Adapting Plan-Based Re-Optimization of Multiway Join Queries for Streaming Data

Fangda Wang

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

IN THE SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Acknowledgments

First and foremost, I would like to express my sincere thanks to my supervisors Prof. Chan Chee Yong and Prof. Tan Kian Lee, for their inspiration, support and encouragement throughout my research progress. Their impressive academic achievements in the database research areas, especially in the area of query processing and optimizing topics attracted me to do the research work in this thesis. Without their expertise and help, this thesis would not have been possible. More importantly, besides the scientific ways to solve problems, their humble attitude to nearly everything will have a profound influence on my entire life. I am fortunate to be one of their students.

I also wish to express my appreciation to my labmates in the Database Research Lab 1, for the precious friendship. They create a comfortable and inspiring working environment, and discussions with them broadened my horizon on research as well.

I also deeply appreciate the kindness that all professors and staff in the School of Computing (SoC) have showered upon me. In the past two years, I have received a lot of technical and administrative helps and I have gained many skills and knowledge from lectures as well. I hope there are chances to make more contributions for SoC in the future.

Last but not the least, I dedicate this work to my parents. It is their unconditional love, tolerance, support and encouragement that accompanied me and kept me going all through this important period.

Contents

Summary

Exploiting a cost model to decide an optimal query execution plan has been widely accepted by the database community. When the plans for running queries are found to be sub-optimal, re-optimization techniques can be applied to generate new plans on the fly. Because plan-based re-optimization techniques can guarantee effectiveness and improve execution efficiency, they achieve success in traditional database systems. However in data-stream management, exploiting re-optimization to improve performance is more challenging, not only because the characteristics of streaming data change rapidly, but also because the re-optimization overheads cannot be easily ignored.

To alleviate these problems, we propose to bridge the gap between exploiting plan-based re-optimization techniques and reacting to the data-stream environments. We describe a new framework to re-optimize multiway join queries over data streams. The aim is to minimize the redundant re-optimization calls but still guarantee sub-optimal plans are detected.

In our scheme, the re-optimizer contains a three-phase re-optimization checking and two-path plan generating component. The three-phase checking component is performed periodically to decide whether re-optimization is needed. Because query optimizers heavily rely on information of cardinality and arrival rate to decide best plans, we evaluate them at checking duration. In the first phase, we quantify arrival rate changes to avoid redundant re-optimization. In the second phase, most recent cardinality values are considered to identify sub-optimality. Finally, in the third phase, we explicitly exploit useful cardinality information to detect local optimality. According to the decision made by the checking component, the plan generating component takes different actions for optimal and sub-optimal plans.

We explored the re-optimization performance over streaming data with different value distributions, arrival rates and window sizes, and we showed that re-optimization could offer significant performance improvement. The experimental results also showed that, traditional re-optimization techniques were able to provide significant performance improvement, if properly adapted to the real-time and constantly-varying environments.

List of Figures

List of Tables

Chapter 1

Introduction

In the last few decades, traditional Database Management Systems (DBMSs) have witnessed great success in dealing with stored data. However, nowadays we are embracing an era of data-stream management : data is generated in the form of data-value sequences at a rapid speed. Stream-based applications, such as those related with sensor networks (Yao and Gehrke, 2003), financial transactions (Zhu and Shasha, 2002) and telecommunication services (Cortes et al., 2000), need platforms to properly monitor, control and make decisions over streaming data.

1.1 Data-Stream Management

To understand the challenge in data stream management, let us begin by considering the following example.

Example 1.1.1 *As mobile applications become increasingly prevalent, it is beneficial to monitor user behavior for marketing and advertising purposes. Assume service providers would like to know users' preferred applications during a time period across a region.*

This requirement would be translated into a query, and the query runs as long as information about application usage can be collected. In case information like application identities, current locations, start time and end time is provided by different data streams, the system firstly needs to assemble (i.e., join) them to obtain comprehensive knowledge with regard to individual users. Then, further processing, such as aggregation, is needed to draw the final conclusions. For such a query that involves multiple source streams and operations, it is challenging to execute it in an efficient way: From the system side, it lacks proper estimation of incoming data, because records of users behaviors can only be known after they are gathered during execution. From the data side, data properties themselves are changing all the time. For example, applications in news or entertainment category may be widely used when people are stuck in traffic on the commute while those business related applications are popular during working hours.

From the above example, it is clear that the data-stream management is not the same as traditional DBMSs. There are many differences distinguishing data-stream management from traditional DBMSs and we describe some important aspects as follows.

- *Nature of Data:* In DBMSs, data is stored and organized well on disk, for example, tuples of the same table can be clustered according to their unique identifications at loading moments. Moreover, it is beneficial to build auxiliary structures like indexes because large-scale updates are less frequent than queries. On the contrary, streaming data are continuous, unbounded, ordered, varying and real-time. These data natures are unfavorable for systems to hold adequate statistics over streaming data in advance. As such, statistics maintained by the systems are constantly changing and are vulnerable to inaccuracy (if they are not updated regularly).

- *Query Semantics:* Traditional databases process one-time queries whose results are produced only on the basis of snapshots of the underlying data at the moment queries are submitted. However, in data stream settings, queries are continuous (Calton et al., 1999; Chen et al., 2000), that is, once registered, queries keep running and results should be constantly delivered as long as the corresponding data flows in. Since stream sources possibly have no time or length bounds, window constraints (Kang, Naughton, and Viglas, 2003; Golab and Özsu, 2003) are used to restrict processing to recent data. These subtle but important points call for re-thinking of the evaluation of data-stream queries.

- *Query Execution:* In traditional databases, data is processed in memory after it is retrieved from disks. However, responsiveness constraints in stream applications are tight such that newly-arriving tuples should be directly managed online; besides, blocking operators that must consume the entire data to produce results cannot be used. Sometimes when loads are too high to react to, accuracy is traded by using approximate techniques (Tatbul et al., 2003; Babcock, Datar, and Motwani, 2004). In addition, due to the long-running feature, continuous queries most likely encounter changes of the underlying data or system conditions throughout their lifetimes.

The above features suggest that adaptability is the most critical ingredient of a stream processing engine, that is, systems should be prepared to adjust their obsolete decisions of query execution based on how data streams and system conditions change. In fact, this is very important as queries are long running, and the use of a sub-optimal plan can result in not only poor query performance but waste of system resources. Clearly, it is infeasible to simply import data into DBMSs and then operate on them there. Therefore, Data Stream Management Systems (DSMSs) have been developed. In order to satisfy data-stream applications' needs, some DSMSs design architectures from scratch, and then develop advanced strategies according to their own specifications (Chandrasekaran et al., 2003; Ives, 2002). Meanwhile, there are another group of DSMSs. They take into account the similar SQL query language and processing operators so they inherit DBMSs' core engine that chooses minimal-cost plans to answer queries (Rundensteiner et al., 2004; Carney et al., 2002; Abadi et al., 2005; Chen et al., 2000). We focus on the latter group of systems that face a main challenge of improving adaptability. Next, we will briefly show reasons for and principle of the means of *run-time re-optimization* that traditional databases proposed to handle the adaptability issue.

1.2 Run-Time Re-Optimization

Run-time re-optimization is initially proposed in traditional DBMSs. Traditional databases exploit plan-based optimization techniques that usually depend on cost models to select minimal-cost plans (Selinger et al., 1979) for submitted queries. At implementation level, optimization techniques heavily rely on cardinalities, that is, the number of tuples of the original data as well as the intermediate results, to evaluate alternative plans. However, sometimes, accurate information of data cardinalities is not available at compile-time, for example, systems are short of appropriate knowledge of a table when it is inserted for the first time. So estimating necessary cardinalities, as a compromise, is used to decide the optimal plans. Unfortunately, it is widely recognized that estimations do not always closely match to the actual, and errors will propagate exponentially with the number of joins or the presence of skewed and correlated data distributions (Christodoulakis, 1984; Ioannidis and Christodoulakis, 1991). Therefore, initially-generated plans easily fail to live up to their potential, and most likely, the actual execution time becomes orders of magnitude slower than expected, leading to degraded system performance.

To guarantee efficiency, DBMSs employ run-time re-optimization, that is, plan execution is interleaved with optimization at run-time (Kabra and DeWitt, 1998; Markl et al., 2004; Babu, Bizarro, and Dewitt, 2005). The principle of these works is to execute plans and monitor data characterisitcs simultaneously, and invoke re-optimization to generate best plans when currently-running plans are deemed to be sub-optimal. The core techniques are explained as follows.

At compile-time, the optimizer chooses some characteristics, e.g., cardinalities of base tables. For each characteristic, the optimizer computes some thresholds according to the characteristic's current estimation. During query execution, the correct information of those chosen characteristics can be gathered. If an actual value violates a threshold, then the corresponding execution plan is considered to be sub-optimal. The execution is paused and re-optimization is invoked to generate a better plan. After that, execution is resumed with the improved plan. Run-time re-optimization can be invoked many times as long as violations occur.

In DBMSs, this plan-based re-optimization performs well. However, these techniques are proposed to deal with stored and static data instead of streaming and time-varying data. They are, unfortunately, not applicable in streaming environments.

1.3 Challenges

Theoretically, DBMSs and DSMSs all need run-time re-optimization for the sake of efficiency. However, due to differences in the underlying data and the processing requirements, challenges remain when applying existing re-optimization techniques on data streams.

Challenge (1): It is not clear whether plan-based re-optimization can work over data streams. On the one hand, DSMSs (Rundensteiner et al., 2004; Carney et al., 2002; Abadi et al., 2005; Chen et al., 2000) that use plan-based optimizers did not explicitly present the way they do re-optimization. On the other hand, careful consideration is needed when applying re-optimization over streaming data. First of all, most data streams exhibit fluctuating arrival rates and varying value distributions. Secondly, in most systems, handling streaming data is I/O-free, meaning re-optimization overhead cannot be ignored, because the gain in execution costs may not always offset the overhead. Existing re-optimization techniques are usually triggered when some cardinality values change. However in streaming environments, invoking re-optimization as long as changes are detected will cause an overhead issue, because data's time-varying feature frequently invokes re-optimization.

Challenge (2): It is non-trivial to decide the significance of cardinality changes. It is unsuitable to use ad hoc thresholds on cardinality changes, because the effect of cardinalities on query optimality is very complex. To handle this issue, existing works (Markl et al., 2004; Babu, Bizarro, and Dewitt, 2005) pre-compute, for each cardinality, an interval around its currently estimated value to represent the range of values for which the current plan remains valid. Intervals can be too narrow to tolerate any variation, and they also can be sufficiently wide such that all available variations are included. During execution, the actual values of the cardinality are collected by a statistics collection component, and they are considered as significant if they go beyond their corresponding intervals. Under this principle, sharp and big variations usually invoke re-optimization. However most likely, redundant re-optimization occurs. We illustrate this with the following example.

Example 1.3.1 *Suppose a query over streams A, B, and C has two join conditions, say, A $\bowtie$ B and B $\bowtie$ C. Initially, the arrival rates of these three streams are all 100 tuples per unit time and the optimal plan is generated according to them. During execution, it is possible for the arrival rates to have dramatic changes simultaneously, say, 500 tuples per unit time. In this case, if those changes are considered individually, pre-computed intervals most likely do not cover the new values, and then re-optimization will be triggered. However, if the data's value distributions remain unchanged, the optimal plan probably remains unchanged. From the viewpoint of effectiveness, the re-optimization effort is wasteful.*

Challenge (3): Underutilization of useful knowledge loses the opportunity to find out better plans. Most existing works (Stillger et al., 2001; Aboulnaga et al., 2004; Kabra and DeWitt, 1998; Markl et al., 2004; Babu, Bizarro, and Dewitt, 2005) re-optimize a query by merely considering cardinality information that are obtained from its own currently-running plan's operators. The advantage is that overhead is low, but it is well-known that the lack of probing more information inevitably makes this strategy risk being stuck in local optimality, that is, even after re-optimization, the generated plan is still suboptimial. An example is shown as follows.

Example 1.3.2 *This example illustrates the importance of considering useful cardinalities' variations when deciding optimal plans. Suppose there are two join queries running, and one has a condition A $\bowtie$ B while the other is a star-join having A $\bowtie$ B, A $\bowtie$ C and A $\bowtie$ D. Additionally, the star-join's currently-running plan is (((A $\bowtie$ D) $\bowtie$ C) $\bowtie$ B), meaning A joins D first and then their intermediate results are routed to join with C, followed by joining with B. During execution, the star-join collects cardinality information of (A $\bowtie$ D), ((A $\bowtie$ D) $\bowtie$ C) and (((A $\bowtie$ D) $\bowtie$ C) $\bowtie$ B) and cardinality values of A, B and C* *still indicate that the current plan is the best. Meanwhile, the execution of the first query obtains the cardinality of (A $\bowtie$ B). It is possible that the cardinality of (A $\bowtie$ B) is lower than that of (A $\bowtie$ D), meaning that (((A $\bowtie$ B) $\bowtie$ D) $\bowtie$ C) is most likley a better plan. Unfortunately, because A $\bowtie$ B is not included in the star-join query's execution path, it fails to detect the sub-optimality.*

1.4 Goals and Contributions

In previous sections, we talk about features of streaming data and the consequential query processing issues. However, for different kinds of queries, these issues have different impacts. First, handling queries that merely involve outer joins and aggregate functions usually has nothing to do with cardinality information, because systems should scan all the corresponding tuples to generate results. Then, for those queries having several filtering conditions over the same stream, orderings of filtering operators need careful arrangements, because the efficiency requirement needs low-selectivity filters to be executed first. However, it is easy and costless to exchange filters' positions such that no severe problems will be caused. Essentially, the difficulty is to deal with inner joins. Processing joins is expensive, moreover, bad plans that execute a multiway join query usually consume lots of extra time and storage resources, so re-optimization is considerately important. Unfortunately, inner joins are involved with combinational number of cardinalities, that is, cardinalities of source streams as well as intermediate results, and hence, they are the most challenging to re-optimize. In this thesis, we concentrate on adapting plan-based re-optimization of multiway join queries over streaming data.

We propose a novel re-optimization strategy for data stream systems. The strategy takes into account variations between the most recent and new cardinality values to continuously refine execution plans of join queries. Our contributions are listed as follows:

- To the best of our knowledge, this work is the first to explicitly extend traditional re-optimization approaches to data-stream management. Specifically, we proposed a method to compute upper and lower bounds in streaming environments.
- We propose a novel re-optimization scheme that consists of a three-phase checking component and two-path plan generating component. The checking component determines if re-optimization is necessary. The first phase quantifies arrival rate changes to avoid redundant re-optimization. The second phase considers cardinality changes to detect sub-optimality. The third phase exploits useful cardinality information to alleviate local optimality.
- We implemented the optimization and re-optimization framework on an open-source system. We explored the re-optimization performance over streaming data with varying value distributions, arrival rates and window sizes. The experimental results showed that, re-optimization was able to provide significant performance improvement by up to a factor of 30%, in the real-time and constantly-varying environments.

The organization of the following thesis is listed as follows: In Chapter 2, we present a survey of optimization strategies proposed in DBMSs and DSMSs. And in Chapter 3, we briefly describe the architecture of Esper, an open-source data stream system that is used in our implementation. Chapter 4 and 5 respectively present the query optimization framework and query re-optimization framework that we implemented on Esper. We show experimental results in Chapter 6. Finally, conclusions are presented in Chapter 7.

Chapter 2

Related Work

The essence of run-time re-optimization is to continuously check whether there are better query plans while still executing those that are supposed to remain optimal, and then to use better plans if they are beneficial enough to replace the currently-running ones. Re-optimization is very critical, especially when performing a multiway join, because a sub-optimal join ordering can result in very poor performance. In this chapter, we first discuss related works about run-time re-optimization strategies in Section 2.1 and 2.2. Then, Section 2.3 talks about join processing over streaming data. Finally in Section 2.4, we briefly review methods for statistics collection that existing re-optimization approaches use to detect current plans' sub-optimality.

2.1 Run-time Re-Optimization for Static Data

In database community, re-optimization has been extensively studied. A great deal of approaches has been developed, and most of them aim to identify plans to answer queries such that the system's efficiency can be maximized. We classify them into two categories: 1) adaptive query processing, which includes some generalized ideas that can be used in traditional databases and data stream systems as well; 2) static query optimization, which is proposed for traditional databases but has a run-time re-optimization consideration.

2.1.1 Adaptive Query Processing

In adaptive query processing, the execution of a query is interleaved with its re-optimization. Despite being proposed for traditional databases, adaptive query processing can also be adopted by data stream systems.

In most traditional databases, optimizers use a cost model to evaluate alternative plans and choose least-cost ones to execute their corresponding queries. Approaches complying with this philosophy belong to *plan-based optimization* category. Although a recent literature (Babu and Bizarro, 2005) made a subdivision in terms of sources of conditions that trigger re-optimization, these plan-based approaches share the same principle, that is, using the most recent knowledge of data characteristics to re-compute plan costs. For this reason, in the following discussion we talk about representative approaches together.

- ReOpt (Kabra and DeWitt, 1998) is the first work to introduce run-time re-optimization of currently-running plans for submitted queries. When initializing plans, special computation is prepared for materialization points, such as processes of sorting or building hash table. Based on the reliability of knowledge on data characteristics that the optimizer uses to evaluate plans, those materialization points are assigned corresponding thresholds. At run-time, the actual information of data characteristics is collected, and if the differences between the actual and the predicted values exceed their thresholds, then there is a possibility that the current plan is sub-optimal. As such, re-optimization is triggered to generate a new plan. If the new plan is indeed better, it replaces the current plan, and processing continues with the newly generated plan. To make use of the intermediate results that are already obtained before re-optimization, ReOpt only allows to re-optimize unprocessed work, that is, sub-plans that stay above those materialization points. Because materialization points are natural to get accurate characteristics, ReOpt has low overhead. However an obvious disadvantage is that the benefits are strictly limited by materialization positions, for example, a query plan without any materialization point or a query plan with the only materialization point as the last step has no chance to be re-optimized at run-time. Meanwhile, even though ReOpt expects to save execution costs by exploiting obtained intermediate results, it may need longer time on query processing instead, because completing current materialization points may need more time than running a totally new plan from scratch.

- POP (Markl et al., 2004) is inspired by ReOpt in the way of detecting re-optimization timing. But it has improvements in two aspects. On the one hand, it separates the responsibility of probing sub-optimality from materialization points by creating a specialized operator named *check*. Like normal operators, check operators can be inserted multiple times in query plan trees. As such, re-optimization has more chances to be triggered. For example, with a check operator appended on top of a query plan tree, the whole plan can be changed from scratch. In such a case, results that are ready to be output can either be discarded or stored temporarily for further processing. On the other hand, it uses a more fine-grained way to detect sub-optimality. To measure the optimality of the current plan, every check operator is associated with a value range on the going-by tuple size. Moreover, the range is calculated from progressive computations instead of ReOpt's specific estimation. At run time, if the collected cardinalities are found to be outside of their corresponding ranges, re-optimization is needed. With more careful pre-computed ranges, POP's performance is more accurate. However, POP still has a drawback. During computation of each range, POP assumes all other information remains the same, so it fails to see the big picture.

- Rio (Babu, Bizarro, and Dewitt, 2005), as a further effort, takes POP's idea of using ranges to measure plans' validity. Similarly, according to each estimated value of data cardinalities, Rio takes estimation errors into consideration and then uses an interval to represent possibly actual values. A significant contribution of Rio is its focus on robustness. Rio considers pairs of related data cardinalities and aims to find a set of plans that perform well within the space of possible values of data cardinalities. Obtaining robust plans is more complicated so that the preparation period of optimization is longer. However, Rio reduces times of excessive re-optimization and hence the possibility of losing previous work.

ReOpt uses single-point values as conditions, while POP and Rio extend to use ranges or intervals. The above works share the common principle that re-optimization is triggered when pre-computed conditions are violated. Due to their proven effectiveness, our approach also follows the main idea. However, due to the different natures in streaming environments, problems disscussed in Section 1.3 would occur when they are applied.

ReOpt, POP and Rio are general approaches that launch re-optimizations without restrictions on the run-time environment. However, some other approaches leverage particular conditions to optimize currently-running queries on the fly. We discuss them as follows.

- Query scrambling (Urhan, Franklin, and Amsaleg, 1998) is a mechanism that can reduce the execution time by taking advantage of delays in arrival of remote data. When data of an operator is unavailable at one moment, query scrambling schedules unaffected execution portions that usually would have been processed at a later stage. If the delay is so long that unaffected portions are all already completed, then the optimizer is able to use those intermediate results to generate completely different plans. The re-optimization's principle is that a new plan should access delayed data as late as possible to take advantage of time wasted to wait for delayed tuples.

- The re-optimization approach of Tukwila (Ives, 2002) is corrective query processing (CQP) (Ives, Halevy, and Weld, 2004). In this strategy, the underlying data is horizontally partitioned and CQP allows several plans, with each applied on one partition of data, to complete the same query. The completion of these partitions is pre-defined as the re-optimization timing, such that CQP re-estimates the current plans' costs based on characteristics information that are monitored from previous execution. If newly-computed costs deviate substantially from the expected, then re-optimization is invoked to generate new plans to process the remaining partitions. From the perspective of query processing, execution and re-optimization can be interleaved many times. This mechanism is error-free for stateless operators, but for a query with stateful operators, such as join, an extra phase is needed to make up results that are generated from data across partitions. The goal of CQP is to provide adaptivity without significant performance compromise, that is, it tolerates sub-optimality of currently-running plans as long as performance is acceptable.

