w are B, 1, and 1, respectively. That is, z and w are not partitioned at all. Alternatively, for the bound formula in Equation 3.8, we observe that join attributes y, and z are both covered unconditionally by `ci`. We therefore dedicate $\sqrt{B}$ buckets to both y and z so that the total number of hash-index combinations is $\sqrt{B} \cdot \sqrt{B} = B$. We provide illustrations for the bounding formulas in Equations 3.7, and 3.8 in Figures 3.3a, and 3.3a, respectively where $B = 4$.

The primary shortcoming to our budgeting strategy is that the index combinations are no longer directly comparable between distinct bounding formulas since each formula now demands a different hashing scheme. The general bounding expression now minimizes over all bounding formulas instead of over all index combinations. While this does remove some inherent gains from fine granularity minimization (i.e. minimizing inside Equation 3.5's summation), our experiments demonstrate that for a single query, a majority of index combinations tend to favor a single bounding formula. Hence, this modification represents a worthwhile trade off between the benefits of fine grained comparison, robust performance, and sufficient generality to include queries with a large number of join attributes. Our new



(a) Partitioning for formula $c_{\texttt{aka}} \cdot d^y_{\texttt{ci}} \cdot d^z_{\texttt{mc}}$, budget $B = 4$. Partitioning of size 4 given to y attribute.



(b) Partitioning for formula $d^y_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^z_{\texttt{mc}}$, budget $B = 4$. Partitioning of size 2 given to both y and z attributes.

Figure 3.3: Hash partition budgeting illustration for formulas in Equations 3.7 and 3.8 regarding the query in Equation 3.6.

general expression is:

$$|Q(D)| \leq \min_{\substack{b\in \\ \text{bounding formulas}}} \left(\sum_{\substack{I\in \\ \text{partition indexes}}} b\left(Q\left(D^I\right)\right) \right) \tag{3.9}$$

This modification also affects the sketch generation step. We observe that even for a single partitioning budget, different bounding formulas might each necessitate a BS from the same relation but of different dimension. However, since relations will generally not exceed three foreign keys, there are few combinations of relevant tensor dimensions leading to few sketches describing the same relation. Furthermore, each sketch is inherently limited to the budgeted number of partitions B (or is many degrees of magnitude smaller) and hence the storage overhead for storing offline sketches is also low.

As an illustrative example, consider again the query in Equation 3.6 which we reproduce below for convenience:

$$Q(x, y, z, w)\text{:-}\texttt{aka}(x, y), \texttt{ci}(y, z), \texttt{mc}(z, w), \texttt{cn}(w)$$

The bounding formulas $d^{y}_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$ will require a $\sqrt{B} \times \sqrt{B}$ dimension BS for relation `ci`, whereas bounding formula $c_{\texttt{aka}} \cdot d^{y}_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$ will require a $B \times 1$ dimension BS (with respect to attribute y). However, since both sketches may be generated in a single pass, we may construct them in parallel.

3.1.5 Bound Formula Generation

We continue with our running example of JOB query `08c`. Selection propagation helps us eliminate one of four join attributes. In this task, we employ a further optimization: ignoring remaining dangling K-FK joins. We do this because the degree multiplier on the key side of the join is one (by definition). It is usually not the case that this information will help to tighten the bound. For example, when considering the join `movie_companies` $\bowtie$ `title`, the resulting output will be determined by the FK side of the join, in this case `movie_companies`. In fact, this is always true assuming the database does not allow dangling FK pointers to nonexistent keys. We therefore ignore joins satisfying this restriction. Note that in JOB query `08c`, the inclusion of the relation `company_name` does not satisfy these conditions because there is a further selection predicate on `company_name.country_code`. While this selection is not sufficiently selective to warrant full selection propagation, it is significant enough to warrant the presence of `company_name` in our bounding formulas. The selection predicate has created a situation where keeping a dangling FK pointer in the `movie_companies` relation might be beneficial. The modified hypergraph following selection propagation and removal of (most of the) dangling K-FK joins may be found in Figure 3.4. Note that we left an unnamed "attribute" represented simply by a dot in the the

$$\sigma_{\texttt{country\_code='[us]'}}(\texttt{company\_name})$$

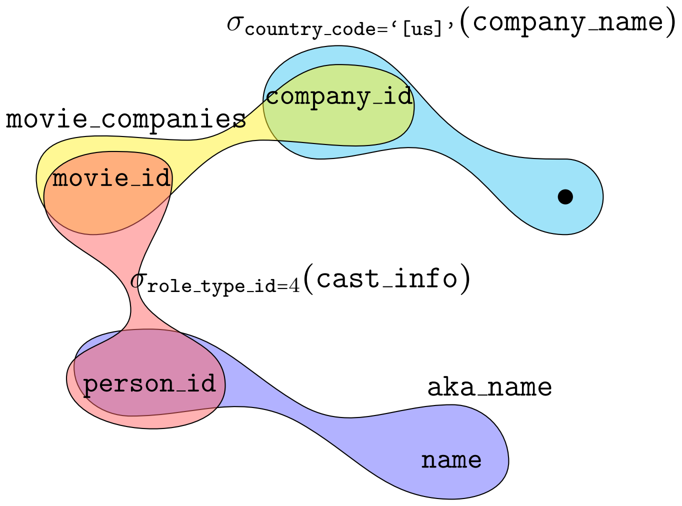

$\sigma_{\texttt{country\_code='[us]'}}$(company_name)
company_id
movie_companies
movie_id
$\sigma_{\texttt{role\_type\_id}=4}$(cast_info)
person_id
aka_name
name

Figure 3.4: Hypergraph of JOB `08c` following propagation of the `role_type` and `country_code` selection predicates.

relation. This is because `company_id` is a foreign key in `company_name` but not a primary key. This inherently implies there exists other attributes which `company_name` must cover and avoids violating the duplicate tuples restriction.

The task of generating bounding formulas is as follows. We first consider all combinations of attribute coverages and filter those that do not fit within our bounding scheme. Using these coverages, we then build our explicit bounds. The pseudocode for generating our bounding formulas and coverages may be found in Algorithms 1 and 2.

The algorithm ingests the attribute-centric hypergraph representation and iterates through coverage assignments for each attribute. That is, for every coverage combination, each attribute will be covered by precisely one of the relations which features that attribute. For every possible coverage combination we consider only those that are expressible by elements of the BS. This implies for all relational hyperedges e, the number of attributes covered by e must take one of three possible values in order for our sketches to be applicable: 0, $|e|$, or $|e| - 1$.

The intuition for these three possibilities is as follows:

Algorithm 1 Bound Formula Generator. Given a hypergraph representation of a query, returns a collection of bounding formulas where each formula takes the form of a 2-tuple: $(c, d = (r, a)) \subseteq R \times (R \times A)$. We let c be the set of relations contributing count statistics. We let $d = (r, a)$ be the set of pairs of relations r and corresponding attributes a, where r contributes a degree statistics with respect to a.

```
1: procedure BOUNDING_FORMULAS($\mathcal{H} = (A, R)$)    ▷ input hypergraph where $A$ are
   attributes (vertices) and $R$ are Relations (hyperedges)
2:     $\mathcal{B} \leftarrow$ Gen_Covers($\mathcal{H}$)                          ▷ generate coverages
3:     $\mathcal{F} \leftarrow \{\}$                                          ▷ bounding formulas
4:     for $\boldsymbol{r} = (r_1, \ldots, r_{|A|}) \in \mathcal{B}$ do
5:         $c \leftarrow \{\}$                            ▷ relations contributing count terms
6:         $d \leftarrow \{\}$                           ▷ relations contributing degree terms
7:         for $r \in R$ do
8:             cover_count $\leftarrow |\{r' \in \boldsymbol{r} : r' = r\}|$
9:             if cover_count = 0 then
10:                pass                  ▷ relation $r$ does not appear in bounding formula
11:            else if cover_count = $|r|$ then
12:                $c \leftarrow c \cup \{r\}$
13:            else if cover_count = $|r| - 1$ then
14:                $a$    ▷ $a$ will be unique attribute in $r$ that is not paired [covered] by $r$ in $\boldsymbol{r}$
15:                for $a' \in r$ do
16:                    if $\boldsymbol{r}[a] \neq r$ then
17:                        $a = a'$
18:                $d \leftarrow d \cup \{(r, a)\}$
19:            else                        ▷ all other possibilities filtered by Gen_Covers
20:        $\mathcal{F} \leftarrow \mathcal{F} \cup \{(c, d)\}$
21:    return $\mathcal{F}$                                          ▷ bounding formulas
```

Algorithm 2 Feasible Coverage Generator. Given a hypergraph representation of a query, returns a restricted collection of attribute to covering relation mappings.

```
1: procedure GEN_COVERS($\mathcal{H} = (A, R)$)                    ▷ input hypergraph
2:     $\mathcal{C} \leftarrow \prod_{a \in A} \{r \in R : a$ covered by $r\}$        ▷ cross product
3:     $\mathcal{B} \leftarrow \varnothing$
4:     for $\boldsymbol{r} = (r_1, \ldots, r_{|A|}) \in \mathcal{C}$ do
5:         safe = True
6:         for $r \in R$ do
7:             if $|\{r' \in \boldsymbol{r} : r' = r\}| \notin \{0, |r|, |r| - 1\}$ then
8:                 safe = False
9:         if safe then
10:            $\mathcal{B} \leftarrow \mathcal{B} \cup \{\boldsymbol{r}\}$
11:    return $\mathcal{B}$                                      ▷ feasible coverages
```

- If a relation has coverage size 0, then it has no coverage responsibilities and need not appear in the resulting bound formula at all.

- If a relation has coverage size equal to the number of attributes present in the relation, then this corresponds to an unconditional entropic term. Therefore, the count term for the relation will appear in the resulting bound formula.

- If a relation has coverage size equal to the number of attributes present in the relation minus 1, then this corresponds to a conditional entropic term conditioned on that missing attribute. Therefore, the degree term for that relation with respect to that missing attribute will appear in the resulting bound formula.

We note that other coverages can also lead to valid bounding formulas. In case a join attribute is covered by more than one relation, the bound will still be valid, but the formula will be dominated by a similar formula where that attribute is instead only covered by a single relation. That is, there exists a formula that must produce a bound that is less than or equal to the multiple coverage bound. Another consideration is those coverages where there exists a relation e which covers at least one attribute, but fewer than the number of join attributes present in e minus 1. This corresponds to a degree term with respect to

	name	peopleID	filmID	companyID	·
1	a1	a1	ci	mc	cn
2	a1	a1	ci	cn	cn
3	a1	a1	mc	mc	cn
4	a1	a1	mc	cn	cn
5	a1	ci	ci	mc	cn
6	a1	ci	ci	cn	cn
7	a1	ci	mc	mc	cn
8	a1	ci	mc	cn	cn

(a) Coverage combinations.

	formula
1	$c_{\texttt{a1}} \cdot d_{\texttt{ci}}^{\texttt{peopleID}} \cdot d_{\texttt{mc}}^{\texttt{filmID}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
2	$c_{\texttt{a1}} \cdot d_{\texttt{ci}}^{\texttt{peopleID}} \cdot c_{\texttt{cn}}$
3	$c_{\texttt{a1}} \cdot c_{\texttt{mc}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
4	$c_{\texttt{a1}} \cdot d_{\texttt{mc}}^{\texttt{companyID}} \cdot c_{\texttt{cn}}$
5	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot c_{\texttt{ci}} \cdot d_{\texttt{mc}}^{\texttt{filmID}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
6	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot c_{\texttt{ci}} \cdot c_{\texttt{cn}}$
7	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot d_{\texttt{ci}}^{\texttt{filmID}} \cdot c_{\texttt{mc}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
8	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot d_{\texttt{ci}}^{\texttt{filmID}} \cdot d_{\texttt{mc}}^{\texttt{companyID}} \cdot c_{\texttt{cn}}$

(b) Bound formulas.

Figure 3.5: JOB query `08c` coverage combinations and their corresponding bound formulae.

greater than one attribute. Since we restrict our BS's to not include these terms, we cannot use these coverages. The generated coverages and corresponding bound formulas for JOB query `08c` are enumerated in Figure 3.5. For readability, we refer to relations with their name abbreviations as it appears in Figure 2.5a. We abuse notation and refer to the filtered relation $\sigma_{\texttt{role\_type\_id=4}}(\texttt{cast\_info})$ simply as `ci`.

Note that some bounding formulas entirely separate the query graph into disjoint subgraphs (formulas 2, 3, 4, and 6). On the other hand, some formulas treat the join as a single long chain where they start with the count term on a single relation and build outward with degree multipliers (formulas 1, 5, 7, and 8). Finally, note that only formulas involving the count term $c_{\texttt{cn}}$ on `company_name` (formulas 2, 4, 6, and 8) are likely to benefit from the selection predicate on `company_name.country_code`. This is because `companyID` is a key in `company_name` but the selection is not highly selective. Hence, the remaining formulas will instead feature $d_{\texttt{cn}}^{\texttt{companyID}}$ terms which will almost always take value one during actual calculation.

3.2 Evaluation

In this section we evaluate our bounds on real world data. We begin by describing our datasets and workloads in Subsections 3.2.1 followed by a brief discussion of our implementation in Subsection 3.2.2. In Subsection 3.2.3 we investigate how effective our partitioning strategy is at tightening bounds. Finally, in Subsection 3.2.4 we demonstrate the effect of our tightened bounds on query execution time.

3.2.1 Datasets and Query Workloads

We focus on two datasets and associated workloads. The first is a collection of 45 GooglePlus community edge-sets [57]. The edge counts within the communities range between 228,521 and 1,614,977, and the cardinality of the self-join triangles derived from these respective communities range between 1,791,588 and 130,322,694. Each tuple (A, B) in an edgeset represents an "*A follows B*" relationship and is therefore not necessarily bidirectional. The edgesets therefore comprise directed graphs.

We design a microbenchmark based on this dataset. We construct 11 distinct query templates, each featuring a unique topology and involving between two and five relations. Examples are give in Appendix D. Each template yields 20 distinct queries based on a different GooglePlus edgeset. The specific community that is chosen for each query is chosen uniformly and without replacement from the collection of 45. Furthermore, each template includes random filter predicates on different relations throughout each query. The filter predicates take the form

$$\text{table.follower\_id}\ \%\ K = x, \quad \text{table.followed\_id}\ \%\ K = x$$

K is tuned for each template in order to control the relative size of each query output. For instance, queries with more relations tend to generate a larger result due to the presence of more FK-FK join attributes. In this situation, K is set to a larger value resulting in a more selective filter. K is set to a lower value for templates with fewer relations. x is chosen at random for each individual query, and independently from each filter within the same query. Note that the microbenchmark is comprised of self-join queries analogous to the subgraph isomorphism count problem within a single GooglePlus community [52]. While joining across different communities is generally possible, the joining columns are often insufficiently correlated to capture a significant FK-FK join blowup. Hence, we rely on self-joins.

The second dataset we use is the IMDb movie and television dataset [9]. It features several entity table as well as association tables relating entity tables to one another. The largest relation in the IMDb dataset is `cast_info` which assigns cast members to specific films and contains 59,906,495 rows. The workload we use with the IMDb dataset is the popular Join Order Benchmark (JOB). The JOB features 33 unique join topologies. Each topology yields several queries by swapping in different filter predicates on base relations to create a new unique query. The JOB consists of 113 unique queries in total [91].

3.2.2 Implementation

All experiments are run on a modified Postgres 9.6.6 instance [122]. Note that the only modification is to allow the optimizer to use our bounds instead of calling the traditional cardinality estimation infrastructure. The bounds are calculated via an external module. The postgres parsing, and execution layers remain unchanged. We use the non-cryptographic Murmur3 algorithm with varying seed values as our hash function [12]. We highlight that our proposed modification is typically a lightweight change leaving almost all of the remaining database engine untouched.

3.2.3 Progressive Bound Tightness

We demonstrate that increasing the partitioning budget may significantly tighten our bounds using the googleplus microbenchmark for partition budgets 1, 8, 64, 512, and 4096. For comparison, we also include the results using Postgres' default query optimizer. Our metric is Relative Error (RE) which we define as follows:

$$\text{RE}(\text{truth}, \text{estimate}) = \begin{cases} \infty & \text{truth} = 0 \text{ \& estimate} > 0 \\ 1.0 & \text{truth} = 0 \text{ \& estimate} = 0 \\ \frac{\text{estimate}}{\text{truth}} & \text{else} \end{cases}$$

Figure 3.6 includes histograms of the RE between the true cardinality, and the estimate or bound of each materialized join and subjoin. Each histogram operates on a logarithmic bucket scale on the x-axis. We observe that as the partitioning budget grows, the observed bounds shrink dramatically. At budget 4096, the majority of estimates hover just above the true value indicating it is likely impractical to increase budget any further. We also note that Postgres does not strongly favor underestimation in these experiments. This is because the queries from the microbenchmark include filters on arbitrary IDs that do not pertain to the real world. They therefore do not introduce the inherent correlation one might expect from a community graph. Note that Postgres' estimates cast a much wider logarithmic RE distribution than using bounds. This suggests the RE between different subqueries can be significantly larger and can lead to poor join orderings.

Finally, we demonstrate that using these bounds translates into plan execution time improvement. For each of the above partitioning budgets and default Postgres, we include an aggregate average plan execution time. This value is the sum of the average plan execution time of five runs for each query in the microbenchmark. Ahead of each such run of five repetitions, we execute an initial un-timed cache "warm-up". The resulting aggregate plan execution times may be found in Table 3.1. While allocating a larger partitioning budget does lead to faster plan execution time, the benefits are not particularly dramatic. In order to better demonstrate the runtime-practicality of our approach over standard cardinality estimation methods, we pivot to the more realistic JOB.

