SCALABLE PROCESSING OF MULTIPLE AGGREGATE CONTINUOUS QUERIES

by

Shenoda Guirguis

M.Sc. in Computer Science, University of Pittsburgh, 2010

M.Sc. in Computer Science, Alexandria University, 2006

B.Eng. in Computer Science and Engineering, Alexandria University, 2001

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Shenoda Guirguis

It was defended on

August 23, 2011

and approved by

Panos K. Chrysanthis, Professor, University of Pittsburgh

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

Kirk Pruhs, Professor, University of Pittsburgh

Mohamed Mokbel, Associate Professor, University of Minnesota

Mohamed Sharaf, Assistant Professor, The University of Queensland

Dissertation Advisors: Panos K. Chrysanthis, Professor, University of Pittsburgh,

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

SCALABLE PROCESSING OF MULTIPLE AGGREGATE CONTINUOUS QUERIES

Shenoda Guirguis, PhD

University of Pittsburgh, 2011

Data Stream Management Systems (DSMSs) were developed to be at the heart of every monitoring application. Monitoring applications typically register hundreds of Continuous Queries (CQs) in DSMSs in order to continuously process unbounded data streams to detect events of interest. DSMSs must be designed to efficiently handle unbounded streams with large volumes of data and large numbers of CQs, i.e., exhibit scalability. This need for scalability means that the underlying processing techniques a DSMS adopts should be optimized for high throughput (i.e., tuple output rate). Towards this, two main approaches were proposed in the literature: (1) Multiple Query Optimization (MQO) and (2) Scheduling. In this dissertation we focus on optimizing the processing of multiple Aggregate Continuous Queries (ACQs), given their high processing cost and popularity in all monitoring applications.

Specifically, in this dissertation, we explore shared processing of ACQs and introduce the concept of *'Weaveability'* as an indicator of the potential gains of sharing the processing of ACQs. We develop *Weave Share*, a multiple ACQs optimizer that considers the different uncorrelated factors of the processing cost, such as the input rate and ACQs' specifications. In order to fully reap the benefits of the new weave-based optimization techniques, we conceptualize a new underlying aggregate operator implementation and realize it in the *TriOps* framework. *TriOps* enables adaptive sharing of multiple ACQs that have different window specification, predicates and group-by attributes. The properties of the proposed techniques are studied analytically and their performance advantages are experimentally evaluated using simulation and in the context of the AQSIOS DSMS prototype.

Keywords Data Streams Management Systems, Continuous Queries, Query Optimization, Scalable Processing, Aggregation, AQSIOS, Weaveability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

PREFACE

No dream comes true without generous support of people and institutions. I am very grateful to the organizations that financially supported this work. Namely, the US National Science Foundation partially supported this work through out the NSF awards IIS-0534531 and IIS-0746696, and the University of Pittsburgh also partially supported this work through the School of Arts and Science Departmental Fellowship and by an Andrew Mellon Predoctoral Fellowship.

I am also truly very grateful to all the people who supported me throughout the years. First of all, I am grateful to my advisers, Panos K. Chrysanthis and Alexandros Labrinidis, who were to me not only advisers, but dear friends. I did learn a lot from them academically as well as personally. Their patience, hard work, pastorship of the work, and their being a model to follow were the main reasons behind my success. Sincere thanks are due to my PhD Committee memebers for their insightful and constructive feedback. Thanks are due to faculty members of the Computer Science department, each taught me something to treasure for the rest of my life. I am also very thankful and grateful to my family, my parents and my two brothers, without whom support and encouragement during hardships I would not have been able to reach to this successful stage. And thanks to the many colleagues and friends who made this journey of graduate school, a pleasant one.

And above all, I am thankful to God, Jesus Christ, Who is always there whenever I need Him.

1.0 INTRODUCTION

Data Stream Management Systems (DSMSs) deviate completely from the *store-then-query* paradigm of traditional database management systems (DBMS). In a DSMS, it is the queries that are stored or registered ahead of time, while the data arrives and is processed without the need for storing it first. DSMSs were developed to be at the heart of every monitoring application, from environmental monitoring, to patient care and outbreaks of disease detection, to network and financial market and cosmic phenomena monitoring (e.g., [6, 3, 7, 2, 73, 77, 39, 74, 56, 22]).

Monitoring applications register *Continuous Queries* (CQs) in DSMSs in order to continuously process unbounded data streams to detect events of interest. DSMSs are designed to efficiently handle unbounded streams with large volumes of data and large numbers of CQs, i.e., exhibit *scalability*. Thus, optimizing the processing of multiple continuous queries is imperative for DSMSs to reach their full potential.

In a typical monitoring application, hundreds of Aggregate Continuous Queries (ACQs) are normally registered [59] to monitor few input data streams. In fact, more than often, these ACQs are also computing the exact same aggregate function, but may have slightly different specifications. In particular, there are three characteristics that can vary among similar ACQs: (1) the window-specifications, (2) pre-aggregation filters (i.e., predicates) and (3) group-by attributes.

For example, a network monitoring application could employ three ACQs to monitor the IP traffic, all of which could compute the `COUNT` of incoming packets. While the first ACQ could report the count in the last minute, updated every five seconds, the second and third ACQs could report the count in the last minute, to be updated every half minute. Further, the first ACQ might be interested in the count of IP traffic originating from a specific source, i.e., have a predicate that the source IP has a certain value. The second and third ACQs, on the other hand, might be counting all received packets, but have them grouped by source IP and destination IP, respectively.

Given the proliferation and similarity of ACQs, and given the high data arrival rates, optimizing the processing of ACQs is crucial for scalability.

The need for scalability means that the underlying processing techniques a DSMS adopts should be optimized for high throughput (i.e., tuple output rate). Towards this, two main approaches were proposed in the literature: (1) *Query processing optimization* (e.g., [46, 47, 45, 81, 28, 29]) and (2) *Scheduling* (e.g., [11, 70, 30]). While query processing optimization aims at minimizing the processing delays, scheduling on the other hand aims to dynamically decide which operator to execute next in order to minimize queuing delays. In this dissertation, we focus on optimizing the processing of multiple Aggregate Continuous Queries (ACQs).

Towards optimizing the processing of ACQs, two categories of optimization techniques have been proposed in the literature: (1) efficient implementation of individual continuous operators and (2) multiple query optimization (MQO). Efficient implementation aims to adopt the most efficient techniques an operator can utilize to perform its task. On the other hand, MQO aims to generate query execution plans, typically statically, in order to minimize the tuple processing delay. The query execution plan consists of the operators of the submitted CQs and their execution order based on their interdependency.

Under the first category of techniques (i.e., operator implementation), *Partial Aggregation* has been proposed to minimize the repeated processing of overlapping data windows within a single ACQ (e.g., [46, 47, 29, 45]). In particular, the *Partial Aggregation* processing scheme aims at processing each input tuple only once and assemble the final aggregate value from a set of partial aggregate values. In this scheme, ACQ processing is modeled as a two-level (i.e., two-operator) query execution plan: in the first level a sub-aggregate function is computed over the data stream generating a stream of partial aggregates, whereas in the second level a final-aggregate function is computed over those partial aggregates.