In the current implementation, we do not have the data delay issue, so scheduling consideration is not involved in our scheme. Besides, our streaming system does not provide permanent storage for the underlying data, and therefore, it is non-trival for CQP to be adapted. Query scrambling and CQP have their own emphasis on when and how to re-optimize current plans, and we view both of them as complementary to our work.

The join operation is frequently used to integrate data from multiple tables, and therefore, some approaches specifically concentrate on join processing. We list below two recent methods that are developed to adaptively react to newly-collected knowledge. The two works are adaptive reordering joins (Li et al., 2007) and adaptive pipelined join processing (Eurviriyanukul, Fernandes, and Paton, 2006; Eurviriyanukul et al., 2010).

- Adaptive Reordering Joins (ARJ) is a method specifically proposed for indexed nested-loop joins. This method dynamically re-orders sequences of joined tables in two stages. Firstly, it fixes the outer-most table for a given query. Then, it uses completion moments of processing a join's outer input to re-compute costs of the remaining joins. If better orderings exist, it proceeds with the changes. Secondly, for each query, when a batch of results is produced, observations from the previous processing are used to evaluate which table should be the best choice as the outer-most table. The outer-most table can be replaced if better options are found. By merging reordering functionality into join operators, ARJ avoids explicit bookkeeping and routing overhead. However obviously, it needs significant modifications of existing operators, and meanwhile it is limited to specific join algorithms.
- Another adaptive join processing algorithm, called Adaptive Pipelined Join Processing, was proposed to generate pipelined plans (Eurviriyanukul, Fernandes, and

Paton, 2006; Eurviriyanukul et al., 2010). The method of Adaptive Pipelined Join Processing (APJP) gives the iterator-model execution engine the ability to do re-optimization if either of the following two conditions are met: 1) a specific number of final results have already been output and 2) updated statistics collected from the previous execution differ more than a pre-defined number than the optimizer's estimation. At a high level, it holds a similar principle as Tukwila in partitioning data into groups to process. The difference is that Adaptive Pipelined Join Processing implements the mechanism at the physical level, that is, it exploits plans' suspending moments to trigger re-optimization. As a consequence, it does not need an explicit cleanup phase to handle cross-partition data.

ARJ and APJP mentioned above are proposed for pipelined joins, which naturally match streaming settings. Compared to them, we use a different architecture that decouples the re-optimization functionality from normal join operators. This choice results in a slightly higher overhead for our approach but the advantage is that the concerns are separated clearly and further extensions will be more convenient.

In spite of employing various techniques, all works we have mentioned till now are originally proposed for statically stored data. However, they all gradually revise the knowledge of the underlying data's characteristics during execution and change the query plans if performance is found to be unsatisfactory. Therefore, their focus on adaptivity is the same with ours.

2.1.2 Static Query Optimization with Re-Optimization Extension

The idea of run-time re-optimization also exists in some static optimization strategies. More precisely, several methods have considered the data characteristics' value intervals to decide query plans including least-expected-cost (LEC) (Chu, Halpern, and Seshadri, 1999), dynamic query execution plans (Graefe and Ward, 1989; Cole and Graefe, 1994), approximate plan diagrams (D., Darera, and Haritsa, 2007; Dey et al., 2008) and parametric query optimization (PQO) (Hulgeri and Sudarshan, 2003; Ioannidis et al., 1997; Bizarro, Bruno, and DeWitt, 2009; Prasad, 1999; Ganguly, 1998; Hulgeri and Sudarshan, 2002).

- LEC aims to search for robust plans that perform well in the situation where characteristics cannot be estimated correctly at compile-time. LEC treats characteristics as random variables, and within the characteristics' space, it chooses least-expected-cost plans to execute queries. Concretely, for each characteristic, LEC partitions its value range and selects a single value in each partition as a representative. Additionally, every representative value is associated with a possibility. By considering all representative values and their corresponding possibilities, LEC generates some plan candidates and computes their expected costs. The least-expected-cost plan is treated as robust and it is chosen as the optimizer's decision. LEC, however, is not very applicable for the environment that we focus on.
- The method of dynamic query execution plans targets the problem that some essential information is not available at compile-time. Therefore, it postpones the decision of picking the optimal plans at compile time to runtime. It introduces a choose-plan operator that evaluates costs of possible plans as an interval and keep alternative plans for run time processing. At run-time, optimal plans are chosen from those alternatives when unknown parameters can be bounded to actual values.

- Picasso (Haritsa, 2010) project visualizes queries' optimal plans over the space of data characteristics as plan diagrams (Reddy and Haritsa, 2005). A query's plan diagram, showing the optimal plan when characteristic values are determined, generally contains many different plans. Therefore, the problem of plan reduction (D., Darera, and Haritsa, 2007) is proposed to minimize the number of optimal plans if some constraints (e.g., degraded performance by 20%) can be accepted. Later, the method of approximate plan diagram (Dey et al., 2008) extends this idea to provide high-quality approximations to plan diagrams by the following steps: Initially, given the parameter space, optimization is done for points chosen by specific algorithms, such as random or grid sampling. Then, these accurate optimization decisions are used to estimate all the other points. To achieve this goal, there are several algorithms that can be employed according to optimizers' decisions, but they all obey the same principle that plans for unknown points are the same with (i.e., RS_kNN algorithm) or similar to (i.e., GS_PQO and ApproxDiffGen algorithm) those of nearby points. The process ends when all points are filled with approximately optimal plans. Essentially, the idea of plan diagram is to decide optimal plans at compile-time, given all possible characteristic values. This is not suitable in streaming environments. On the one hand, it is too expensive to compute all situations at compile-time and maintain them at run-time. On the other hand, due to the varying nature of streaming data, it is not easy to limit ranges of characteristic values. Moreover, according to the above description, plan reduction and approximate plan diagram aim to identify robust plans over a given space, and therefore they are beyond the scope of the current focus in this thesis.

- PQO has an appealing idea that it prepares at compile-time multiple plan candidates, each of which is chosen for a range of buffer sizes. At run-time, the actual buffer size can be known so that the pre-determined optimal plan is chosen to run. Given the range of possible values of buffer sizes, PQO produces the set of plan candidates as follows: First, for each value, it initializes an optimal plan based on randomized optimization algorithms (i.e., iterative improvement, simulated annealing and two-phase optimization). Due to the nature of randomized optimization, those initialized plans are most likely to be local minima. To alleviate this problem, PQO enhances the basic optimization with the ability of performing *sideways information passing*. For a given buffer size value v and the corresponding optimal plan p, PQO chooses a set of values that are numerically close to the specific value v. Then, PQO compares the optimal plan p with those plans perceived as best choices when the buffer size value equals to anyone in the chosen set of values. If the optimal plan p has a lower cost, then nothing is changed. Otherwise, it is regarded as a local minimum plan and replaced with the lower-cost one. PQO terminates after considering all values. Considering that only the parameter of buffer size is concerned in the original PQO method, later works (Ganguly, 1998; Prasad, 1999) proposed a solution when the optimizer deals with 2 and 3 parameters. To generalize the original PQO idea, Hulgeri and Sudarshan (Hulgeri and Sudarshan, 2002; Hulgeri and Sudarshan, 2003) proposed heuristic solutions for the cases when the cost functions used by the optimizer are linear or piecewise linear or even nonlinear in the given parameters. A more recent work PPQO (Bizarro, Bruno, and DeWitt, 2009) specifically focuses on query-dependant parameters, that is, query predicates' parameters that users define at run-time. When a new value of a parameter is submitted, PPQO consults the *Parametric Plan* (PP) data structure to get an optimal or a near-optimal plan without doing real re-optimization. PP considers the monotonicity principle of optimal regions and returns plans under some sub-optimality restrictions. If such plans cannot be found, real optimization is invoked to generate optimal plans, and those plans along with parameter values are added into PP for further consultation.

All the mentioned static optimization strategies provide valuable viewpoints of optimization, and they theoretically adapt to newly-observed characteristics at run-time. However, because the search space of optimal plans is super-exponential in the number of characteristics considered, they are too expensive to use in data-stream environments. Besides, in order to be cost-effective, it is important for parameter values that those works concern to cover only a small space of their own domains. This requirement cannot be easily satisfied in data-stream environments.

2.2 Optimization for Streaming Data

In this section, we will review approaches that are especially put forward for streaming settings, where queries are submitted only once and results are continuously delivered to users as long as new data are streamed into the system. These queries are known as continuous queries (CQ) and it is more critical for such queries to be re-optimized. First, streaming data's characteristics change dynamically so that the currently-running plans most likely become sub-optimal. Second, these queries are long running, and hence if a sub-optimal plan is not replaced by a more optimal one, the performance degradation will be severe.

Many data stream systems, like StreaMon (Babu and Widom, 2004), TelegraphCQ (Chandrasekaran et al., 2003), CAPE (Rundensteiner et al., 2004), NiagaraCQ (Chen et al., 2000) and Borealis(Aurora) (Carney et al., 2002; Abadi et al., 2005), pay attention on CQ-based adaptive query processing. Among them, StreaMon and TelegraphCQ explicitly address the adaptivity issue.

- StreaMon proposes two methods, A-Greedy (Babu et al., 2004) and A-Caching (Babu et al., 2005). A-Greedy dynamically reorders filters, that is, projection operators, that streaming data goes through. For each query, it obtains each filter's selectivity by counting the number of tuples that are dropped by the filter. In the meanwhile, it maintains a matrix indicating all filters' selectivities in order to choose a greedy ordering so that it takes the minimal cost for all inputs to pass. The matrix, as a global view of all filters in a plan, allows A-Greedy to capture correlations among them. However, the consequential drawback is the high monitoring overhead. Therefore, variants of A-Greedy that carefully choose a subset of selectivities to keep track of has also been proposed. A-Caching is an adaptive mechanism to improve join performance in StreaMon. Since join operators drop tuples if no match can be founded, StreaMon also treats join operators as filters and thus arranges join orderings by the A-Greedy strategy, which unifies optimization strategies but fails to consider materializing intermediate results to reduce execution costs. A-Caching is proposed to handle this limitation. The concrete method is to reduce total costs by exploiting cached intermediate results. Given multiple joins, A-Caching works iteratively. Every time, it carefully chooses candidates of intermediate results to profile. After collecting observations for these candidates, it runs an offline algorithm to pick up the optimal set by which most execution costs can be saved. Because the optimization of join orderings and the determination of materializing intermediate results are separated, it is possible that the join ordering changes before the next iteration starts. In this case, A-Caching would remove all contexts and re-compute candidates for further usage.

- TelegraphCQ, being another data stream system, has a completely different architecture for query processing. The architecture has no conventional optimizer or explicit execution plans; instead, it uses a *Eddy* (Avnur and Hellerstein, 2000) operator to handle intermediate results' transition. This method is regarded as *routing-based optimization*, because the routing mechanism of the Eddy operator is similar to deciding execution plans that traditional optimizers do. The Eddy operator dynamically determines the orderings of necessary operators for every tuple to go through according to their most recent selectivities. To route correctly, Eddy associates every tuple with additional information, indicating whether its corresponding operators have been gone through. This method has two shortcomings: 1) Eddy uses a ticket-based routing policy, thus each tuple merely goes through a locally optimal path instead of the globally optimal one, and 2) Eddy operator does optimization at the tuple level, imposing a big overhead in steady state. Even though further work (Deshpande, 2004) has proposed to arrange an unified path for a batch of tuples in order to reduce scheduling overhead, the routing work for individual tuple cannot be eliminated. Moreover, in this less aggressive variant, the scheduling period is fixed.

A-Greedy and A-Caching are able to cope with characteristic changes, but as a re-optimization mechanism, A-Caching's fixed re-optimization interval is not flexible enough. Moreover, they are proposed for a specific system and therefore it is limited for them to be applied on other systems. Compared to them, our approach, based on plan-based optimizers, is much easier to be implemented in many data stream systems, without modifying their existing architecture. Besides, although TelegraphCQ's approach essentially performs re-optimization, its architecture does not have a plan-based optimizer and therefore it is beyond the scope of our focus.

Next, among the remaining works in the field of data-stream management, we briefly review some representative ones that involve some form of optimization.

- Temporal constraints (i.e., responsiveness) are important when dealing with data streams and there is a work (Hammad et al., 2003) that explicitly concerns the scheduling issue for shared window joins. Three methods, namely Largest Window Only (LWO), Shortest Window First (SWF) and Maximum Query Throughput (MQT), are proposed for different requirements. LWO allows newly-incoming inputs to join with all counterpart tuples under the largest window size of shared queries. Instead of processing new inputs one by one according to their arrival time, SWF suspends the current processing as long as a new input needs to join tuples with a smaller window size. As a combination of LWO and SWF, MQT aims at maximizing throughput, that is, serving the maximal number of queries per unit time.

- State-Slice (Wang et al., 2006) improves performance by sharing results of common sub-expressions. One big contribution of State-Slice is that it considers selections and joins together. For such queries, the general ways that systems interleave selection operators with join operators are *pull-up* and *push-down* methods. The pull-up method is to perform selections after the completion of joins while the push-down method is to filter underlying data by selection operators as early as possible before feeding them into join operators. State-Slice identifies that neither methods can be optimal, given some storage and computing resource constraints. Therefore, it separates every single stream buffer into several slices based on window sizes of queries submitted. In each slice, it is able to compute intermediate results by executing selections and joins together, and route those results for different queries. At a system level, it chains up all slices of the same stream in sequence to guarantee system correctness. By using this finer-grained method, State-Slice achieves either maximal memory-efficiency or CPU-efficiency goal.

- Plan migration (Zhu, Rundensteiner, and Heineman, 2004) indicates the link-up process between a pair of current plan and new plan. Its first attempt is *pause-drain-resume* approach, proposed in Aurora system(Carney et al., 2002). This work is easy to implement but it causes errors when processing queries involving stateful operators (such as join and aggregation), because it either misses results or causes deadlock. To guarantee correctness, CAPE (Rundensteiner et al., 2004) introduces two strategies, that is, *moving state* (MS) and *parallel track* (PT) in the literature (Zhu, Rundensteiner, and Heineman, 2004). Given a new plan, MS first suspends the current plan. Next, it finds out all pairs of intermediate results with identical schemas between the current plan and the new plan so that corresponding tuples can be transferred in the next stage. Then, it computes the remaining intermediate results needed by the new plan. Finally, it discards the current plan and starts the execution of the new plan. During migration, there is no output generated. Unlike MS, PT runs the current plan and the new plan simultaneously and keeps the current plan until it cannot generate legal results any more. In the meanwhile, additional processing is added to the top-most join operator of the current plan in order to avoid duplication. To achieve better efficiency, HybMig (Yang et al., 2007) is proposed to combine MS and PT together. By carefully choosing data to process, HybMig generates non-overlapping final results. As such, the whole migration duration becomes shorter. Another work (Esmaili et al., 2011) expands the idea of plan migration to plan modification, which adapts the current plan to changes of either query semantics or system load. This plan modification method exploits punctuations to control plans' start or stop points. According to interactions among punctuations, it also provides some variants to satisfy different requirements of correctness. But the up-to-date techniques can only be applied to stateless operations, for example, selection and projection.

Albeit showing significant potential for improving system performance, they are described for specific usages, and if necessary, they can be integrated as a pre-computing or post-computing process with our plan-based methodology.

2.3 Processing Joins over Streaming Data

Join algorithms have been specifically studied in the context of data-stream management. Since our work focuses on re-optimizing multiway joins over streams, we next review some relevant works on this topic.

- The operator of *State Module* (SteM) is proposed in the work (Madden et al., 2002). A SteM has the access control to a stream by encapsulating a single index built on it with a particular attribute as the key. There are two scenarios that a SteM for stream s has: One is that a singleton tuple of s comes, then it is inserted into the SteM as well as other SteMs that build indexes on s, before the tuple is processed. The other one is an intermediate tuple that needs to join with tuples of s comes, then it is used to probe the index that the SteM manages. If there is no match, the intermediate tuple is discarded. Otherwise, the SteM revises the tuple's state information to indicate the completion of current processing and delivers the newly joined intermediate result to following operators. To handle joins, the system has a global operator named Eddy (Avnur and Hellerstein, 2000) that determines the orderings of SteMs for every singleton tuple to go through. Eddy dynamically observes processing costs of SteMs and chooses reasonable orderings.

- The operator of STAIR (Deshpande and Hellerstein, 2004) addresses a limitation, that is, in the routing-based architecture, previous decisions may affect join orderings in future, even if new decisions are supposed to be applied. This problem arises for multiway joins, because joining orderings (i.e., routing decisions) cannot be changed, once intermediate results were generated and state information was modified correspondingly. STAIRs' solution is to undo the earlier work and pre-compute necessary work when the join ordering is changed. A STAIR operator is actually a Symmetric Hash Join (SHJ) (Wilschut and Apers, 1991) operator with four extra interfaces. Two interfaces are *insert* and *probe*; while the former stores a tuple, and the latter returns matching tuples inside the STAIR. The third one is *demote* and it restores intermediate results by removing the portion related to a specific stream. The last interface, named *promote*, joins its stored data with more streams and retains the matches. With these four interfaces, STAIRs guarantee that the join ordering for a tuple can be adjusted at run-time, according to most recent statistics.

- XJoin (Urhan and Franklin, 2000) based on Symmetric Hash Join (SHJ) (Wilschut and Apers, 1991), is a non-blocking operator to handle multiway joins. It extends the original SHJ to use secondary storage and reactively schedules processings for in-memory and on-disk tuples. It divides join processing into three stages: The first stage joins memory-resident tuples, which is essentially the same as the original SHJ does. At the second stage, disk-resident tuples from one source stream are chosen to probe memory-resident tuples from the other source stream. The third stage performs necessary matching to complement results missed by the first two stages. During XJoin's scheduling, the first stage is given the highest priority so that in-memory results are processed as long as new tuples arrive. When the first stage does not have any new tuple to process, the second stage is triggered. Since the first stage cannot make progress, the second stage in effect hides intermittent delays in data arrival. Finally, the third stage executes after all tuples have been received.

- MJoin (Viglas, Naughton, and Burger, 2003) is a more generalized Symmetric Hash Join (SHJ) (Wilschut and Apers, 1991) algorithm for multiway joins. A MJoin operator that is assigned to process a join is completely symmetric with respect to related streams: It builds a hash index for each stream, such that an incoming tuple from any source stream can be used to generate and propagate results in a single step, without going through a multi-stage binary execution pipeline. The sequence for a tuple to join with other streams is based on join selectivities, and every time the most selective join of the remaining ones is chosen to be evaluated first. For tuples from different streams, orders are not the same.

- The first work that explicitly considers multiway joins for sliding windows is the literature (Golab and Özsu, 2003). It points out that, join algorithms are affected by two issues: One is tuple processing strategy - while *eager evaluation* processes a tuple immediately when it arrives, and *lazy evaluation* processes newly-incoming tuples periodically. The other one is tuple expiration strategy - while *eager expiration* removes old tuples whenever a new tuple comes continuously, and *lazy expiration* does pruning periodically. Join algorithms are designed as follows: *Ea-*

ger Multiway Nested Loop Join (NLJ) processes tuples in the eager way. This method can be used with either eager or lazy expiration strategy. In contrast, *Lazy Multiway NLJ* uses the lazy evaluation strategy and only can be applied with lazy expiration. Accordingly, *Multiway Hash Join* has both eager and lazy versions. A *Hybrid NLJ-Hash Join* is also proposed in order to use available hash indexes. All the mentioned algorithms have in common the policy that in each lookup stage on a stream, only tuples that satisfy the timestamp requirement can be processed. The main difference between the NLJ and Hash groups is that NLJ algorithms need to scan entire windows but Hash algorithms only probe a bucket of tuples. Moreover, this work proposes some heuristics to decide join orders. The main idea is to order joins in the descending order of their selectivities so that less selective joins are executed later. Additionally, if one or more streams have faster arrival rates than the others, then it tries to move forwards these fast streams' positions in the ordering.

Among the works mentioned above, XJoin is simply a join algorithm. The way that SteM does re-optimization actually splits the recording of state information from the optimizer. However, both SteM and STAIR must cooperate with a routing-based processing logic (Avnur and Hellerstein, 2000), and therefore, it cannot be compared in the context of the thesis. The last two choose join orders by the same heuristic, which is shown to be effective but has no guarantee of optimality. Moreover, they do not deal with re-optimization.

There are still some works proposed with optimization purposes, although they do not specifically target multiway joins. We briefly describe them as follows.

- Rate-based optimization for streaming data is first proposed in the work (Viglas and Naughton, 2002). The work builds up a cost model as a funtion of streams' arrival rates to estimate output rates of Select-Project-Join queries. With respect to join methods, it considers Symmetric Hash Join and Nested Loop Join. Based on the cost model, it discusses two optimization cases: One is *local rate maximization*, that is, given a specific time point, the optimizer selects a plan to maximize the output size. Solving this problem is straightforward: By using the cost model, output rates of all possible plans can be obtained and the plan with maximal output rate should be chosen. The other case is *local time minimization*, that is, given a specific number of outputs, the optimizer selects a plan to minimize the execution time. The work is implemented by a divide-and-conquer strategy with the sub-problem of finding the fastest plan to join a specific set of streams. It is worth pointing out that the computation can be terminated if a satisfying plan is identified for a subset of streams.