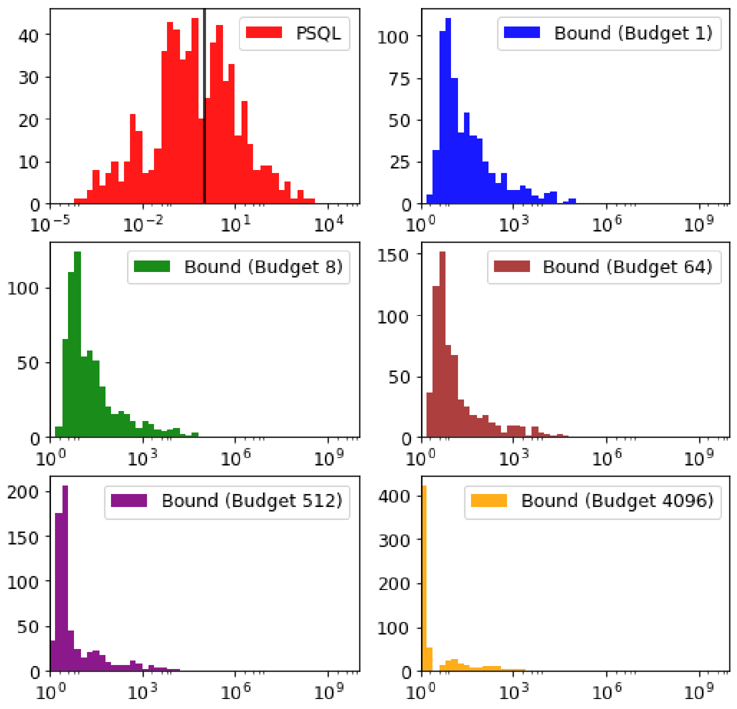
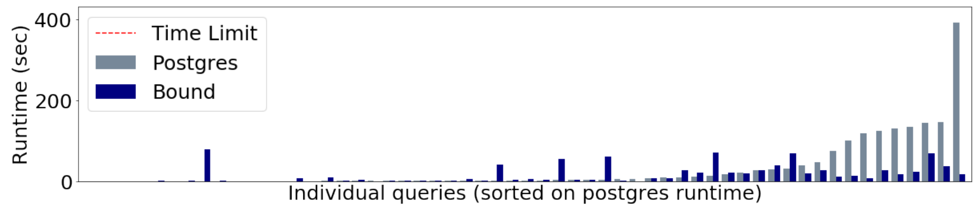


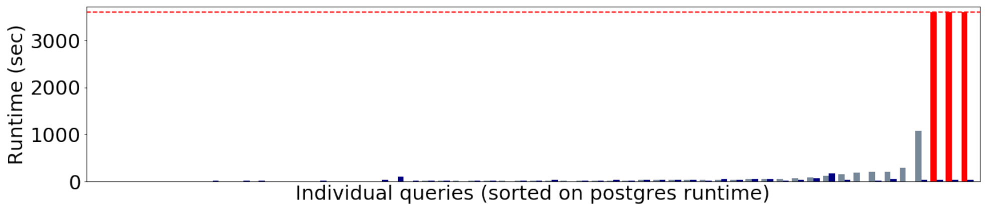
Figure 3.6: Relative Error of the default Postgres optimizer (labeled *Postgres*), and of our bounds for an increasing hash partitioning budget over the GooglePlus microbenchmark. Using bounds with partitioning budget one corresponds to the traditional KNS bound.

3.2.4 Plan Execution Time Improvement

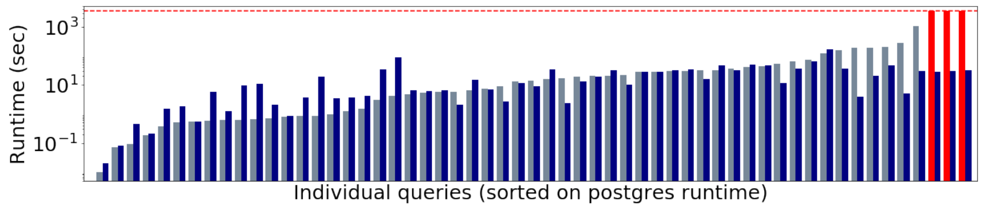
We demonstrate that our pessimistic query optimization strategy generates more robust query plans. As an initial comparison, we have modified a Postgres instance to use our bound values directly as cardinality estimates instead of the default Postgres query optimizer's estimates. We investigate two common settings: when the database has precomputed FK indexes, and when it hasn't. We relate this comparison to one between experienced and inexperienced users: an experienced user will precompute these indexes dependent on the nature of her workload, while an inexperienced user might not. As in many other systems, Postgres FK indexes are only precomputed through explicit user defined commands and not set as a default. Query plan execution times in the precomputed FK index setting may be found in Figure 3.7a. Query plan execution times in the FK index absent setting may be



(a) Linear scale JOB plan execution time when FK indexes available.



(b) Linear scale JOB plan execution time when FK indexes unavailable.



(c) Log scale JOB plan execution time when FK indexes unavailable.

Figure 3.7: Individual JOB query plan execution times for Postgres optimizer using Postgres' default cardinality estimation (gray) versus optimizers using cardinality bounds (blue). Queries highlighted in red represent those default Postgres plans that failed to complete ahead of the one hour time limit (No plans generated using bounds hit time cutoff). Each graph independently sorts on default Postgres cardinality estimation plan execution time. For readability, only approximately half of all 113 JOB queries are displayed.

Optimizer Setting	Aggregate Plan Runtime
Default Postgres	46.9 minutes
Budget 1	39.1 minutes
Budget 8	39.1 minutes
Budget 64	37.2 minutes
Budget 512	36.9 minutes
Budget 4096	36.8 minutes

Table 3.1: GooglePlus microbenchmark aggregate plan execution time and added preprocessing time. We vary the optimizer settings with partitioning budgets 1, 8, 64, 512, and 4096 as well as default Postgres optimizer

found in Figures 3.7b, and 3.7c. Each query is run six times in total: the first run to warm up cache and the reported time is the average of the remaining five runs.

In the presence of FK indexes, we find the aggregate plan execution time over all JOB queries to be 3,190 seconds when using the plans generated via Postgres' default cardinality estimates. In contrast, the aggregate execution time for plans generated using bounds is only 1,831 seconds, representing a saving of nearly 43%. While most queries yield approximately equivalent plan execution times for those queries that are relatively inexpensive (w.r.t. default Postgres plan execution time), using bounds yields much faster plans for expensive queries. This supports the notion that using bounds leads to more robust query optimization.

When FK indexes are stripped away, this divide between bound-generated and default Postgres-generated plans widens. This is because FK accesses have become significantly more costly and more computation must be performed at runtime. We find the aggregate plan execution time over all JOB queries to be 24,725 seconds when using the plans generated via Postgres' default cardinality estimates compared to only 2,304 seconds when using bounds. This is an order of magnitude difference.

We also place a one hour cutoff on each plan execution time: this limit is enforced on five of the default Postgres plans. Note that only approximately half of all JOB queries are included in Figures 3.7b, and 3.7c, for readability. This is why only three of the five aforementioned time-limit aborted queries appear. While we do not penalize those queries beyond the one hour mark, their actual plan execution time can be significantly longer. Some queries may fail entirely. Note that the bound-generated aggregate plan execution times with and without foreign keys are not significantly different. This suggests that bounds can help produce efficient plans even when modifying the amount of metadata (in this case, foreign key indexes).

The downside of using the BS is the increased optimization time. We populate our sketches using a naive algorithm based on the SQL query found in Figure B.1, which we feed to our modified Postgres instance. While this method is sufficient to demonstrate that more robust plans are possible, it is not optimized for efficient optimization time. In the presence of FK indexes, the additional optimization time over the JOB is 4,795 seconds. This additional optimization time is longer than execution time. Without FK indexes, the additional optimization time is 6,450 seconds. Again, the additional optimization time is longer than actual execution time but insignificant compared to the execution time for plans generated by default Postgres. These issues are corroborated by independent investigations in G-Care [117].

Further investigation into the individual plans suggests that using bounds pushes the optimizer to make more conservative planning decisions. This generally means that the optimizer is more likely to generate shallower, bushy tree plans rather than the typical left deep plan. We find that default Postgres severely underestimates intermediate join size higher up the tree. This will naturally encourage the optimizer to generate a left deep plan in order to avoid the cost of building a hash table on, or sorting some intermediate product and instead stream what it predicts will be few intermediate tuples past preexisting base table indexes. This wrong assumption is indeed where default Postgres suffers the most. Alternatively, plans based on bounds suffer the most when the bounds far overshoot and materializing intermediate products and data structures is suboptimal. Nevertheless, we argue that these mistakes are generally less costly at runtime than those associated with underestimation.

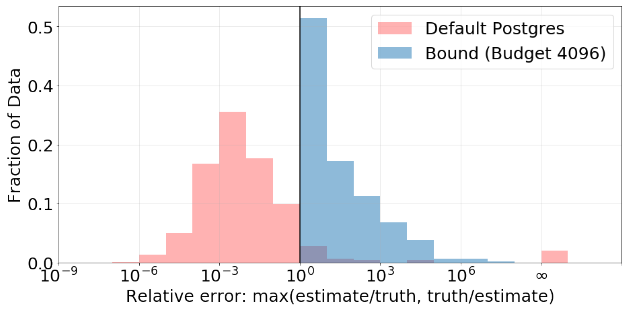
We have included a comparison between the RE for our bounds and default Postgres' cardinality estimates in Figure 3.8a. The PDFs include the relative error of bounds from all the intermediate relations appearing in either the plan produced by using our bounds or from default Postgres. Even though the object of our methods is not to produce highly accurate cardinality estimates, we still enjoy a generally lower RE than default Postgres. In order to compare the tightness of both distributions to the true value, we employ q-error [108].

$$\text{q-error}(\text{truth}, \text{estimate}) = \begin{cases} \infty & \text{truth} = 0 \;\&\; \text{estimate} > 0 \\ 1.0 & \text{truth} = 0 \;\&\; \text{estimate} = 0 \\ \max\left(\frac{\text{estimate}}{\text{truth}}, \frac{\text{truth}}{\text{estimate}}\right) & \text{else} \end{cases}$$

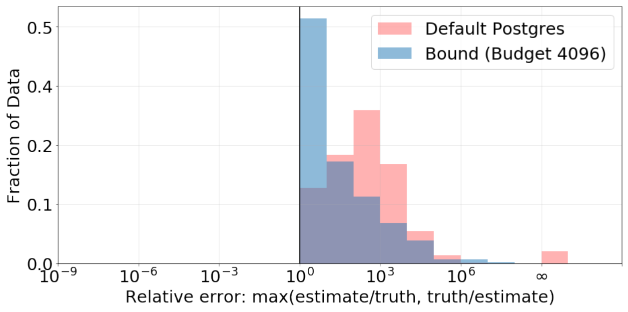
That is, q-error is equivalent to RE when the estimate is greater than the true value. When the estimate is below the true value, we invert the RE. The inverted or equivalent relationship between RE and q-error is highlighted by the similarity in log distributions (see Subfigures 3.8a, and 3.8b). Note that when using upper bounds, RE and q-error are equivalent. We include the PDFs, and CDFs of the q-error for our bounds and default Postgres' cardinality estimates in Figures 3.8b, and 3.8c, respectively. We highlight that our bounds are consistently tighter to the true value.

3.3 Conclusion

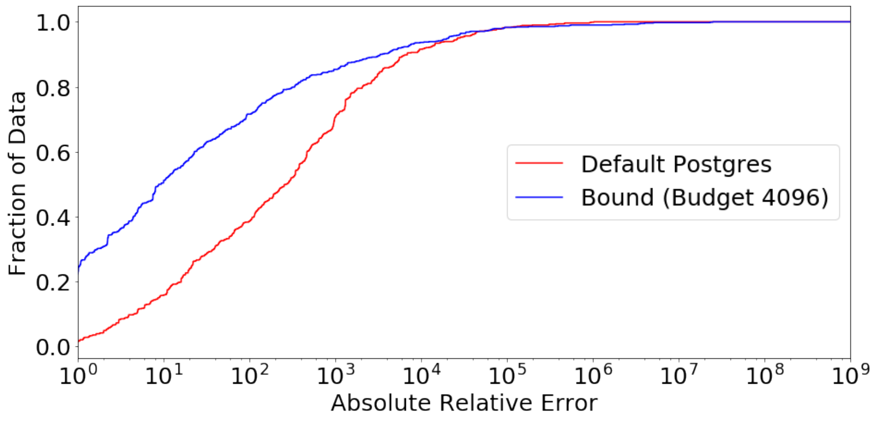
In this chapter we argue that using bounds is a legitimate strategy for achieving robust query optimization. We emphasize that our bounds are known ahead of time to be single sided (overestimates), whereas traditional methods may produce both overestimates and underestimates which will drive the RE between different intermediate relations even further apart. I.e. if intermediate relation A is overestimated by a factor of 10, whereas intermediate relation B is underestimated by a factor of 0.1, their RE between the two intermediate relations is a factor of 100. Moreover, we demonstrate using challenging workloads on real world data that our bounds outperform traditional methods on single node production systems via lightweight changes to the optimizer. Our evaluation demonstrates that the majority of our gains come from safe plans on a few minority “disaster” queries. For the remainder of queries where traditional methods are already sufficient, using bounds leads to plans that are approximately on par. In the following chapter, we extend this work and address some key shortcomings to our prototype implementation.



(a) PDFs of JOB intermediate product cardinality bounds RE (with partitioning budget 4096) versus default Postgres cardinality estimate RE.



(b) PDFs of JOB intermediate product cardinality bounds q-error (with partitioning budget 4096) versus default Postgres cardinality estimate q-error.



(c) CDFs of JOB intermediate product cardinality bounds q-error (with partitioning budget 4096) versus default Postgres cardinality estimate q-error.

Figure 3.8: JOB relative and Q-Error distributions

Chapter 4
GENERALIZING CARDINALITY BOUNDS

In the previous chapter we build the framework necessary for using tightened entropic cardinality bounds in place of traditional estimates in a Database Management System (DBMS). In doing so we combat the problem of systematic cardinality underestimation that pervades modern production systems [91]. While using cardinality bounds shows promise in single node systems, our first attempt leaves many open questions (both theoretical and practical) regarding large scale parallelized query processing. As the field moves towards flexible fleets of commodity hardware, questions regarding optimization cost, scaling/parallelizability, and adaptivity need to be addressed. Note that each of these challenges cannot be addressed in a vacuum. Many share the same root causes and hence will also benefit from the same solutions. However, for organizational purposes, we treat them separately and highlight when the issues intersect.

We consider these challenges in the following sections and preview a practical implementation. We emphasize Spark as it represents an accepted mainstream big data system designed for the parallelized environment [15, 148, 149].

4.1 Integrating Bounding into the Optimizer

In this section we detail a closer integration of our bounding technique into the standard query optimizer structure. We do so to combat arguably the greatest drawback of using our bounds; optimization overhead. I.e. the time and resources spent by the optimizer to generate a physical plan. While the prototype implementation was by no means an optimized library, the issue remains that in order to successfully execute our bounds optimization time represents a severe hit to the planning phase.

The most obvious source of increased increased optimization time comes is the handling of filter predicates at runtime. Ad hoc queries will naturally contain various selection clauses that might drastically change the attribute value distributions of different base relations. This renders precomputed bound sketches (See Chapter 3.1.1) useless and fresh sketches reflecting the filter predicate. This problem is somewhat intractable due to the unpredictability of ad hoc queries but we nevertheless present some optimizations in Subsection 4.1.1.

A more subtle source of increased optimization time comes as an impedance mismatch between entropic bounding formulas and standard join enumeration. Traditional cardinality estimation methods integrate well into the enumeration process. Entropic bounds and the associated sketches, on the other hand, do not immediately have a convenient one-to-one mapping to specific subqueries. The result is that in our prototype implementation the bounding formula is forced to be an entire separate module from the join enumerator. This separation is inefficient and leads to a more complex engineering task. In Subsection 4.1.2 we preview the typical join enumeration process. In Subsections 4.1.3 and 4.1.4 we develop a framework for integrating our bounding technique conveniently into the enumerator.

4.1.1 Pushing Filter Predicates through Bounding Formulas

The first issue we address is filter predicates. The need for base table statistics to reflect filter predicates is particularly challenging, as the only reasonable way of achieving this is to access the base data. Our initial implementation skirts the issue by executing subqueries via our filter predicate propagation described in Chapter 3.1.3. However, this is inefficient. One solution is that the results from the filter propagation should be reused in the full query execution. This is a tempting direction since pushing filters down to base relations is a good idea and will appear in the optimal query plan in almost all analytical workloads. For this reason, the cost of accessing base tables to assess the affect of filter predicates may be amortized into execution time if the results of this filter is then operated on directly by the remainder of the physical plan. This idea is similar to some sampling systems where join tuples generated during optimization are reused during execution [92]. However, this isn't a perfect elimination of repeated work as it will require the explicit materialization of the filtered base table. This materialization can be resource intensive depending on the size of the join samples. In the case of a highly selective filter, materializing the filtered table is relatively cheap. However, if the filter is not selective then materializing a filtered table that is almost the size of the original relation is prohibitively expensive and the physical plan should just access the base table directly. In the scenario where the base table has an index, it is possible that accessing the sample materialization might actually be slower. Hertzschuch et al. avoid this issue via a sampling procedure that approximates the selectivity of complex filters on base relations [68]. However, it should be noted that their procedure does not re-estimate the degree statistic reflecting the filter predicate.

Finally, materializing a subset of some intermediate relation runs counter to the tuple or mini-batch-at-a-time architecture that dominates production systems. Modern fast query execution typically does not materialize intermediate results but instead pipelines data from one operator directly to the next operator [112].

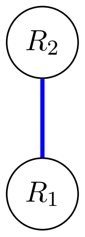
4.1.2 Join Enumeration

We shift our focus to the algorithmic features of integrating bounds into the optimizer. Much of this involves integrating the use of bounds into the standard join enumeration stage. In Chapter 2 we previewed the typical bottom-up join enumeration procedure. In this section we go into further detail on the enumerator and describe how bounding can be elegantly integrated into this enumeration procedure.