Under the second category of techniques (i.e., multiple query optimization), the general principle is to minimize (or eliminate) the repeated processing of overlapping operations across multiple ACQs. This repetition occurs as a result of processing the same data by different queries, which exhibit an overlap in at least one of the three specifications, as mentioned earlier: (1) window specifications, (2) predicate conditions, or (3) group-by attributes.

In general, MQO is well known to be NP-hard for traditional database systems [67] as well

as for DSMSs [83]. Therefore, MQO techniques are typically based on heuristics that aim to share the processing of *common sub-expressions* among the execution plans of the various queries. This raises the challenge of identifying which sub-expressions are beneficial to share, if any [85, 42]. The optimization of multiple ACQs goes beyond the classic identification of common sub-expressions to exploiting the window semantics, the overlap of predicates, the overlap of the group-by attributes, and the *Partial Aggregation*; this is the challenge we are addressing in this dissertation.

On one hand, leveraging overlaps in predicate conditions and group-by attributes across different queries has been the focus of intense research in classical multiple query optimization. On the other hand, the shared processing of overlapping windows is a new area of research that has emerged with the paradigm shift for handling continuous queries with the use of scalable DSMSs.

A first step towards optimizing multiple ACQs, *Partial Aggregation* has been utilized to share the processing of multiple similar ACQs with different windows and predicates, but same group-by attributes, assuming it is always beneficial to share the partial aggregation [45]. The assumption that it is always beneficial to share is based on the premise that data arrival rate is the predominant factor in determining the sharing decision, where a high rate is always a precursor for plan sharing. It is also based on the observation that, in most practical applications, data streams do in fact exhibit a high rate. This approximation, however, considers only one facet of the problem (i.e., the characteristics of data streams) while diminishing the impact of the other facet (i.e., the characteristics of the registered ACQs).

Orthogonally, [59] proposed an extension to the classical subsumption-based multiple query optimization techniques towards sharing the processing of multiple ACQs with varying group-by attributes, but similar windows and predicates. This technique utilizes *Partial Aggregation* in a hierarchical fashion (i.e., more than two levels of aggregate operators). Regardless of the differences between the above multiple query optimization techniques, they all rely on the same concept of partial aggregation.

Given the crucially of optimizing the processing of multiple similar ACQs, and given the lack of a general technique that handles all different cases, one is intrigued to ask the following questions.

Q1. In addition to the data streams input rate, what other factors of the workload characteristics and ACQs properties affect the cost of a shared query plan? And more importantly, how do these factors interact with each other to affect the cost of a query plan?

Q2. Given our understanding of how the factors that affect the cost of the shared plan interact, can we design a multiple ACQs optimizer that considers all these factors while making the sharing decision? Could this new optimizer comprehensively handle all three cases of variability in the ACQs specifications (i.e., windows, predicates and group-by attributes)?

Q3. Given that ACQs are added to, and deleted from, the DSMS over time, and given that input rates also fluctuates, what is the best adaptive sharing strategy? In other words, when the workload characteristics changes, should the query plan be recomputed or be incrementally updated?

Q4. Is the currently widely-accepted *Partial Aggregation* technique the best continuous aggregation operator implementation for the shared processing of multiple ACQs?

In this dissertation, in answering the above questions, we argue that the properties and specifications of the installed ACQs are of equal importance to the workload characteristics (i.e., input rate and number of ACQs) in determining the sharing decision. In fact, the main thesis of this dissertation is that *intelligent shared processing of ACQs that (1) considers all factors that affect the processing cost of ACQs and (2) handles all cases of variability in ACQs specifications, is the key solution for achieving scalability in DSMSs*. We discuss our approach to address the above questions next.

1.1 APPROACH AND CHALLENGES

The objective of this dissertation is to identify the best strategies a DSMS should adopt in order to optimize the processing of ACQs, to achieve the desired level of system scalability. Towards achieving this goal, this dissertation tries to answer the main four intriguing questions mentioned above. Briefly, we first study the interaction between the properties of ACQs and the characteristics of the workload that affect the cost of the shared processing of ACQs. Once we understand the

interaction between the different factors, we can devise a multiple ACQs optimizer that utilizes this knowledge. Then, given the insights gained from the previous step, we explore the efficient processing schemes searching for an efficient scheme that is more suitable for shared processing and that is efficiently adapting to changes in the workload. Finally, we revisit the multiple ACQs optimization problem given the new processing scheme. Below, we detail these steps and the challenges involved with each step.

Addressing Q1: What other factors of the workload characteristics and ACQs properties affect the cost of a shared query plan?

To address this question, we first formulate the problem and build the cost model of the shared query plan. Given this cost model, we can identify the tradeoffs involved in optimizing the shared processing of multiple ACQs. We can further study the tradeoffs by performing a thorough experimental analysis which considers many different settings of the workload parameters. Once we identify the tradeoffs, the hope is to develop a formula that captures the interaction between the different workload characteristics that affect the cost of the shared query plan.

Challenge: Uncorrelated Factors

We observed that the most challenging part in this task is that the performance of an aggregate operator relies on a set of uncorrelated, and sometimes contradicting, factors. These factors are the input arrival rate, the size of the workload (i.e., number of ACQs), as well as per ACQ specifications (e.g., window specifications, predicate and group-by attributes). The effect of these factors varies depending on the underlying single ACQ processing scheme. For example, in the *Partial Aggregation* processing scheme not only the number of ACQs, but also which ACQs are being shared affects the processing overhead. Unfortunately, all these factors are not correlated. Prior work in multiple ACQ optimization has considered only one factor. Therefore, there is a need for solutions that consider all factors comprehensively in order to achieve the best possible performance.

Addressing Q2: Can we design a multiple ACQs optimizer that considers all these factors comprehensively while making the sharing decision?

Once we have devised a formula that captures the interaction of the different factors that affect the cost of processing ACQs, we search for an optimizer that utilizes this formula. We first consider

the simple case where all similar ACQs vary in window specifications only. Then we consider the more general case when ACQs can arbitrary vary in any specification, i.e., windows, predicates and group-by attributes.

Challenge: Exponential Search Space

As it is the case with traditional multiple query optimization (MQO), finding the optimal query plan of multiple ACQs, given a set of workload parameters, is an NP-Hard problem [67]. In terms of search space, the number of possible plans is exponential in the number of ACQs. The research community has investigated several heuristics to approximate the optimal query plan in both traditional MQO (e.g., [66]) as well as in multiple ACQs (e.g., [83]). The challenge is to develop a multiple ACQs optimizer that efficiently prunes the search space without compromising the quality of the generated query plan.

Addressing Q3: To recompute or not to recompute?

Given that the factors that affect the cost of a query plan, in addition to being uncorrelated, are dynamic, i.e., change over time, we investigate how to adapt the query plan efficiently. We consider different options between over-provisioning and recomputation from scratch in order to find the best online strategy.

Challenge: Quality versus overhead tradeoff

There are many factors that are dynamic. The stream arrival rate is known to be of bursty nature. ACQs can be added and deleted on demand, at run time. This further complicates the task of finding the optimal query plan and calls for solutions capable to adapt to workload changes in real time. That is, the optimizer should be able to choose the best way to update the query plan when faced with different types of changes, balancing the tradeoff between the quality of the query plan and the overhead needed to compute the updated plan or to recompute the new query plan from scratch.