- Rate-based Progressive Join (RPJ) (Tao et al., 2005) is proposed to maximize the output rate by choosing orderings of processing data portions that are stored in memory or on disk. It is essentially an advanced version of XJoin with a flexible joining and complex flushing strategy. It includes three phases. The first one that joins memory-resident tuples is the same as that of XJoin. The second one is to reactively choose one disk-resident portion from one stream to process. This portion can be joined with either a memory-resident or disk-resident portion, as long as the output rate is estimated to be maximal. Because XJoin handles two disk-resident portions only when no more in-memory portions await processing, in this sense, RPJ is more flexible than XJoin. The third one is a clean-up phase that generates missed results. The other aspect where RPJ is superior to XJoin is that, RPJ flushes a memory-resident portion to disk based on a probalistic analysis.

It estimates the probabilities that the next incoming tuple belongs to each portion. For a portion *p* with the smallest probability, RPJ flushes the corresponding portion that *p* is supposed to be joined with. This process is repeatedly done until enough space is released.

- Another representative work (Kang, Naughton, and Viglas, 2003) introduces a unit-time cost model and focuses on processing a single binary join. It considers three scenarios. The first one is one stream arrives much faster than the other and the work aims to maximize the join efficiency (i.e., execution time). In the latter two situations, the system has computing and memory restrictions respectively, and the goal is to maximize the number of results generated. This work has some interesting results obtained from both experimental and analytical studies. An important conclusion is that asymmetric combination of Hash Join and Nested Loop Join for two join sides outperforms the other options in the first scenario. Besides, in the second situation, allocating half of the computing resources to each stream is the best choice and hash join is most suggested. In the third situation, it suggests that the stream with the slower arrival rate should be assigned all available memory and a hash index shoule be built on it for the other stream to probe.

Obviously, the rate-based works have a different emphasis in terms of re-optimization direction. However, our scheme actually covers the consideration of arrival rates by comparing cardinality changes of source streams. Moreover, the last work mentioned above needs to extend the cost model to full query plans so as to be incorporated into the re-optimization routine. Finally, note that, although some of those listed works consider memory requirements, our work that currently assumes queries and streaming data can fit into memory, is complementary.

2.4 Statistics Collection

As discussed, it is obvious that the re-optimization process usually depends on most recent statistics of the underlying data to detect sub-optimality. As such, statistics collection is very important for re-optimization. Next, we discuss three groups of statistics collection, namely exploitation-oriented, exploration-oriented and balanced approaches.

- Exploitation-oriented approaches follow the principle that collecting statistics should be based on currently-used plans' execution routines. DB2's Automated statistics collection (ASC) (Aboulnaga et al., 2004), LEO (Stillger et al., 2001), Rio (Babu, Bizarro, and Dewitt, 2005), ReOpt (Kabra and DeWitt, 1998) and POP (Markl et al., 2004) all belong to this group. ASC, as a component in DB2 UDB, is comprised of a update-delete-insert (UDI) driven and a query-feedback(QF) driven process. The UDI-driven process monitors table activities that change statistics of the underlying table. The QF-driven process monitors query results and populates these information to update optimizer's cardinality estimation. LEO also leverages feedbacks to correct estimation errors in currently-running plans. It collects actual information and computes an individual adjustment factor for every characteristic. Moreover, based on isolated characteristics, LEO is able to identify correlation between predicates. Similar to LEO, Rio renders feedbacks of characteristic estimations that current plans need. It uses a random sampling strategy and merges this process with regular query execution process. The objective of statistics collection in ReOpt and POP is to verify the estimations of used characteristics. ReOpt puts very low burden on statistics collection, because it only allows gathering knowledge at materialization points (e.g., sorting or building hash table) at run-time. To obtain more information, POP aggressively materializes intermediate results wherever *check* operators are inserted. Exploitation-oriented approaches guarantee the least overhead, but the main problem is that they fail to have accurate information of characteristics that are absent from currently-running plans' routines. This may lead to a problem of local optimality, that is, some plans remain sub-optimal even after they are re-optimized many times.

- Exploration-oriented approaches aims to alleviate the local optimality problem of exploitation-oriented approaches so they proactively probe statistics that current plans are unaware of. The representatives are the pay-as-you-go approach (Chaudhuri and Ramamurthy, 2008) , exact cardinality query optimization (Chaudhuri, Narasayya, and Ramamurthy, 2009) and Oracle's ATO (Belknap et al., 2009). The pay-as-you-go approach takes as input a user-defined threshold and deliberately executes sub-optimal plans, which are transformed from optimal ones in order to obtain missed statistics. To achieve maximized benefits, it uses a set cover algorithm to greedily select characteristics. Exact cardinality query optimization targets obtaining all exact cardinalities in order to conduct optimizer testing. Although it also relies on executing query expressions and leverages feedback for further choices, it works separately from re-optimization. Given query expressions, it classifies them into different sets according to their own *signature*s, which are actually representations of related schemas. Then, queries are generated for each set and they are greedily chosen to run until all expressions are covered. Lastly, chosen queries are executed to obtain exact cardinalities. ATO differentiates itself from the other schemes by doing additional processing to reduce the uncertainty of cardinality estimations. Precisely, ATO performs random sampling to validate possible characteristics (e.g., cardinalities of base tables, selectivities of join conditions) before executing queries. The collected statistics may cause the optimization decisions to change, indexes to create and so on. Exploration-oriented approaches also have a clear disadvantage: The overhead of acquiring the comprehension of all statistics is prohibitively high. More importantly, all of these works are proposed and used in traditional databases, and it is not clear whether they can be embeded into the data-stream systems.

- Xplus (Herodotos and Shivnath, 2010) is a recent work balancing utilization of exploitation and exploration. Its decisions on which statistic to monitor has two phases. In the first phase, it uses a policy to choose a *plan neighborhood*, within which plans are structurally similar to each other. Then, the goal of the second phase is to pick up a plan with least cost in its corresponding neighborhood. Lastly, after running chosen plans, gathered statistics are enough to decide optimal plans. To keep the balance, Xplus needs to guarantee the total collection overhead is under a specific bound, such as a user-defined bound.

Our scheme also needs to monitor the most recent statistics information. Our statistics collection is based on the exploitation-oriented idea. Additionally, we are able to obtain a more comprehensive view of statistics by analyzing statistics collected from different plans. The combination of collecting and analyzing achieves a balance between overhead and accuracy, and further improves the re-optimization performance.

Chapter 3

Esper: An Event Stream Processing Engine

We implemented our scheme on Esper (Esper, 2013), an open-source software designed for handling data streams and complex queries over them. In this chapter, we depict essential features and components of Esper. First, we show the architecture of Esper in Section 3.1. Then, Section 3.2 explains the data models containing the window concept and tuple expiration issue. Next, storage management, join algorithms and execution plan types are illustrated in Section 3.3. Finally, we introduce Esper's original optimization method in Section 3.4.

3.1 Architecture

In order to support streaming environments well, Esper (the free version) offers in-memory implementation, namely, the system stores data in main memory and uses online algorithms to manage data. In spite of being a data-stream management system, Esper still uses the architecture and plan-based optimizer as most conventional databases do.

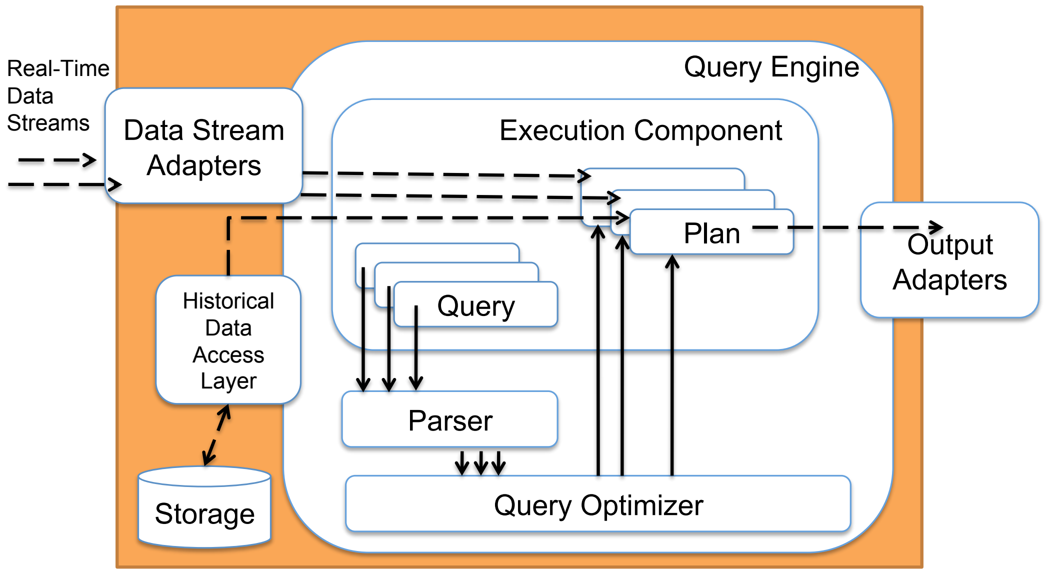


Figure 3.1: Esper's architecture

The main architecture consists of interfaces, storage and query engine. Interfaces are primarily data stream adapters and output adapters to receive data and deliver results, respectively. The storage management component deals with storage of original data and their statistics. More importantly, providing the system's core functionality, query engine includes parser, optimizer and execution engine. As preparation units, parser and optimizer are in charge of analyzing query semantics and deciding optimal plans, respectively. The execution engine processes data and answers queries. The architecture mentioned above is shown in Figure 3.1, where dashed lines represent flows of data streams and solid lines represent control flows after queries are submitted.

At a high level, Esper works like a DBMS as follows: Once registered, application queries are stored in the system. As data flows in at real-time, Esper checks whether data properties match any stored query. If so, the data will be processed and results will be organized and output. Otherwise, the data will be temporarily stored until their expiration. In the following subsections, we explain how Esper deals with streaming data and answers stored queries.

3.2 Data Models

A data stream that Esper accepts is a sequence of tuples that arrive at real-time in some order. Stream tuples can be represented by a rich range of forms, such as Java objects, XML documents and simple name value pairs. The form of an individual stream must be registered before any arrival of its data. All data are stored in main memory.

Because data are potentially unbounded in size, data size would outgrow the memory capacity eventually. Like many other data-stream systems (Abadi et al., 2005; Chen et al., 2000; Chandrasekaran et al., 2003; Babu and Widom, 2004), Esper handles this problem by using the concept of *window*. Windows restrict the scope of data that the system stores and processes and they move gradually. As time goes, data that fall beyond their corresponding windows are expired and discarded. For different streams, windows can be different, according to users' definition in queries' declaration. In the context of query processing over data streams, two most popular schemes of windows are supported in Esper. They can be classified according to the following criteria:

- *Definition of window size:* A window can be defined by either *time* or *length*. A time-based window always stores newly-incoming data for a time period, while a length-based window only saves a specific number of the most recent tuples.

- *Movement method of window:* There are *batch* and *sliding* windows. A batch window is actually a tumbling window that batches data and releases them when a given number of tuples are collected (length-based) or a given time interval is passed (time-based). In this sense, data from different windows of the same stream do not overlap. On the contrary, a sliding window, via moving forward its endpoints, releases expired data gradually as new tuples arrive (length-based) or time goes (time-based).

Actually, Esper supports more window schemes. For example, it provides windows with computing ability, such as sorting and averaging. These kinds of windows are Esper's featured functionalities, so in our context, we only focus on windows that we described above.

3.3 Storage and Query Processing

Storage and query processing in Esper are heavily tied up with each other, and therefore, we illustrate them together.

Esper has a storage component to store all tuples required during query execution. Additionally, simple statistics, mainly including queries' output size and execution time, are collected at run-time and stored in the storage component.

Esper supports continuous queries (real-time processing) and it provides a SQL-like query language that allows filtering, join, aggregation and pattern recognition over data streams. In the context of this thesis, we concentrate on multiway join queries. Specifically, as discussed in Section 1.4, the set of outer join results is essentially the Cartesian product of related data streams, so it is difficult for the re-optimizer to significantly influence the execution performance. Therefore, to facilitate discussion, we only focus on inner joins in the remaining thesis, unless specifically stated.

Multiway joins have two kinds of requirements on storage: On the one hand, tuples that need to be handled can be incoming data from stream sources, and on the other hand, the system needs to maintain intermediate results for further processing. We talk about the corresponding schemes as follows:

- *Indexes for Stream Sources:* Every stream source requires main memory space to store its valid tuples, the size of which depends on the maximal size of windows on it. Specifically, for fast query processing, Esper builds indexes on streams' attributes when parsing and analyzing query statements. Concretely, for equality join queries, hash indexes are built on attributes shown in a query's *where* clause. For non-equal joins, the index used is a self-balancing binary search tree. According to the query statement, it is possible for a single stream to have more than one indexes built on it.

- *Temporary Buffer for Intermediate Results:* Esper does not materialize intermediate results. In some sense, there is no explicit storage scheme for them. Intermediate results are temporarily stored inside the operator that produces them. Every time an intermediate result is generated, the operator allocates a space to hold it, and feeds it into the next operator. The next operator looks up the corresponding tuples, and if matches are found, new intermediate results are formed. At the completion of the lookup, this intermediate result is discarded and its space is released. In a nut shell, an intermediate result is temporarily stored in the system between its generation and the time that the next operator completes processing it.

Based on the storage schemes, methods that Esper uses to process joins are pipelined hash join and nested indexed (BTree-like) join. Another important aspect of join processing is types of join plans. In Esper, bushy plans are not supported. In other words, Esper only provides binary plans to answer queries.

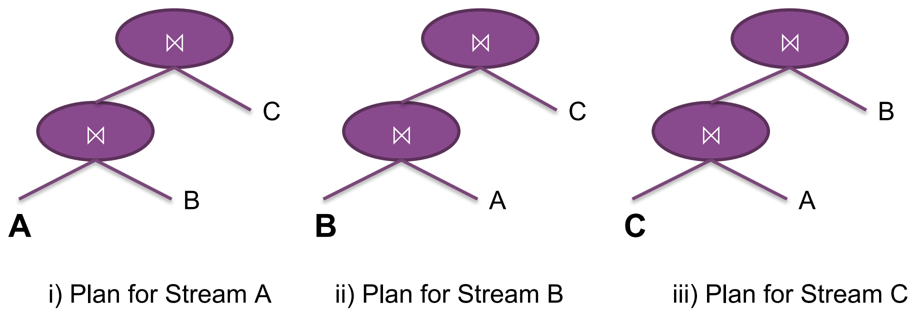


Figure 3.2: Esper's multiple-plan-per-query strategy

Specifically, as demonstrated in Section 1.1, data streams display uneven and dynamic speeds. Therefore for multiway joins, the traditional *one-plan-per-query* strategy leads to a latency issue that newly-incoming tuples must wait, if processing them depends on the completion of processing other streams. Esper, as a stream-oriented system, is designed to operate in the way that data are immediately processed as they arrive, no matter which stream they come from. It uses a *multiple-plan-per-query* strategy that allows multiple execution plans running for one join query. More concretely, given a join query, the optimizer assigns an individual plan for each source stream. The following example illustrates the multiple-plan-per-query strategy.

Example 3.3.1 *Suppose A, B and C are source streams and over them there is a query with two join conditions, A.id = B.id and B.id = C.id. For this query, the optimizer would produce three execution plans, one for each source stream. Assume the plans are those shown in Figure 3.2, and the source streams that the plans are generated for are emphasized. Conceptually, the plans are all different because they use different source streams' tuples to trigger join operations. At run-time, newly-arriving tuples that belong to different source streams will be fed into different plans. Take the middle plan in Figure 3.2 for example, a new B tuple will follow the steps of stream B's plan to join stream A's tuples first, and the result, if any, is then joined with stream C's tuples.*

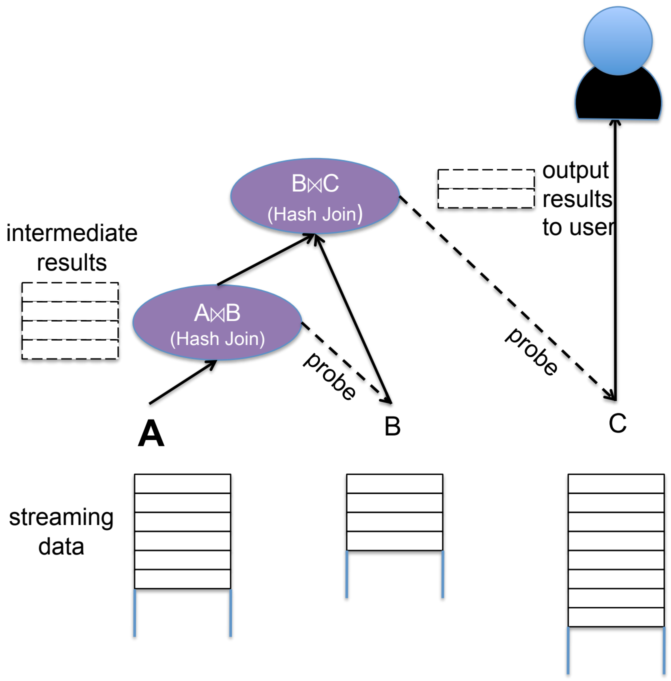


Figure 3.3: Storage and query plan for the join in Example 3.3.2

Finally, putting them together, we give an example to illustrate the storage and query plan issue for a multiway join.

Example 3.3.2 *Suppose A, B and C are source streams and over them there is a join query with two conditions, A.a = B.b1 and C.c = B.b2. Figure 3.3 shows a plan (for stream A) to answer the query. Tuples of data streams are represented by rectangles with solid lines, and intermediate results temporarily retained by join operators are represented by rectangles with dashed lines. Hash indexes are built on the attribute* b1 *of stream B and the attribute* c *of stream C. Suppose a tuple* t *of stream A arrives, then the stream B is first probed. If stream B contains a tuple matching* t.a, *then the joined tuple* jt *is generated and held by the operator A ⋈ B. The intermediate result* jt *is used to probe stream C with* jt.b2. *If a matching is found, then a final result is generated and it can be output to the user. Otherwise, if no matches are contained in stream C, then* jt *is* *discarded.*

3.4 Query Optimization

In this section, we first show the plan representation. Then, we illustrate how Esper's original re-optimizer works for a query.

Based on the discussion in the previous section, for a multiway join query, Esper generates multiple plans, the number of which is equal to that of involved source streams. More importantly, for every plan, only tuples of the specific source stream are able to trigger the processing, and only intermediate results containing these tuples can probe other streams to generate final results. Therefore, a plan actually degrades to an *ordering*. We give the definition and a detailed example as follows.

Definition 3.4.1 *An* ordering *is the processing sequence of source streams involved in a multiway join query. For a N-way join, an ordering is represented by* $o_0 : o_1 \rightarrow ... \rightarrow o_i \rightarrow o_j ... \rightarrow o_{N-1}$. o_0 *indicates the* start-up *stream, to which the ordering is assigned. And the rest represents the* join sequence*, where* o_i *is the* i*th stream to be processed. Note that, arrows in the ordering representations just highlight the join sequence, so it is safe to ignore them on the purpose of simplicity. Therefore, the ordering can also be represented as* $o_0 : o_1 ... o_i o_j ... o_{N-1}$.

Example 3.4.1 *Consider plans shown in Figure 3.2. Stream A's plan can be represented as the ordering A:B→C (or A:BC), with A indicating the start-up stream and B→C showing the join sequence. Similarly, B:A→C (or B:AC) and C:A→B (or C:AB) indicate plans of stream B and C, respectively.*

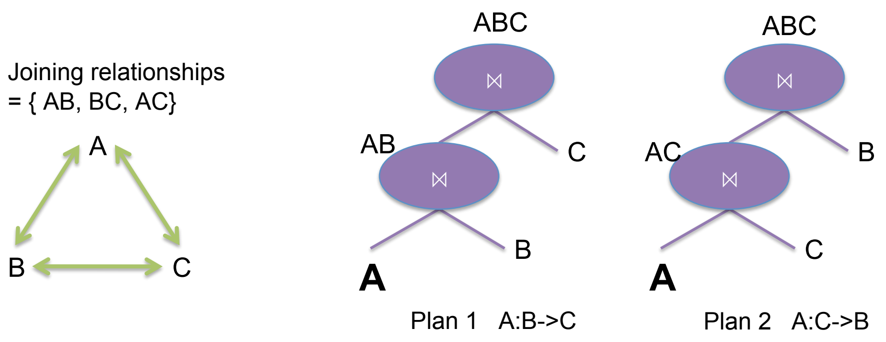


Figure 3.4: Optimization process to generate stream A's plan in Figure 3.2

In the rest of the thesis, the concepts of ordering and execution plan indicate each other and are used interchangeably. Moreover, considering that execution plans are usually illustrated with the binary tree structure, we still use this form.