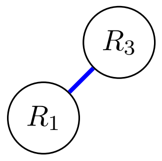
Typical the optimizer must efficiently enumerate all subqueries. Consider the following four relation query:

$$Q(x_1, x_2, a_1, a_2, a_3, a_4) :- R_1(x_1, a_1), R_2(x_1, a_2), R_3(x_1, x_2, a_3), R_4(x_2, a_4) \tag{4.1}$$

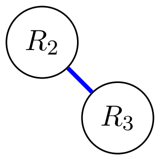
Note that the x_i are the only join attributes in the above query. The a_i are dummy attributes included to allow for repeated tuples in the relations. A graphical representation for the above



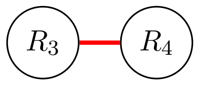
(a) R_1, R_2



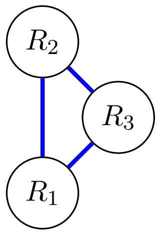
(b) R_1, R_3



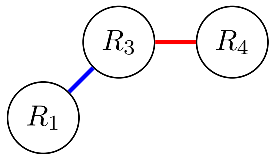
(c) R_2, R_3



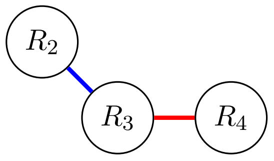
(d) R_3, R_4



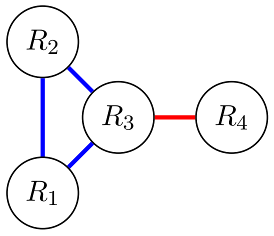
(e) R_1, R_2, R_3



(f) R_1, R_3, R_4



(g) R_2, R_3, R_4



(h) R_1, R_2, R_3, R_4 (full join)

Figure 4.1: All subqueries for query in Equation 4.1. Blue edges represent joins on attribute x_1. Red edges represent joins on attribute x_2.

query along with the graphical representations for all subqueries not including cross products may be found in Figure 4.1. The full query graph representation appears in Figure 4.1h.

Optimizers avoid cross products because cross products have necessarily multiplicative size and will likely be impractically large. Instead, the optimizer wisely opts to take advantage of the filtering property of each join predicate.

In Figure, 4.1, we proceed in the order typical of a bottom-up query enumeration process. We begin with subqueries of size two (joins of base tables). These comprise Figures 4.1a, 4.1b, 4.1c, and 4.1d. In this phase the optimizer must choose which join algorithm to use if the subquery would appear as part of the final join plan for the full query. This generally involves the consideration of which physical join algorithm to use, which base table should be the “inner” relation and which should be the “outer”, as well as keeping track of special properties that are required and/or a byproduct of certain join orders. These properties include special orderings of the output data as well as controlling the partitioning of the input and output tuples. The optimizer will settle on a prospective best plan for each binary subquery. The optimizer then proceeds by building three-table subqueries in much the same way. These queries are detailed in Figures 4.1e, 4.1f, and 4.1g. However, for the three-table subqueries, the optimizer will have more choices on how to construct the subquery plan than with two-table subqueries. For example, consider the R_1, R_2, R_3 subquery in Figure 4.1e. The optimizer can build a plan by first taking the plan for R_1, R_2 and then joining the intermediate product with R_3. Alternatively, it can start with the plan for R_1, R_3 and then join R_2. The last option is to start with the plan for R_2, R_3 and then join R_1. Each is a valid option and will lead to a semantically equivalent relation. But again the optimizer must settle on a single "best" plan that it would use were that subquery to appear as part of the final join tree. As query topology grows more complex, subqueries will also become more complex and will likely yield several valid join plans from which to choose from. In this way, the optimizer builds queries of size k on top of the plans it has generated for all subqueries of size $< k$. At the end of the join enumeration procedure, the optimizer will have settled on a single plan built on the plans it has generated for all possible subqueries. I.e. the optimizer will construct a single final plan for the full query (Figure 4.1h). This dynamic programming approach is exhaustive and is guaranteed to deliver an optimal plan assuming the optimizer always guesses costs correctly.

Note that there exist alternatives to bottom-up join enumeration such as top-down/transformational methods [60, 59]. Many of these systems employ heuristics to prune the plan space; sacrificing exhaustive search for increased search efficiency and extensibility. However, in almost all cases, the enumeration process is still predicated on the iterative application of binary join algorithms to generate the final query output. Thus, while we focus on the exhaustive bottom-up search, the extension to other join enumeration procedures should be straightforward.

4.1.3 Bounds Integration

Clearly, cardinality/cost estimation and join enumeration are interwoven procedures. In this section, we describe how to replace standard cardinality estimation with our bounding strategy. Consider the following four relation chain join expressed in datalog notation.

$$Q(x, y, z, w) :- R(x), S(x, y), T(y, z), W(z)$$

Consider further the following two bounding formulas:

$$c_R \cdot d_S^x \cdot d_T^y \cdot d_W^z \tag{4.2}$$

$$d_R^x \cdot c_S \cdot d_T^y \cdot d_W^z \tag{4.3}$$

Assume that we use a partition factor of 4 for each of the x, y, z join attributes. The associated summations for the bounding formulas in Equations 4.2, and 4.3 may be found in Equations 4.4, and 4.5, respectively. Note that these formulas are instantiations of the bounding formula model found in Equation 3.9.

$$\sum_{0 \le h_x, h_y, h_z < 4} c_{R^{h_x}} \cdot d^x_{S^{h_x,h_y}} \cdot d^y_{T^{h_y,h_z}} \cdot d^z_{W^{h_z}} \tag{4.4}$$

$$\sum_{0 \le h_x, h_y, h_z < 4} d^x_{R^{h_x}} \cdot c_{S^{h_x,h_y}} \cdot d^y_{T^{h_y,h_z}} \cdot d^z_{W^{h_z}} \tag{4.5}$$

However, this method repeats calculations. These repeated computations are understood most easily when the formulas are rewritten in matrix multiplication notation. In Equations 4.6, and 4.7 we include the equivalent multiplicative matrix computation (filled in with numeric values) for Equations 4.4, and 4.5, respectively.

$$\underbrace{\begin{bmatrix}20 & 50 & 0 & 10\end{bmatrix}}_{c_R}\underbrace{\begin{bmatrix}1 & 1 & 3 & 2\\0 & 3 & 1 & 5\\1 & 2 & 1 & 6\\4 & 4 & 1 & 2\end{bmatrix}}_{d_S^x}\underbrace{\underbrace{\begin{bmatrix}1 & 1 & 4 & 0\\2 & 1 & 1 & 3\\0 & 0 & 1 & 3\\3 & 1 & 1 & 2\end{bmatrix}}_{d_T^y}\underbrace{\begin{bmatrix}3\\0\\1\\5\end{bmatrix}}_{d_W^z}}_{=\begin{bmatrix}7 & 22 & 16 & 20\end{bmatrix}^T} = 13160 \tag{4.6}$$

$$\underbrace{\begin{bmatrix}1 & 4 & 0 & 2\end{bmatrix}}_{d_R^x}\underbrace{\begin{bmatrix}10 & 20 & 60 & 10\\0 & 10 & 20 & 40\\10 & 40 & 10 & 20\\30 & 50 & 10 & 10\end{bmatrix}}_{c_S}\underbrace{\underbrace{\begin{bmatrix}1 & 1 & 4 & 0\\2 & 1 & 1 & 3\\0 & 0 & 1 & 3\\3 & 1 & 1 & 2\end{bmatrix}}_{d_T^y}\underbrace{\begin{bmatrix}3\\0\\1\\5\end{bmatrix}}_{d_W^z}}_{=\begin{bmatrix}7 & 22 & 16 & 20\end{bmatrix}^T} = 10370 \tag{4.7}$$

Note that matrix multiplication operation $d_T^y \cdot d_W^z$ appears in both Equations 4.6, and 4.7. Ideally, we should only need to run this calculation once. In some sense this is analogous to common subexpression elimination [124]. However, for ease of implementation, this is neglected in our initial single node prototype. Instead, we loop over all combinations of partition index values (See Equation 3.5).

The ideal arrangement would be to integrate our bounding algorithm with the typical join enumeration procedure. This design choice will help with ease of implementation for other systems and followup research since modern systems already blend the join the enumeration and the cardinality estimation processes. This is simple because most optimizers handle cardinality estimation by breaking the join into the logical cross product and filter predicate operators. The cross product is calculated by taking the product of the cardinality (estimates) for both sides of the join. The selectivity of the filter predicate is then estimated by direct accesses to the base columns. However, this will cause issues when calculating our bounds because direct column statistics likely will not match column statistics for the same logical column in the intermediate relation. For example, the degree statistics we store for attribute y in relation T will likely not match the statistics for attribute y in intermediate relation $T \bowtie_z W$. This is because following joining with W, some rows in T might not contribute to any rows in $T \bowtie_z W$ while other rows in T might join with multiple rows in W. Thus tuples can contribute to fewer or more tuples in $T \bowtie_z W$ which can effect the count and fanout statistics. Thus, if we rely on base table stats, we risk calculating invalid bounds. Static column statics cannot capture this. Hence we must associate richer information with logical intermediate relations during join enumeration. The obvious solution is to compute a new (set of) bound sketches for each intermediate relation.

At this point, the observant reader will note that not all bound formulas may be neatly framed in terms of matrix multiplication. In fact, any join topology that isn't an example of a chain join will break the matrix multiplication structure. For instance, consider a star schema join:

$$Q(x_1, \dots, x_k, y_1, y_2, \dots) :- R(x_1, \dots, x_k), S_1(x_1, y_1), \dots, S_n(x_k, y_k)$$

The relation R contains greater than 2 join attributes and neither the count nor fanout portion of the bound sketch portion can be denoted as a matrix. Instead, it must be formulated as a k-dimensional tensor where each dimension corresponds to one join attribute. This naturally clashes with the concept of traditional 2D matrix multiplication. Another example that breaks standard joins is where greater than two relations include the same join attribute. Consider the following example:

$$Q(x, y, z, w) :- R(x, y), S(x, z), T(x, w) \tag{4.8}$$

While each relation may be encoded as a 2D matrix, matrix multiplication would force us to sum away the x attribute when computing the bound sketch that would represent any two-table subquery of Q. This would eliminate our ability to use the sketch for the subquery to construct the sketch for Q since a dimension mapping to the x attribute would be necessary to combine with the remaining third base relation. Instead of vanilla matrix multiplication, we must fall back on the more general *tensor contraction* operation which allows for summing over arbitrary finite many join attributes [25, 144]. Tensor contraction takes m tensors $T^1, \ldots, T^m$ as input. We assume there exists a set of attributes A and each tensor T^i takes dimensions over some subset $A^i \subseteq A$. That is, each tensor T^i has dimension $|A^i|$. We further assume the A^i are exhaustive; $\cup_i A^i = A$. Finally, suppose there exists a subset of attributes $\mathcal{A} \subseteq A$ (possibly empty) over which we marginalise (sum over). The tensor contraction produces a new tensor of dimension $|A \setminus \mathcal{A}|$ whose entries (indexed by the attribute set $A \setminus \mathcal{A}$) may be computed as follows:

$$X_{A \setminus \mathcal{A}} = \sum_{\mathcal{A}} \left(\prod_i T^i_{A^i} \right) \tag{4.9}$$

Observe that matrix multiplication can be written as a specific instance of tensor contraction. Matrix multiplication requires there are two input tensors of dimension two (matrices) who share exactly one attribute. Moreover, this shared attribute is the lone member of the $\mathcal{A}$ subset over which we marginalise.

Recall the example in Equation 4.8. Tensor contraction allows us to marginalise the x attribute over all three relations R, S, T at once or can handle the inclusion of each joining relation one by one. It does so by maintaining a set of “necessary” remaining join attributes (i.e. those attributes that would join with other relations later on) and not marginalising away still necessary attributes. We establish the two-dimensional tensors for each of the base relations above. For simplicity, we abuse notation by overloading relation names to also refer to the associated tensors.

$$[R_{i,j}]\,,\quad [S_{i,k}]\,,\quad [T_{i,\ell}]$$

Let the i, j, k, ℓ indexes correspond to the x, y, z, w attributes, respectively. Assume the query in Equation 4.8 is a subquery of some larger query. Assume further that the necessary attributes are y, z, w. Therefore we wish to generate a 3-dimensional tensor to represent Q during query optimization. Ideally, the tensor will take the form

$$Q_{j,k,\ell} = \sum_i R_{i,j} \cdot S_{i,k} \cdot T_{i,\ell} \tag{4.10}$$

However, as we have established earlier, the plan enumeration process proceeds one additional relation at a time. So we cannot jump immediately into a three relation join. We must first calculate a two relation subquery with which we should be able to build Equation 4.10. For illustrative purposes, let's consider the subquery involving only the R and S base relations.

$$Q'(x, y, z) :– R(x, y), S(x, z)$$

The bound sketch tensor for Q' should take the form:

$$Q'_{i,j,k} = R_{i,j} \cdot S_{i,k}$$

Note that we do not marginalise away the i index (corresponding to the x attribute). This is because the x attribute is the join variable for $Q'(x, y, z)$ and $S(x, w)$. We can now construct the Q tensor from the Q' and S tensors as follows:

$$Q_{j,k,\ell} = \sum_i Q'_{i,j,k} \cdot T_{i,\ell} = \sum_i (R_{i,j} \cdot S_{i,k}) \cdot T_{i,\ell}$$

Thus, using careful applications of tensor contractions, we may maintain necessary sketches for each subquery and fully integrate bounding into the standard exhaustive join enumeration procedure. This integration allows the optimizer to generate bounds without repeated work and with minimal overall changes to existing enumeration procedures resulting in faster optimization time.

4.1.4 Further Integration Optimizations

In this section we describe further optimizations of integrating bounding into the join enumeration task. In particular we make a calculated decision to continue ignoring certain bounding formulas in favor of less state stored at each logical subquery.

We first note that our bounding formula is stable w.r.t ordering of tensor contractions. This is a byproduct of tensor contraction being an associative operation. Consider the explicit bound calculation in Equation 4.6. The result will be the same if we begin by calculating $c_R d_S^x$, or $d_S^x d_T^y$, or $d_T^y d_W^z$. Thus, given a subquery and a bound formula for that subquery, it would not make sense to recalculate the same bound for every possible pairing of subquery plans that would produce the desired subquery.

We avoid this issue by only storing a count-sketch with each logical subquery plan. Consider a subquery of size k. We first iterate over all subqueries of size $k-1$ that are one relation away from the desired subquery. We may then generate the desired bound by taking the minimum over all bounds derived from adding the missing relation to each of the size $k-1$ subplans. Note that some size $k-1$ subplans will yield repeat bounding formulas.

In this manner we will necessarily miss some valid entropic bounding formulas. In particular, we will miss any bounding formula where the associated entropic formula features multiple relations that cover their attributes unconditionally. This translates to only considering cardinality bounding formulas with a single count statistic. This is a heuristic choice but will almost certainly not affect the overall minimum bound. When considering acyclic joins, having multiple count statistics will guarantee the appearance of cross products (direct or indirect multiplication of count statistics) in the bounding formula. Similar to how typical optimizers eschew cross products because they generally lead to large intermediate products, it is unlikely that a multi-count-statistic bounding formula will be tighter than competing single-count-statistic bounding formulas. Thus, this heuristic choice is highly unlikely to affect the performance of our bounding formula in practice. Note that the above procedure of gleaning a minimum count sketch for a query of size k from the count sketches of all subqueries of size $k-1$ is guaranteed to cover all single count statistic bounding formulas. This is done either explicitly, or implicitly via the same minimization that occurred for smaller subqueries.

There are two primary benefits to unifying our bound partitioning and associating a single count sketch with each logical plan. The first is that less data needs to be stored per logical plan. In our previous implementation, there would exist a single partition budget which could dictate different partitioning allotments for each bounding formula. For each relation, each unique partition budget allotment with respect to that relation would require a single count sketch and one degree sketch per join attribute. Instead, we now store a single count sketch per subquery and all degree statistics are integrated in from base relations. Their tensor contraction would again represent another count sketch to be stored with the resulting plan. While this may at first seem like a move towards greater memory utilization for each subquery, note that these sketches would still have to be calculated in our previous bound calculation model. The difference is that the sketches associated with common subqueries would often be recalculated several times. We are therefore trading slightly more storage overhead for significantly greater computational efficiency.

The second primary benefit is finer tuned minimization. That is, we may now return to localized minimization over logical partitions of the underlying data. Consider the following chain join query and subquery:

$$Q(x,y) :- R(x), S(x,y), T(y)$$
$$Q'(x,y) :- R(x), S(x,y)$$

Assume that both attributes x, y have a partition budget of 2. Assume further that the count and fanout (with respect to x) tensors for relation R and S appear as follows:

$$c_R = \begin{bmatrix} 1 & 99 \end{bmatrix}, \quad d_R^x = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad c_S = \begin{bmatrix} 1 & 99 \\ 1 & 99 \end{bmatrix}, \quad d_S^x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Let's consider the count sketch that we may produce for $R \bowtie_x S = Q'$ by using the bound formula $c_R d_R^x$ versus the count sketch that produced by $d_R^x c_S$:

$$c_R d_R^x = \begin{bmatrix} 1 & 99 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 100 & 100 \end{bmatrix}$$

$$d_R^x c_S = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 99 \\ 1 & 99 \end{bmatrix} = \begin{bmatrix} 2 & 200 \end{bmatrix}$$

It isn't immediately clear which count sketch is better. Depending on the distribution of values in the d_T^y tensor, either could be a superior candidate to store with the Q' subquery during join enumeration. However, we observe that both tensors describe the same partitioning of the Q' intermediate relation. Specifically, the first position count statistic in the tensor derived from formula $c_R d_R^x$ as well as the first position count statistic in the tensor derived from formula $d_R^x c_S$ are both bounds on the number of tuples in Q' whose y attribute hashes to 0. Similarly, the second position count statistic in the $c_R d_R^x$ as well as the second position count statistic in the $d_R^x c_S$ are both bounds on the number of tuples in Q' whose y attribute hashes to 1. Thus we may take the element-wise minimum of both tensors to create the final tensor for $c_{Q'}$:

$$c_{Q'} = \min\left(c_R d_R^x, d_R^x c_S\right) = \min\left(\begin{bmatrix} 100 & 100 \end{bmatrix}, \begin{bmatrix} 2 & 200 \end{bmatrix}\right) = \begin{bmatrix} 2 & 100 \end{bmatrix}$$

In effect we allow each logical partition of Q' to pick whichever bound formula happens to work best independent of other logical partitions. The left position chooses $c_R d_R^x$, and the right position chooses $c_R d_R^x$. In more detail, by decoupling a partition budget from a specific bound formula along with restricting intermediate products to only store count sketches and not degree sketches, we recapture the fine grained minimization that we sacrificed in our first iteration described in Chapter 3. Thus we get to keep the best-of-both-worlds from both the bound formula in Equation 3.5, as well as the bound formula in Equation 3.9.