Addressing Q4: Is the *Partial Aggregation* technique the best continuous aggregation operator implementation?

Based on the experience of developing a new optimizer, we investigate whether we can improve the underlying aggregation operator processing scheme. Specifically, we revisit the *Partial Aggregation* scheme to devise a new scheme that better suits the new sharing scheme (i.e., the new

optimizer), and the dynamic nature of the problem. Given this new sharing-friendly processing scheme, we revisit the multiple ACQs optimizer to explorer potential chances for further improvements.

Challenge: Multiple dimensions to optimize for

The new processing scheme should be designed to optimize the shared processing of similar ACQs that have arbitrary different specifications. The challenge is that it is unclear which dimension of these specifications should the new scheme prioritize to optimize for to promote sharing. Further, in such dynamic environment of changing workload characteristics, the new processing scheme should be online by design, i.e., support adaptive optimization of multiple ACQs.

1.2 CONTRIBUTIONS

In this dissertation, we address all the above challenges and make contributions to the theory and practice of efficient processing of multiple ACQs in DSMSs alike. We identify the sharing tradeoff and introduce the concept of *Weaveability* of ACQs. The *Weaveability* of a set of ACQs is an indicator of the potential gains from sharing their processing.

We then propose *Weave Share*, a cost-based multiple ACQs optimizer, which exploits weaveability to optimize the shared processing of multiple ACQs that vary in window specifications only. *Weave Share* selectively groups ACQs into multiple execution trees to minimize the total plan cost by considering all factors that affect the cost of the shared query plan. We experimentally evaluate and analyze the performance of *Weave Share* in terms of quality of the generated plans using all possible settings of workload characteristics. Our experimental analysis shows that *Weave Share* generates up to four orders of magnitude better quality plans compared to the best alternative sharing schemes. We develop and experimentally evaluate a practical implementation and several optimizations that dramatically improve the efficiency of the *Weave Share* optimizer in generating high quality plans.

Given the *Weave Share* optimizer, and in order to efficiently handle the addition and deletion of ACQs in an online fashion, we propose *Incremental Weave Share* that efficiently weaves the new ACQs into the execution trees of an existing plan, as long as the quality of the query plan is not

compromised, i.e., remains within specified tolerance limits. *Incremental Weave Share* balances the tradeoff between quality and efficiency by triggering a recomputation from scratch whenever the quality of the incrementally maintained query plan deteriorates beyond a certain threshold. We experimentally evaluate *Incremental Weave Share* and perform sensitivity analysis of the threshold with respect to performance.

To fully reap the benefits of the new *Weave Share* multiple ACQs optimizer, we investigate a new underlying aggregate operator implementation. This implementation allows more flexibility in the data flow between the sub-aggregation and final-aggregation levels so that partial aggregate results are easily pipelined to different final-aggregate operators, or equivalently, to different trees of operators as in the case of *Weave Share*. Specifically, we propose *TriOps*, a new aggregate operator implementation that works in synergy with the new *Weave Share* optimizer towards minimizing the total cost of processing multiple ACQs. *TriOps* employs a novel three-level data processing model that minimizes the repetition of operations at the sub-aggregate level. Given the *TriOps* framework, we propose *TriWeave* a *TriOps*-aware multiple ACQs weave-based optimizer. We experimentally demonstrate the performance gains provided by *TriWeave* and show it is superior to other alternatives including *Weave Share* based on the two-level *Partial Aggregation* technique.

In addition to those gains, *TriOps* still maintains all the attractive features of the two-level aggregation model, which allows it to directly incorporate traditional multiple query optimization techniques for exploiting overlapping predicates and group-by attributes. As such, we generalize *TriWeave* to integrate the classical subsumption-based multiple query optimization techniques (i.e., overlapping predicates and group-by attributes) with the new weaveability-based multiple ACQs optimization.

A design goal of all solutions proposed in this dissertation is to adapt to changes in workload characteristics in an online fashion. The *TriOps* framework enables a smooth and efficient query plan switching and, therefore, enables adaptivity to changes to all workload settings.

We extended the AQSIOS DSMS prototype [6, 19], which implements *Weave Share* and the two-operators shared processing scheme. Previous version of AQSIOS did not support sliding windows, a serious shortage. Our realization of *Weave Share* and the two-operator processing scheme introduces the support for the sliding windows. The basic evaluation of *Weave Share* in AQSIOS confirmed our simulation results. Finally, the implementation of *Weave Share* in AQSIOS

(v. 3.0) sets up the stage for future studies, in a real-system, of the synergy between the query optimizer and the other DSMS modules, such as the scheduler, load shedder and admission control. The interaction with the scheduler is particularly important given the underlying two-operators scheme, where each operator is typically scheduled independently. As mentioned earlier, different scheduling techniques and query optimizers have been proposed independently to optimize the performance of ACQs. It is crucial that the different adopted techniques for the different DSMS modules work well together to avoid the undesired situation where the optimization efforts of one module, e.g., scheduler, are canceled by the optimization efforts of another module, e.g., query optimizer. AQSIOS 3.0 is expected to enable the design of new synergistic strategies among the DSMS modules.

In summary, the contributions of this dissertation are:

- *Weaveability*, a new concept that captures the potential gains from sharing the processing of multiple ACQs. We introduce the concept of weaveability after we identify and demonstrate the tradeoffs in optimizing the shared processing of multiple ACQs.
- *Weave Share*, a new multiple ACQs optimizer that comprehensively considers all cost factors and applies to ACQs with different window specifications as well as different predicates.
- *Incremental Weave Share*, the online version of *Weave Share* that efficiently handles the addition and deletion of ACQs.
- *TriOps*, a new aggregation processing scheme that is designed to optimize the shared processing of ACQs that can vary in any specification (i.e., window, predicates or group-by attributes). *TriOps* also enables efficient adaptive processing to changes in the workload in terms of changes in the input rate and addition or deletion of ACQs.
- AQSIOS 3.0, which is our realization of *Weave Share* optimizer in the AQSIOS DSMS prototype.
- Extensive performance evaluation and sensitivity analysis of multiple ACQs optimization techniques using the developed simulation platform and the AQSIOS prototype.

Road map: The rest of the dissertation is organized as follows: the background and related work are discussed in Chapter 2. In Chapter 3 we define weaveability of ACQs and how to exploit it to achieve better plans using the *Weave Share* optimizer. The online incremental version of *Weave Share* that handles the addition and deletion of ACQs is discussed in Chapter 4. We then propose *TriOps*, a new aggregation processing scheme, that enables adaptive weaving of stream aggregation with different window specifications, predicates and group-by attributes in Chapter 5. *TriWeave*, the generalized multiple ACQs optimizer that utilizes *TriOps* is also presented in Chapter 5. Then we summarize the realization of our proposed solutions in AQSIOS prototype in Chapter 6. We finally conclude the dissertation discussing the future work and how to generalize our proposed weaving techniques to handle more complex CQs in Chapter 7.

2.0 BACKGROUND AND RELATED WORK

In order to set up the stage for the rest of this dissertation, in this chapter, we first furnish the necessary background on data streams management systems (DSMSs). Second, we discuss our assumed underlying model for aggregation over data streams. Then we describe our experimental platform. Finally, we provide a survey of other related work on Aggregate Continuous Queries (ACQs) processing.