At a high level, Esper uses a *optimize-once* scheme, that is, queries are optimized only when they are submitted and the generated plans would be used throughout the execution. Moreover, since an execution plan has a start-up stream, to decide a plan is essentially to arrange a sequence of the other streams involved in the query. Therefore, given a join query, Esper's optimizer runs as follows: The optimizer initially identifies the joining relationships among all data streams. After that, the following steps are done for each source stream: First, the optimizer randomizes an ordering of all streams except the source stream. Next, the optimizer, according to the joining relationships, checks whether every stream in the ordering can join with at least one of previous streams. If the checking is not passed, another ordering will be tried until the optimizer finds out a satisfying ordering and uses it as the execution plan. A concrete example is shown as follows.

Example 3.4.2 *Consider again the Example 3.3.1 and the corresponding plans shown in Figure 3.2. We illustrate how the optimizer produces those plans. For the query, the* *set of joining relationships is {AB, BC and AC}, where AB and BC both come from original join conditions and BC is derived according to transitivity. On the left side of Figure 3.4, the set is visualized as a graph, with vertices denoting the source streams and edges denoting that joining related streams is an inner join. Generating execution plans is essentially finding out an ordering for each source stream. Because the process for each individual stream is the same, we only illustrate stream A as an example for simplicity. For stream A, all permutations of join sequences are B→C and C→B, and their corresponding plans are shown in Figure 3.4. Either of them can be chosen randomly at the first attempt. Assume the optimizer chooses B→C and then checks the members in the sequence one by one: According to join relationships, B is able to join with A, meanwhile, C can join with either A or B, so A:B→C (or A:BC) is selected as stream A's execution plan. The same process runs for stream B and C, and the optimizer's final decisions are shown in Figure 3.2.*

Obviously, Esper's optimization method is able to provide an ordering very quickly, however, it cannot guarantee the optimality of the ordering. Moreover, Esper has no capability to perform re-optimization. Our scheme aims to overcome these two problems, and we will discuss the detailed techniques in the following chapter.

Chapter 4

Query Optimization Framework

Recall that for a join query, Esper has multiple plans to answer the query, one of which being assigned to a source stream (represented as multiple-plan-per-query strategy in Section 3.3). An optimal plan is actually an ordering in which streams are processed one by one (explained in Section 3.4). Moreover, the original optimizer in Esper decides a plan by randomly choosing a join ordering according to the joining relationships of related data streams (illustrated in Section 3.4). Obviously, this kind of implementation cannot guarantee the quality of plans produced. In order to introduce our re-optimization scheme, we in this chapter present our way of enhancing Esper's original optimizer. The modified query optimization framework includes a basic optimization method, statistics collection issues and a cost model to perform a cost-based optimization.

We summarize in Table 4.1 notations that are frequently used in this Chapter.

Table 4.1: Notations frequently used in the query optimization framework

Symbol	**Description**
Card(s, t)	the cardinality value for s (source stream or intermediate result) at time t
Card(s)	the cardinality value for expression s (source stream or intermediate result) at present
Card($s_1 ... s_n$)	the cardinality for intermediate result of joining streams s_0 to s_n
$o_0 : o_1 ... o_N$	execution plan for source stream o_0 that joins streams o_0 to o_N

4.1 Optimization using Dynamic Programming

A crucial extension that we made on Esper's original optimizer is the abilities of plan enumeration and pruning for join queries. They are encapsulated in a dynamic programming (DP) algorithm, which is used to identify the optimal plans.

The DP algorithm computes the best plan for a query incrementally, based on optimal plans of its plans. As shown in Algorithm 1, the recursive process is implemented in a bottom-up fashion. For every start-up stream (see Definition 3.4.1) in a query, dynamic programming starts with the information of streams that need to be joined (line 2 and 3). Every time, it enumerates a new (intermediate) join ordering by appending one more stream to the current one (line 8). Those (intermediate) join orderings are temporarily stored by the optimizer, and if the optimizer observes that two orderings involve the same set of streams, then they are compared (line 9). The worse ordering is pruned (line 10), and the better one is saved to represent the currently-optimal plan for this set of streams. The recursion terminates until all streams are considered. The optimal plan is the ordering that is ultimately saved for the set containing all related streams.

The optimizer decides which join ordering is better by comparing their costs and the joining cost largely depends on how many tuples should be processed, because the execution is done in memory. Before introducing the cost model, we first present the definition for the size of valid tuples.

4.2 Cardinality

The concept of *cardinality* is widely used in traditional databases to indicate relation size. However, it is not explicitly proposed in streaming environments. Next, we introduce the way that cardinalities are measured in our context.

Algorithm 1: Dynamic Programming to Generate an Optimal Plan

```
input : The number of streams $N$,
        the startup stream $s_{startup}$,
        the set $S$ of related source streams except $s_{startup}$
output: The optimal plan
1  Let $List_n (0 < n \leq N)$ store the currently-optimal (sub-)plans that contain $n$
   different streams;
2  foreach stream $s$ in $S$ do
3      Put $s$ in $List_1$;                                // Initialization
4  for $n \leftarrow 2$ to $N$ do
5      foreach currently-optimal sub-plan $o_{pre}$ in $List_{n-1}$ do
6          foreach stream $s$ in $S$ do
7              if $s$ is not contained in $o_{pre}$ then
8                  Generate a new (sub-)plan $o_{crnt}$ by appending $s$ to $o_{pre}$;
9                  if $List_n$ contains a plan $o_{opt}$ involving all streams in $o_{crnt}$ then
10                     if $o_{crnt}$ has a smaller cost then Replace $o_{opt}$ with $o_{crnt}$;
11                 else put $o_{crnt}$ in $List_n$;
12 return $s_{startup} : List_n.getTheOnlyElement()$;
```

4.2.1 Definition of Cardinality

In data-stream management, the cardinality (i.e., the number of valid tuples) is actually varying with respect to time, subject to the arrival rate of streaming data and the window semantics of query processing. For example, Figure 4.1 illustrates how the number of valid tuples in a source stream *s* can change in a period of time. The initial value is 0, because no data flows into the system. As time goes, the value changes, as shown in Figure 4.1.

Theoretically, the number of valid tuples should be a continuous function of time. However, at the implementation level, it is prohibitively expensive to simulate every single increment or decrement, instead, considering the total effect after a time interval (or unit) is more cost-efficient. Therefore, we use discrete points to represent the continuous function of the number of tuples, and it is acceptable when the time interval is small. The

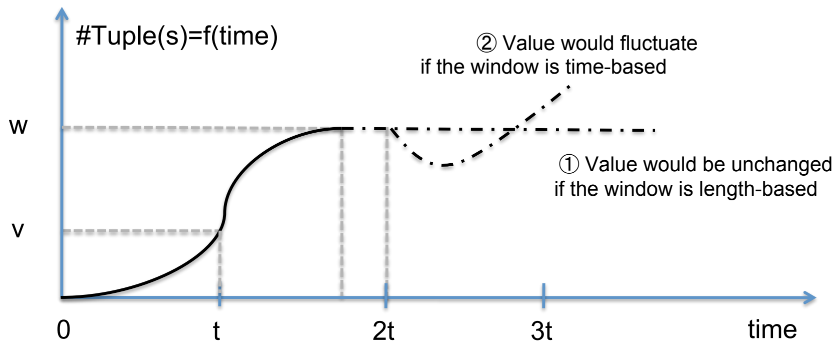


Figure 4.1: The number of a source stream's valid tuples in a window

concrete definition is given as below.

Definition 4.2.1 *Consider a data source* s *that provides tuples for an operator* o *to process. The source* s *can be an original data stream or the intermediate result generated by the prior operator. The* cardinality *for* s *at the present time interval* t*, denoted as* Card(s, t)*, is the number of valid tuples accumulated during this interval. Concretely, if* s *is an original data stream, then* Card(s, t) *is the number of tuples that still reside within its window (described in Section 3.2) at the end of* t*. Otherwise, if* s *indicates an intermediate result,* Card(s, t) *are the number of tuples fed into* o *during this period.*

According to the definition, cardinalities shown in Figure 4.1 are: $Card(s, t) = v$ and $Card(s, 2t) = w$.

We note that, for a source stream, the definition has different effects with respect to the kind of window used (discussed in Section 3.2). If the window is length-based, the cardinality of the source stream would be fixed as the window size at steady state (① in Figure 4.1). On the other hand, if the window is time-based, there is no specific upper bound on its cardinality, and the value would fluctuate according to the arrival rate (② in Figure 4.1).

As most applications, our scheme is interested in cardinalities over recent data. Most of our discussion means the present time interval, and therefore, we omit the time parameter in the denotation, unless specifically stated. Concretely, we use *Card(s)* to represent *Card(s,t)* in the remaining thesis, unless specifically stated.

4.2.2 Estimating Cardinality Information

We note that, source streams' cardinalities can be easily obtained by counting the number of newly arriving tuples and considering the window constraints, and therefore, the only problem is to decide the intermediate results' cardinalities. Given a query Q and the set of currently running plans $\{P_1, \ldots, P_N\}$, there are two cases when considering a particular cardinality $Card(s_1 ... s_n)$ $(1 \leq n \leq N)$. In the case that joining streams s_1 to s_n corresponds to a sub-plan of P_i $(1 \leq i \leq N)$, the cardinality value can be collected from the execution routine. Otherwise, the cardinality value cannot be directly obtained.

One approach of obtaining such cardinalities is to design plans that include those cardinalities. By executing them, actual values can be acquired. Unfortunately, this method is not suitable for streaming environments. On the one hand, it takes more computing resources to execute these purposely-designed and sub-optimal plans. Even if those plans' results can be delivered to users, it is still costly to integrate them with other plans to answer the query (e.g., avoiding redundant outputs). On the other hand, the validity of cardinality values cannot last for long, due to the time-varying nature of streaming data. Therefore, occasionally applying purposely-designed plans has no significant effect on guaranteeing the cardinalities' accuracy during continuous processing.

We propose an analytical method to estimate intermediate results' cardinalities. Moreover, it is obvious that the estimation for inner and outer joins should differ, because the cardinality of an outer join is the Cartesian product of involved streams' cardinalities, but inner joins' are not. To classify the intermediate results, we introduce the definition of *inner join set* first.

Definition 4.2.2 *Consider a set of streams* S *and a set of join conditions* C *on* S*. If there is a join condition indicating that streams* u *and* v *($u, v \in S$) are joined on an attribute, then the join between* u *and* v *is an inner join. Given a (sub-)set* T *of* S*, we regard it as an* inner join set*, if each stream* w *in* T *at least has an inner join with one stream in* T-{w}.

The following is an example on how intermediate results are classified according to the above definition.

Example 4.2.1 *Assume there is a query on a set of streams {A, B, C, D}. Suppose the set of join conditions is {A.b = B.b, A.c = C.c, A.d = D.d}. The set {A, C, D} is an inner join set, because A and C are joined on the common attribute* c*, and A and D are joined on the common attribute* d*. However, the set {B, C, D} is not an inner join set, according to our definition. Therefore, the cardinalities of ACD and BCD should be estimated in different ways. In another case, suppose the set of join conditions is {A.id = B.id, A.id = C.id and A.id = D.id}, then the set {B, C, D} is an inner join set. This is because i) the join between B and C is inner join due to the transitivity of join conditions (B can be joined with A and A can be joined with B on the common attribute* id*) and ii) similarly, the join between C and D is also inner join.*

Consider an intermediate result *s* that does not correspond to any running sub-plan. *S* is the set of involved streams. We estimate its cardinality *Card(s)* according to the following three cases: Firstly, if there are no join conditions declared between streams in *S*, then *Card(s)* is simply the product of cardinality values of all related source streams. Secondly, in the case where *S* is an inner join set according to Definition 4.2.2, then we consider newly-collected cardinalities that contain S. Furthermore, for this kind of cardinalities, the fewer streams it contains, the closer its value is to the actual value of *Card(s)*. Therefore, we classify these cardinalities according to the number of streams involved. The heuristic is to find out the minimal value of the cardinalities with the fewest streams. Then, the value is used as the estimation for $Card(s)$. Finally, if a subset *T* of *S* is an inner join set, then we estimate the cardinality of the intermediate result *t* composed of streams in *T* by using the method of estimating inner joins. And then *s*'s cardinality can be computed by considering *t* and streams in *S* - *T* as outer joins.

Algorithm 2 shows the technique for the above discussion. We recursively estimate cardinalities with the most number of streams (line 4). Note that, the loop does not start with N, because the cardinality that contains all source streams is the final output and hence its value can be collected. Besides, as discussed previously, source streams' cardinalities also can be gathered, and hence the loop ends with 2.

For each targeted cardinality (line 7), we directly estimate its value, if the first two cases were met (from line 4 to 12). In the case that all joins are outer joins, it is straightforward to assign the product of single streams' values (line 12). On the other hand, if all joins are inner joins, the value is computed via a recursive function Min (line 9). In the function Min, if cardinalities that involves $(n+1)$ streams as well as contains $Set(s)$ can be found, the minimal value is identified and returned. Otherwise, the function searches cardinalities that contains more streams by calling $Min(Set(s), n+2, N, C)$. In the worst case, the process terminates when the second and third parameters are equal and the cardinality value of the query's output is returned. Finally, for cardinalities belonging to the third case, cardinalities are estimated by combining the inner join portion and outer join portion together (line 20).

Algorithm 2: Estimating Absent Cardinalities

```
input  : The set $C$ of newly-gathered cardinality values about a query,
         the set $S$ of source streams,
         the number of source streams $N$
output: The set of cardinality values that the optimizer needs

Let $T$ store all cardinalities that are absent from optimal plans' routines;
Let $Card(s)$ indicate the cardinality value for intermediate result $s$ in $T$;
Let $num$ indicate the number of source streams contained in a cardinality;
for $n \leftarrow N - 1$ to $2$ do
    foreach cardinality $Card(s)$ in $T$ do
        $num \leftarrow$ the number of source streams in $s$;
        if $num$ is equal to $n$ then
            if $Set(s)$ is an inner join set then
                $Card(s)$ = Min ($Set(s)$, $n + 1$, $N$, $C$);
                Put $Card(s)$ into $C$;
            else if $Set(s)$ contains no inner joins then
                $Card(s)$ = Product ($Set(s)$, $S$, $C$);
                Put $Card(s)$ into $C$;
for $n \leftarrow N - 1$ to $2$ do
    foreach cardinality $Card(s)$ in $T$ do
        if $Card(s)$ has been computed then continue;
        Identify the inner join set $Subset(s)$;
        $V_{inner}$ = Min ($Subset(s)$, $n + 1$, $N$, $C$);
        $V_{outer}$ = Product ($Set(s) - Subset(s)$, $S$, $C$,);
        $Card(s) = V_{inner} \times V_{outer}$;
return $T$;
```

We give the example below to illustrate the process.

Example 4.2.2 *Consider a query on streams A, B and C. The cardinalities for A, B and C are 10, 20 and 30, respectively. The number of final output ABC is 8. Plans are A:B→C, B:A→C and C:A→B. The absent cardinality is only BC. If {B, C} is not an inner join set (e.g., when join conditions are A.b = B.b and A.c = C.c), then the cardinality value is estimated as* $20 \times 30 = 600$. *Otherwise, there is a case that {B, C} is an inner join set, for example, when join conditions are A.id = B.id and A.id = C.id.*

In this example, the cardinality containing BC is only ABC, and hence 8 is used as the estimation for BC.

4.3 Cost Model

Based on the discussion on cardinalities, we now present the cost model used by the optimizer. Intuitively, the cost model should consider the costs of inserting newly-arriving tuples and invalidating expired tuples. However, as discussed in Section 3.3, once the query is submitted, the kinds of indexes for streams are decided, meaning that both of those costs are fixed. Therefore, we only consider the probing cost. As follows, we describe analytical formulas for the join selectivity and the joining cost.

4.3.1 Join Selectivity

In our context, join plans are illustrated by left-deep trees, and the task of each join operator (tree node) is to use the tuples of the left-side input (left child) to find matches in the right-side input (right child). Given a binary join involving streams o_0 to o_n, we denote the selectivity as $S(o_0...o_{n-1}, o_n)$, where $o_0...o_{n-1}$ and o_n respectively represent the left and right inputs.

The join selectivity is usually defined as the ratio of the number of matched tuples, that is, output's cardinality, over the Cartesian product of the inputs' cardinalities. In terms of the selectivity evaluation, there are two cases: (i) If necessary cardinalities can be accurately obtained by the statistics collection component, we compute the selectivity by using these cardinalities. (ii) In the case that some of the related cardinalities cannot be gathered, estimation is needed. Usually, join selectivities are estimated according to columns' distinct values. However, in our context, maintaining histograms that record these information is prohibitively expensive, because update is needed when new data arrive, and the influence revocation should be performed when the corresponding tuples get expired. Based on the practical consideration, we assign the minimal value among all pairwise selectivities that involve o_n. The selectivity in the above two cases can be formally represented as:

$$S(o_0...o_{n-1}, o_n) = \begin{cases} \dfrac{Card(o_0...o_n)}{Card(o_0...o_{n-1}) \times Card(o_n)}, & \text{case (i)} \quad (4.1) \\ min\{S(o_i, o_n) | 0 \leq i < n\}, & \text{case (ii)} \quad (4.2) \end{cases}$$

The underlying assumptions for Equation 4.2 are independency of join conditions and the uniform distribution of join columns' values. As long as the assumptions hold, Equation 4.2 provides the correct estimation. Moreover, in Equation 4.2, the pairwise selectivity $S(o_i, o_j)$ $(0 \leq i < j)$ also needs estimation if any cardinality of $Card(o_i o_j)$, $Card(o_i)$ and $Card(o_j)$ cannot be collected and hence the selectivity cannot be computed via Equation 4.1. Obviously, if o_i and o_j have no common attribute to be joined, then $S(o_i, o_j)$ should be 1. Otherwise, we estimate the selectivity by using estimated cardinalities. The estimation approach has been discussed in Section 4.2.2.

We use the following example to illustrate the above discussion.

Example 4.3.1 *Suppose there is a join query on streams A, B, C and D. Join conditions are $A.attr_1 = B.attr_1$, $B.attr_1 = C.attr_1$ and $D.attr_2 = A.attr_2$. The leftmost graph in Figure 4.2 shows the joining relationships among streams, and its edges are annotated with the attribute name that streams join on. Moreover, running plans that correspond to different start-up streams are shown in Figure 4.2. The statistics collection component is able to gather cardinalities of all source streams and intermediate results in these plans. Therefore, some selectivities can be computed using these cardinalities. For instance,*

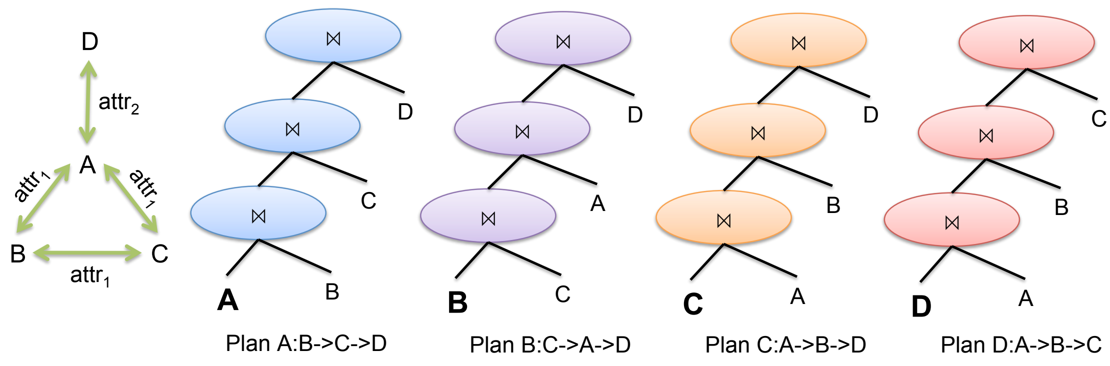


Figure 4.2: Join selectivity Computation and Estimation

S(AB, C) is assigned the value $\frac{Card(AB)}{Card(A)\times Card(B)}$*. Still, there are some selectivities need estimation, such as S(AC, D). According to Equation 4.2, it should be estimated as the minimal value between S(A, D) and S(C, D). Due to lack of Card(CD), we need to estimate S(C, D)'s value. According to the joining relationships in Figure 4.2, C and D have no join attribute, and therefore, S(C, D) is equal to 1. Since S(A, D) is not larger than S(C, D), S(AC, D) can further be decided as S(A, D).*

4.3.2 Cost Model

Table 4.2: Symbols used in the cost model

Symbol	**Description**
S(o_0...o_i, o_j)	the join selectivity of intermediate result o_0...o_i and stream o_j
Num(o_0...o_i)	the number of intermediate result produced by joining stream o_0 to o_i
Num(o_i)	the number of accessed tuples in stream o_i
Inc(o_0)	the number of newly-incoming tuples in the start-up stream
C	cost of accessing one tuple
C_h	cost of accessing one tuple via hashing
C_b	cost of accessing one tuple via binary searching
B_{o_i}	number of hash buckets in stream o_i

For a multiway join over *N* streams, we denote a plan candidate (i.e., ordering) as

$o_0 : o_1...o_{N-1}$, with o_0 indicating the start-up stream and $o_1...o_{N-1}$ representing the join sequence (discussed in Section 3.4).