4.2 Distributed Fanout

The natural follow-up to improved optimization time is the ability to scale. That is, how can the bounds approach be generalized to large scale distributed database settings? We take this to mean that base relations are horizontally partitioned; the rows from a single relation are partitioned across multiple nodes. This is generally a solution to massive datasets that cannot fit on a single server. This is in contrast to vertical partitioning where different columns are stored on each server. We often refer to such a subset of rows residing on the same node as a *shard*. The major mechanical issue comes from the calculation of the max degree/fanout statistic. Unlike the simple count statistic, which is a linear function, the max degree statistic would require a full distributed grouped aggregation to calculate exactly. What this may look like is that each shard of a dataset would need to calculate a count for each attribute value present in the shard and then transmit the results of the grouped aggregation over the network to its neighbors. In the worst case, the number of distinct values would be on the order of the size of the entire dataset, in which case the grouped aggregation is essentially a full data *shuffle*. A data shuffle is a re-partitioning of all rows from a single relation to a new node. Often this re-partitioning is in preparation for a distributed hash join and mapping of a specific row to a specific node follows the associated hash function. Note that using partitioning will also increase the number of groups in the distributed aggregation since a separate aggregation will need to be executed for every logical partition of a relation. If a relation contains greater than one join attribute, then tuples with the same value for one column may end up sorted into different logical partitions based on the value of other join columns. This is obviously prohibitively expensive when all the bounding method really requires is the max of the grouped counts; the max degree.

The clear solution is to fall back on classical frequent item data sketches. The first major category of such sketches are counter-based sketches such as the *Misra-Gries*, *lossy counting*, *space saving*, and *unbiased space saving* sketches [106, 104, 100, 136]. This family of sketches provides absolute error bounds and members of the family are known to perform better empirically than their theoretical analysis would suggest. The second major category is linear sketches such the *count-min*, *AMS*, and *count sketch* [44, 8, 34] which can handle deletions but require greater computation to generate a max frequency approximation. A major caveat is that our upper bounds are no longer guaranteed. Instead, sketch based aggregation implies that we are using estimates of theoretically guaranteed upper bounds. Still, the fact that we target one-sided error still makes underestimation highly unlikely.

The final consideration to choosing a distributed max degree estimation sketch is the sketch's merging performance. We argue that counter-based sketches hold the upper hand over linear sketches. Linear sketches feature straightforward merging thanks to a common internal hash functions used to create sketches locally at each shard. However, it is likely that the transmitted sketch would be prohibitively expensive as it would have to have the same dimension as the ideal sketch that would be constructed on the full logical dataset and not on each individual shard. For the count-min sketch, compression such as run length encoding may be an option. However, data shards will often still produce dense count-min sketches and the storage savings from zero-runs will likely not cover the cost of compression ahead of transmission and decompression following transmission. The AMS sketch and count sketch, similar in architecture, would likely not benefit from compression as each data item will issue an update to each position in the sketch. Counter based sketches provide an equally intuitive merging process with arguably less network usage. However, they lack the same clean error guarantees following the merging process. This merging process is relatively straightforward as the sketch contributed by each shard will take the same form. That is, each shard contributes a list of attribute values that are frequent on that shard along with approximated counts. A set of sketches may be merged by simply concatenating all lists and

value	count
seattle	100
san francisco	50
atlanta	10
vancouver	5
new york	5
austin	1

(a) Shard 1 MG sketch

value	count
new york	100
san francisco	100
london	50
miami	10
seattle	5
austin	1

(b) Shard 2 MG sketch

value	count
san francisco	150
seattle	105
new york	105
london	50
atlanta	10
miami	10
vancouver	5
austin	2

(c) Merge MG sketches from shards 1, 2

Figure 4.2: Misra-Gries sketch merge example

summing any counts that are associated with the same attribute value. An example merging of two Misra-Gries sketches may be found in Figure 4.2. In this example the merge of the two sketches is not pared down to a given budget. Instead, all pairs are maintained in the merged sketch. Merging without dropping lower rank attribute values is generally paired with a single round of merging; all shards send their sketches to a single aggregating node; in this case it is assumed that the merging node has sufficient memory to store all incoming sketches. Alternatively, merging with dropping lower rank attribute values is generally paired with multiple rounds of merging; usually following a bottom up approach on a tree. It is not clear which blend of approaches is the outright superior for the purposes of approximating max degree but the simplicity of a single round of aggregation makes it most attractive.

Observe that for all datasets featuring logical partitioning, a sketch will need to be constructed for each logical partition. This will add to memory and network usage. For example, consider a single shard from some two join attribute relation R. We include an illustration where R's logical partitioning is depicted as a $k \times k$ matrix. Each logical partition will contribute a single sketch.

$$\begin{bmatrix} \text{Sketch}(R^{(0,0)}) & \cdots & \text{Sketch}(R^{(0,k-1)}) \\ \vdots & \ddots & \vdots \\ \text{Sketch}(R^{(k-1,0)}) & \cdots & \text{Sketch}(R^{(k-1,k-1)}) \end{bmatrix}$$

Note that for both counter-based and linear sketches, the greatest challenge comes from overall low skew data where noise is proportional to the size of the entire dataset and may result in relative error far beyond the actual magnitude of the true max degree. The best case scenario is to have data pre-partitioned based on join attributes. This is common practice since distributed hash joins would otherwise require reshuffling data to achieve the same physical partitioning. If logical partitioning for bounding matches physical partitioning, then the variance of the max degree estimators will be greatly decreased and in some cases distributed aggregation of sketches may be avoided entirely.

4.3 Adaptive Execution

Adaptive query optimization (AQO) has become a mainstay in production distributed systems. In brief, AQO allows for plans to be modified in the middle of execution. That is, there exists a feedback loop between the optimizer and the execution layers where accurate statistics from the executor can be used to affect the query plans moving forward [75, 19, 4, 35]. AQO can thus be considered a sibling technique to using bounds since both methods aim to achieve a more robust query plan. While using bounds focuses on avoiding disastrous plans by starting conservative and assuming a worst case scenario plan, AQO allows the DBMS to pivot or even backtrack during execution. Note that plan adaptivity is not as simple as rearranging the nodes higher up the physical join tree. This is because the execution of a physical plan is not normally executed on a node-by-node or physical-operator-by-physical-operator basis. Instead, operators are pipelined to improve efficiency and avoid the cost of materializing intermediate products [112]. When an operator requires a full materialization or more generally when an operator needs to be completed before proceeding to the next operator, this is known as a *pipeline breaker*. Pipeline breakers are extremely helpful for re-optimization techniques as these actions represents natural pauses where the completed subtree can be neatly rooted without any wasted work. Pipeline breakers are thus the natural points during execution for adaptive execution.

Luckily, AQO and bounding do not directly contradict each other and allow for relatively easy combination. In this section we discuss considerations for using both approaches in tandem. There are two general approaches to combining bounds and adaptivity. The first is to start aggressive and then transition to a more conservative plan. Alternatively, one may start conservatively using bounds and then see how severely the optimizer overestimates. If explicitly calculated cardinalities suggest that the optimizer was overly conservative, the executor may pivot to a more aggressive plans higher up the tree.

We first consider the aggressive-to-conservative approach. This would involve allowing the optimizer to fall back on strong assumptions about the underlying data for initial plan creation. The optimizer could then use bounds later on in the more smaller subquery bounding tasks. Use of bounds could also be triggered if the execution layer encounters unexpectedly large intermediate relations during runtime. One could argue that this is an ideal scenario as it will likely work well for the majority of queries. This is because both conservative and aggressive plans will likely perform similarly on these “easy” queries but conservative plans would involve more up front work and hence more work overall. Examples include data reshuffling/partitioning, sorting, and/or the collection of more in-depth statistics. If the execution went unnecessarily down an overly conservative path, these operations would be time and resource intensive wasted work. On the other hand, if the optimizer is overly aggressive and chooses a plan that might lead to catastrophic intermediate result blow-up, it is probable that the execution will need to start over and scrap any intermediate product. This is particularly dangerous if the system is allowed to progress up the join tree, dedicating significant time and resources to a suboptimal plan.

The key to aggressive-to-conservative adaptive plans then is being able to adapt early at these natural break points. This is a promising direction since the execution of leaves in a distributed setting often involves the application of filters and shuffling data appropriately ahead of initial join operators. In this case we will be able to collect necessary statistics to perform pessimistic cardinality estimation in parallel with these base table operators thus amortizing the optimization cost. Note that not all of the statistic collection will be perfectly amortized since there will be a necessary round of message passing where the statistics/sketches are aggregated. This message-passing round might not always be masked by existing shuffles in the query plan. However, even in the non-amortized scenario these messages should be relatively cheap as the data structures are designed to be highly compact. One challenge is that the original aggressive plan is built on an entirely different collection of cardinality parameters compared to the bounds. This might prompt pivoting to a completely different –now pessimistic– query plan from the plan originally started by the executor. The challenge can then be boiled down to the problem of transitioning gracefully to a pessimistic plan but not being overly eager to do so and in turn discarding previously performed work.

The second option is to begin with conservative plans and allow the optimizer to make more aggressive decisions if it becomes clear during execution that the optimizer was overly conservative. We call this the conservative-to-aggressive approach. This is a tempting model as many systems solidify a join order during initial optimization and only use adaptive execution to modify access patterns for base relations. This is the case of our target system, Spark. Spark employs a volcano style optimizer where the physical plan is optimized using transforming rules [149, 59, 60]. A rule to implement subquery join reorders would be complex and expensive. As such, dynamic join reordering is not yet enabled in the Spark catalyst optimizer. Given these limitations, we believe that it would be best to focus on this scenario: use bounds to settle on a conservative and safe join order while allowing the adaptive capabilities of Spark to pivot to faster access paths and physical join algorithms if it deems it necessary.

4.4 Conclusion

In this chapter we address key shortcomings of our initial pessimistic query optimization prototype. Furthermore, we discuss the challenges of shifting from a single node architecture to a distributed setting. The primary challenge we face is an unacceptably long optimization time associated with using entropic bounds. We address optimization time by demonstrating how bounding may be more tightly integrated into the standard join enumeration process as well as how to handle ad hoc filter predicates. We continue by discussing how one may maintain the necessary count and degree statistics in a distributed setting. Finally, we briefly discuss how bounding might interact with a sibling method; adaptive execution.

Chapter 5

THRESHOLD FUNCTIONS OVER STREAM JOINS

Joins over two or more data streams are ubiquitous in many applications. Most existing data stream management systems (DSMS's), such as Apache Spark [14] and Apache Flink [28], all support stream joins. Typical join processing techniques are the classic *symmetric hash join* algorithm and its variants [55], which need to buffer intermediate join state in main memory. This raises challenges when processing stream joins with memory constraints, e.g., Internet of Things (IoT) applications that run on *edge* devices such as Raspberry Pi or cell phones, or when the join state is very large [11].

In this chapter, we study one common class of stream queries in practice, namely applying a threshold function over a stream join to compute an alert. Our main observation is that if the threshold function satisfies the mathematical property of *quasiconvexity*, then we can drastically reduce the memory required for the join. Our technique omits tuples from the input streams without affecting the query result.

Consider machinery on a manufacturing assembly line. A common requirement is that machines remain dry, since water would interfere with their operation. To detect when equipment is in danger of collecting moisture, the maintenance staff monitors three sensors: relative humidity h, air temperature a, and surface temperature s. The sensors emit measurements for these values in three separate data streams H, A and S. Values from these streams are then combined using the Magnus Formula [130]:

$$\phi = \ln\left(\frac{h}{100}\right) + \frac{18.678a}{257.14 + a} - s \tag{5.1}$$

If ϕ exceeds 0, the surface temperature of the machinery has dropped below the dew point of the air. This implies water will condense at a higher rate than it evaporates, causing water to collect on the surface of the machinery. This should trigger an alarm.

The Magnus formula should be applied to triples of sensor readings that are within a time window of each other, that is, tuples in the result of a temporal join. For example, if the window size is ten seconds and stream elements $s \in S$, $h \in H$, and $a \in A$ are within ten seconds of each other, then the function should be applied to the triple of readings $[s, h, a]$ to determine if they exceed the dew point threshold.

Our primary observation is that for a broad class of threshold functions we do not need to keep all tuples. Many tuples can simply be omitted without any knowledge of tuples from other streams and without ever missing an alarm.

During normal operation, the memory savings from omitting these tuples is significant, because the factory may have thousands of machines, each with its own sensors. The savings is also important during abnormal operation. For example, suppose that the humidity sensor output significantly lags the machines' surface-temperature sensor output. Then the DSMS needs to cache all of the surface temperature readings following the most recent humidity reading until their corresponding humidity readings arrive. This costs memory. Also, when the humidity readings finally arrive, the DSMS's catch-up processing risks a delay in triggering the alarm. Even worse, if the cached surface temperature readings overflow memory, the system might crash and alarms that should have been raised are permanently lost.

If tuples are only needed to trigger the alarm, then our technique can be used to filter them out on an edge device, thereby avoiding the expense of sending them to a datacenter. However, sometimes it may be necessary to capture all tuples in persistent storage for later analysis. In this case, even though all tuples must be read in, most of them can be streamed immediately to storage. Only the small number that are needed for the threshold calculation will consume main memory for a non-negligible time period.

To illustrate this intuition, consider a simple case where we have only two streams $R \bowtie S$ and the threshold formula is $f(r, s) = r + s$. Assume that tuples $r \in R$ and $s \in S$ join if their timestamps are within a time window of length $\omega/2$ from each other. This is known as an

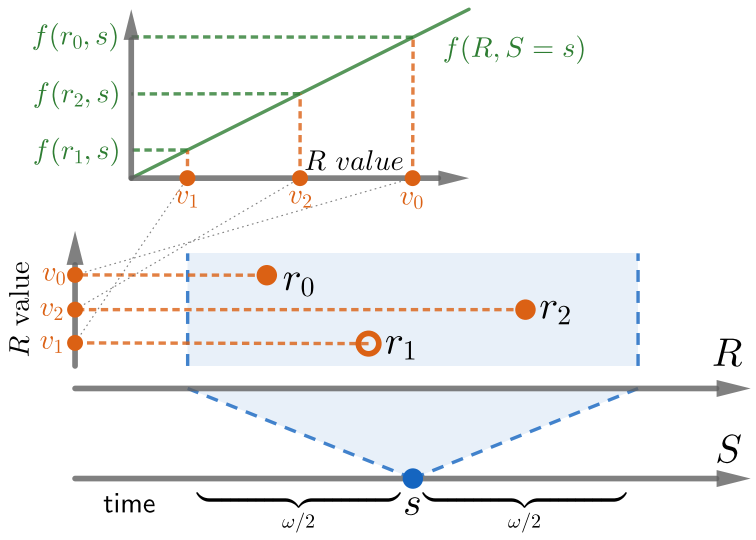


Figure 5.1: Simple tuple omission depiction.

interval join and is explained in detail in Section 2.5.1. Suppose that R sees a sequence of tuples (t, v), where t is a timestamp and v is the value of a sensor reading.

$$r_0 = (t_0, v_0), \quad r_1 = (t_1, v_1), \quad r_2 = (t_2, v_2)$$

Assume that $t_0 < t_1 < t_2$, $\{r_0, r_1, r_2\}$ are within ω of each other (i.e., $t_2 - t_0$ ¡ ω), $v_1 < v_0$, and $v_1 < v_2$. (See the bottom of Figure 5.1.) Observe that if r_1 joins with a tuple s from stream S, then s must join with at least one of r_0 and r_2. Since r_0 carries a higher value, if $f(v_1, s) = v_1 + s > \mathcal{T}$, then $f(v_0, s) = v_0 + s > v_1 + s > \mathcal{T}$. (See the top of Figure 5.1.) Similarly for r_2 and $f(v_2, s)$. Since at least one of $r_0 \bowtie s$ and $r_2 \bowtie s$ will be passed to the threshold function f, r_1 is redundant. That is, if we omit r_1, an alarm is still raised and the monitoring system still accomplishes its goal. The remainder of this chapter formalizes and generalizes this simple intuition.

Our contributions are as follows

1. We show that for a large class of threshold functions over the result of a temporal join of two streams, the system needs to retain very few tuples per-data-stream in each

time interval yet never miss a alarm.

2. We prove that our technique is optimal in the sense that it deletes as many tuples as possible.

3. We generalize our technique to multi-stream joins and show for which join topologies it does and does not work.

4. We provide experimental evidence that validates the technique's substantial memory savings.

The chapter is arranged as follows. Section 5.1 describes our omission algorithm in greater detail as well as its limitations. Section 5.2 generalizes our omission algorithm to the multi-stream join environment. Section 5.3 explores the question of automatically checking if a threshold function is amenable to our omission algorithm. Section 5.4 reports on experiments that evaluate our method against synthetic and real world workloads. Section 2.5.2 discusses past work and how it relates to our contribution.

5.1 Method

The basic form of our problem is as follows: We are given two streams R and S. Each tuple of R takes the form $r = (t_r, v_r)$ where t_r is the event time and v_r is the value. Similarly, each tuple of S takes the form $s = (t_s, v_s)$. We are also given a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ where f ingests values taken from an interval join $R \bowtie S$. Our goal is to trigger an alarm whenever there exists some tuples $r \in R$ and $s \in S$ that join and $f(v_r, v_s) > \mathcal{T}$ for a given threshold $\mathcal{T}$. Going forward, we will often abuse notation and write $f(r, s)$ instead of $f(v_r, v_s)$.

Our primary observation is this: if f is quasiconvex w.r.t. S, and there exist tuples $s_1, s_2, s_3 \in S$ each of which joins with some tuple $r \in R$, then we may omit the tuple of S with middle value. That is, if $v_{s_1} \leq v_{s_2} \leq v_{s_3}$ then as far as tuple r is concerned, we may omit s_2. The reason is that quasiconvexity guarantees that if $f(s_2, r) > \mathcal{T}$, then at least one of

$f(s_1,r)$, $f(s_3,r)$ also exceeds $\mathcal{T}$ and an alarm will be raised even with s_2 omitted. Stated formally, if f is quasiconvex w.r.t. S, then

$$f(s_2,r) > \mathcal{T} \Longrightarrow \Big(f(s_1,r) \geq f(s_2,r)\Big) \vee \Big(f(s_3,r) \geq f(s_2,r)\Big).$$

This observation generalizes to a sufficient condition for omitting tuples from the maintained state of S independent of any specific tuple from R: for all tuples $s \in S$, if there exists a pair of tuples that occurs before and after s, have values *greater* than v_s, and occur within ω of each other, and similarly there exists a pair of tuples that occurs before and after s, have values *less* than v_s, and occur within ω of each other, then we may omit s from S's cache. We call this tuple s *doubly bracketed.* This suggests a policy for reducing the amount of memory by deleting tuples that are doubly bracketed.