2.1 DATA STREAM MANAGEMENT SYSTEMS

As mentioned in the Introduction chapter, DSMSs deviate completely from the *store-then-query* paradigm of traditional database management systems (DBMS). Specifically, in a DSMS, the continuous queries are registered ahead of time and continuously process arriving data. The data, therefore, need not to be stored except for archival purposes. Data arrives in the form of unbounded streams from different data sources, where the arrival of new data is similar to an *insertion* operation in traditional database systems. A DSMS is typically connected to different data sources and a single data stream might feed more than one query.

That is, in DSMSs, users of monitoring applications register Continuous Queries (CQs) which continuously process unbounded data streams looking for data that represent events of interest to the end user. DSMSs are designed to efficiently handle such large and burst volumes of data and large number of continuous queries. That is DSMS are designed to exhibit scalability, while at the same time providing fast response times. Towards developing a scalable DSMS, several prototypes have been proposed (e.g., [6, 3, 7, 2, 39, 56]) as well as several commercial products (e.g., [73, 77, 39, 74, 56, 22]).

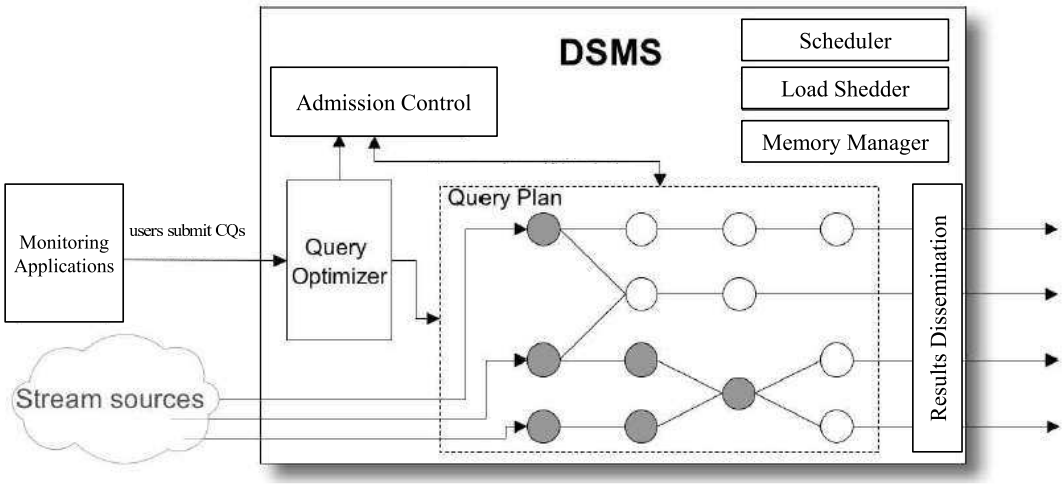


Figure 1: DSMS Architecture.

The typical architecture of a DSMS is illustrated in Figure 1 which shows the main modules of the DSMS, namely, the *Query Optimizer*, *Query Scheduler*, *Buffer/Memory Manager*, *Load Shedder*, *Admission Control* and *Results Dissemination*. We now briefly describe the functionality of each of these modules.

As CQs are registered to the DSMS, the *Query Optimizer* is invoked to generate an optimized CQ evaluation plan that minimizes the processing overhead. A CQ evaluation plan can be conceptualized as a data flow tree [15, 11] (e.g., the Query Plan in Figure 1). The nodes are operators that process tuples and edges represent the flow of tuples from one operator to another. An edge from operator o_i to o_j means that the output of o_i is an input to o_j. Each operator is associated with a *queue* where input tuples are buffered until they are processed. A continuous query evaluation plan is usually, and in this dissertation hereafter, referred to as an *execution tree* as well.

Optimizing the execution tree to minimize the processing overhead is achieved in two levels. First is optimizing the query plan of a single CQs by applying traditional query optimization heuristics, such as pushing selections and projections down the execution tree. Second, at a more global level, is the multiple query optimization, which exploits common sub-expressions among the individual query execution trees to avoid repeating processing of the same operators on the same data.

The query optimizer also utilizes as much information available about the data semantics and meta-data, whenever available, to generate the most possible efficient evaluation plan [82]. Since the data semantics (e.g., data ranges and distributions) and meta-data (e.g., workload settings such as arrival rate and CQs) change over time, a continuous adaption of the query evaluation plan is also needed [50, 49, 9, 10].

Once the query plan is generated, the DSMS utilizes the *Scheduler* to chose the order at which CQ operators shall be executed. Different metrics have been proposed to capture the requirements of difference monitoring applications. Different scheduling algorithms are needed to optimize, at run time, the execution of the queries for the different metrics [11, 70]. Two categories of metrics have been utilized, namely, *Quality of Service* (QoS) metrics and *Quality of Data* (QoD) metrics. The most common QoS metric is the *response time*, which is the time span between when an event of interest occurs (i.e., a data tuple arrives) and when it was produced as an output of the interested CQs (i.e., all CQs that this input tuple is part of their output). *Data freshness*, on the other hand, is usually used to capture the QoD of the output. In [71, 69], the output tuple is fresh if and only if it is still valid, i.e., no overwriting output tuple is due.

The *Scheduler* is also sometimes utilized as a query optimizer, in the sense that it is used to synchronize the execution of operators that share partial processing [30, 38, 81] to avoid repeating that processing and minimize the memory used for storing intermediate results [11].

The *Memory Manager* (*Buffer Manager*) role is to dynamically allocate memory buffers to different queues and operators. Shared queues to store intermediate results among different operators are typically utilized [14]. The memory manager is a crucial module, given the memory intense nature of the DSMSs. That is, given the real-time requirements of the monitoring applications DSMSs support, all data should fit into main memory (similar to main-memory databases). Since a DSMSs support a large number of registered CQs, as well as the large and bursty volumes of input data streams, main memory could become a bottleneck. This memory intense nature might lead to overloading situations, where the DSMS might fail to meet the promised performance guarantees. In such cases, load shedding is employed to resolve this issue [3, 48, 65, 64, 78].

Load shedding is one of the two approaches to load management in a DSMS. Under overloading situations, the *Load Shedder* module would select certain tuples to discard without processing them in order to reduce the memory and processing requirements at the moment. This is per-

formed on the hope that by reducing the load, the DSMS could meet the QoS guarantees [27, 75]. Typically this comes at the expense of deteriorating the QoD. Thus, the load shedder main goal is to improve the QoS while minimizing the deterioration of the QoD. Semantics of the input data streams could be utilized to better select which tuples to shed [48, 64, 65].

Orthogonal to load shedding, *Admission Control* tries to avoid overloading situations. The *Admission Control* module would decide which CQs to admit into the DSMS, so that to minimize the chances for the system to run into an overload situation. The *Admission Control* module would selectively admit CQs to be registered into the the system, given the system capacity, in order to guarantee the promised QoS and QoD. One can regard the *Admission Control* as avoidance scheme, while the *Load shedder* is a detect and resolution scheme. In [52], a game theoretic approach was proposed to pick the set of CQs to admit in a way to maximize the system profit while maintaining user satisfaction.