For the startup join operation $o_0 \bowtie o_1$, it is newly-incoming tuples of o_0 that are used to probe tuples in stream o_1. For any other join operator $o_0...o_{i-1} \bowtie o_i$ $(1 < i < N)$, it is the intermediate results *t*s of joining stream o_0 to o_{i-1} $(1 < i < N)$ that trigger the task of probing tuples in stream o_i. With the symbols shown in Table 4.2, the cost model can be represented as:

$$Cost(o_0 : o_1...o_{N-1}) = C \times Num(o_1) \times Inc(o_0) + \sum_{i=2}^{N-1} C \times Num(o_i) \times Num(o_0...o_{i-1}) \tag{4.3}$$

The above equation captures the cost of all join operations in a plan. Amongst the used factors, $Num(o_0...o_i)$ $(0 < i < N-1)$, representing the number of the intermediate result, is defined as follows:

$$Num(o_0...o_i) = S(o_0...o_{i-1}, o_i) \times Card(o_0...o_{i-1}) \times Card(o_i) \tag{4.4}$$

Furthermore, as discussed in Section 3.3, a hash index or binary-search tree index would be built on the attribute of the stream that are probed. Therefore, the $C \times Num(o_i)$ $(0 < i < N)$ portion of every join operation in Equation 4.3 can be replaced:

$$C \times Num(o_i) = \begin{cases} C_h \times Card(o_i)/B_{o_i}, & \text{via hashing} \quad (4.5) \\ C_b \times \log_2 Card(o_i), & \text{via binary-searching} \quad (4.6) \end{cases}$$

The cost model is used in this way: The factors $Card$, Inc are collected statistics, and S are derived according to the discussion in Section 4.3.1. When re-optimization is needed, the optimizer re-evaluates plan costs by substituting the most recent values for these factors. The results generated actually are plan costs during the previous time interval, but we use these costs as indicators to distinguish best plans, assuming that a plan that runs quickly in the past would show good performance for a while.

The cost model mainly depends on cardinalities due to the following two reasons: On the one hand, cardinalities of source streams and intermediate results naturally themselves are very important factors in plans' execution costs. On the other hand, maintaining statistics in data stream management is very expensive. Among all kinds of data characteristics, cardinality is the easiest to be obtained by counting numbers of tuples that stream by, and therefore, it is the most cost-effective.

Chapter 5

Query Re-Optimization Framework

In this chapter, we propose a new framework to perform run-time re-optimization for multiway join queries over streaming data. First, we give an overview of the re-optimization process in Section 5.1. Next, we describe the main algorithms from Section 5.2 to Section 5.4. Section 5.2 illustrates the basis algorithm that identifies re-optimization conditions, and the way it is implemented and used at the time of plan generation and re-optimization checking. We discuss the limitations of the basis algorithm and propose the corresponding improvements in Section 5.3 and 5.4. Finally, at the end of Section 5.4, we give a complete example to illustrate how re-optimization is performed.

5.1 Overview of Re-Optimization Process

Our scheme's main idea is to dynamically detect and correct sub-optimal plans by comparing cardinality estimation used by the optimizer with actual values measured on the fly. Figure 5.1 shows an overview of our scheme. For clarity, we distinguish between the initialization run (i.e., the process that the optimizer performs before query execution) and the re-optimization run.

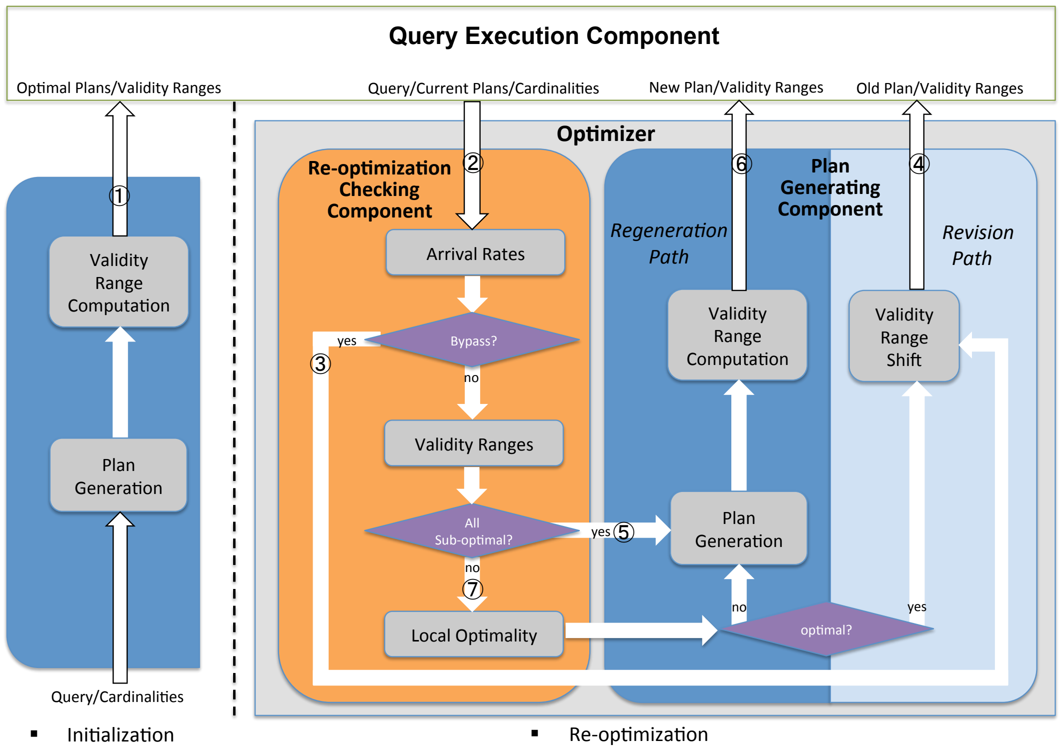


Figure 5.1: Re-Optimizer's overview

Given a query, the initialization run is performed only once. The optimizer generates the best plan along with *validity range*s computed for its cardinalities (① in the Figure 5.1). The validity range, a borrowed notation from the classic re-optimization work (Markl et al., 2004), essentially indicates a value interval of a cardinality, and it is an important data structure used to invoke re-optimization. We will talk about the issues of validity ranges in detail in Section 5.2.

During query execution, the re-optimization may run any number of times as long as query sub-optimality is found. The re-optimizer includes two components: One is a re-optimization checking component. This component has three phases, which are indicated as rounded rectangles with *arrival rates*, *validity ranges* and *local optimality*, and it checks whether re-optimization process is needed. The other one is a two-path plan generating component. For sub-optimal plans, the component generates new plans and validity ranges. For optimal ones, the existing validity ranges are revised. According to their tasks, we call them *regeneration* and *revison path*, respectively (see Figure 5.1). We note that the initialization process discussed in the previous paragraph is essentially the regeneration path in the plan-generating component.

Re-optimization checking for join queries is performed periodically. For a join query, during each period, its currently-running plans and newly-detected cardinalities are fed into the three-phase checking component (② in Figure 5.1). There are three phases. In the first phase, the checking component examines *arrival rates* of source streams. The less the arrival rates vary from the previous information, the more possible the remaining checking processes are completely bypassed (③ in Figure 5.1). In the case that further checking is bypassed, the plan-generating component revises validity ranges and outputs the old plans to the execution engine (④ in Figure 5.1). Otherwise, the checking component continues to do the second phase by individually comparing the current value of a cardinality with its corresponding validity range. A plan can be associated with many validity ranges and it is determined to be sub-optimal as long as a cardinality value falls beyond its corresponding range. At the completion of the second phase, if all plans are found to be sub-optimal, they are directly delivered to the plan-generating component to generate better plans and compute new validity ranges as well (⑤ and ⑥ in Figure 5.1). Also, it is possible that some plans are detected to become sub-optimal while some others are not. The third phase checks whether sub-optimality exists in those plans that are currently regarded as optimal. At the completion of the third phase, plans that remain optimal will still be used (④ in Figure 5.1) while sub-optimal ones will be replaced with better plans (⑥ in Figure 5.1).

The method associated with validity ranges is the basis of both re-optimization checking and plan generating component, and therefore, we will first introduce this method in the following section.

5.2 Identifying Re-Optimization Conditions

Optimizers are well designed to produce the optimal plans by using cost models. Although cost models differ from system to system, they all heavily rely on cardinality estimation. Generally, an optimal plan becomes sub-optimal when actual cardinality values significantly differ from the estimation. In these scenarios, a straightforward way is to call the optimizer and generate a new plan. However, this kind of *optimize-always* strategy has a clear disadvantage, that is, some re-optimization runs are wasteful because they may generate the same plan as the one used before re-optimization. To minimize the risk of redundant re-optimizations, some traditional DBMSs propose an alternative way: When choosing a plan as the execution plan, the optimizer also identifies conditions where the plan keeps optimality. During execution, re-optimization is invoked only when the pre-computed conditions are violated.

As discussed in Chapter 1, data-stream management also has the re-optimization issue. Due to the time-varying nature of streaming data, cardinalities detected at runtime are most likely different from the optimizer's previous information, meaning that running plans probably become sub-optimal. In the I/O-free environment, the cost of frequent re-optimizations is prohibitively high, and therefore, it is only acceptable to perform re-optimization under some conditions.

5.2.1 Computing Validity Ranges

We follow the main idea of computing *validity range*s as optimality conditions, which is proposed by an existing and representative work *Progressive Query Optimization* (POP) (Markl et al., 2004). The motivation is that, cost models heavily rely on cardinalities, and moreover, some cardinalities can be easily monitored when executing the currently-running plan. Therefore, regions of optimality can be characterized by value-based cardinalities. POP pre-computes validity ranges for those cardinalities, one for each cardinality. Those validity ranges indicate the conditions where the currently-chosen plan remains optimal. At run-time, when actual cardinality values fall inside their corresponding ranges, re-optimization can be avoided without hurting the quality of the optimizer's decisions.

A validity range is essentially a value interval limited by an upper and lower bound. Most importantly, among the values included, a validity range must contain the current value of the corresponding cardinality within it. Although we still use the concept of validity range, we modified the concrete techniques. In the original work, validity ranges are merely defined by upper bounds. However, considering that in data-stream environment, cardinality values are time-varying, we explicitly compute the lower bounds. Furthermore, there are two important considerations for computing validity ranges: Firstly, our cost model, as given in Section 4.3, is monotonic with respect to all cardinalities. Secondly, a binary join associates with two inputs and hence two cardinalities, that is, when considering a particular join operation, we actually compute validity ranges for the two cardinalities.

Technically, the computation for a validity range means determining the upper and lower bounds. Based on considerations mentioned above, we illustrate the corresponding methods in the following two subsections.

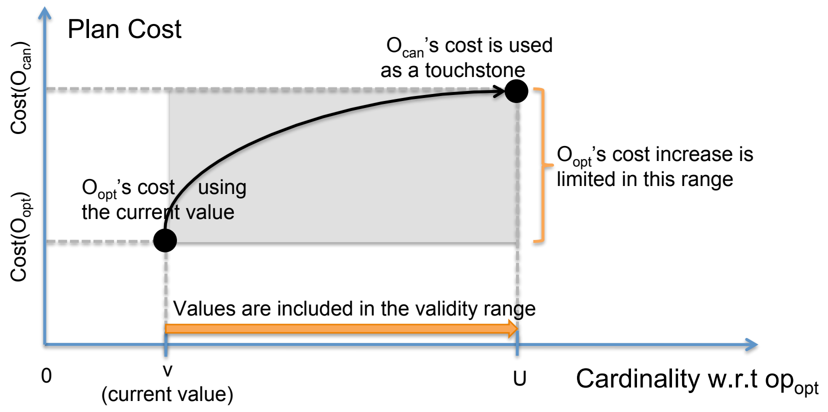

Plan Cost
Cost(O_can)
Cost(O_opt)
O_can's cost is used as a touchstone
O_opt's cost increase is limited in this range
O_opt's cost using the current value
Values are included in the validity range
0
v
(current value)
u
Cardinality w.r.t op_opt

Figure 5.2: Intuition of computing an upper bound

5.2.2 Determining Upper Bounds

The intuition of computing the upper bound is given as follows: For a given start-up stream, consider the top-most join operators op_{opt} and op_{can} of two corresponding orderings O_{opt} and O_{can}, which involve the same source streams and join conditions. O_{opt} has a smaller cost under the current cardinalities. According to the monotonicity, for a particular cardinality associated with op_{opt}, if its value increases, then the cost of O_{opt} would also increase, which is shown as the curve in Figure 5.2. Therefore, using O_{can}'s cost as a touchstone, we can bound values within the validity range until the cost of O_{opt} is larger than O_{can}'s for the first time. And this value is determined as the upper bound. We illustrate this intuition in the following example.

Example 5.2.1 *Consider two orderings A:B→C→D and A:B→D→C. They can be compared because both of their top-most join operators generate results joining streams A, B, C and D. Assume the cost of A:B→C→D is smaller. Its top-most join operation is ABC ⋈ D and hence the associated cardinalities are ABC and D. We will separately find upper bounds for cardinalities ABC and D, such that under them the cost of A:B→C→D reaches that of A:B→D→C.*

Algorithm 3: Computing the Upper Bound of a Validity Range

```
input  : Currently optimal ordering $O_{opt}$,
         a cardinality $Card$ associated with the top-most operator in $O_{opt}$,
         the current value $V_{opt}$ of $Card$,
         the current upper bound $U_{cur}$ of $Card$'s validity range,
         an ordering candidate $O_{can}$ and its cost $C_{can}$
output: Upper bound of $Card$'s validity range
1  Let $newDif$ and $curDif$ indicate the differences between the costs of $O_{opt}$ and
   $O_{can}$;
2  Let $low$ indicate a value such that under it $O_{opt}$'s cost is still smaller than $C_{can}$;
3  Let $high$ indicate a value such that under it $O_{opt}$'s cost is larger than $C_{can}$;
4  Let $gradient$ indicate the factor when exploring a larger value;
5  $low$ = $V_{opt}$;
6  $high$ = $V_{opt} \times FACTOR$;                    // FACTOR is more than 1
7  $curDif$ = $C_{can}$ - $C_{opt}$;
8  $newDif$ = $C_{can}$ - Cost ($O_{opt}$, $newValue$);
9  while $newDiff > 0$ do
10     $gradient$ = ($curDif$ - $newDif$) / ($newValue$ - $curValue$);
11     $low$ = $high$;
12     $curDif$ = $newDif$;
13     $high$ += $newDif$ / $gradient$ + 1;
14     $newDif$ = $C_{can}$ - Cost ($O_{opt}$, $newValue$);
15 return BinarySearch($low$, $high$, $C_{can}$);
```

We show the detailed techniques of the above description in Algorithm 3. Because our cost model is not continuous, we treat the computation as a numerical solving problem. Based on the current value (line 5), we iteratively explore a scope, within which the upper bound would reside (from line 9 to line 14). When computing, we use the gradient (line 10) that is derived from the current and previous loop to save exploration times. After the *while* loop, the scope is defined by variables *low* and *high*. The variable *low* indicates a specific value and under it, the O_{opt}'s cost is smaller. On the contrary, *high* indicates a value such that by using it, O_{opt}'s cost is larger than O_{can}'s. Finally, a binary search is applied. The search returns the smallest value within the scope that

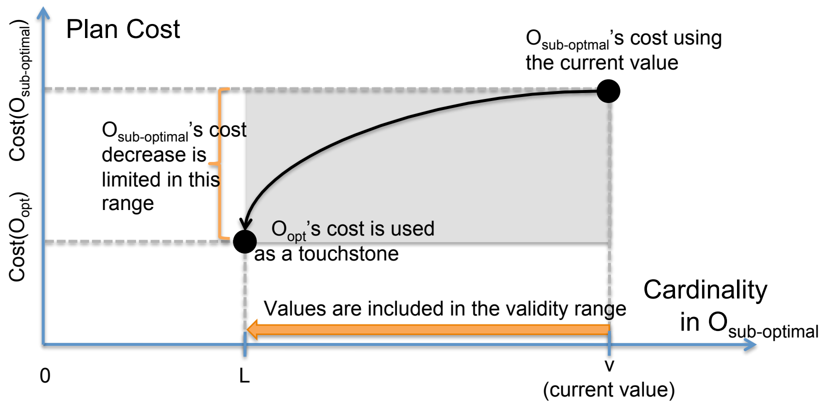


Figure 5.3: Intuition of computing a lower bound

causes an inversion between costs of O_{opt} and O_{can} (line 15). This returned value is the upper bound.

5.2.3 Determining Lower Bounds

The heuristic of computing upper bounds uses larger-cost plans to limit the maximal values that cardinalities can reach. Similarly, when we identify lower bounds, those larger-cost plans also can be utilized to obtain the minimal cardinality values. However, the details of deciding lower bounds are slightly different.

Given a query, if we have a best plan to execute it, then its cost can be used as the base line of how much a sub-optimal plan costs. Therefore, as illustrated in Figure 5.3, the cardinality values should be restricted such that the best plan remains optimal, even if the costs of sub-optimal plans are estimated under those values. Consider the top-most join operator of a sub-optimal plan shown in Figure 5.4. We know that the smallest cost of joining streams s_1, s_2, s_3 and s_4 is $Cost_{opt}$, which is the optimal plan's cost. We gradually decrease cardinality s_4's values and identify a value, when using it, the current sub-optimal plan has a smaller cost than $Cost_{opt}$. This value is determined as the lower

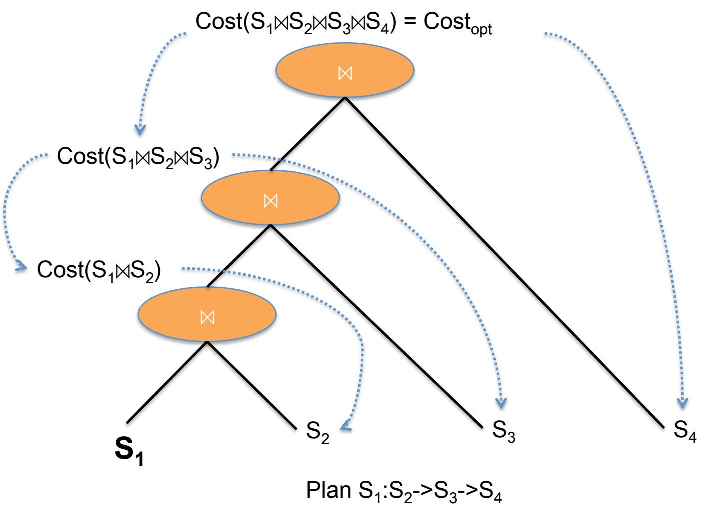


Figure 5.4: Base line distribution when computing a lower bound

bound of cardinality s_4 in the optimal plan. Furthermore, this strategy can be applied on the join sequence all the way forward, as indicated with dashed lines in Figure 5.4. For instance, the cost of joining s_1, s_2, s_3 and s_4 can be decomposed into two parts: the cost of $s_1 \bowtie s_2 \bowtie s_3$ and the cost of $s_1 s_2 s_3 \bowtie s_4$. Since the latter can be computed, we consequently know the base-line cost of the sub-plan $s_1 \bowtie s_2 \bowtie s_3$. Therefore, the lower bound of cardinality s_3 would be calculated. The computation for s_2 is similar.

We use a recursive program to perform the above progress over a sub-optimal plan. In each loop, The core computation is performed via binary search as line 15 in Algorithm 3, with parameters being 0, current value and the base-line cost. We omit the detailed algorithm for simplicity and implementation issues will be discussed in Section 5.2.4.

This method is able to determine the lower bounds of all source streams. However, the cardinalities of intermediate results remain. Considering the monotonicity property of our cost model, we set all lower bounds of intermediate results to be 0, because the optimality of the chosen plan is enhanced, if intermediate results' actual values are smaller than the optimizer's estimation.

5.2.4 Implementation in the Plan Generating Component

Having introduced the detailed algorithm, now we discuss how validity range computation is incorporated in our (re-)optimization framework. As discussed in Section 5.1, validity ranges are involved with both *regeneration* and *revision* paths of the plan generating component.

5.2.4.1 Regeneration Path

The main computation discussed in the previous two subsections is implemented in the regeneration path that generates an optimal plan as well as its corresponding validity ranges. As mentioned in Section 5.1, the regeneration path performs at both initialization and re-optimization time. It is natural to merge the computation into the plan enumeration and pruning phases of dynamic programming (discussed in Section 4.1): Suppose for a given query, there is an ordering O_{opt} which is a currently-optimal (sub-)plan for joining a set of streams. During plan enumeration (line 8 in Algorithm 1), the optimizer compares a candidate O_{can} with O_{opt}. If O_{can}'s cost is larger, before O_{can} is pruned (line 10 in Algorithm 1), its cost is used to refine the upper and lower bounds of O_{opt}'s validity ranges. Otherwise, the optimal ordering should be replaced with O_{can} and validity ranges of the new optimal orderings' cardinalities would be computed. As such, at the completion of dynamic programming, an optimal plan is produced and validity ranges of the associated cardinalities are also obtained.

It is worth pointing out that, even though the validity range computation and the plan generation can be combined together as the above description, we show them separately in Figure 5.1. This is because the combination causes a problem: If the currently-optimal plan O_{opt} is replaced by a newly-enumerated plan O_{can} that has a smaller cost, the previous computation for O_{opt} is wasteful. To avoid this, we apply an efficient implementation that additionally records the smallest and second-smallest costs for executing every set of join conditions. According to those information, we only compute validity ranges when the optimal plan is decided.