This observation can be specialized for monotonic functions. In this scenario, a singly bracketed tuple may safely be omitted. More specifically, if the function is monotonic increasing (decreasing) we may omit any tuple that is bracketed by two tuples with greater (lesser) value.

Is this omission criterion "optimal"? That is, is the ability to omit future tuples unaffected by omitting a tuple s? The answer is yes. Before proving it Section 5.1.3, we define some useful terminology and present the algorithm for omitting tuples.

5.1.1 Terminology

Intuitively a tuple s is above bracketed if there exist tuples before and after s whose values are greater than v_s. We formalize this and related concepts in the following definitions.

Definition 5.1.1 (Above/Below Bracketed). *A tuple s is* above *(resp.* below*)* bracketed *iff there exist tuples s_e, s_ℓ s.t.*

1. $t_{s_e} < t_s < t_{s_\ell}$

2. $v_{s_e}, v_{s_\ell} > v_s$ *resp.* $(v_{s_e}, v_{s_\ell} < v_s)$

3. $t_{s_\ell} - t_{s_e} \leq \omega$

(I.e., e and l are shorthands for "earlier" and "later".) Tuples s_e, s_ℓ, are called an above bracket *(resp.* below bracket*). We refer to a quartet of tuples that form an above and below bracket of the same tuple as simply a* bracket*.*

Definition 5.1.2 (Maximal/Minimal)**.** *A tuple s is* maximal *(resp.* minimal*) iff there exists an interval I of length ω containing s where for all tuples $s' \in I$, $v_s \geq v_{s'}$ (resp. $v_s \leq v_{s'}$).*

Definition 5.1.3 (Maximally/Minimally Bracketed)**.** *A tuple s is* maximally *(resp.* minimally*)* bracketed *iff there exist tuples s_e, s_ℓ s.t.*

1. $t_{s_e} < t_s < t_{s_\ell}$

2. $v_{s_e}, v_{s_\ell} > v_s$ *(resp.* $v_{s_e}, v_{s_\ell} < v_s$*)*

3. $t_{s_\ell} - t_{s_e} \leq \omega$

4. s_ℓ *and* s_e *are maximal (resp. minimal)*

An above bracketed tuple cannot be maximal because any window containing s would also have to contain s_e and/or s_ℓ, implying that s is not maximal in that window. Equivalently, any maximal tuple cannot be above bracketed. Similarly, no below bracketed tuple can be minimal.

5.1.2 Greedy Algorithm

Algorithm 3 presents a greedy approach to omitting tuples using the above reasoning. As tuples arrive from the input stream, they are inserted into a generic tuple store ($\mathcal{TS}$). In most cases, the store will sort the entries on event time. If we regard the $\mathcal{TS}$ as sorted left-to-right with newest (highest timestamp) elements on the right, we can use left and right as synonyms for earlier and later.

We assume that $\mathcal{TS}$ provides a generic $\mathcal{TS}$.search(t) function that provides a pointer to the element in $\mathcal{TS}$ with largest timestamp $\leq t$. If $\mathcal{TS}$ allows duplicate timestamps, then we arbitrarily break ties by the stored value and $\mathcal{TS}$.search(t) returns the "first" such tuple.

Each tuple is stored in $\mathcal{TS}$ with four additional attributes: lb_s, la_s, rb_s, ra_s. They denote the time difference between the given tuple s and the chronologically closest known tuple on its left with value below, left with value above, right with value below, and right with value above, respectively. These values are initialized to ∞ at line 5 in the pseudocode.

When a new tuple s arrives, we probe the tree using the tuple's timestamp. Assuming $\mathcal{TS}$ allows duplicate timestamps (i.e., not a red-black tree), we first check any tuples that may have the same timestamp (lines 7 - 16). We then traverse left (lines 19 - 26) and right (lines 29 - 36) searching for above and below bracketing pairs. If s completes a bracket for a tuple in $\mathcal{TS}$, then the newly bracketed tuple is dropped from $\mathcal{TS}$. If the probe reveals a bracket of s that already exists in $\mathcal{TS}$, we omit s.

The pseudocode assumes the use of doubly-closed interval boundary semantics. For doubly-open interval boundary semantics, swap $\leq$ / $\geq$ for $<$ / $>$ (or vice-versa) at lines 19, 29 in Algorithm 3, and line 2 in Algorithm 4. We discuss mixed boundary interval join semantics in Section 5.1.4.

5.1.3 Global Optimality of Greedy Algorithm

Algorithm 3 omits a tuple when it finds an above bracketing pair and a below bracketing pair of neighboring tuples. We want to verify that doing so will not harm our chances of omitting other tuples later. First, we will consider above bracketing tuples when deciding whether to omit the tuple. It will become clear that a symmetric argument holds for below bracketing pairs and that the conditions may be considered separately. We simply evaluate the conjunction of both conditions to decide on omission.

Theorem 5.1.1. *Algorithm 3 leads to a globally optimal state.*

It is sufficient to prove that if a tuple is above bracketed, then it is also maximally

Algorithm 3 Maintains a minimal collection of necessary tuples from stream S.

```
1: procedure GREEDY($S$)
2:     $\mathcal{TS} \leftarrow$ Store[$\mathbf{t}, v, la, lb, ra, rb$]()        ▷ empty tuple store
3:     while $S$.has_next() do
4:         $s \leftarrow S$.next()
5:         $la_s, lb_s, ra_s, rb_s \leftarrow \infty$
6:         $s' \leftarrow \mathcal{TS}$.search($t_s$)
7:         while $t_s = t_{s'}$ do
8:             if $v_{s'} = v_s$ then
9:                 continue        ▷ $s$ is duplicate of $s'$: omit $s$.
10:            else if $v_s < v_{s'}$ then
11:                $la_s, ra_s, lb_{s'}, rb_{s'} \leftarrow 0$
12:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
13:            else
14:                $lb_s, rb_s, la_{s'}, ra_{s'} \leftarrow 0$
15:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
16:            $s' \leftarrow$ succ($s'$)
17:        $s' \leftarrow \mathcal{TS}$.search($t_s$)
18:        if $t_s = t_{s'}$ then $s' \leftarrow$ prev($s'$)
19:        while $(la_s = \infty \vee lb_s = \infty) \wedge t_s - t_{s'} \leq \omega$ do
20:            if $v_s < v_{s'}$ then
21:                $la_s, rb_{s'} \leftarrow \min(la_s, t_s - t_{s'}), \min(rb_{s'}, t_s - t_{s'})$
22:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
23:            else if $v_s > v_{s'}$ then
24:                $lb_s, ra_{s'} \leftarrow \min(lb_s, t_s - t_{s'}), \min(ra_{s'}, t_s - t_{s'})$
25:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
26:            $s' \leftarrow$ prev($s'$)
27:        $s' \leftarrow \mathcal{TS}$.search($t_s$)
28:        while $t_s = t_{s'}$ do $s' \leftarrow$ succ($s'$)
29:        while $(ra_s = \infty \vee rb_s = \infty) \wedge t_{s'} - t_s \leq \omega$ do
30:            if $v_s < v_{s'}$ then
31:                $ra_s, lb_{s'} \leftarrow \min(ra_s, t_{s'} - t_s), \min(lb_{s'}, t_{s'} - t_s)$
32:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
33:            else if $v_s > v_{s'}$ then
34:                $rb_s, la_{s'} \leftarrow \min(rb_s, t_{s'} - t_s), \min(la_{s'}, t_{s'} - t_s)$
35:                if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
36:            $s' \leftarrow$ succ($s'$)
37:        if $\neg$Bracketed($s$) then
38:            $\mathcal{TS}$.insert($t_s, v_s, la_s, lb_s, ra_s, rb_s$)
```

Algorithm 4 Checks if tuples s is bracketed

```
1: procedure BRACKETED($s$)
2:     if $la_s + ra_s \le \omega \wedge lb_s + rb_s \le \omega$ then
3:         return True
4:     else
5:         return False
```

bracketed, because any above bracketed tuple must have an above bracketing pair of tuples both of which are maximal. Since a maximal tuple cannot be above bracketed, it will never be omitted. This implies that the maximal tuples will always be available to above bracket the original tuple in question, and thereby justify its omission.

Lemma 5.1.2. *A tuple is above bracketed iff it is maximally bracketed.*

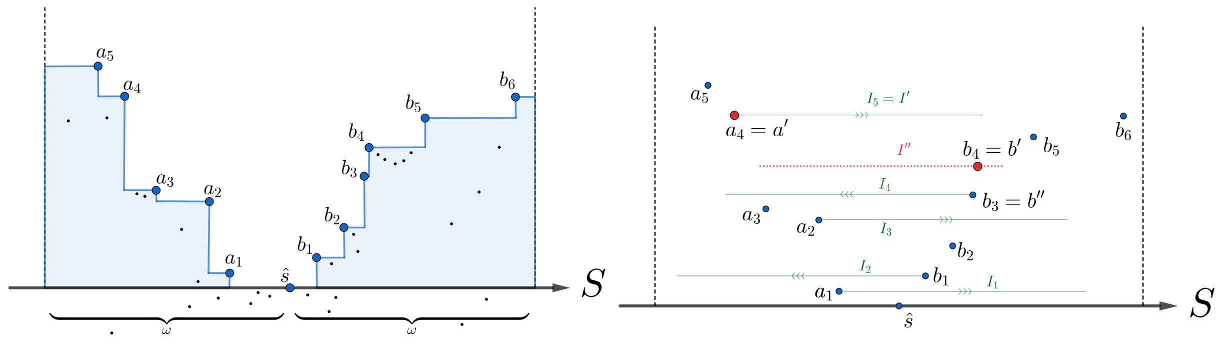
The reader may skip the proof of Lemma 5.1.2 without having it detract from understanding the rest of the chapter.

Proof. Assume that we are using closed interval semantics. The proof for open interval semantics is nearly identical. In the proof, we will refer to above bracketed tuples simply as bracketed. If two tuples have equal value, we arbitrarily choose the later one to have the "higher" value. Trivially, maximally bracketed implies bracketed. It remains to prove that bracketed implies maximally bracketed. The proof is constructive. Suppose that there exists a bracketed tuple $\hat{s}$ with bracketing pair s_1, s_2. Define two sequences of tuples $a_0, a_1, a_2, \ldots$, and $b_0, b_1, b_2, \ldots$ that extend backwards and forwards in time from $\hat{s}$. Let $a_0 = b_0 = \hat{s}$. For $i > 0$ let a_i be recursively defined as the latest element earlier than a_{i-1}, and later than $t_{\hat{s}} - \omega$, with value greater than a_{i-1}. More formally, for $i > 0$:

$$a_i = \text{argmax}_{s \in S} \left\{ t_s | t_{\hat{s}} - \omega \le t_s < t_{a_{i-1}}, v_s > v_{a_{i-1}} \right\}$$

Similarly for b_j where $j > 0$:

$$b_j = \text{argmin}_{s \in S} \left\{ t_s | t_{b_{j-1}} < t_s \le t_{\hat{s}} + \omega, v_s > v_{b_{j-1}} \right\}$$



(a) $\{a_i\}$ and $\{b_j\}$ sequences.

(b) $\{I_k\}$ sequence.

Figure 5.2: Visualization of constructive proof.

A visualization of these sequences is in Figure 5.2a. Other tuples may appear in the time-line but not all tuples end up as elements of $\{a_i\}$ or $\{b_j\}$. Although the bracketing pair s_1, s_2 might not actually be elements of the $\{a_i\}, \{b_j\}$ sequences, the existence of s_1, s_2 guarantees that the sequences are nonempty and $t_{b_1} - t_{a_1} \leq \omega$.

We construct a sequence of intervals and associated heights starting with whichever of a_1 or b_1 has a lesser value. WLOG assume it is a_1. The first interval under consideration is $I_1 = [t_{a_1}, t_{a_1} + \omega]$ with initial height v_{a_1}. Intervals (and heights) are recursively defined as follows: If there exist values from the opposite tuple sequence that fall inside the interval I_k, and whose value exceeds h_k, then we select the closest (with respect to time) such tuple as the new starting point for I_{k+1} and set the new height h_{k+1} as the value of said tuple. If the opposite tuple sequence is $\{a_i\}$, we choose the latest such a_i. If the opposite tuple sequence is $\{b_j\}$, we choose the earliest such b_j. By construction of the $\{a_i\}, \{b_j\}$, the tuple that is time-wise closest value to $\hat{s}$ on the opposite side of $\hat{s}$ must be a member of either $\{a_i\}$ or $\{b_j\}$. See Figure 5.2b.

Note that h_k is a strictly increasing sequence of values with upper bound

$$\max(\max_i(a_i), \max_j(b_j))$$

. Thus there must exist some final interval I' (with height h') where there does not exist tuples on the opposite side of $\hat{s}$ that exceed h'. WLOG assume I' is anchored on an element a' of $\{a_i\}$. In Figure 5.2b, this interval is I_5 with anchor $a' = a_4$ and height $h' = v_{a_4}$. By construction of the $\{I_k\}$ sequence, the interval contains $\hat{s}$ and must contain at least one element b' of $\{b_j\}$. This is because the previous interval was anchored at a point b where a' is within ω of b. (In Figure 5.2b, this tuple is $b' = b_4$.) By construction, a' and b' form a bracketing pair on $\hat{s}$. Furthermore, a' is maximal on the interval I'. It remains to prove that b' is also maximal. Consider the interval $I'' = [t_{a'} + \delta, t_{a'} + \omega + \delta]$ where we choose δ sufficiently small that no tuples appear in the intervals $(t_{a'}, t_{a'} + \delta]$ and $(t_{a'} + \omega, t_{a'} + \omega + \delta]$. This is only guaranteed possible when using doubly-closed, or doubly-open join interval boundary semantics. When using mixed interval boundaries (left-closed-right-open, left-open-right-closed), the existence of such a δ is not guaranteed. Section 5.1.4 presents a counterexample to Theorem 5.1.1 when using mixed boundaries, along with further discussion.

We wish to show that b' is maximal in I''. By construction, there cannot exist any tuples to the right of $\hat{s}$ whose value is greater than $v_{b'}$. Assume towards contradiction that there exists some a'' that lies inside I'' and whose value exceeds $v_{b'}$. This would imply that a'' lives in the most recent interval anchored at a tuple from $\{b_j\}$. Call this tuple b''. We have $v_{a''} > v_{b'} \implies v_{a''} > v_{b''}$. We know that such an interval exists in $\{I_k\}$ since we began anchoring at the lower of a_1 and b_1 and hence at least two intervals (at least one anchored from both $\{a_i\}$ and $\{b_j\}$) are in the sequence $\{I_k\}$. In Figure 5.2b, this tuple is $b'' = b_3$. This is a contradiction because it would imply that a'' should have been chosen to anchor I' instead of a'. Recall the anchors for the $\{I_k\}$ based on *closest* (w.r.t. time) higher value in the opposite sequence, not simply the highest. Therefore, no such a'' can exist. In more detail, no element to the left of $\hat{s}$ in the interval I'' may exceed $v_{b'}$ and thus b' is maximal in I''. Hence a' and b' are both maximal in some interval and form a maximal bracket on $\hat{s}$. $\square$

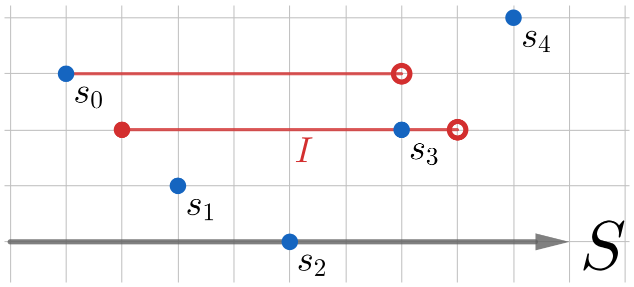


Figure 5.3: Mixed boundary counter example.

5.1.4 Mixed Boundary Interval Semantics

We present a counter example to Theorem 5.1.1 when using mixed boundary interval semantics. Let the join interval be over a left-closed-right-open interval with $\omega = 3$. Our workload consists of 5 tuples (see Figure 5.3):

$$s_0 = (0,3), s_1 = (1,1), s_2 = (2,0), s_3 = (3,2), s_4 = (4,4)$$

We wish to justify the omission of s_2. The interval $I = [0.5, 3.5)$ and pair s_1 and s_3 imply s_2 is above bracketed. It remains to show that s_2 is not maximally bracketed. Clearly, with value 1, tuple s_1 is not maximal: any interval containing s_1 must contain either s_0 or s_3. Therefore any maximal bracketing of s_2 must involve s_0. However, no interval exists that contains both s_0 and the closest tuple right of s_2, namely s_3. Therefore no maximal bracket exists and the statement of Theorem 5.1.1 fails. This workload also yields a counter example for left-open-right-closed interval join semantics.

We emphasize that it is still “safe” to deploy our omission policy here. It just lacks a guarantee of global optimality. However, the practitioner should be confident that the retained tuples are extremely close to optimal, although not easily provably so. We may simply pretend that the query is operating under doubly-open interval boundary semantics and omit tuples based on this assumption. For example, if the semantics are left-closed-right-open with interval length ω, we execute our omission strategy based on left-open-right-open interval of length ω. This might lead to slightly more tuples being retained than necessary, but the system will not suffer any false negatives while still omitting many tuples.

5.1.5 Further Discussion

We wish to highlight a few additional aspects of our greedy approach. First, our algorithm is robust to out-of-order streams. This is true both in terms of delay differences between two streams and out-of-order behavior within the same stream. In the different streams scenario two tuples r, s from different streams arrive out of order. That is, we have $t_r < t_s$ but s arrives ahead of r due to differences in stream latency. It is less obvious how tuples from the same stream may arrive out of order, but it is certainly possible, e.g., if a single logical stream is the union of multiple physical streams.

Second, our omission algorithm may be pushed down to the emitter. This saves network bandwidth, not just memory usage. Furthermore, in the event of a network failure the memory required to cache results before delivering them to the central processing node would be significantly reduced. This extends the state savings benefits across the entire system.

Finally, while we focus on joins, our omission policy may be applied in a setting where a single stream provides input parameters to the function. However, if the threshold function is quasiconvex, the calculation of the function is likely to be relatively simple, in which case it is efficient simply to execute the function at the edge node and skip any kind of state management. Still, if the execution of the function is an external service or otherwise ”expensive”, then our method may again be useful.

5.2 Generalization to Multijoins

This section generalizes our strategy to multijoin settings. In the two stream scenario, Algorithm 3 may be applied to a single stream independent of the join partner and relying only on knowledge of the interval size ω. This is because the options for join predicates are highly limited. With more streams the choice of join topology is more complex, so the omission policy must be applied with greater care.