The *Results Disseminator* task is to timely return query results to users. Each ACQ is associated with a socket, or network connection, which has a limit depending on the multi-programming level (MPL) of the system. The objective of the *Results Disseminator* is to prioritize results delivery, i.e., scheduling the connections, in order to minimize the response time as perceived by the end user. In a wireless setting, whenever the delivery media is a broadcast network, the *Results Disseminator* also schedules the delivery of the results with the objective of minimizing the energy consumption [61, 62].

Thus far, we have described the different main modules of a general purpose DSMS. In addition, there is a lot of work done in specialized DSMSs, such as for spatio-temporal applications [56, 58, 5, 55]. For example, PLACE [56, 57] and its extension SOLE [55] presented a scalable scheme of processing moving queries over a stream of moving objects. Moving queries are spatio-temporal queries with continually changing (i.e., moving) spatio-temporal predicates. PLACE [56] is a location-aware database server that utilizes a set of spatio-temporal operators. SOLE [55] achieves scalability by keeping track of only the significant spatio-temporal objects. A spatio-temporal *Load Shedder* is utilized to handle overloading situations.

In the following section we discuss our assumed underlying model for aggregation processing over data streams.

2.2 AGGREGATION OVER DATA STREAMS

Multiple query optimization (MQO) of ACQs poses a challenge because of the variability in window specifications, predicates and group-by attribute across multiple ACQs. In particular, it goes beyond the level of identifying common sub-expressions as in traditional MQO (e.g., [66, 25, 43]) to the level of exploiting the window semantics, the overlap of predicates and the overlap of group-by attributes and the partial aggregation. In the following sections, we first discuss the window semantics of ACQs and explain the *Partial Aggregation* processing scheme. Then, we discuss the problem of sharing multiple ACQs.

2.2.1 ACQ Semantics

Since the input stream is continuous, i.e., unbounded, an ACQ is defined over a sliding window which is specified in terms of two intervals: 1) *range (r)*, and 2) *slide (s)*. For example, in a stock monitoring application, a user may register an ACQ that computes the average stock price over the last hour (range) and update it every 30 minutes (slide). In addition, an optional predicate could be used to filter tuples before performing the aggregation.

The settings of both the range and slide parameters per ACQ determine its semantics. For instance, whenever the slide equals the range ($r = s$), the window is called a *tumbling window*, and if the slide is greater than the range ($s > r$) then it is called a *hopping window*. Otherwise, it is a *sliding* or *overlapping window*. Further, both the range and slide intervals could be defined either as tuple-based or time-based, where the former is set to a certain number of tuples, whereas the latter is set to a certain time period. In this dissertation, we consider the more general time-based definition for both the range and slide parameters.

The settings of range and slide also determine the data processing requirements per ACQ. In particular, producing a new result requires processing each subset of tuples within the range interval. Slide, on the other hand, defines how the window boundaries move over the continuous data stream. For instance, when slide is less then range (sliding window), different consecutive windows overlap and a single tuple will belong to more than one window, hence, it is involved in the computation of different aggregate instants.

Example 1. *Consider an ACQ with range = 1 hour and slide = 10 minutes. For this window definition, a boundary line is reached every 10 minutes and an aggregation is performed over the tuples within the last 1 hour. Hence, each input tuples will be involved in the computation of 6 consecutive windows (=1 hour/10 minutes).*

In a straight forward implementation of aggregates, input tuples are buffered and once a window boundary is reached, the aggregate function is evaluated using the buffered tuples that are within the range boundaries. After evaluating the aggregate, the range boundaries are shifted and all the buffered tuples that fall outside the new boundaries are expired and discarded since they cannot contribute to in any future computation.

2.2.2 The *Paired Window* Technique

Towards efficient processing of a single ACQs over sliding window, current techniques exploit *Partial Aggregation*, where the final aggregate value is assembled from a set of partial aggregate values (e.g., [46, 45]). In general, under such techniques the input data is divided into a set of partitions where an initial *sub-aggregate* function is applied on each partition separately to generate a partial aggregate. Then a *final-aggregate* function is applied on the set of partial aggregates to generate the final result. This concept is materialized via an aggregate query plan composed of two operators: 1) sub-aggregate operator, and 2) final-aggregate operator.

As an example, under the *Partial Aggregation* scheme, an aggregate `COUNT(*)` over a certain window range is computed using (1) a `COUNT(*)` on each partition and (2) a `SUM(*)` over the partial count of each partition. Clearly, partial aggregation is applicable over all distributive and algebraic aggregate functions that are widely used in database systems and monitoring applications such as: `MAX`, `MIN`, `SUM`, `COUNT`, and `AVERAGE`. Formally:

Definition 1. *For a dataset G of disjoint fragments g_1,g_2, ..., g_n, an aggregate function $\mathcal{A}$ over G can be computed from sub-aggregate function $\mathcal{S}$ over each dataset g_i and a final-aggregate function $\mathcal{F}$ over the partial aggregates.*

$$\mathcal{A}(G) = \mathcal{F}(\{\mathcal{S}(g_i)|1 \leq i \leq n\}) \tag{2.1}$$

Partial Aggregation allows for reducing data processing cost since each input tuple is processed only once by the sub-aggregate function/operator. As the window slides, only partial aggregates are buffered and processed to generate new results, whereas individual input tuples are processed only once to produce those partial aggregates then discarded. *Partial Aggregation* also reduces the memory needed, since input tuples are buffered until they are aggregated once (as opposed to the number of overlapping windows it belongs to). Further, storing partial aggregates requires less space.

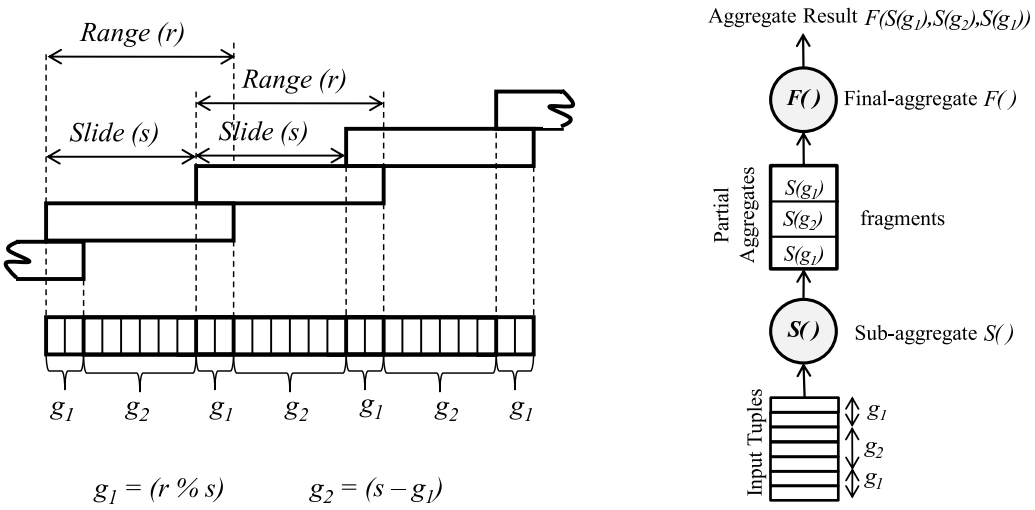
Clearly, the less the number of partial aggregates in a window, the better because it means that the final-aggregate operator performs fewer operations per output. The work in [45] proposes what is called the *Paired Window* technique for ACQ processing and shows that it is possible to partition each slide into at most two *fragments* (i.e., a pair). Hence, producing a final aggregate requires at most $2\lceil \frac{r}{s} \rceil$ operations, where $\lceil \frac{r}{s} \rceil$ is the number of slides per sliding window and 2 is the maximum number of fragments per slide. Formally:

Definition 2. *Under the* Paired Window *technique, the slide s of an ACQ with range r is split into two fragments g_1 and g_2, where:*

$$g_1 = r\%s, \text{ and } g_2 = s - g_1 \tag{2.2}$$

Figure 2(a) illustrates the idea of a *Paired Window*. The lower part of Figure 2(a) shows the set of input tuples, while the upper part shows different overlapping window instances of a window of size r and slide s. As shown in Figure 2(a), each *slide* is paired into exactly two *fragments* of length: g_1 and g_2, where $g_1 = r\%s$ and $g_2 = s - g_1$. In the partitioning of Figure 2(a), the range consists of a sequence of g_1, g_2, g_1 fragments, where the length of a pair of fragments equals $g_1 + g_2 = s$. In general, when $r > 2s$, the range would consist of a sequence $g_1, g_2, g_1, g_2, ..., g_1$ fragments. Notice that if r is a multiple of s, then only one fragment per slide is produced. In the rest of this dissertation, however, we will consider the general case of 2 fragments per slide.

Figure 2(b) shows the execution tree of an ACQ, assuming the *Paired Window* processing scheme. The plan consists of a sub- and final-aggregation operators, and the input queue of each operators. As illustrated in Figure 2(b), the sub-aggregate operator processes the input tuples as they arrive generating a sequence of partial aggregates. Specifically, the end of each fragment g corresponds to an *edge*, where the tuples in g are assembled into a partial aggregate. Since the ACQ



(a) Range, slide and fragments

(b) Query Plan

Figure 2: *Paired Window* technique

results are due every slide (i.e., two edges), the final aggregation is computed every two edges. All fragments (partial aggregates) that are within the new window boundaries (i.e., within the range r) are aggregated by the final-aggregation operator to produce a results. This is further illustrated in the following example.

Example 2. *Consider an ACQ illustrated in Figures 2(a) and 2(b). The figures shows that the range r consists of exactly three fragments (i.e., $r = g_1 + g_2 + g_1$). A window instance is due every two fragments g_1 and g_2 and is computed by applying the final-aggregation function $\mathcal{F}$ to exactly three fragments, that is $\mathcal{F}(\mathcal{S}(g_1), \mathcal{S}(g_2), \mathcal{S}(g_1))$. Then the first pair of fragments can be discarded immediately since they do not contribute to any further computations.*

Notice that the sub-aggregate operator processes each tuple exactly once and produces a sequence of non-overlapping fragments. On the other hand, the final-aggregate operator needs to process each fragment several times to generate the sequence of overlapping windows results.

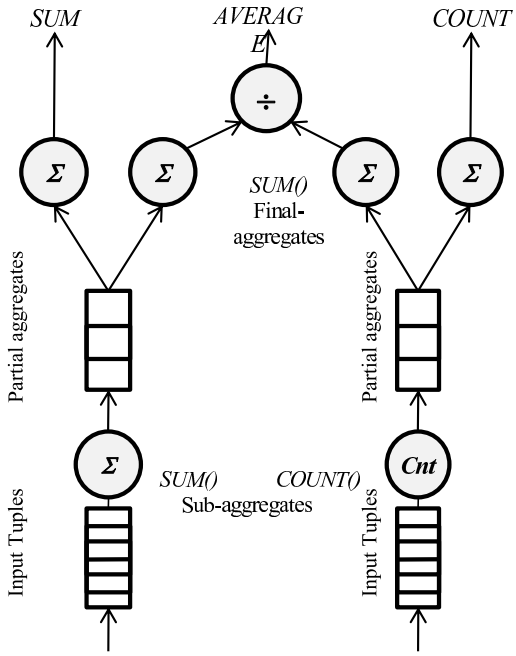


Figure 3: Sharing the partial aggregations.

2.2.3 Sharing Multiple ACQs

Several shared processing schemes as well multiple ACQs optimizers that utilize the *Paired Window* technique have been proposed [45, 59]. Below we describe the *Shared Time Slices* technique which handles the case of varying windows, the *Shared Data Shards* which handles varying predicates, and the *Intermediate Aggregates* which handles varying group-by attributes.

2.2.3.1 Shared Time Slices The *Shared Time Slices* [45] (we refer to as *Shared* hereafter) technique was proposed to share the processing of multiple ACQs with varying windows but same predicates and same group-by attributes. The main idea behind *Shared* is to share the sub-aggregation operators and to generate fine grained fragments in a way to satisfy all varying windows. Specifically, the *Partial aggregation* scheme allows for sharing the computation performed at the tuple-level (i.e., sub-aggregate level) across multiple queries and, hence, avoid repeating that phase for each and every query individually.

For example, if the system has three ACQs: a `SUM`, `COUNT` and `AVERAGE`. The `SUM` is rewrit-

ten as SUM of sub-SUMs, the COUNT as SUM of sub-COUNTs, and the AVERAGE as the SUM of sub-SUMs divided by the SUM of sub-COUNTs. Then the sub-aggregate operator that performs the sub-SUM can be shared among the SUM and AVERAGE queries, while the sub-aggregate operator that performs the sub-COUNT can be shared among the COUNT and AVERAGE queries. The execution tree of this example is illustrated in Figure 3.

In order to utilize *Paired Window* scheme, while at the same time allow for sharing between multiple ACQs, the process of fragment generation is extended to accommodate shared processing. That is, the fragments produced by the *Paired Window* technique are merged to support the variability in ranges and slides exhibited by the different ACQs sharing the same sub-aggregate operator.

For a set of ACQs $\{q_1, q_2, ..., q_n\}$ with slides $\{s_1, s_2, ..., s_n\}$, to determine the new sequence of fragments (or equivalently, edges) under the *Paired Window* scheme, the following three steps are needed:

1. Define composite slide: Multiple slides are integrated into a new *composite slide (CS)*, where the period (length) of the composite slide is the lowest common multiple of the slides of individual ACQs (i.e., length of $CS = lcm_i(s_i)$).
2. Stretch individual slides: Each slide s_i is then stretched into a new slide s'_i where the length of s'_i is equal to CS. Further, the edges (i.e., end of each fragment) in each slide s_i are then copied and repeated to the length of s'_i (=repeated $\frac{s'_i}{s_i}$ times, or equivalently = $\frac{CS}{s_i}$).
3. Merge edges: The fragments in the composite slide are created by overlaying the edges from each individual slide s'_i onto the new composite slide CS. Specifically, each individual slide s'_i is scanned and each edge is added to the new composite slide unless it already exists (i.e., common edge).