5.2.4.2 Revision Path

Validity ranges are also involved with the revision path of the plan generating component. This path is only invoked at run-time, when a currently-running plan is found to be optimal after re-optimization checking. In this case, the associated cardinalities are unchanged and hence there is no need to completely re-compute their validity ranges. A straightforward thought is to leave the current validity ranges untouched. Theoretically, there is nothing wrong. However, because these validity ranges are not computed for newly-detected cardinality values, violations most likely occur at the next checking. Therefore, a simple heuristic we use as a compromise is to shift the validity ranges according to the differences between their current and previous cardinality values. Consider a validity range for data source *s*, with *lb* and *ub* indicating the lower and upper bound, respectively. The plan keeps optimal when the cardinality value is *Card(s, t-1)* and *Card(s, t)*, respectively. According to our heuristic, the upper bound of the validity range is moved to *ub+Card(s,t)-Card(s,t-1)*, and similarly, the lower bound is changed to *lb+Card(s,t)-Card(s,t-1)*. This compensation heuristic is not proposed from the accuracy viewpoint, instead, it aims to delay the real re-optimization to the place where it is indeed required.

Table 5.1: Modification on Algorithm 1 for a stream with length-based window

Conditions	**Actions**
$ws \leq low$	ws is returned as the upper bound
$low < ws < high$	return BinarySearch(low, ws, C_{can})
$high \leq ws$	return BinarySearch(low, hig, C_{can})

5.2.4.3 Considerations for Streams with Length-based Windows

Previous methods are proposed for general cases, regardless of whether validity ranges are processed for source streams or intermediate results, or whether there are window constraints. We now show considerations and implementation, when we deal with a cardinality of a source stream, which is restricted by a length-based window.

The size of the length-based window, denoted as *ws*, is naturally an upper bound. Recall that when computing the stream's upper bound (Algorithm 1), we explore two values, that is, *low* and *high*, to limit the range where the upper bound would be found (from line 9 to line 14). In this case, before feeding them into the binary search (line 15), we perform additional checking of comparing *ws* with values of *low* and *high*. Corresponding actions are shown in Table 5.1.

Moreover, as discussed in Section 4.2.1, an important property of the length-based window is that the cardinality keeps unchanged in a steady state and the value is essentially the window size. Therefore, when the stream's cardinality reaches the window size, its upper bound's computation (Section 5.2.2) and shift (Section 5.2.4.2) can be avoided as long as the current plan remains optimal.

5.2.5 Checking Validity Ranges

At run-time, re-optimization checking is performed periodically. Cardinality information gathered during execution is fed into the checking component. Now we illustrate how validity ranges are used to detect sub-optimality.

As shown in Figure 5.1, checking validity ranges is the second phase. For plans transited into this phase, checking is only performed on those that are not ensured to be sub-optimal. For each plan, the most recent values of their cardinalities are compared to the corresponding validity ranges. As long as newly-changed values fall inside their own validity ranges, the plan is still optimal. Otherwise, re-optimization is needed because violations indicate that better plans are possible. If all plans associated with the same query are found to become sub-optimal, then the optimizer immediately generates better plans and computes new validity ranges. Otherwise, further checking would be performed by the next phase. An example is shown as below.

Example 5.2.2 *Consider again Example 3.4.2 on Page 42. The orderings A:B→C, B:A→C and C:A→B are selected as optimal plans for the corresponding streams. Assume at the beginning streams A, B and C's cardinalities are 10, 20 and 30, respectively. Take the plan A:B→C for stream A as an example. The associated cardinalities are B, C and AB, and their validity ranges have been computed. Note that, A's cardinality is also involved in the plan, however as the start-up stream, A's tuples are always used to probe other streams and hence it is meaningless to compute its validity range. Furthermore, assume the validity range for B is [0, 30]. After a period of execution, if B's cardinality value is detected to be larger than 30, then the re-optimization process will be invoked to generate a better ordering for stream A and validity ranges are computed from scratch. Otherwise, if no violation occurs, validity ranges of B, C and AB are shifted. For example, if B's value is 25, then the validity range is shifted to [5, 35].*

In this section, sub-optimality is detected by comparing the most recent cardinality values with their corresponding validity ranges. However, there are two limitations:

On the one hand, the checking mechanism reduces complexity and guarantees that re-optimization checking is completely independent with each other, but correspondingly, it loses the big picture on how relative those value changes are. On the other hand, considering that optimization cost is considerately high in the I/O-free environment, we avoid exploring all possible combinations of cardinality values. Instead, when computing upper and lower bounds for a particular cardinality, we assume values of all the other cardinalities stay unchanged. Therefore, re-optimization chances might be missed.

To alleviate the problems, we propose two efficient algorithms in the following two sections.

5.3 Considering Arrival Rates

As shown in Example 1.3.1, the problem of redundant re-optimization might occur under the validity range method discussed in the previous section. We note that, after re-optimization, join plans are supposed to be optimal. Moreover, we assume that, data properties, such as value distribution, would less likely have a significant change in a short while. In this case, the less arrival rates vary from the most recent ones, the more possible re-optimization can be bypassed.

Even though the concept of arrival rates is widely used in data-stream management, we explicitly give the definitions of *arrival rate* and *arrival rate vector* under re-optimization considerations in the following subsection.

5.3.1 Definition of Arrival Rate

Definition 5.3.1 *Consider that In our optimization framework, cardinality information was updated periodically and the time interval is fixed (discussed in Section 4.2). For* *a source stream, its* arrival rate *is the number of newly arrived tuples since the last re-optimization checking.*

Definition 5.3.2 *For a query joining N streams s_1 to s_N, its* arrival rate vector, *denoted as $V^{(t)}_{s_1,\ldots,s_N}$, is a list of all source streams' arrival rates at time* t. *Given an ordering of source streams, $V^{(t)}_{s_1,\ldots,s_N}$ can also be represented as $V^{(t)}$, for simplicity.*

The following is an example to illustrate the two definitions.

Example 5.3.1 *Suppose there is a join query on streams A, B, C and D. The most recent and present checking are performed at time t_{last} and $t_{present}$, respectively. Within the period between $t_{present}$ and t_{last}, the newly arriving tuples are 10, 20, 30 and 40 for A, B, C and D, respectively. Therefore, the arrival rates for A, B, C and D are 10, 20, 30 and 40, respectively. In this case, the arrival rate vector $V^{(t_{present})}_{A,B,C,D}$ is (10, 20, 30, 40).*

5.3.2 A Probabilistic Model

As we pointed out previously, the less arrival rates vary from the most recent ones, the more possible currently running plans stay optimal and hence re-optimization can be bypassed. Given two arrival rate vectors V^{t_1} and V^{t_2}, to quantify their difference, we define a distance function by using L1 norm:

$$d(V^{t_1}, V^{t_2}) = \frac{1}{2} \times \|\hat{V}^{t_1} - \hat{V}^{t_2}\|_1$$

where $\hat{V}^{t_i} = V^{t_i}/\|V^{t_i}\|_1$ is the normalized vector of vector V^{t_i}. Moreover, since $\|\hat{V}^{t_i}\|_1$ equals to 1, the component $\|\hat{V}^{t_1} - \hat{V}^{t_2}\|_1$ falls into the range [0, 2] ($\|\hat{V}^{t_1} - \hat{V}^{t_2}\|_1 \leq \|\hat{V}^{t_1} + \hat{V}^{t_2}\|_1$ = 2). Therefore, we use the factor $\frac{1}{2}$ to normalize the value to the range [0, 1].

A straightforward idea is to use $d(V^{(t_{last})}, V^{(t_{present})})$ to decide whether re-optimization is redundant and hence can be bypassed. We compute a probability p as follows:

$$p = \alpha + (1 - \alpha) \times d(V^{(t_{last})}, V^{(t_{present})}). \tag{5.1}$$

Considering that a small cardinality change might be sufficiently significant to lead to a new plan, we use a lower bound $\alpha \in [0, 1]$ in Equation 5.1. The factor α inditates that the probability of performing re-optimization is at least α and it is a user-defined constant. We note that, given the value of α, the smaller the probability p is, the more similar the two most recent arrival rate vectors are, meaning that the less likely currently running plans become sub-optimal. Therefore, we bypass re-optimization in the probability of $(1 - p)$ at the present checking.

An example below illustrates the main idea.

Example 5.3.2 *Consider a join query on streams A, B, C and D. Assume at the most recent checking, the arrival rate vector* $V^{(t_{last})}_{A,B,C,D}$ *is* $(100, 200, 300, 400)$. *At the present checking, the vector* $V^{(t_{present})}_{A,B,C,D}$ *becomes* $(200, 100, 300, 400)$. *Obviously, the normalized vectors* $\hat{V}^{(t_{last})}_{A,B,C,D}$ *and* $\hat{V}^{(t_{present})}_{A,B,C,D}$ *are* $(0.1, 0.2, 0.3, 0.4)$ *and* $(0.2, 0.1, 0.3, 0.4)$*, respectively. If the constant* α *is set to be 1/3, then according to the Equation 5.1, the probability* p *is computed to be* $1/3+(1-1/3)\times(|0.1-0.2|+|0.2-0.1|+|0.3-0.3|+|0.4-0.4|)\times 1/2$ *= 0.4. Therefore, we bypass re-optimization in the probability of 0.6.*

5.3.3 Checking Arrival Rates

In this subsection, we first depict the implementation of checking arrival rates, and then we show the actions that the (re-)optimizer takes after the checking.

Given a query, we first compute the probability p according to Equation 5.1 at the checking time. Then, in the probability of $(1-p)$, re-optimization is bypassed. As shown in Figure 5.1, this method is positioned at the first place of the checking component. For a query, there are two situations after arrival rates' difference is assessed. One is that re-optimization is bypassed. Then, the query is directly conveyed to the process of validity ranges shift (discussed in Section 5.2.4) in the plan generating component. After that, execution is resumed. In the other situation where re-optimization is not bypassed, all plans are fed into the second phase for validity range checking.

5.4 Detecting Local Optimality

As we discussed previously, re-optimization chances might be missed under the validity range method. We handle this problem in this section.

The commonality between the methods about validity ranges and arrival rates is that, they are based on cardinality values that are collected from the optimal plan's execution routine to decide whether it is still optimal. This causes the problem of local optimality, that is, even after re-optimization, the plan that it generates is still sub-optimal. As demonstrated in Example 1.3.2, local optimal occurs usually because the optimizer fails to utilize available and useful information.

In Esper, for a multiway join query, the optimizer gives every source stream an individual ordering. All these plans only uses newly-incoming tuples of its corresponding start-up stream to trigger join processing. Due to this property, there are chances that more combinations of source streams can be seen and hence more cardinalities can be collected. Therefore, we intend to use these cardinalities to detect local optimality.

5.4.1 Definition of Comparable Cardinality

For the purpose of exploiting available cardinalities, we first introduce the definitions of *comparable cardinalities* and *peer plans*.

Definition 5.4.1 *Consider two intermediate results u and v. The set of source streams contained in u and v are denoted as $S(u)$ and $S(v)$, respectively. We regard $Card(u)$ and $Card(v)$ to be comparable if the following requirements are met: 1) The numbers of elements in $S(u)$ and $S(v)$ are equal; 2) $S(u)$ and $S(v)$ are not equal, and 3) $S(u)$ and $S(v)$ contain some common source stream s. In this case, $Card(u)$ and $Card(v)$ are comparable w.r.t s. Note that this property is transitive, that is, if $Card(u)$ and $Card(v)$ are comparable w.r.t s, and $Card(w)$ and $Card(v)$ are comparable w.r.t s, then $Card(u)$ and $Card(w)$ are also comparable w.r.t s.*

Definition 5.4.2 *Recall that for a multiway join query involving N streams, Esper generates N join plans (i.e., join orderings), one for each source stream. We call these N plans as* peer plans *for simplicity in the remaining thesis.*

The following is an example to illustrate the above definitions.

Example 5.4.1 *Consider the three plans A:B→C, B:A→C and C:A→B shown in Figure 3.2 on Page 39. They are generated to answer the same query but for different source streams, they are peer plans under Definition 5.4.2. Moreover, $Card(AB)$ and $Card(AC)$, that are intermediate results of A and C's plans respectively, are comparable w.r.t A, according to Definition 5.4.1.*

5.4.2 Combating Local Optimality

To minimize the risk of being stuck in local optimality, we focus on cardinalities collected from the startup join operators of all peer plans of the same query.

Our heuristic is based on the observation that, the fewer intermediate results the plan must process, the quicker it runs. Furthermore, the output size of the startup join operator forms the basis of the total number of intermediate results. Therefore, for the plan assigned for a specific stream, if we can guarantee that its startup join operator produces the fewest intermediate results by probing the current stream instead of any other choice, the probability that the plan is locally optimal is very small.

Algorithm 4: Detecting Local Optimality for a Join Ordering

```
input : A multiway join query $Q$,
        an associated join ordering $o_0 : o_1 ... o_{N-1}$,
        the set $P$ of cardinalities collected from all plans of $Q$
output: Whether the plan $o_0 : o_1 ... o_{N-1}$ is locally optimal
1 Let $ir$ indicate the startup join output of the given plan and initialize it to be $o_0 o_1$;
2 foreach cardinality $p$ in $P$ do
3     if $p$ and $ir$ are comparable w.r.t $o_0$ then
4         if $Card(p) < Card(ir)$ then
5             return $YES$;
6 return $NO$;
```

The corresponding technique is simple as shown in Algorithm 5. To enable the comparison, we take as input the cardinality information among all peers of the same query. This means, our heuristic allows intra-query sharing but still maintains inter-query independency. The advantage is that it minimizes the potential influence on the system performance, such as, the limitation on parallelism. For each plan's startup join operation, we identify comparable cardinalities (line 3) and check whether they produce fewer intermediate results(line 4). If the condition is satisfied, then local optimality exists.

5.4.3 Checking Local Optimality

The method demonstrated in the previous subsection is implemented as the third (last) phase of the re-optimization checking component. Therefore, only if there are plans that are suspected to be optimal, Algorithm 5 is applied. After the checking, plans would be fed into the plan generating component. An example is shown as below.

Example 5.4.2 *Consider the query mentioned in Example 3.4.2 on Page 42. The optimizer assigns plans A:B→C, B:A→C and C:A→B for streams A, B and C, respectively. Suppose stream A's plan is not ensured to be sub-optimal. Its intermediate result is only AB in this case. The set of cardinalities collected from all plans is {A, B, C, AB, AC, ABC}. According to the Algorithm 3, cardinalities of AB and AC are compared. Suppose the newly-gathered cardinality values of AB and AC are 5 and 10, it is better for A to join with B first and hence we regard the plan is still optimal.*

Putting-it-all-together: Finally, we show a comprehensive example illustrating how the (re-)optimizer works.

Example 5.4.3 *Consider Figure 5.5. Three plans A:B→C, B:A→C and C:A→B, are running to execute the same query. At the time of re-optimization checking, the re-optimizer takes as inputs those plans and newly-gathered cardinality information. Firstly, plans are checked by the phase about arrival rates (Section 5.3). If re-optimization checkings are decided to be bypassed, no plans would be changed, but still validity ranges would be shifted. Then, the execution is resumed with current plans. Otherwise, if there is at least one plan is detected to be sub-optimal, all plans are fed into the second phase about validity ranges (Section 5.2). Checking is performed as shown in Example 5.2.2. After that, if all plans are found to become sub-optimal, then further checking is bypassed and the re-optimizer generates better plans along with new validity ranges. Then,*

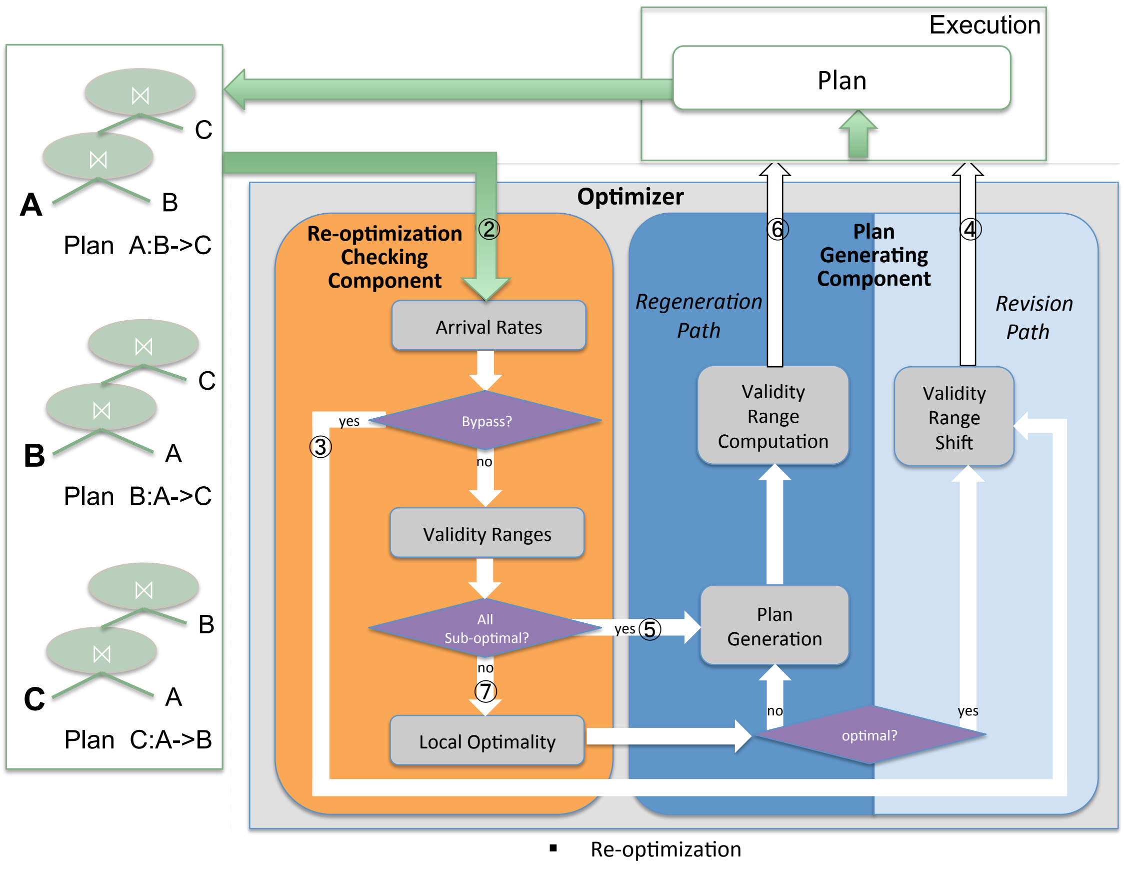


Figure 5.5: Re-Optimization progress

the execution is resumed with new plans. If at the completion of the second phase, some plans are still not ensured to be sub-optimal, then they are checked, as shown in Example 5.4.2, by the third (last) phase of local optimality (Section 5.4). Different actions would be taken by the plan generating component, according to the final decision on whether the plan remains optimal.

Chapter 6

Performance Study

We implemented our scheme on Esper's source code (http://esper.codehaus.org). Major modifications were implementing techniques discussed in Chapters **??** and 5. Morevoer, as Esper only counts the number of query output, we extended it to collect cardinality values of source streams and intermediate results.

Our goal is to validate our re-optimization framework, compare performance of various schemes on multiway joins and explore complexity associated with query processing in streaming environments. To do so, the modified Esper provided three re-optimization modes. The first was an optimize-once solution, that is, all queries were executed via plans produced by dynamic programming and no run-time re-optimization process was possible. The second one was the adapted mechanism that used validity ranges to detect sub-optimality. The third mode was the framework that consisted of a three-phase checking and two-path plan generating component. For simplicity, we denote them as $BASE$, POP' and $CARD$ respectively.

6.1 Experimental Setup

In our experiments, source streams were generated synthetically. Each stream instance is essentially a file containing tuples in a way they would appear if exported from a relational database. We simply viewed the files as prefixes of (potentially infinite) streams. All data files used in our experiments contained 1 million tuples.

Table 6.1: Attribute description of stream tuples

Attribute	**Length**	**Description**
Identification number (ID)	8bytes	Primary key
Timestamp (TS)	8bytes	Necessary property of a stream tuple
Common (COM)	8bytes	Common attribute for queries to be applied on
Other (OTHER)	76bytes	Complementary information

The schemas of streams in our experiments were the same, as shown in Table 6.1. Every tuple had four attributes, that is, ID, TS, COM and OTHER. The ID attribute that uniquely defined a tuple was represented by integers. Every value of the TS attribute was initially assigned the system time when the tuple was generated. Then, we added a random small delay of at most 1 second to simulate network traffic. The OTHER attribute was filled with character strings such that every tuple had a length of 100 bytes. Most importantly, as in (Golab and Özsu, 2003), COM, represented by integers, was the common attribute that joins were performed on.