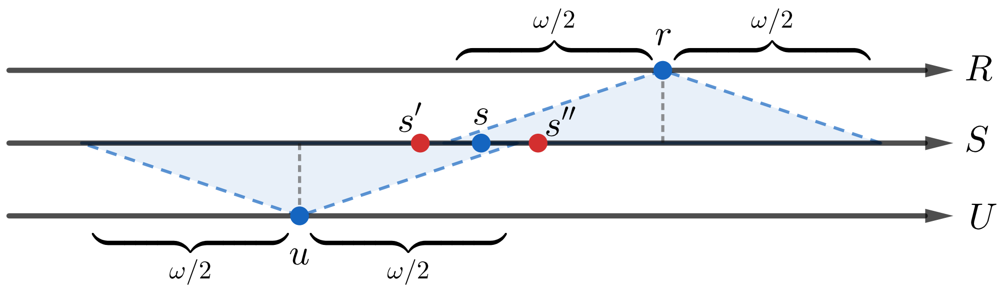
Consider the chain joins defined by the SQL query in Figure 5.4a. Streams R and U directly join to S but not to each other. This relationship is depicted in Figure 5.4b. It is not safe to omit a bracketed tuple from the state of stream S using Algorithm 3. For example, suppose tuple s is bracketed by s' and s'' and thus omitted. Tuple s' fails to join with r while tuple s'' fails to join with u. The omission of s has led to no joined output tuples being delivered to the threshold function.

However, it is still possible to apply the omission policy to tuples from streams R and U. This is because they do not have to worry about the intersections of multiple intervals.

In general, we may omit tuples from any *peripheral* streams in the join topology, that is, any stream that is temporally joined directly with only one other stream. Conversely, a stream is *internal* if it is temporally joined to more than one other stream.

```
SELECT f(R.v, S.v, U.v)
FROM R, S, U
WHERE  |R.t - S.t| <= 1
AND |U.t - S.t| <= 1;
```

(a) Multistream query SQL.

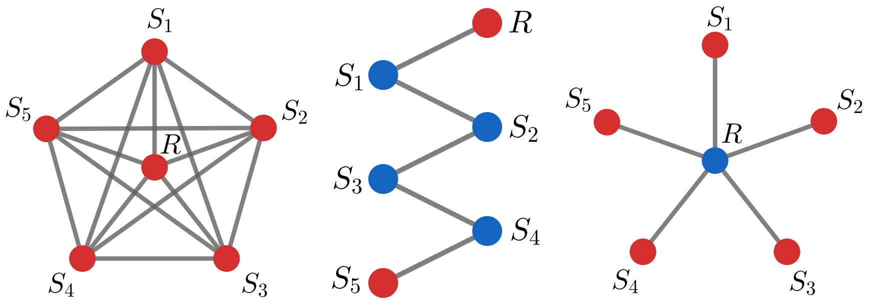


(b) Multistream interval intersection.

Figure 5.4: Multistream Query.

This suggests that join topologies heavily influence the applicability of our omission policy. For instance, a chain join only allows tuples from the two end streams to be omitted while all internal streams must be cached in full (see Figure 5.5b). However, such a chain join in the context of streams seems unlikely, since the timestamps from contributing base tuples might stretch unnaturally over the event-time space. An alternative is a star join topology where one stream is chosen to be the internal center of the star while all other streams are peripheral (see Figure 5.5c). The star query allows the application of our omission policy on all but the central stream, potentially a major savings.

Arguably the most natural join topology is where every pair of tuples contributing to a join output must be within ω of one another[1], that is, a clique query (see Figure 5.5a). One might expect a clique to be the worst case join topology for our omission policy, since there are no peripheral streams. Surprisingly however, in a clique query our omission algorithm may be applied to every stream!



(a) Clique. (b) Chain. (c) Star.

Figure 5.5: Applicability of our omission policy across different join topologies. Streams highlighted in red may omit tuples safely.

We define the neighborhood of a stream S in a join topology to be set of all of streams with which S joins directly. For any query topology, our omission policy may be applied to every stream whose neighborhood is a clique. That is, for any stream S, for any pair of distinct streams S', S'' where S joins with S' and S'' directly, then S' and S'' must join

[1]We switch from pairwise distance $\omega/2$ to ω in the multijoin scenario. The two stream scenario is a special case where we are allowed to omit tuples using intervals whose length is twice the pairwise distance bound.

directly as well in order to omit tuples from S. We formalize this statement in the following Theorem and Corollary.

Theorem 5.2.1. *Algorithm 3 may be applied safely to every stream whose neighborhood in the join topology is a clique.*

Corollary 5.2.1.1. *Algorithm 3 may be applied safely to every stream in a clique query.*

In order to prove this we start with a short lemma.

Lemma 5.2.2. *Consider a clique query $Q = (\bowtie_i S_i)$, where all joins are interval joins over an interval of length ω. For every output $(\bowtie_i s_i)$ of Q, there is an interval I of length ω that contains the timestamps of all tuples in $(\bowtie_i s_i)$.*

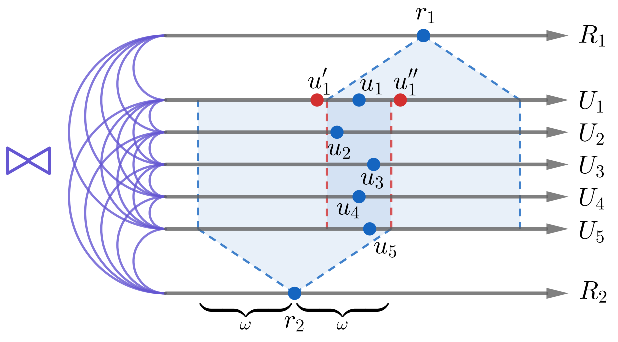
Proof. Let t_i be the timestamp of s_i for all i. Let $t_{\min} = \min_i t_i$, the minimum over all contributing tuples' timestamps, and similarly, $t_{\max} = \max_i t_i$. By definition of the clique, $t_{\max} - t_{\min} < \omega$ and $t_{\min} \leq t_i \leq t_{\max}$ for all i. Thus, $[t_{\min}, t_{\max}]$ satisfies the definition of I. □

We now prove Theorem 5.2.1.

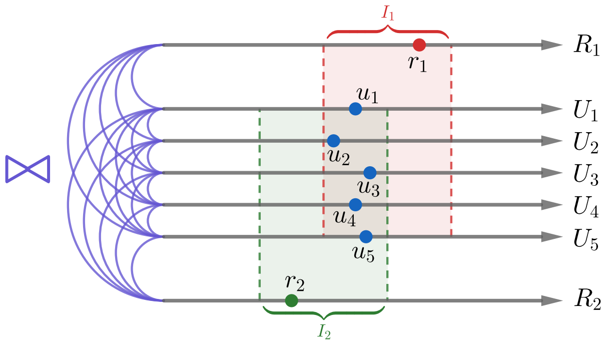
Proof. We will show that the omission of one tuple cannot lead to a false negative and generalize inductively. Consider a query $(\bowtie_i S_i) \bowtie (\bowtie_j R_j)$ where subquery $(\bowtie_i S_i)$ is a clique (the R_j need not form a clique). Let the neighborhood of stream S_1 be the clique $(\bowtie_i S_i)$. Thus, S_1 does not join directly with any R_j. Pick an arbitrary join output $(\bowtie_i s_i) \bowtie (\bowtie_j r_j)$. For each i, let t_i be the timestamp of tuple s_i. By Lemma 5.2.2, there exists an interval I of length ω where $t_i \in I$ for all i. Assume that there exists a bracket on s_1. For both the above and below bracketing pair, at least one tuple from the pair falls inside I. Therefore at least one of the four bracketing tuples (call it s'_1) may *replace* s_1 and $(s'_1 \bowtie (\bowtie_{i \neq 1} s_i)) \bowtie (\bowtie_j r_j)$ will still raise the alarm. Moreover, we know that no join predicate between pairs of streams other than S_1 has been violated since all other base tuples have been held constant. This logic may be applied inductively to any chain of omissions and subsequent replacements within the same stream, agnostic of omissions and replacements in other streams. Therefore, we need not worry about a replacement tuple being replaced itself, so this will not cause false negatives and the omission is safe. □

In a clique query the neighborhood of each stream is the entire query. Thus Corollary 5.2.1.1 follows trivially. In Section 5.2.1 we discuss a scenario where a single join predicate is dropped from a clique. Note that the neighborhood of any peripheral stream is automatically a clique subquery as the clique of size 2 only has one edge. The only difference is that cliques of size 2 allow us to use intervals of double length in our omission policy (See Footnote 1).

5.2.1 Near Clique



(a) Omitting u_1 may lead to false positive.



(b) Omitting r_1 or r_2 is safe.

Figure 5.6: Near-clique join missing predicate $(R1, R_2)$.

Consider the near clique query in Figure 5.6. Except for streams R_1 and R_2, all pairs of streams share a join predicate. Figure 5.6a shows how a false negative can occur when omitting u_1. The join tuple $x = r_1 \bowtie u_1 \bowtie \cdots \bowtie u_5 \bowtie r_2$ satisfies all join predicates and thus should be passed to the threshold function. Assume that x would trigger the alarm. Now suppose that u_1 is bracketed by tuples u_1' and u_1'' and hence omitted. Observe that u_1' joins with r_2 but not with r_1. Conversely, u_1'' joins with r_1 but not with r_2. Thus, in this scenario the omission of u_1 due to the bracket u_1', u_1'' can lead to a false negative.

On the other hand, omitting r_1 is safe, because the neighborhood of stream R_1 is $\{R_1, U_1, \ldots, U_5\}$, which forms a clique. As depicted in Figure 5.6b any bracket on r_1 will necessarily leave at least one replacement in the interval I_1. Thus we may apply our omission policy to R_1. A similar argument holds for tuple r_2.

5.2.2 Multiquery Workloads

The consideration of mulitijoin streaming queries begs the question of how to apply our omission policy in a multiquery setting. In this scenario, the central processing node is executing multiple queries and a stream may be an input to some or all of them.

Suppose that each query is amenable to our omission policy individually. That is, it includes a threshold function applied to the join of some stream inputs. Consider some stream S. We will explain how each query that takes S as input enforces a minimal interval size restriction on S. Recall that the set of tuples that are maintained for interval size ω is the set of all tuples for which there exists some interval I of length ω in which the tuple is minimal/maximal. I.e. the set of all minimal and/or maximal tuples. Let this subset of tuples be S_ω. For any tuple, if there exists an interval I' of length $\omega' > \omega$ in which the tuple is minimal/maximal, then there must exist some interval $I \subset I'$ which has length ω and in which the tuple is still minimal/maximal. Therefore, the set of tuples that need to be maintained for some interval size ω is a superset of those tuples that need to be maintained for any larger interval size $\omega' > \omega$. More formally,

$$\omega' > \omega \implies S_{\omega'} \subseteq S_\omega$$

Thus, the shorter the interval size, the more tuples we have to maintain to guarantee no false negatives. This implies that if we have two queries both operating on some stream S, it suffices to maintain the necessary tuples to whichever query is applying the smaller interval size, i.e. the more restrictive query. By the argument above, all necessary tuples for the less restrictive query are maintained as a byproduct of maintaining tuples for the more restrictive query.

If the query is a simple join of two streams and computes a quasiconvex function with respect to both inputs, then both streams must maintain min and max values for any interval of length 2ω where ω is the pairwise distance bound to guarantee no false negatives. If the query is a clique join of three or more streams, then all streams must maintain min and max values for any interval of length ω (see footnote 1). If the query is a non-clique multijoin, then any stream whose neighborhood is a clique must again only maintain min and max values for any interval of length ω, where ω might differ from one query to the next. Finally, for any stream S, if there is a query Q whose threshold function is not quasiconvex with respect to S or if S's neighborhood in Q's join graph is not a clique, then we will need to maintain every tuple from S. That is, the query enforces a length 0 interval size restriction.

For each stream, it suffices to maintain a subset of tuples corresponding to the narrowest interval size restriction. This is disappointing since the existence of less restrictive queries in the workload will not help us to reduce overall maintained state; the DSMS will still need to maintain the necessary tuples for the most restrictive query. However, this is also a "best case scenario" as maintaining the bare minimum amount of state to satisfy the most restrictive query will be sufficient for all other queries and would be required to evaluate that strict query anyway. That is, the less restrictive queries do not incur any extra work beyond what is required for the most restrictive query.

5.3 Certifying Quasiconvexity

Ensuring that a function is linear or monotonic step is simple but requires a restrictive user interface. On the other hand, general verification of quasiconvexity is difficult. Simple functions may be rewritten in innumerable complex but still equivalent ways. Even normal convexity is difficult to verify since there is no generic base formulation with which all convex functions may be written. In fact, just checking the convexity of polynomials is NP-hard [6].

One imprecise solution is to evaluate the function at every point in a discretization of the domain and check to see if quasiconvexity holds for those points. For the motivating example, this involves evaluating Equation 5.1 for each point (s,h,a) in some discretization of $[-273.15,T]\times[0,1]\times[-273.15,T]$ where we choose T to be a reasonable upper bound on a temperature reading.

A further refinement is automatically detecting if the threshold function is a (positive) linear combination of smaller *sub-functions*. Each sub-function may then be certified against the cross product of only those streams that appear in that sub-function. For the motivating example, the sub-functions are:

$$\phi_1 = -s, \quad \phi_2 = \ln\left(\frac{h}{100}\right), \quad \phi_3 = \frac{18.678a}{257.14+a}$$

This technique may be used to cut down on the exponential size of the discretization: the exponent of the complexity of the evaluation drops from the number of stream inputs to the number of streams in the sub-function with the most stream inputs.

Even if one sub-function fails, our omission policy can still be applied to some streams. It only discounts streams that were inputs to the failed sub-function. For example, if ϕ_1 fails, but ϕ_2 and ϕ_3 succeeds, then we can still safely omit tuples from H and A.

If the discretization is not sufficiently fine, then this test might miss unsafe functions. In this scenario, small dips/spikes might appear between discretization steps creating the illusion of quasiconvex behavior. However, the expense of using a finer discretization can be affordable, since this analysis need only be run once to test the applicability of a function.

5.4 Evaluation

In this section, we demonstrate the effectiveness of our omission policy empirically. We first briefly describe our implementation, then compare different tuple stores, and finally investigate raw tuple retention.

5.4.1 Implementation Details

Many DSMS's assume that the data is timestamp-ordered after ingestion [32, 28]. In these cases, the first step in a query plan is a stateful *incremental sort* operator that sorts the input during ingestion [31]. With this in mind, we prototyped our omission policy as an online sorting algorithm that also omits bracketed and thus unnecessary tuples. We call it a *threshold sorter*. This design allows us to plug the component easily into an existing DSMS. The simplicity of the bracketing condition implies that the threshold sorter requires only a few hundred lines of code to implement, not including the skip-list implementation [74].

5.4.2 Data Sets

We evaluate our method against both synthetic and real world data. The synthetic data streams are comprised of (timestamp, value) pairs where the timestamps are chosen uniformly at random from time range $[0, T)$. The pairs are sorted into chronological order. Values are drawn iid from a uniform distribution over some range. We refer to such a dataset as "synth-uniform" or simply s-unif. Alternatively, we may choose values non-independently. We simulate samples from a Wiener Process where the time discretization corresponds to the already chosen random timestamps. We refer to such a dataset as "synth-wiener" or simply s-wiener.

Disordered behavior may be introduced to a stream by adding random noise to the timestamp to simulate disparity between event time and processing time. We use Gaussian noise and vary the standard deviation parameter to increase or decrease the degree of disorder. The degree of disorder can be measured using the "inversion" rate measure: the fraction of all pairs of tuples in the stream that are disordered (i.e., *inverted*) [84]. Formally, given some sequence of values $x_0, x_1, \ldots, x_n$, the inversion rate of the sequence is defined as:

$$|\{i < j : x_i > x_j\}| / |\{i < j\}|$$

5.4.3 Tuple Store Comparison

We compare the performance of Algorithm 3's greedy tuple retention policy using three different tuple store data structures: linked list (LL), skip-list (SL)[123], and red-black trees (RB)[24]. All data structures use timestamp as the sort key with tuple values used to break ties.

LL offers worst-case linear search/insertion/deletion. However, if the stream is in-order, then the algorithm needs to follow only a single pointer to the tail of the linked list. By contrast, SL and RB both offer logarithmic expected worst case complexity for search/insertion/deletion.

Like LL, SL offers convenient lateral traversal over neighbors. While RB also offers lateral traversal, there is potentially "wasted work" in that some nodes in the tree represent tuples that are not within the interval ω of the new tuple's timestamp but need to be traversed to reach tuples that are within the interval. (See the loops starting at lines 19 and 29 in Algorithm 3.) Moreover, RB does not support duplicate timestamps. Thus, the timestamps in synthetic streams are drawn without replacement from the timestamp range. In summary, we should expect SL to outperform LL and RB in the presence of disordered tuples and underperform LL slightly if the stream is perfectly in order.

The following experiments measure the time needed to process each dataset without any data arrival latency. That is, each implementation ingests the streams sequentially and without any waiting between sucessive tuples. We report the total time spent on the stream. In each experiment the stream consists of 10^6 tuples with timestamps drawn from the range $[0, 10^7)$ (approximately 1 tuple every 10 time units). The interval size is kept at a constant 100 units for all experiments. With these parameters, approximately 40% of all tuples are preserved after a full pass. Each experiment yields the average runtime of 5 trials. Results along with empirical inversion rates may be found in Table 5.1

While LL performs best for in-order streams, as the degree of disorder increases LL performance suffers a sharp decline. In contrast, while SL lags behind LL due to higher

Stream	sigma	inv rate	LL	SL	RB
s-unif	0 (in order)	$< 10^{-6}$	0.727	1.184	4.565
s-wiener	0 (in order)	$< 10^{-6}$	0.753	1.247	4.824
s-unif	1	$< 10^{-6}$	0.754	1.243	4.620
s-wiener	1	$< 10^{-6}$	0.747	1.200	4.754
s-unif	10	$< 10^{-6}$	0.768	1.232	4.821
s-wiener	10	$< 10^{-6}$	0.787	1.254	5.058
s-unif	100	$1.2 \cdot 10^{-5}$	0.793	1.277	4.634
s-wiener	100	$1.2 \cdot 10^{-5}$	0.853	1.344	4.984
s-unif	1000	0.000111	1.031	1.373	4.801
s-wiener	1000	0.000102	1.113	1.440	4.861
s-unif	10000	0.001078	5.103	1.645	4.910
s-wiener	10000	0.001181	5.279	1.666	4.994
s-unif	100000	0.011324	84.341	2.594	6.036
s-wiener	100000	0.011216	80.047	2.522	6.131

Table 5.1: Stream processing times in seconds for linked-list (LL), skip-list (SL), and red-black tree (RB) tuple store implementations on synthetic datasets with varying Gaussian noise standard deviation (sigma). Stream size 10^6, timestamp range $[0, 10^7 - 1)$, interval length 100.

bookkeeping costs of the underlying tuple-storage data structure, the logarithmic probe complexity leads to graceful performance degradation as the stream becomes increasingly disordered. RB follows a similar performance trend as SL with respect to degree of disorder in the stream but consistently lags behind SL because it traverses more nodes than is strictly necessary. Thus, if the stream is guaranteed to be in order, LL is the obvious simple choice. However, if the stream might be disordered, SL is the more resilient data store.