Example 3. *Consider two ACQs q_a and q_b with ranges $r_a = 12$ and $r_b = 10$, and slides $s_a = 9$ and $s_b = 6$ seconds. As illustrated in Figure 4, the fragments in q_a's slide are of length $g_{a,1} = 3$ and $g_{a,2} = 6$ and for q_b, the fragments are $g_{b,1} = 4$ and $g_{b,2} = 2$. For this setting, the $lcm(s_1, s_2)$ is of length 18. By stretching each slide to the lcm period, the edges for q_a appear at times (3, 9, 12, 18) and those for q_b appear at (4, 6, 10, 12, 16, 18) and the edges in the composite slide appear at times (3, 4, 6, 9, 10, 12, 16, 18).*

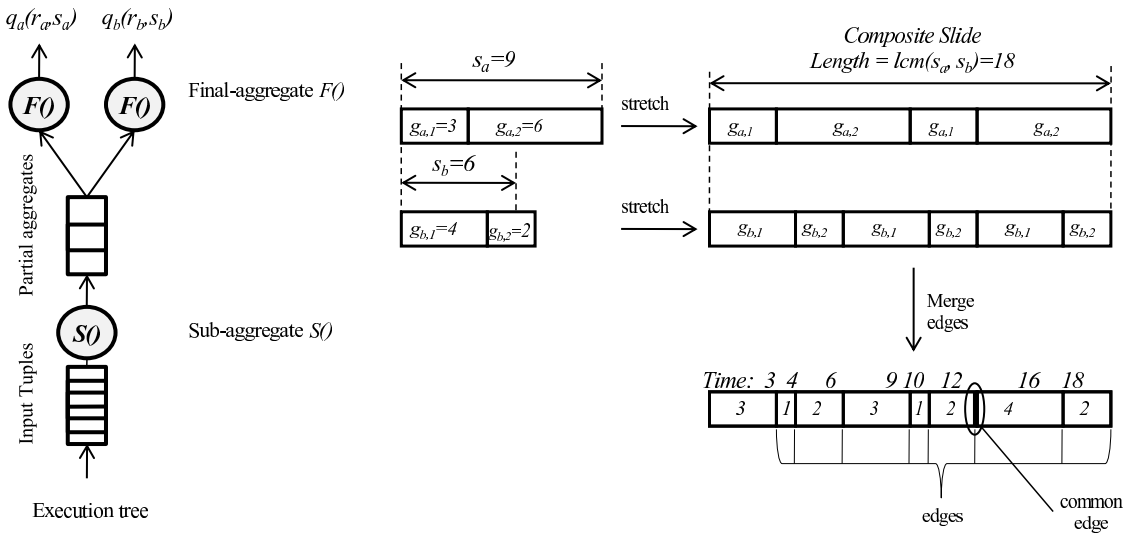


Figure 4: Example 3: stretching slides, merging edges, and shared plan.

2.2.3.2 Shared Data Shards The *Shared Data Shards* (SDS) [45] technique was proposed to share the processing of multiple ACQs with different predicates but same windows specifications and group-by attributes, assuming the *Paired Window* technique. The main advantage of *SDS* is that it avoids any unnecessary repeated evaluation of predicates. In particular, each predicate is evaluated for each tuple exactly once in a preprocessing phase prior to the sub-aggregation level. The outcome of this preprocessing phase is a set of augmented tuples, where each tuple is augmented with a predicates signature which encodes the results of evaluating all the predicates for this tuple. This signature is a bitmap vector, where each bit represents a predicate and is set to one only if this predicate evaluates to 1 for this tuple. In this way, this signature identifies which set of ACQs this tuple belongs to.

Given the set of augmented tuples, the sub-aggregation operator then aggregates tuples of identical signatures together, resulting in a set of fragment-signature pairs. Once an edge is due, each fragment-signature pair is pushed and routed into the input buffers of all ACQs that it satisfies their predicates, i.e., have matching signature.

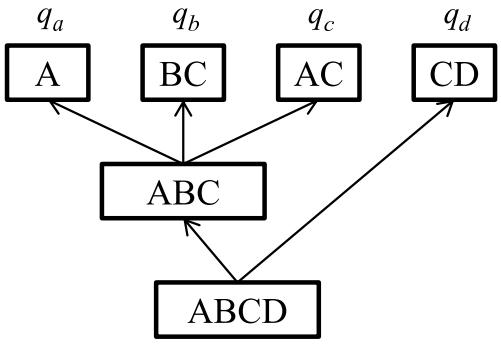


Figure 5: Intermediate Aggregates tree

2.2.3.3 Intermediate Aggregates The problem of optimizing multiple ACQs with different group-by attributes, but same windows and same predicates was addressed in [83, 84, 59]. Motivated by the constrained memory on the Network Interface Card (NIC), a shared processing scheme called *Phantoms* [83, 84] was proposed. *Phantoms* essentially introduces a set of sub-aggregates (i.e., *Phantoms*) to share the processing of the ACQs that have identical window specifications, same predicates, but have different group-by attributes. *Phantoms* technique is developed for the specific architecture of Gigascope [23], where the sub-aggregation is performed in the limited memory of the NIC. In this architecture, the goal is to minimize the rate with which data is copied between the NIC and the main memory. In [59], the *Intermediate Aggregates* technique generalizes the *Phantoms* technique to general, yet memory-constrained, DSMS. In the latter case, moving data from one operator to another is not the most expensive operation anymore, rather, it is the number of aggregation operations.

The main concept of *Intermediate Aggregates* is that if two similar ACQs have different group-by attributes, then a sub-aggregation that uses for its group-by attributes the union of the group-by attributes of both ACQs can be shared among the two ACQs. For more than two ACQs, the same concept is recursively applicable to sub-sets of the final-aggregation operators (which are essentially ACQs). Thus, *Intermediate Aggregates* aims to exploit the overlapping of group-by attributes to generate the best possible execution tree. We further illustrate the idea in the example below.

Example 4. *Assume four ACQs q_a, q_b, q_c and a_d with same window specifications and same predicates. Assume also that the ACQs have group-by attributes: A, BC, AC and CD, respectively, as illustrated in Figure 5. The label of each node in the figure represent the set of group-by attributes used by this node. That is, each node represent an aggregation operator that performs a group-by aggregation using this set of attributes. The figure shows that q_a,q_b and q_c share a sub-aggregation that uses the union of attributes ABC for its group-by attributes. This sub-aggregation further shares with q_c another sub-aggregation operator that uses the union of attributes $ABCD$ for its group-by attributes.*

It was mentioned without further details in [59] that if the ACQs have different windows specifications, then either an epoch (i.e., a fragment) of size equals to the greatest common divisor of the ranges or the *Paired Windows* scheme can be utilized. In this dissertation, we investigate strategies that best minimize the processing overhead of multiple ACQs, with varying windows, predicates and group-by attributes, exploiting shared *Partial Aggregation*.

2.3 EXPERIMENTAL PLATFORM

In order to study the sharing trade off and to evaluate the schemes we propose in this dissertation in a controlled environment, we built a simulation platform in C++. We validated our simulation model by reproducing same results trends as reported in the related work and by running exhaustive search to find the optimal plan for small cases that we solved by hand. In this section, we list the baseline algorithms we used in our experiments and describe the generated workload characteristics, experiments parameters and the performance metrics.