We generated three data sets, based on different distributions of COM. In the first set, which is indicated as *Uni-Set*, all source streams were generated with uniform distribution, but different streams had different value ranges. To ensure that result sets were not empty, we followed the principle of inclusion. For a *k*-stream join, we firstly generated a stream with a value range of (0,*500000*] on its COM attribute. The value ranges of the remaining streams' COM attributes were randomly chosen from (0,*500*], (0,*1000*], (0,*2000*], (0,*10000*], (0,*50000*] and (0,*100000*]. In the second set, indicated as *pUni-Set*, we also used uniform distribution, but we kept all streams' COM attributes in the same value range of (0,*500000*]. Moreover, for every single stream, every 100 thousand tuples, its value range was changed. Everytime, a new range was chosen randomly from (0,*1000*], (0,*2000*], (0,*10000*], (0,*50000*] and (0,*500000*]. Finally on the third data set *Zipf-Set*, Zipf distribution was used. We generated tuples whose value was *i* out of emphN numbers with probability according to the following equation:

$$p(i,N) = \frac{\frac{1}{i^s}}{\sum_{i=1}^{N} \frac{1}{i^s}}$$

Options for skew factor *s* and value range (0,*N*] in the above equation are given in Table 6.2. In the cases that skew factor was 0.2 or 0.4, all streams were generated with Zipf distribution. However, when skew factor was 0.6, we only let half of joined streams follow Zipf distribution, and their value ranges were chosen randomly from (0,*1000000*] and (0,*1500000*]. The other streams were generated uniformly, with value ranges chosen randomly from (0,*10000*], (0,*20000*] and (0,*50000*]. When skew factor was 0.8, streams were generated in a similar way.

Table 6.2: Zipf Distribution for Data Generation

Skew Factor	**# streams of Zipf**	**# streams of Uniform**	**Value Ranges used in Zipf**	**Value ranges used in Uniform**
0.2	6	0	(0,10000], (0,20000], (0,100000] and (0,200000]	-
0.4	6	0	(0,100000], (0,200000], (0,300000] and (0,1000000]	-
0.6	3	3	(0,1000000] and (0,1500000]	(0,10000], (0,20000] and (0,50000]
0.8	2	4	(0,10000000] and (0,20000000]	(0,5000], (0,10000] and (0,50000]

Table 6.3 lists the experimental parameters we used. For queries, we consider star-joins on the COM attributes. To present it more concretely, assuming S_1 to S_6 are

Table 6.3: Parameters used in experiments

Parameter	**Explanation**
Join attribute (ATT)	COM
Re-optimization checking frequency (Period Length)	1, 2 and 3 time unit
Number of streams (#)	3, 4, 5 and 6
Window Size (length-based)	5k, 10k and 15k
Value Distribution of COM	Uni-Set, pUni-Set and Zipf-Set
α (used in the 1-phase checking component)	1/3

the source streams, *where* clauses of our experimental queries was of the form:

$$\textbf{where } S_1.COM = S_2.COM \textbf{ and } S_1.COM = S_3.COM \textbf{ and}$$

$$S_1.COM = S_4.COM \textbf{ and } S_1.COM = S_5.COM \textbf{ and } S_1.COM = S_6.COM$$

In our schemes, we periodically checked if re-optimization was needed. We called this *period length* and it was defined by the number of time unit, which was the time interval of accessing and processing 1000 tuples. Moreover, we tested the system performance under different number of streams, length-based (also called tuple-based in some literature) window constraints.

All experiments were conducted in the following fashion: For a query involving *k* $(3 \leq k \leq 6)$ streams, we generated *k* streams by satisfying the constraints of corresponding experimental parameters. Moreover, to simulate general cases in streaming environments, we did not keep constant arrival rates for source streams. For each stream, we randomized an arrival rate periodically and then used it to control the subsequent few input reads. Also, we did not keep constant orderings of source streams from which tuples were fed into the system. After collecting enough tuples from all streams, we scrambled the tuples before feeding them into the system, in order to simulate the scenario that every arriving tuple came from a random source stream. We run the query and measured the time taken to complete the execution. Moreover, we measured the time taken on re-optimization checking and real re-optimization, when POP' and $CARD$ were used. For each experiment, we repeated the generation and execution procedure 10 times and reported the average time. We quantified the performance improvement, according to the following formula:

$$\frac{T_{BASE} - T_{re-opt}}{T_{BASE}} \times 100\%$$

where T_{BASE} indicates the average time taken to complete the query under the mode of BASE, and T_{re-opt} indicates the average time when re-optimization (POP' or $CARD$) was used.

All experiments were conducted on a Unix-based operating system with a 2.4GHz Intel Core 2 Duo Processor and 4 GB main memory. All algorithms are implemented in Java.

6.2 Overall Performance

Our first set of experiments study the overall performance of the three re-optimization modes ($BASE$, POP' and $CARD$) on the three data sets. We set the window size as 10K for each stream and the parameter α used in our first phase checking component as $\frac{1}{3}$. We varied the number of joining streams (#) and period length (PL).

6.2.1 Performance on Uni-Set

We began by executing queries (shown in Section 6.1) over data of the *Uni-Set* (discussed in Section 6.1) and compared the average execution time taken by $BASE$, POP' and $CARD$.

Detailed runtime breakdown are presented in Figures 6.1 to 6.4. Every bar represents the average execution time spent to complete the *k*-stream ($3 \leq k \leq 6$) join, where dashed part represents the reoptimization overhead and solid part represents the time taken to process tuples.

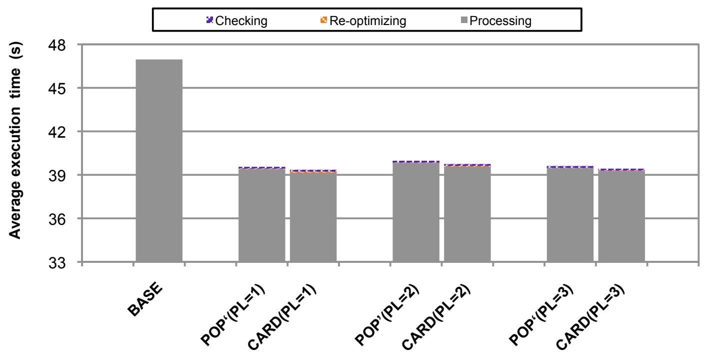


Figure 6.1: Runtime breakdown for 3-stream joins on Uni-Set

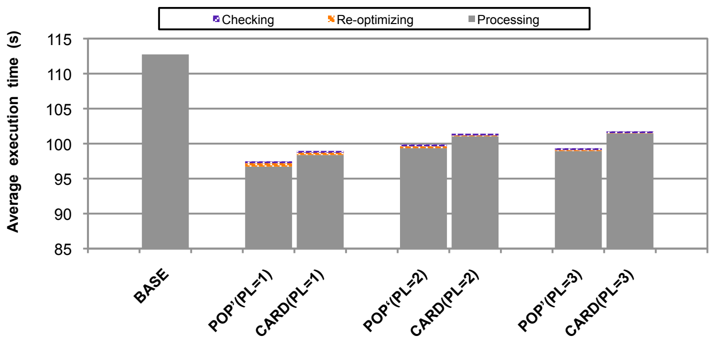


Figure 6.2: Runtime breakdown for 4-stream joins on Uni-Set

We see from Figures 6.1 to 6.4 that POP' and $CARD$ significantly outperformed $BASE$, and POP' and $CARD$ were robust, when the period length varied from 1 to 3. Moreover, the re-optimization costs increased as the number of streams became larger, because more possible plans needed to be explored. However, over the Uni-Set data, since the savings in execution time (solid bars) dominated, the re-optimization overhead (dashed bars) became very insignificant.

Table 6.4 summarized the performance improvement with respect to tuple processing and total execution time, when POP' and $CARD$ were compared to $BASE$

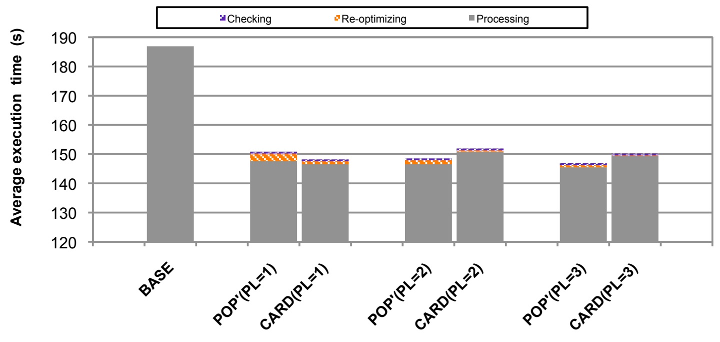


Figure 6.3: Runtime breakdown for 5-stream joins on Uni-Set

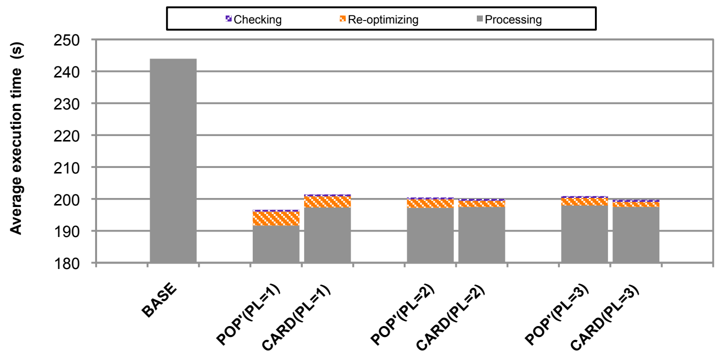


Figure 6.4: Runtime breakdown for 6-stream joins on Uni-Set

Table 6.4: Performance improvement (%) between three re-optimization modes over Uni-Set

#	**PL=1**				**PL=2**				**PL=3**			
	POP'		**CARD**		**POP'**		**CARD**		**POP'**		**CARD**	
	tuple	total	tuple	total	tuple	total	total	total	tuple	total	tuple	total
3	16.0	15.8	16.5	16.2	15.0	14.9	15.5	15.4	15.7	15.6	16.2	16.0
4	14.2	13.5	12.8	12.2	11.9	11.5	10.4	10.0	12.3	11.9	10.0	9.7
5	21.0	19.3	21.6	20.7	21.5	20.5	19.3	18.7	22.2	21.4	20.0	19.6
6	21.4	19.4	19.1	17.4	19.1	17.8	19.1	18.0	18.9	17.6	19.0	18.1

respectively. The notations *tuple* and *total* indicate the performance gain in terms of tuple processing time and total execution time, respectively. We see that optimization provided significant performance improvement by up to 21% over Uni-Set data. This reason is given as follows. The query plans that $BASE$ initially chose were more likely bad orderings. Furthermore, as $BASE$ was unable to do re-optimization, the bad plans needed much longer time to complete the queries.

6.2.2 Performance on pUni-Set

It is worth pointing out that, Esper (i.e., $BASE$) is very efficient for streams with similar properties. The first reason is that the multiple-plan-per-query strategy (discussed in Section 3.3) decreases the chance of choosing costly plans as well as limits the effect of costly plans. For example, if the plan assigned for stream S is inefficient, only the processing for S's newly arriving tuples would be influenced. Besides, since the plans assigned for different streams are independent, there are chances that other streams' plans are efficient. On the other hand, for equi-joins, Esper hard coded the hash join method (as discussed in Section 3.3), which is robust and efficient for processing data with non-skewed distribution.

We verified the observation experimentally as follows. We purposely generated streams such that the best and worst join plans could be identified beforehand. Therefore, we hard coded the best and worst join plans in the optimizer and tested their execution time, respectively. The experimental results showed no significant difference, indicating that (re-)optimization cannot provide much performance gain in the case that underlying streams share similar properties.

Therefore, it is expected that performance gain over data of the *pUni-Set* (discussed in Section 6.1) cannot be as much as it was over *Uni-Set*.

Detailed runtime breakdown are presented in Figures 6.5 to 6.8. Every bar represents the average execution time spent to complete the k-stream $(3 \leq k \leq 6)$ join, where dashed parts represent the time taken for the purpose of re-optimization and solid part represents the time taken to process tuples. These figures show that performance was improved when re-optimization was used.

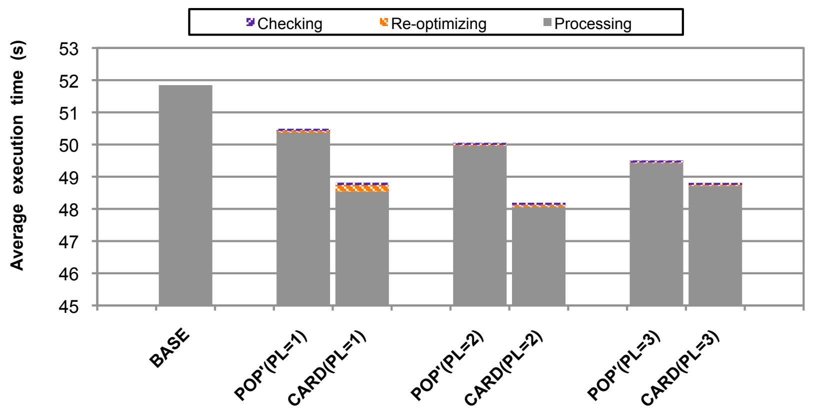


Figure 6.5: Runtime breakdown for 3-stream joins on pUni-Set

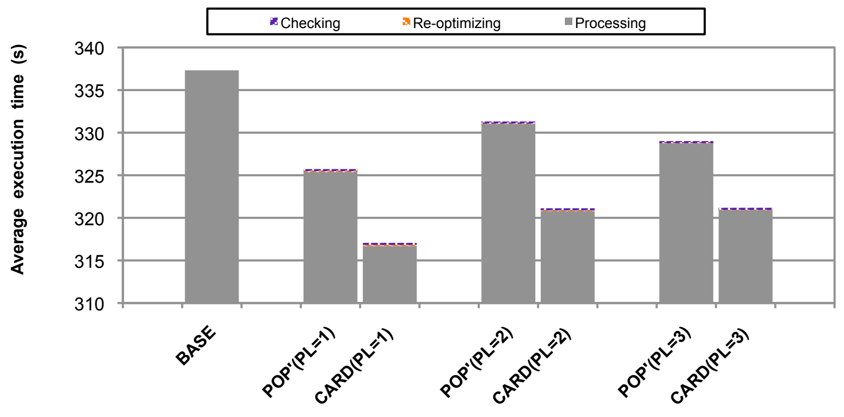


Figure 6.6: Runtime breakdown for 4-stream joins on pUni-Set

Table 6.5 summarized the performance improvement, when POP' and $CARD$ were compared to $BASE$ respectively, for pUni-Set data. The notations *tuple* and *total* indicate the performance gain in terms of tuple processing time and total execution

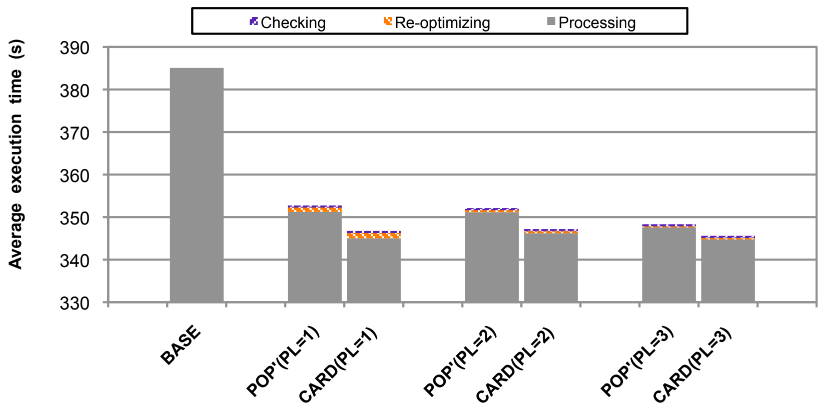


Figure 6.7: Runtime breakdown for 5-stream joins on pUni-Set

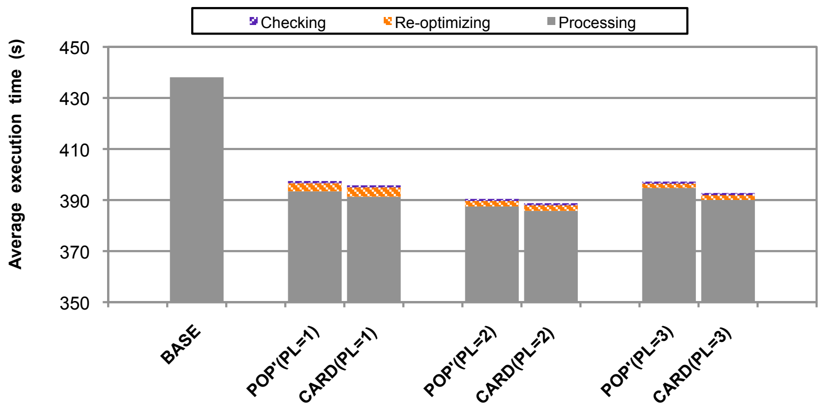


Figure 6.8: Runtime breakdown for 6-stream joins on pUni-Set

time, respectively. As we discussed previously, the performance gain was not as much as that of pUni-Set, compared to Table 6.4. Moreover, we note that, re-optimization benefit was gradually enhanced when the number of streams increased. This is because, the more number of streams are joined, the more possible plans can be chosen by the optimizer, and therefore the chances that $BASE$ would choose a good plan initially becomes smaller.. Moreover, over the pUni-Set data, $CARD$ improved POP' by about 3%, because $CARD$ could detect and correct more sub-optimality.

Table 6.5: Performance improvement ($\%$) between three re-optimization modes over pUni-Set data

#	**PL=1**				**PL=2**				**PL=3**			
	POP'		**CARD**		**POP'**		**CARD**		**POP'**		**CARD**	
	tuple	total	tuple	total	tuple	total	total	total	tuple	total	tuple	total
3	2.8	2.6	6.4	5.8	3.6	3.5	7.3	7.0	4.6	4.5	6.0	5.9
4	3.5	3.4	6.1	6.0	1.9	1.8	4.9	4.8	2.5	2.5	4.8	4.8
5	8.8	8.4	10.4	10.0	8.8	8.6	10.1	9.8	9.7	9.5	10.5	10.3
6	10.2	9.3	10.7	9.7	11.6	10.9	11.9	11.3	9.9	9.3	11.0	10.3

6.2.3 Performance on Zipf-Set

In this set of experiments, we only tested performances over 6-stream joins over skewed data. Figure 6.9 shows runtime breakdown of $BASE$, POP' and $CARD$. Every bar represents the average execution time, where grey bars represents the time taken to process tuples and bars of other colors represent the time taken for the purpose of re-optimization and solid part. Moreover, performance improvements of execution time were shown on top of bars of POP' and $CARD$.

From Figure 6.9, we only see around 5% performance improvement when skew factor was 0.2, 0.4 and 0.6. This is because the value ranges we chose to generate data did not produce tuples whose COM values occur many times, where $BASE$'s hash join

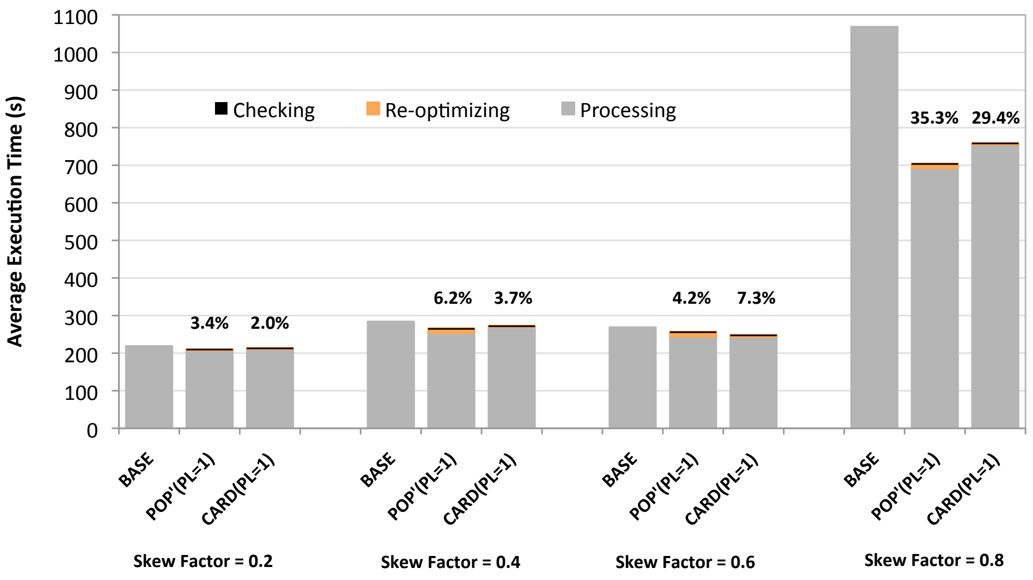


Figure 6.9: Runtime breakdown for 6-stream joins on Zipf-Set

method was very efficient. However, when skew factor was 0.8, equal values dramatically increased according to the features of Zipf distribution. In this case, POP' and $CARD$ were able to choose suitable streams to join first, leading to performance improvements by up to 35%.

6.3 Effect of Window Size

Window semantics, as constraints on tuples that would be processed, is an indispensable parameter while dealing with data streams. In this experiment set, we varied the window sizes imposed on streams. Window sizes were changed from a small size (i.e, 5000) to a medium one (i.e, 10000), and a larger size (i.e., 15000), in order to test their impact on join processing and re-optimization. We used re-optimization modes with a period length of 1 time unit as representatives.

6.3.1 Performance on Uni-Set and pUni-Set

On Uni-Set and pUni-Set data, we varied the number of streams from 3 to 6. The experimental results are shown in Figures 6.10 and 6.11, respectively. Moreover, we summarized the performance improvement when POP' and $CARD$ are compared with $BASE$ in Tables 6.6 and 6.7.