Another consideration is trimming data. Often streaming query engines will drop data that is no longer relevant. This is the case in normal operation when failures and network partitions are not experienced. The engine assumes that no tuple will arrive with a timestamp earlier than some punctuation. While SL and RB will both find that cutoff point in logarithmic time, RB will have to re-balance the tree after trimming. LL will again take linear time to find the cutoff point. Both SL and LL can simply drop all tuples left of the punctuation, a constant time operation. In this scenario SL has a clear theoretical advantage over LL and RB.

5.4.4 Tuple Retention Savings

While the ability to deploy the algorithm is dependent on the join topology and threshold function, the execution is entirely agnostic of external streams. Thus to demonstrate how effectively our omission policy reduces the state, it suffices to demonstrate it on a single stream. We evaluate the effectiveness of our approach on multiple streams as described below.

- **DEBS 2012 Grand Challenge** (DEBS) is a dataset consisting of monitoring data from manufacturing equipment [73]. We demonstrate the effectiveness of our omission policy by pairing the given timestamps with the value of column `mf01`: the electrical power main phase sensor reading. The stream consists of just over 32 million tuples.

- **Gamma-Wiener** is a synthetic dataset consisting of 10 million tuples. Timestamps start at 0 with each subsequent timestamp adding an independent draw from a Gamma distribution. Values are taken from a Wiener process with the discretization chosen with respect to the timestamps.

For each stream, we assume that the value is an input parameter to a quasiconvex threshold function. Recall that the specific nature of the stream or any joining stream is irrelevant given that the threshold function is quasiconvex. We wish to demonstrate that the performance of our omission policy depends mostly on the number of tuples that appear inside any given interval. As we vary interval length, we expect the proportion of omitted tuples to vary approximately linearly with respect to the average number of tuples per interval.

Longer intervals are semantically more inclusive: one would expect the true join output to grow exponentially with respect to the interval length, where the exponent is the number of streams being joined. However, longer intervals also lead to more base tuples being omitted

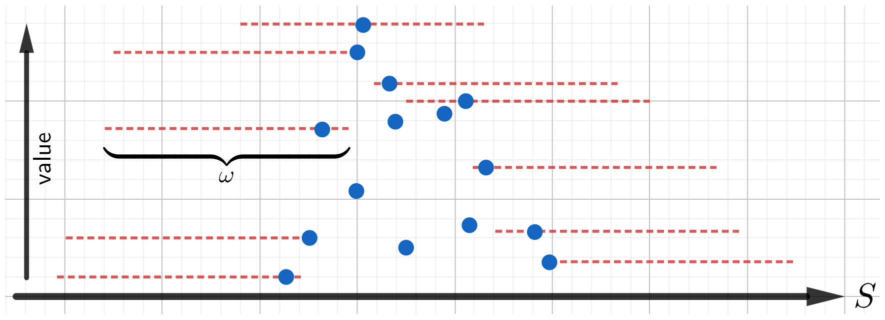
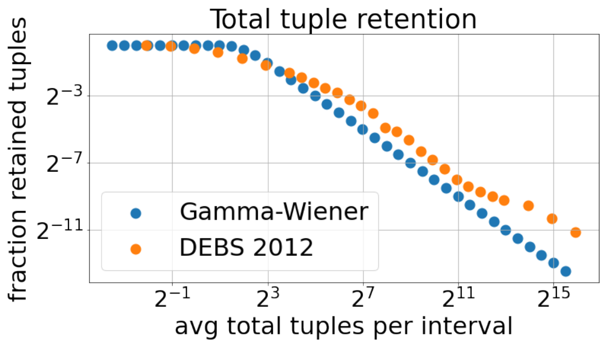
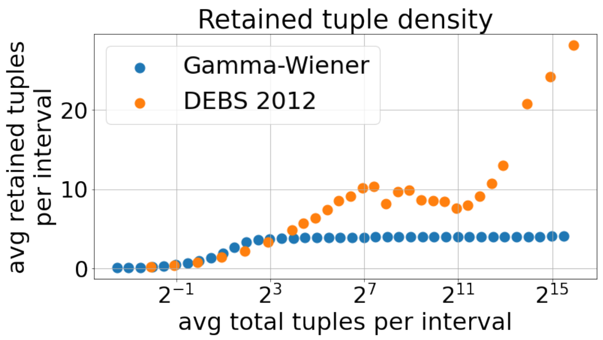


Figure 5.7: Retained tuples in the presence of significant time gaps. Each retained tuple is paired with an interval in which that tuple is maximal thus forcing the inclusion of the tuple. Of the 15 total depicted tuples, we are forced to retain 10.

as the range of tuples that may comprise a bracket grows. In the following experiments we omit tuples if they are both above and below bracketed. The results are in Figures 5.8a and 5.8b.



(a) Total tuple retention.



(b) Average per interval tuple retention.

Figure 5.8: Tuple omission performance when varying interval length.

Results for Gamma-Wiener are noticeably uniform, because large time gaps and longer sustained shifts in the value parameter are less likely. For very narrow intervals, the true number of tuples per interval is exceedingly small. Hence the fraction of retained tuples over total tuples is very high (lefthand side of Figure 5.8a), while the average number of tuples that are retained per interval is very low: there just are not enough tuples per interval. However, once the total tuples per interval grows to about four, the fraction of retained tuples drops dramatically. This trend is most easily understood in Figure 5.8b where the average number of retained tuples per interval for Gamma-Wiener flat-lines at four. Given that we omit a tuple when it is above and below bracketed, this average of four tuples per interval is not surprising. Intuitively, each omitted tuple requires one above and one below bracket. If one draws a loose connection between an omitted tuple and an interval it represents, the four points in the above and below bracket are the 4 retained tuples per interval.

Results are less uniform for DEBS. As one might expect, when intervals are exceedingly narrow and on average few tuples appear for any given interval, the fraction of total retained tuples stays close to one since there are rarely enough close neighbors to form a bracket.

Notice that DEBS begins omitting tuples at a lower average tuple density than Gamma-Wiener. This is because the timestamp distribution for DEBS is highly skewed with large gaps between clusters of timestamps.

Clustering tuples increases the probability of forming brackets and omitting tuples. While this timestamp skew helps at lower density, it actually does the opposite at higher average tuple densities. Observe in Figure 5.8b that DEBS seems to level off around 8 retained tuples per interval. This is partially because the large timestamp gaps create boundary effects that force us to retain disproportionate numbers of tuples just before and just after these gaps. Simply put, the lack of tuples in a time gap often eliminates the ability to construct brackets forcing us to retain more tuples. Thus, disproportionately many tuples are retained near the boundaries of time gaps. An example scenario is in Figure 5.7.

As we increase interval length, often these gaps may be bridged and the time gap boundary retained tuples decrease proportional to the total tuples per interval. This is characterized as the slight downward trend in average retained tuples per interval for DEBS between 2^7 and 2^{11} tuples per interval. However, as intervals grow but still fail to bridge larger gaps, many time gap boundary retained tuples remain. Thus the number retained tuples per interval also increases proportionately. This explains the spike in average retained tuples per interval for DEBS on the right hand side of Figure 5.8b. However, at this point the average tuples per interval is very large making this scenario unlikely in practice.

Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

In this dissertation we present novel work in applying data sketching techniques during the optimization and execution of join queries. We emphasize that joins are a fundamental data operation, which still cause headaches for production systems.

For analytic multijoin queries we argue that existing methods fail to capture complex correlations across tables. Typical production systems rely on strong assumption about the underlying data which leaves the executor vulnerable to severe cardinality underestimation from the optimization process. Instead we champion the use of theoretically guaranteed cardinality upper bounds by demonstrating how to tighten existing entropic bounding formulae. This marks the newest practical advancement in entropic bounds and their relationship with relational joins. Moreover, our experiments suggest that using bounds is a possible key to robust query optimization; a goal of multijoin and multiquery optimization that has been sought after for decades. We go on to outline key challenges with our tightened bounding approach and how one may rectify these issues. These challenges include tighter integration with existing join optimization procedures, the approximation of max fanout in a distributed setting, as well as the interaction between bounds and adaptive query optimization and execution techniques.

Our work invites further experimentation and optimization across a broader collection of workloads and architectures. In particular, future work might include practical implementations in existing production systems such as Apache Spark, Amazon Redshift, Microsoft SQL Server, and Snowflake [149, 63, 61, 45]. While we outline solutions for the most obvious challenges of such an implementation, other lower level of issues such as the exact partitioning size/strategy and interactions with physical data layouts still remain. Other future directions include a hybrid optimization strategy which can be used to deploy bounds when there is indication that traditional methods might generate disastrous plans. The challenge with this hybrid strategy is defining the exact nature of this disaster indicator. In order to test these practical implementations, more challenging workloads will also need to be developed and made widely available. While the IMDb workload showcases the dangers of correlated and skewed data from the real world, the raw size of the dataset is insufficient for testing the big data systems of the future. For now, the largest real world datasets are proprietary and unlikely to be released publicly by the large corporations that maintain them.

We also explore the problem of joins in the streaming data model where a combination of unreliable connectivity, limited resources, and the demand for low latency often lead to system failures. In the common case of threshold functions applied to the result of temporal joins, we present a theoretically optimal tuple omission policy that may be executed at low computational overhead. The omission policy is a pure push-down method that may be applied before the join even at the stream emitter leading to decreased network usage, as well as memory usage at the central DSMS. Experiments demonstrate that the empirical behavior approximately follows theoretical analysis. Furthermore, our omission policy may be generalized to the multijoin and multiquery setting.

While threshold function on stream joins is a narrower area than analytical query optimization, we emphasize that it is a growth market. In fact, the global market for IoT devices –a driving factor behind stream query processing– is expected to reach close to 2 trillion dollars by 2028 [53]. From a practical standpoint, our tuple omission policy can benefit from further optimizations to the greedy omission algorithm, as well as an explicit implementation of our proposed quasiconvexity detection algorithm. Furthermore, more thorough testing on real world threshold stream join workloads (as versus experiments on expected single stream performance) would highlight the costs and benefits of our framework.

Final Remarks: The main takeaway from this dissertation is that fundamental principles still govern data processing. Understanding these constraints is necessary to making systems robust and performant. Our focus on joins highlights that even longstanding problems can be solved or at least mitigated with carefully constructed and deployed data sketches. We believe that future work will extend these principled methods to issues that will inevitably arise not just in the processing of joins but across the entire data processing stack.

BIBLIOGRAPHY

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 277–289. www.cidrdb.org, 2005.

[2] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 181–192, New York, NY, USA, 1999. Association for Computing Machinery.

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

[4] S. Agarwal, S. Kandula, N. Bruno, Ming-Chuan Wu, I. Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *NSDI*, 2012.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[6] Amir Ahmadi, Alex Olshevsky, Pablo Parrilo, and John Tsitsiklis. Np-hardness of deciding convexity of quartic polynomials and related problems. *Mathematical Programming*, 137, 12 2010.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[8] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 20–29, New York, NY, USA, 1996. Association for Computing Machinery.

[9] Amazon. *IMDb*, 2018.

[10] Amazon. *Amazon Timestream*, 2020.

[11] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.

[12] Austin Appleby. *SMHasher*, 2008.

[13] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. PIQL: success-tolerant query processing in the cloud. *CoRR*, abs/1111.7166, 2011.

[14] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *SIGMOD*, pages 601–613, 2018.

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

[16] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Similarity search on time series based on threshold queries. In *EDBT*, pages 276–294, 2006.

[17] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. T-time: Threshold-based data mining on time series. In *ICDE*, pages 1620–1623, 2008.

[18] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[19] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

[20] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, 2004.

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

[22] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, page 350, USA, 2004. IEEE Computer Society.

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

[24] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1(4):290–306, December 1972.

[25] Richard L. Bishop and Samuel I. Goldberg. *Tensor analysis on manifolds*. Dover, 1980.

[26] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 18–35, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Walter Cai, Philip A. Bernstein, Wentao Wu, and Badrish Chandramouli. Optimization of threshold functions over streams. *Proc. VLDB Endow.*, 14(6):878–889, February 2021.

[28] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[30] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 45(6):363–375, 2010.

[31] B. Chandramouli, J. Goldstein, and Y. Li. Impatience is a virtue: Revisiting disorder in high-performance log analytics. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 677–688, 2018.

[32] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, December 2014.

[33] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 203–214. VLDB Endowment, 2002.

[34] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, page 693–703, Berlin, Heidelberg, 2002. Springer-Verlag.

[35] S. Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[36] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. *SIGMOD Rec.*, 28(2):263–274, June 1999.

[37] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. *SIGMOD Rec.*, 23(2):161–172, May 1994.

[38] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 161–172, New York, NY, USA, 1994. Association for Computing Machinery.

[39] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 379–390, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 759–774, New York, NY, USA, 2017. ACM.

[41] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 63–78, New York, NY, USA, 2015. Association for Computing Machinery.

[42] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[43] Graham Cormode. Sketch techniques for approximate query processing. 2010.

[44] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

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

[46] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 40–51, New York, NY, USA, 2003. Association for Computing Machinery.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[48] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, October 2020.

[50] Cristian Estan and Jeffrey F. Naughton. End-biased samples for join cardinality estimation. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 20–, Washington, DC, USA, 2006. IEEE Computer Society.

[51] Leonidas Fegaras. A new heuristic for optimizing large queries. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, DEXA '98, pages 726–735, London, UK, UK, 1998. Springer-Verlag.

[52] Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and B.V. Raghavendra Rao. Faster algorithms for finding and counting subgraphs. *J. Comput. Syst. Sci.*, 78(3):698–706, May 2012.

[53] Fortune Business Insights. *Internet of Things (IoT) Market Worth USD 1,854.76 Billion by 2028; Critical Need to Virtually Monitor Operations to Boost Growth*, 2021.

[54] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. Prediction-based geometric monitoring over distributed data streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 265–276, New York, NY, USA, 2012. Association for Computing Machinery.

[55] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[56] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[57] Google. *GooglePlus*, 2017.

[58] Google. *Holt-Winters' Seasonal Method*, 2020.

[59] G. Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

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

[61] Jim Gray. Microsoft sql server. January 1997.

[62] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 289–298, 2006.

[63] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1917–1923, New York, NY, USA, 2015. Association for Computing Machinery.

[64] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proc. VLDB Endow.*, 11(4):499–512, December 2017.

[65] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1035–1050, New York, NY, USA, 2020. Association for Computing Machinery.

[66] J. Hellerstein. What goes around comes around by michael stonebraker. 2004.

[67] J. M. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, Kris Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23:7–18, 2000.

[68] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Simplicity done right for join ordering. In *CIDR*, 2021.

[69] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2013.

[70] Robin John Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice.* OTexts, Australia, 2nd edition, 2018.

[71] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, September 1984.

[72] influxData. *InfluxDB*, 2020.

[73] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. The debs 2012 grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, page 393–398, New York, NY, USA, 2012. Association for Computing Machinery.

[74] David Jeske. Bdskiplist. last accessed 2020-09-30.

[75] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery.

[76] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Rec.*, 27(2):106–117, June 1998.

[77] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. Cuttlefish: A lightweight primitive for adaptive query processing, 2018.

[78] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 631–646, New York, NY, USA, 2016. ACM.

[79] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[80] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. Computing join queries with functional dependencies. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 327–342, 2016.

[81] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? *CoRR*, abs/1612.02503, 2016.

[82] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *CoRR*, abs/1809.00677, 2018.

[83] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1937–1940, New York, NY, USA, 2019. Association for Computing Machinery.

[84] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[85] Ilya Kolchinsky and Assaf Schuster. Efficient adaptive detection of complex event patterns. *Proc. VLDB Endow.*, 11(11):1346–1359, July 2018.

[86] Ilya Kolchinsky and Assaf Schuster. Real-time multi-pattern detection over event streams. In *Proceedings of the 2019 International Conference on Management of Data*,

SIGMOD '19, page 589–606, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[88] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 128–137, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[90] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning, 2019.

[91] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[92] Viktor Leis, Bernharde Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[93] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*, June 2014.

[94] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 615–629, New York, NY, USA, 2016. Association for Computing Machinery.

[95] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, page 1–11, New York, NY, USA, 1990. Association for Computing Machinery.

[96] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. *SIGMOD Rec.*, 19(2):1–11, May 1990.

[97] G. Lohman. Is query optimization a 'solved' problem? 1989.

[98] Mary E. S. Loomis. The 78 codasyl database model: A comparison with preceding specifications. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*, SIGMOD '80, page 30–44, New York, NY, USA, 1980. Association for Computing Machinery.

[99] S. Madden and M.J. Franklin. Fjording the stream: an architecture for queries over streaming sensor data. In *Proceedings 18th International Conference on Data Engineering*, pages 555–566, 2002.

[100] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 346–357. VLDB Endowment, 2002.

[101] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery.

[102] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[103] Julian Mcauley and Jure Leskovec. Discovering social circles in ego networks. *ACM Trans. Knowl. Discov. Data*, 8(1), February 2014.

[104] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, page 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[105] Microsoft. *Azure Stream Analytics*, 2020.

[106] J. Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[107] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *In SIGMOD*, pages 539–552, 2008.

[108] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.*, 2(1):982–993, August 2009.

[109] Magnus Müller, Guido Moerkotte, and Oliver Kolb. Improved selectivity estimation by combining knowledge from sampling and synopses. *Proc. VLDB Endow.*, 11(9):1016–1028, May 2018.