Table 6.6: Performance improvement (%) between three re-optimization modes under different window sizes over Uni-Set

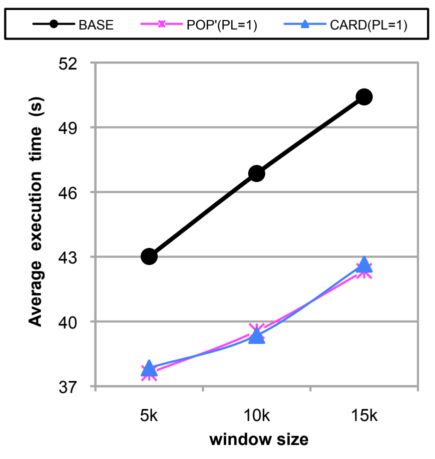
Window Size	**3-stream**		**4-stream**		**5-stream**		**6-stream**	
	POP'	CARD	POP'	CARD	POP'	CARD	POP'	CARD
5k	12.6	12.0	13.0	17.1	10.7	13.27	7.7	10.3
10k	15.6	16.0	13.6	12.2	19.3	20.7	19.4	17.4
15k	16.0	15.4	16.8	21.9	21.1	20.6	21.7	19.7

Table 6.7: Performance improvement (%) between three re-optimization modes under different window sizes over pUni-Set

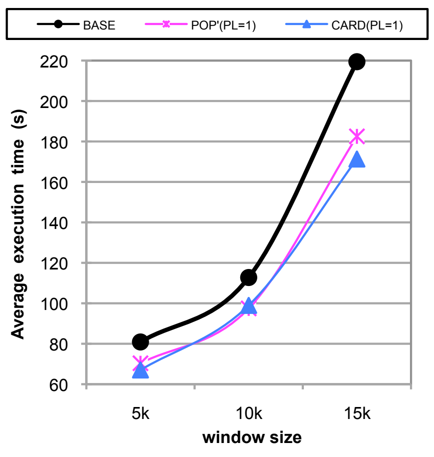
Window Size	**3-stream**		**4-stream**		**5-stream**		**6-stream**	
	POP'	CARD	POP'	CARD	POP'	CARD	POP'	CARD
5k	-0.01	2.0	0.9	3.6	7.1	5.8	6.5	7.0
10k	2.6	5.8	3.4	6.0	8.4	10.0	9.3	9.6
15k	2.5	6.1	6.1	9.7	19.1	30.0	18.9	30.1

We see from Tables 6.6 and 6.7 that re-optimization's benefit was steadily enhanced as window sizes became larger. This is because that under larger window sizes, bad join orderings (i.e., plans) would do more work on generating unnecessary intermediate results, but re-optimization schemes (POP' and $CARD$) were able to detect and hence avoid such sub-optimality efficiently. More importantly, when window sizes were 15000, $CARD$ outperformed POP' by 11%, because more sub-optimality could be detected.

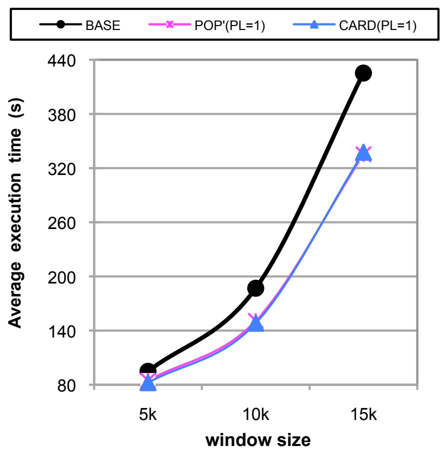
Over Uni-Set data, POP' and $CARD$ showed significant performance improvement by up to 30% in comparison to $BASE$. However, over pUni-Set data, we note



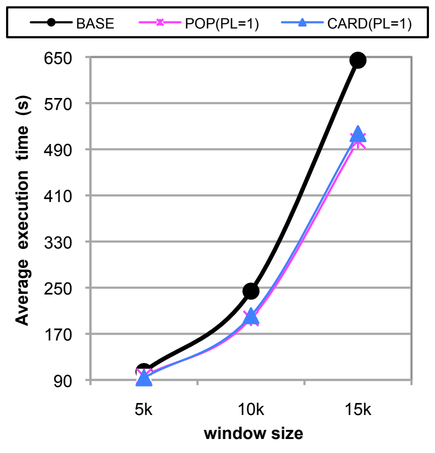
(a) 3-stream



(b) 4-stream

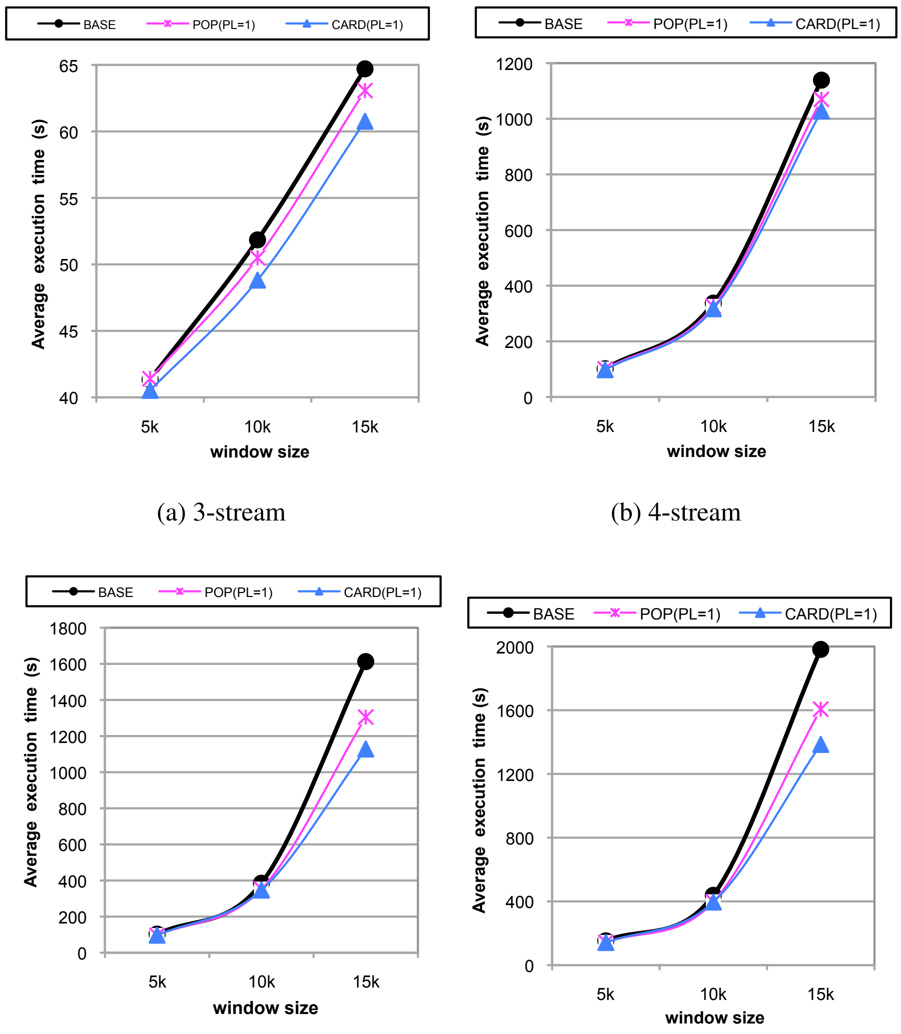


(c) 5-stream



(d) 6-stream

Figure 6.10: Performance of joins on Uni-Set w.r.t different window sizes



(a) 3-stream

(b) 4-stream

(c) 5-stream

(d) 6-stream

Figure 6.11: Performance of joins on pUni-Set w.r.t different window sizes

that the 3-stream join performance of POP' was worse than that of $BASE$, when window size was 5000, because that the re-optimization benefit was overshadowed by the re-optimization costs. This verified that in data-stream management, re-optimization overhead, compromised of checking and re-optimizing costs, cannot be ignored as in traditional DBMSs. Therefore, re-optimization should be used carefully, especially when the number of joining streams are few and the window sizes are small.

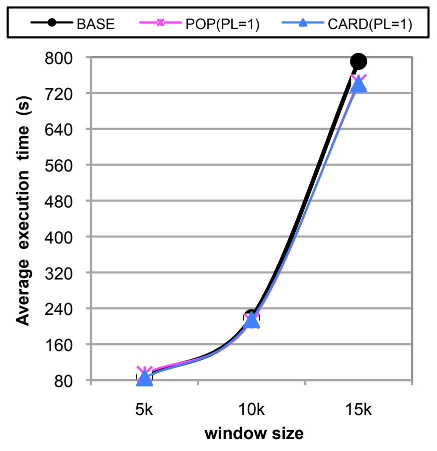
6.3.2 Performance on Zipf-Set

We tested 6-stream joins over Zipf-Set with different skew factors. The experimental results are shown in Figure 6.12. Moreover, we summarized the performance improvements when POP' and $CARD$ are compared with $BASE$ in Table 6.8.

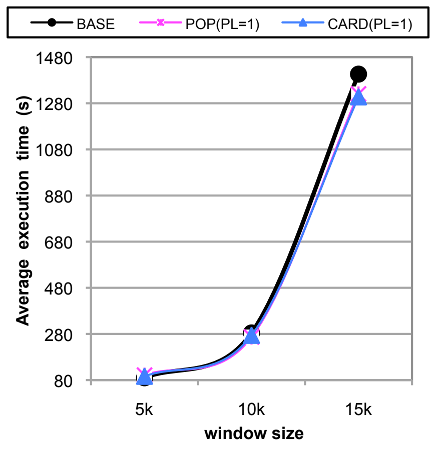
Table 6.8: Performance improvement (%) between three re-optimization modes under different window sizes over Zipf-Set

Window Size	**skew factor = 0.2**		**skew factor = 0.4**		**skew factor = 0.6**		**skew factor = 0.8**	
	POP'	CARD	POP'	CARD	POP'	CARD	POP'	CARD
5k	-7.2	2.8	-12.8	-7.0	-13.6	-5.3	-7.1	0.0
10k	3.4	2.0	6.2	3.7	4.2	7.3	35.3	29.4
15k	5.8	6.3	6.1	7.0	5.3	10.3	22.7	22.8

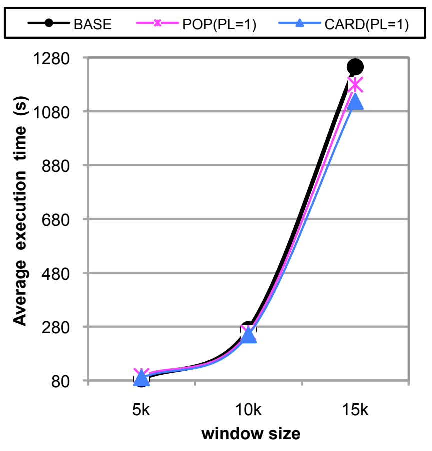
From Table 6.8, re-optimization was able to provide performance improvements by up to 35%. However, we see significant performance degradation when POP' was used. This is because under window size of 5000, there is little room for performance improvement, even worse, $POP's$ and $CARD$ needed more re-optimization runs over skewed data, causing significant overhead.



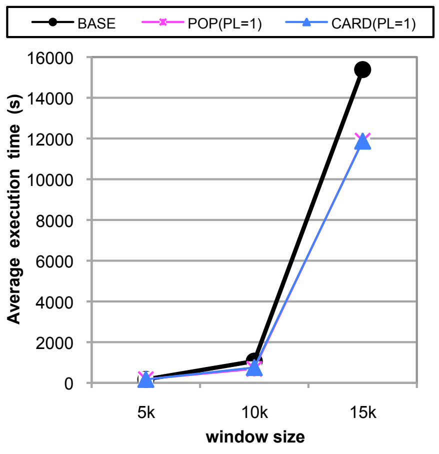
(a) skew factor = 0.2



(b) skew factor = 0.4



(c) skew factor = 0.6



(d) skew factor = 0.8

Figure 6.12: Performance of joins on Zipf-Set w.r.t different window sizes

Chapter 7

Conclusion and Future Work

Over the last few decades, the database community has largely relied on the same architecture for query processing: statistics collection, query optimization based on these statistics, and execution of query plans the the optimizer generates. This architecture has achieved great success in traditional DBMSs such that data-stream management also uses it.

In this kind of architecture, re-optimization methods play an important role in ensuring system efficiency. However, due to different features of traditional and streaming data, the way of performing re-optimization needs to be re-thought.

In this thesis, we propose a new re-optimization framework for multiway join queries over streaming data. We propose a novel re-optimization scheme that consists of a three-phase checking component and two-path plan generating component. The checking component determines if re-optimization is necessary. The first phase quantifies arrival rate changes to avoid redundant re-optimization. The second phase considers cardinality changes to detect sub-optimality. The third phase exploits useful cardinality information to alleviate local optimality. Additionally, we propose an analytical method to estimate cardinality values if they cannot be collected during execution.

We have implemented our scheme on Esper, a commercial stream engine. We explored the re-optimization performance over streaming data with varying value distributions, arrival rates and window sizes. Our experimental study shows that re-optimization techniques are able to provide significant performance improvement by up to 35%, in the real-time and constantly-varying environments.

Currently, we only consider cardinality information and arrival rates to decide whether re-optimization is needed. In future work, we plan to explore more kinds of statistics. Moreover, our heuristic of estimating cardinality values is quite simple, and we plan to explore how to obtain more accurate estimation, without taking too much computing resources or hurting the system performance.

References

Abadi, D. J., Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. 2005. The design of the borealis stream processing engine. In *CIDR*, pages 277–289.

Aboulnaga, A., P. J. Haas, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. 2004. Automated statistics collection in db2 udb. In *VLDB*, pages 1146–1157.

Avnur, R. and J. Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272.

Babcock, B., M. Datar, and R. Motwani. 2004. Load shedding for aggregation queries over data streams. In *ICDE Conference*, pages 350–361.

Babu, A. and P. Bizarro. 2005. Adaptive query processing in the looking glass. In *CIDR Conference*, pages 238–249.

Babu, S., P. Bizarro, and D. Dewitt. 2005. Proactive re-optimization. In *SIGMOD Conference*, pages 107–118.

Babu, S., R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. 2004. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418.

Babu, S., K. Munagala, J. Widom, and R. Motwani. 2005. Adaptive caching for continuous queries. In *ICDE Conference*, pages 118–129.

Babu, S. and J. Widom. 2004. Streamon: an adaptive engine for stream query processing. In *SIGMOD Conference*, pages 931–932.

Belknap, P., B. Dageville, K. Dias, and K. Yagoub. 2009. Self-tuning for sql performance in oracle database 11g. In *ICDE Conference*, pages 1694–1700.

Bizarro, P., N. Bruno, and D. J. DeWitt. 2009. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Eng.*, 21(4):582–594.

Calton, Ling Liu, Ling Liu, Calton Pu, and Wei Tang. 1999. Continual queries for internet-scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng.*, 11:610–628.

Carney, D., U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. 2002. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226.

Chandrasekaran, S., O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. 2003. Telegraphcq: Continuous dataflow processing. In *SIGMOD Conference*, page 668.

Chaudhuri, S., V. R. Narasayya, and R. Ramamurthy. 2009. Exact cardinality query optimization for optimizer testing. *PVLDB*, 2(1):994–1005.

Chaudhuri, S. and V. Narasayyaand R. Ramamurthy. 2008. A pay-as-you-go framework for query execution feedback. In *VLDB Conference*, pages 1141–1152.

Chen, J. J., D. DeWitt, F. Tian, and Y. Wang. 2000. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390.

Christodoulakis, S. 1984. Implications of certain assumptions in database performance evaluation. *TODS*, 9(2):163–186.

Chu, F., J. Y. Halpern, and Praveen Seshadri. 1999. Least expected cost query optimization: An exercise in utility. In *PODS Conference*, pages 138–147.

Cole, R. L. and G. Graefe. 1994. Optimization of dynamic query evaluation plans. In *SIGMOD Conference*, pages 150–160.

Cortes, C., K. Fisher, D. Pregibon, A. Rogers, and F. Smith. 2000. Hancock: A language for extracting signatures from data streams. In *SIGMOD Conference*, pages 9–17.

D., H., P. N. Darera, and J. R. Haritsa. 2007. On the production of anorexic plan diagrams. In *VLDB Conference*, pages 1081–1092.

Deshpande, A. and J. M. Hellerstein. 2004. Lifting the burden of history from adaptive query processing. In *VLDB Conference*, pages 948–959.

Deshpande, Amol. 2004. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49.

Dey, A., S. Bhaumik, Harish D., and J. R. Haritsa. 2008. Efficiently approximating query optimizer plan diagrams. *PVLDB*, 1(2):1325–1336.

Esmaili, K. S., T. Sanamrad, P. M. Fischer, and N. Tatbul. 2011. Changing flights in mid-air: a model for safely modifying continuous queries. In *SIGMOD Conference*, pages 613–624.

Esper. 2013. Esper. /urlhttp://esper.codehaus.org.

Eurviriyanukul, K., A. A. A. Fernandes, and N. W. Paton. 2006. A foundation for the replacement of pipelined physical join operators in adaptive query processing. In *EDBT Conference*, pages 589–600.

Eurviriyanukul, K., N. W. Paton, A. A. A. Fernandes, and S. J. Lynden. 2010. Adaptive join processing in pipelined plans. In *EDBT Conference*, pages 183–194.

Ganguly, S. 1998. Design and analysis of parametric query optimization algorithms. In *VLDB Conference*, pages 228–238.

Golab, L. and M. T. Özsu. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB Conference*, pages 500–511.

Graefe, G. and K. Ward. 1989. Dynamic query evaluation plans. In *SIGMOD Conference*, pages 358–366.

Hammad, M. A., M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. 2003. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308.

Haritsa, J. R. 2010. The picasso database query optimizer visualizer. *PVLDB*, 3(2):1517–1520.

Herodotos, Herodotou and Babu Shivnath. 2010. Xplus: a sql-tuning-aware query optimizer. *PVLDB*, 3(1):1149–1160.

Hulgeri, A. and S. Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB Conference*, pages 167–178.

Hulgeri, A. and S. Sudarshan. 2003. Anipqo: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *SIGMOD Conference*, pages 766–777.

Ioannidis, Y. and S. Christodoulakis. 1991. On the propagation of errors in the size of join results. In *SIGMOD Conference*, pages 268–277.

Ioannidis, Y. E., R. T. Ng, K. Shim, and T. K. Sellis. 1997. Parametric query optimization. In *VLDB Conference*, pages 132–151.

Ives, Z. G. 2002. *Efficient Query Processing for Data Integration*. Ph.D. thesis, The University of Washington.

Ives, Z. G., A. Y. Halevy, and D. S. Weld. 2004. Adapting to source properties in processing data integration queries. In *SIGMOD Conference*, pages 395–406.

Kabra, N. and D. DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD Conference*, pages 106–117.

Kang, J., J. F. Naughton, and S. Viglas. 2003. Evaluating window joins over unbounded streams. In *ICDE Conference*, pages 341–352.

Li, Q., M. Shao, V. Markl, K. S. Beyer, L. S. Colby, and G. M. Lohman. 2007. Adaptively reordering joins during query execution. In *ICDE Conference*, pages 26–35.

Madden, S., M. A. Shah, J. M. Hellerstein, and V. Raman. 2002. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60.

Markl, V., V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. 2004. Robust query processing through progressive optimization. In *SIGMOD Conference*, pages 659–670.

Prasad, V. G. V. 1999. Parametric query optimization: a geometric approach. Technical report.

Reddy, N. and J. Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *VLDB Conference*, pages 1228–1240.

Rundensteiner, E. A., L. P. Ding, T. M. Sutherland, Y. L. Zhu, B. Pielech, and N. Mehta. 2004. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Conference*, pages 1353–1356.

Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34.

Stillger, M., G. M. Lohman, V. Markl, and M. Kandil. 2001. Leo - db2's learning optimizer. In *VLDB Conference*, pages 19–28.

Tao, Y., M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. 2005. Rpj: Producing fast join results on streams through rate-based optimization. In *SIGMOD Conference*, pages 371–382.

Tatbul, N., U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. 2003. Load shedding in a data stream manager. In *VLDB Conference*, pages 309–320.

Urhan, T. and M. J. Franklin. 2000. Xjoin: a reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33.

Urhan, T., M. J. Franklin, and L. Amsaleg. 1998. Cost-based query scrambling for initial delays. In *SIGMOD Conference*, pages 130–141.

Viglas, S. and J. F. Naughton. 2002. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48.

Viglas, S., J. F. Naughton, and J. Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB Conference*, pages 285–296.

Wang, S., E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. 2006. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB Conference*, pages 619–630.

Wilschut, A. and P. Apers. 1991. Dataflow query execution in a parallel main-memory environment. In *International Conference on Parallel and Distributed Information System*, pages 68–77.

Yang, Y., J. Krämer, D. Papadias, and B. Seeger. 2007. Hybmig: A hybrid approach to dynamic plan migration for continuous queries. *IEEE Trans. Knowl. Data Eng.*, 19(3):398–411.

Yao, Y. and J. Gehrke. 2003. Query processing in sensor networks. In *CIDR Conference*, pages 233–244.

Zhu, Y. and D. Shasha. 2002. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB Conference*, pages 358–369.

Zhu, Y. L., E. A. Rundensteiner, and G. T. Heineman. 2004. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Conference*, pages 431–442.