[110] M. Muralikrishna and David J. DeWitt. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 28–36, New York, NY, USA, 1988. Association for Computing Machinery.

[111] Thomas Neumann. *Classification of Join Ordering Problems*, 2009.

[112] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[113] Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 677–692, New York, NY, USA, 2018. ACM.

[114] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, February 2014.

[115] Frank Olken and Doron Rotem. Simple random sampling from relational databases. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, page 160–169, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[116] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18, New York, NY, USA, 2018. Association for Computing Machinery.

[117] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-care: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1099–1114, New York, NY, USA, 2020. Association for Computing Machinery.

[118] Tiago P. Peixoto. *The Netzschleuder network catalogue and repository*, August 2021.

[119] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305, June 1996.

[120] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 294–305, New York, NY, USA, 1996. Association for Computing Machinery.

[121] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A. Rundensteiner. Complete event trend detection in high-rate event streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 109–124, New York, NY, USA, 2017. Association for Computing Machinery.

[122] Postgres Development Core Team. *PostgreSQL*, 2017.

[123] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[124] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002.

[125] Guy Sagy, Daniel Keren, Izchak Sharfman, and Assaf Schuster. Distributed threshold querying of general functions by a difference of monotonic representation. *Proc. VLDB Endow.*, 4(2):46–57, November 2010.

[126] S. Sujin Issac Samuel. A review of connectivity challenges in iot-smart home. In *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, pages 1–4, 2016.

[127] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[128] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32(4):23–es, November 2007.

[129] Izchak Sharfman, Assaf Schuster, and Daniel Keren. Shape sensitive geometric monitoring. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, page 301–310, New York, NY, USA, 2008. Association for Computing Machinery.

[130] D. Sonntag. Important new values of the physical constants of 1986, vapour pressure formulations based on the its-90, and psychrometer formulae. 1990.

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

[132] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, November 2019.

[133] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8(7):702–713, 2015.

[134] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 309–320. VLDB Endowment, 2003.

[135] Timescale. *TimescaleDB*, 2020.

[136] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1129–1140, New York, NY, USA, 2018. ACM.

[137] Howell Tong. Threshold models in time series analysis–30 years on. *Statistics and its Interface*, 4(2):107–118, 2011.

[138] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Scotty: Efficient window aggregation for out-of-order stream processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1300–1303. IEEE, 2018.

[139] Twitter. *Twitter*, 2017.

[140] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm, 2013.

[141] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8:1530–1541, 2015.

[142] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

[143] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The myria big data management and analytics system and cloud services. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[144] wikipedia. *Tensor contraction*, 2021.

[145] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[146] Lucas Woltmann, C. Hartmann, D. Habich, and W. Lehner. Machine learning-based cardinality estimation in dbms on pre-aggregated data. *ArXiv*, abs/2005.09367, 2020.

[147] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. *Proc. VLDB Endow.*, 12(3):210–222, November 2018.

[148] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[149] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[150] U. Çetintemel, D. Abadi, Yanif Ahmad, H. Balakrishnan, M. Balazinska, Mitch Cherniack, J. Hwang, S. Madden, Anurag Maskey, A. Rasin, Esther Ryvkina, M. Stonebraker, Nesime Tatbul, Ying Xing, and S. Zdonik. The aurora and borealis stream processing engines. In *Data Stream Management*, 2016.

Appendix A
THE AGM BOUND

The proof may be originally found [18] and is reproduced below:

Theorem A.0.1 (The AGM Bound)**.** *Consider query Q over relational schema σ and let $D = (R_1, \dots, R_m)$ be a database instance of σ. For all fractional edge covers $(u_1, \dots, u_m)$ of Q we have*

$$|Q(D)| \leq \prod_{j=1}^{m} |R_j(D)|^{u_j}$$

Proof. Due to the density of rationals in the reals, WLOG we may assume $u_j \in \mathbb{Q}$. Thus there exists $\{v_j\}$ and w s.t. $u_j = v_j/w$ for all j. Let $\sum_j v_j = V$. Consider a possibly repeating collection of subsets $\boldsymbol{a}$, the attributes appearing in Q:

$$\tilde{\boldsymbol{a}}_1, \dots, \tilde{\boldsymbol{a}}_V \in 2^{\boldsymbol{a}}$$

where the collection contains precisely v_j copies of each subset $\boldsymbol{a}_j$. That is,

$$\left| \{k : \boldsymbol{a}_j = \tilde{\boldsymbol{a}}_k\} \right| = v_j$$

We may therefore assume that for all i, the attribute $a_i \in \boldsymbol{a}$ appears in at least w elements of the collection since

$$\left| \{k : a_i \in \tilde{\boldsymbol{a}}_k\} \right| = \sum_{j : a_i \in \boldsymbol{a}_j} v_j = w \cdot \sum_{j : x \in \boldsymbol{x}_j} u_j \geq w$$

by definition of a fractional edge cover.

Finally, let X be a tuple of random variables

$$\boldsymbol{X} = (X_1, \ldots, X_m)$$

where each X_i corresponds to attribute a_i in $\boldsymbol{a}$. Let X be uniformly distributed on $Q(D)$. That is, for all tuples $t \in Q(D)$, $\mathbb{P}(X = t) = |Q(D)|^{-1}$. Because $\boldsymbol{X}$ is uniformly distributed on a space of size $|Q(D)|$ we have the entropy of $\boldsymbol{X}$

$$h[\boldsymbol{X}] = \log |Q(D)|$$

We may now apply Shearer's Lemma (A.0.2) on the variable tuple $\boldsymbol{X}$ as well as the collection of attribute subsets $\tilde{\boldsymbol{a}}_k$:

$$b \cdot \log |Q(D)| = b \cdot h[\boldsymbol{X}] \overset{A.0.2}{\leq} \sum_{k=1}^{A} h\left[\boldsymbol{X}_{\tilde{\boldsymbol{a}}_k}\right] = \sum_{j=1}^{m} v_j h\left[\boldsymbol{X}_{\boldsymbol{a}_j}\right]$$

EXPLAIN IN WORDS HOW WE APPLY SHEARER'S LEMMA. For each relation R_j, we have the marginal entropy $h[X_{\boldsymbol{a}_j}]$ of the variable tuples on those attributes appearing in R_j is bounded above by the entropy of the uniform distribution on the relational instance $R_j(D)$. We may therefore continue:

$$\sum_{j=1}^{m} v_j h\left[\boldsymbol{X}_{\boldsymbol{a}_j}\right] \leq \sum_{j=1}^{m} v_j \log |R_j(D)|$$

We conclude:

$$|Q(D)| \leq 2^{\frac{1}{w} \sum_{j=1}^{m} v_j \log |R_j(D)|} = \prod_{j=1}^{m} |R_j(D)|^{v_j/w} = \prod_{j=1}^{m} |R_j(D)|^{u_j}$$

$\square$

Lemma A.0.2 (Shearer's Lemma). *Let $\boldsymbol{X} = (X_1, \ldots, X_m)$ be a tuple of random variables and let $\tilde{a}_1, \ldots, \tilde{\boldsymbol{a}}_V$ be a not necessarily distinct collection of subsets of the index set $[m]$ where* *for each $X_i \in \boldsymbol{X}$, X appears in at least w elements of $\tilde{\boldsymbol{a}}_1, \dots, \tilde{\boldsymbol{a}}_V$. That is, for all i*

$$|\{k : i \in \tilde{\boldsymbol{a}}_k\}| \geq w$$

For each index subset $I \subset [m]$, define the variable tuple $\boldsymbol{X}_I = (X_i : i \in I)$. We may bound the entropy of X in terms of the marginal entropies of the $X_{\tilde{\boldsymbol{a}}_k}$ and w:

$$w \cdot h[\boldsymbol{X}] \leq \sum_{k=1}^{V} h\left[X_{\tilde{\boldsymbol{a}}_k}\right]$$

Appendix B

SINGLE PASS ALGORITHM TO COMPUTE BOUND SKETCH

The downside of using the BS is the increased optimization time. We populate our sketches using a naive algorithm based on the SQL query found in Figure B.1 which we feed to our modified postgres instance. While this method is sufficient to demonstrate that more robust plans are possible, it is not optimized for efficient optimization time. In the presence of FK indexes, the additional optimization time over the JOB is 4,795 seconds. This additional optimization time is longer than execution time. Without FK indexes, the additional optimization time is 6,450 seconds. Again, the additional optimization time is longer than actual execution time but insignificant compared to the execution time for plans generated by default Postgres.

The BS may calculated in a single pass following the Algorithm 5. For simplicity, we assume a binary relation R with two join variables x, y and hash partition sizes M_x, M_y. However, this method is easily generalized to any number of join variables and may be executed concurrently in the same table scan but with alternative hash partition sizes.

While this algorithm is linear in runtime, it also requires linear additional storage in the worst case. Alternatively, one degree sketch (with respect to x) as well as the count sketch could also be populated using the nested SQL query found in Figure B.1. To populate both degree sketches, a query of this form would need to be executed twice. However, it is often the case that only a single degree statistic is needed for all bounding formulas in which case the degree sketch with respect to y need not be calculated.

The primary downside of using the BS is the increased optimization time. We populate our sketches using a naive algorithm based on the SQL query found in Figure B.1 which we

Algorithm 5 Bound Sketch Generator.

```
1:  procedure BOUND SKETCH($R(x,y), M_x, M_y$)    ▷ input relation $R(x,y)$ with hash partitions sizes $M_x, M_y$.
2:      $c_R \leftarrow [0]_{M_x \times M_y}$    ▷ the count tensor, and both degree tensors are initiated as $M_x \times M_y$ tensors of zeros.
3:      $d_R^x \leftarrow [0]_{M_x \times M_y}$
4:      $d_R^y \leftarrow [0]_{M_x \times M_y}$
5:      $\mathrm{HM}_x \leftarrow$ new hashmap: $([M_y], W) \rightarrow \mathbb{Z}_+$
6:      $\mathrm{HM}_y \leftarrow$ new hashmap: $([M_x], W) \rightarrow \mathbb{Z}_+$
7:      for $t \in R(x,y)$ do
8:          $m_x = H(t[x])$
9:          $m_y = H(t[y])$
10:         $c_R[m_x, m_y] ++$
11:         $\mathrm{HM}_x(m_y, t[x]) ++$
12:         $\mathrm{HM}_y(m_x, t[y]) ++$
13:         if $\mathrm{HM}_x(m_y, t[x]) > d_R^x[m_x, m_y]$ then
14:             $d_R^x[m_x, m_y] \leftarrow \mathrm{HM}_x(m_y, t[x])$
15:         if $\mathrm{HM}_y(m_x, t[y]) > d_R^y[m_x, m_y]$ then
16:             $d_R^y[m_x, m_y] \leftarrow \mathrm{HM}_y(m_x, t[y])$
17:     return $(c_R, d_R^x, d_R^y)$    ▷ return sketches
```

```
SELECT
    R_inner.hx AS hx
    R_inner.hy AS hy,
    SUM(R_inner.cnt) AS cnt,
    MAX(R_inner.cnt) AS max_degree
FROM(
    SELECT
        hash(x) AS hx,
        hash(y) AS hy,
        x,
        COUNT(*) AS cnt
    FROM R
    GROUP BY hx, hy, x) AS R_inner
GROUP BY hx, hy;
```

Figure B.1: Nested SQL query used to generate BS.

feed to our modified postgres instance. In the case of multiple join attributes in a single table, we must submit the query multiple times. While this method is sufficient to demonstrate that more robust plans are possible, it is not optimized for efficient optimization time. In the presence of FK indexes, the additional optimization time over the JOB is 4,795 seconds. This additional optimization time is longer than the plan execution time for both default Postgres and using our bounds. Without FK indexes, the additional optimization time is 6,450 seconds. Again, the additional optimization time is longer than plan execution time for the plans generated by bounds, but insignificant compared to the plan execution time for plans generated by default Postgres.

Appendix C

EXAMPLE OF NON-MONOTONICITY OF THE DEGREE BOUND

We wish to demonstrate that the Degree-Bound formula may fail to be monotonic non-increasing as the hash size grows. That is, we will construct an explicit example where increasing the hash size will result in a higher join size bound. Note that the bounds presented below differ from the partition budgeting scheme presented in Subsection 3.1.4. Consider the following conjunctive query:

$$Q\left(x,y,z\right) \text{:-} R\left(x,y\right), S\left(y,z\right), T\left(z,w\right)$$

We populate the relational instances as follows:

$$\underbrace{\begin{array}{|c|c|} \hline x & y \\ \hline 0 & 0 \\ \hline 0 & 1 \\ \hline 1 & 0 \\ \hline 1 & 1 \\ \hline \end{array}}_{R} \bowtie \underbrace{\begin{array}{|c|c|} \hline y & z \\ \hline 0 & 0 \\ \hline 1 & 0 \\ \hline 2 & 1 \\ \hline 3 & 1 \\ \hline \end{array}}_{S} \bowtie \underbrace{\begin{array}{|c|c|} \hline z & w \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 2 & 2 \\ \hline 3 & 3 \\ \hline \end{array}}_{T} = \underbrace{\begin{array}{|c|c|c|c|} \hline x & y & z & w \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}}_{Q}$$

We begin by considering hash size 1. That is, the generic (non-partitioned) degree bound formula. We have 3 candidate bound formulas:

$$\left|Q(x,y,z)\right| \leq \min \begin{cases} c_R \cdot d_S^y \cdot d_T^z \\ d_R^y \cdot c_S \cdot d_T^z \\ d_R^y \cdot d_S^z \cdot c_T \end{cases}$$

Note the first formula above is in fact tight to the true cardinality of the join:

$$c_{R(0)} \cdot d_{S(0,0)}^y \cdot d_{T(0)}^z = 4 \cdot 1 \cdot 1 = 4$$

We consider a hash size of 2. Define hash function $h(u_i) = i\%2$. That is, simply the modulo-2 function of the attribute value. We define the exact mapping below:

$$h(0) = h(2) = 0$$
$$h(1) = h(3) = 1$$

We may now explicitly describe the hash size 2 degree bound:

$$\left|Q(x,y,z)\right| \leq \sum_{\substack{i,j \\ \in\{0,1\}}} \min \begin{cases} c_{R(i)} \cdot d_{S(i,j)}^y \cdot d_{T(j)}^z \\ d_{R(i)}^y \cdot c_{S(i,j)} \cdot d_{T(j)}^z \\ d_{R(i)}^y \cdot d_{S(i,j)}^z \cdot c_{T(j)} \end{cases} \tag{C.1}$$

This is where these formulas differ from the budgeting scheme. When using budgeting, the first formula would not introduce partitioning to attribute z and the third formula would not introduce partitioning to attribute y. We have $c_{R(i)} = 2$, $c_{S(i,j)} = 1$, and $c_{T(j)} = 2$ for all values $i,j \in \{0,1\}$. This also implies that the degree statistic for each relation, with respect to either attribute, for each partition is also at least 1. That is $d_{R(i)}^y, d_{S(i,j)}^y, d_{S(i,j)}^z, d_{T(j)}^z \geq 1$ for all values $i,j \in \{0,1\}$. Furthermore, $d_{R(i)}^y = 2$ for all $i \in \{0,1\}$. This implies that Equation

C.1 may be bounded below as follows:

$$\sum_{\substack{i,j\\ \in\{0,1\}}} \min \begin{cases} c_{R^{(i)}} \cdot d^{y}_{S^{(i,j)}} \cdot d^{z}_{T^{(j)}} \\ d^{y}_{R^{(i)}} \cdot c_{S^{(i,j)}} \cdot d^{z}_{T^{(j)}} \\ d^{y}_{R^{(i)}} \cdot d^{z}_{S^{(i,j)}} \cdot c_{T^{(j)}} \end{cases} \geq \sum_{\substack{i,j\\ \in\{0,1\}}} \min \begin{cases} 2 \cdot 1 \cdot 1 \\ 2 \cdot 1 \cdot 1 \\ 2 \cdot 1 \cdot 2 \end{cases} = \sum_{\substack{i,j\\ \in\{0,1\}}} 2 = 8$$

Observe that the bound has increased despite the fact that hash size has increased. Moreover, we have demonstrated this behavior in a strict sub-partitioning scenario. That is, each partition using hash size 2 is a subset of a partition using hash size 1. This is in contrast to a non strict sub-partitioning (i.e. where tuples are mixed between buckets during a change in hash size).

Appendix D
GOOGLEPLUS MICROBENCHMARK TEMPLATE EXAMPLES

The googleplus microbenchmark consists of aritificial multiqueries joining moderate size social media graph communities. The data consits of several clusters drawn from Google's now defunt social media platform, GooglePlus [57, 103]. Data was originally taken from Stanford SNAP [93] and may still be downloaded from Netzschleuder [118]. Benchmark queries are constructed using a collection of standardized query templates where the community is chosen at random. Random filter predicates are implemented as simple modulo predicates on the community member's id. Example queries from the benchmark may be found in Figures D.1, D.2, D.3.

```sql
SELECT COUNT(*)
FROM
    community_44 AS t0,
    community_44 AS t1,
    community_44 AS t2,
    community_44 AS t3
WHERE
    t0.object = t1.subject AND
    t1.object = t2.subject AND
    t2.object = t3.subject AND
    t0.subject % 512 = 89 AND
    t3.object % 512 = 174;
```

Figure D.1: Template 4a, Googleplus Community 44

```sql
SELECT COUNT(*)
FROM
    community_30 AS t0,
    community_30 AS t1,
    community_30 AS t2,
    community_30 AS t3,
    community_30 AS t4
WHERE
    t0.object = t1.subject AND
    t0.object = t2.subject AND
    t0.object = t3.subject AND
    t3.object = t4.subject AND
    t0.subject % 256 = 49 AND
    t1.object % 256 = 213 AND
    t2.object % 256 = 152 AND
    t4.object % 256 = 248;
    AND ci.movie_id = mc.movie_id;
```

Figure D.2: Template 5c, Googleplus Community 30

```sql
SELECT COUNT(*)
FROM
      community_5 AS t0,
      community_5 AS t1,
      community_5 AS t2,
      community_5 AS t3,
      community_5 AS t4
WHERE
      t0.object = t1.subject AND
      t1.object = t2.subject AND
      t2.object = t3.subject AND
      t2.object = t4.subject AND
      t0.subject % 1024 = 615 AND
      t3.object % 1024 = 765 AND
      t4.object % 1024 = 384;
```

Figure D.3: Template 5e, Googleplus Community 5