Algorithms: We have implemented the *Paired Window* processing scheme as the underlying ACQ processing scheme. In terms of multiple ACQs optimizers, we implemented *Random*, *Exhaustive Search*, *Shared* (where all ACQs are merged in one single tree [45]), *No Share* (as a base line, where each ACQ is executed separately) and an adapted version of *Local Search* (LS) (see Appendix A).

ACQs: We generated ACQs randomly with different parameters summarized in Table 1. We used the `SUM` as the aggregate function, so that the sub- and final-aggregate operators both perform a

Table 1: Queries parameters

Parameter	Values
Slide Length (s)	[1–100000] using Zipf distribution
Slide Skewness	[0.0–3.3] (skewed to large-slide)
Overlap Factor	[1–Ω_{max}]

SUM. The type of the aggregate function, however, does not affect the performance of the optimization algorithms. The parameters of the queries are as follows:

- Slide (s): randomly generated from a Zipf distribution with minimum 1 second and maximum 100000 seconds. The Zipf's skewness parameter is in the range [0.0–3.3], where a value of 0.0 is equivalent to uniform distribution, whereas higher values result in a more skewed distribution that has more large-slide ACQs. This reflects the different levels of interest the users typically have in real world applications.
- Range (r): the range (r) of each query was generated relative to the query's slide using an overlap factor (Ω). That is, for each query q_i, $r_i = \Omega_i \times s_i$, where Ω_i is generated randomly from a uniform distribution in the range [1–Ω_{max}]. Ω_{max} is the *maximum overlap factor* which is a simulation parameter.

Experimental Parameters: In addition to changing the ACQs parameters, each experiment has a set of parameters that are summarized in Table 2. Briefly, these parameters are the number of ACQs, the input rate and the initial state and the iterations bound of *Local Search* (LS). The input rate values are chosen to cover the input rate ranges of all different monitoring applications. Specifically, the 1M tuples/sec covers network monitoring applications, while the 10^5 tuples/sec covers Web Analysis applications. Financial applications, however, have a typical input rate of couple hundreds [60] and sensor networks and phenomena monitoring applications have a typical input rate of few tuples, or less, per second.

Dataset: We chose to use synthetic workload, which allows us to control the system parameters, in order to conduct detailed sensitivity analysis and gain better insight into the behavior of the

Table 2: Experimental Parameters

Parameter	Values
Number of ACQs	[50–1000]
Input arrival rate	[0.5–1,000,000] tuples/sec.
Max. Overlap Factor (Ω_{max})	[50 - 2000]
LS initial state	{No Share, Random}
LS iterations bound	{2x, 5x, 10x} proposed optimizer iterations

different techniques by setting the parameters to cover all possible real scenarios. For instance, controlling the skewness of windows specifications allows us to depict different cases, from the simple pre-specified time-scale classes as in *Truviso* to the more demanding uniform distribution as in [45]. Our choices based on [45] were also for fairness and validation.

Performance Metrics: We measured the quality of the optimized shared plans in terms of their total cost. The cost is measured as the number of aggregate operations per second (which also indicates the throughput). We chose this metric because it provides an accurate and fair measure of the performance, regardless of the platform used to conduct the experiments. We also evaluate the different algorithms efficiency in terms of their overhead to generate the query plan.

2.4 OTHER RELATED WORK

As discussed in Section 2.2, the main idea of paired windows scheme is to split the slide into two fragments to be processed using the sub-aggregation operator. The *Pane* [46] scheme was the first to propose the idea of rewriting the ACQ as a final-aggregation of sub-aggregation and split the slide into fragments. Opposed to *Paired Window*, *Pane* splits the window into equal sized fragments. *Paired Window* improves over *Pane* by splitting the slide into exactly two fragments to minimize the processing needed at the final-aggregation operator.

The *Window-ID* (WID) technique proposed in [47] improves the performance of an ACQ by maintaining multiple aggregates of multiple window extents at the same time. A bucket operator is utilized to tag the input tuples appending a range of window extents that the tuple belongs to, or may contribute to. The tagged tuple is then aggregated to all window extents at once. It was shown in [47] that *WID* can be easily integrated with *Pane* scheme and since *Paired Window* scheme is a variation of *Pane*, it can similarly be integrated with *WID*.

There are also alternative approaches for processing ACQs. For example, optimization techniques for processing sliding-window queries that utilize the *negative tuples* approach have been proposed in [29]. ACQs are instances of sliding-window queries. In the negative tuples approach, tuple expiration is determined when a negative tuple is inserted. This doubles the number of tuples through the query plan. Also, in [29], query operators were classified into two classes according to whether an operator can avoid the processing of negative tuples or not. Based on this classification, several optimization techniques over the negative tuples approach where presented to reduce the overhead of processing negative tuples.

At the multiple ACQ processing level, sharing results of aggregation among different queries has also been proposed in [30], where a scheduling technique to optimize the execution of ACQs has been developed. This technique utilizes a window-aware scheduling scheme that synchronizes the re-execution times of similar queries to execute common parts only once.

In general, there is a rich literature on multiple query optimization (MQO) in traditional databases [68, 66, 42, 51, 40, 80] as well as in data streams [80, 3, 81, 49]. Multi-query optimization in traditional databases aims at exploiting common sub-expressions to reduce evaluation cost [66]. Similarly, shared processing is exploited in multiple continuous query optimization. In both cases, finding the optimal query plan is an NP-Hard problem [67], and hence the data management research community has investigated heuristic approaches to optimize the generated query plans.

For instance, two cost-based and one greedy heuristics where proposed in [66]. The main idea behind the two cost-based heuristics in [66] is to extend the *Volcano* [32] query optimizer, which performs a depth-first search in the state space of alternative query plans. Different alternatives are represented using AND-OR graphs. The proposed heuristics improves the performance of *Volcano* by augmenting the AND-OR DAG representation to enable the detection of common sub-

expressions across different queries as well as expressions' subsumption. Thus, while performing the depth-first search-and-prune phase, Volcano can generate much more efficient plans.

Recently, DSMSs and monitoring applications are being moved to the cloud [21]. In [21] in particular, a demonstration of implementing event monitoring application using the modified *Hadoop* framework was presented. This shows the importance of optimizing the processing of aggregate continuous queries.

Closely related to the DSMSs in the cloud is the distributed DSMSs (D-DSMSs). In [18], D-DSMSs have been motivated by the fact that monitoring applications are inherently geographically distributed. The Medusa [13] was proposed to address the main new challenge of D-DSMSs, which is adaptive load balancing. Special operator implementations for D-DSMSs have also been proposed, such as the binary join [76] and aggregation approximation, with differential accuracy requirements per data items [37]. In general, moving traditional relational operators to the data streams involves new queuing requirements that has been studied in [41].

Finally, adaptive processing of queries traditional DBMSs has been proposed [9, 10] as well as adaptive processing of CQs in DSMSs [49]. In the former case, the motivation was the changing characteristics of resources and data distributions in large-federated and shared-nothing databases. In such settings, assumptions made at query optimization may not hold at the execution time. The Eddies [9] adaptive processing scheme continuously determine the order of operators in a query plan, per tuple, depending on operators selectivities and resources characteristics. The continuously adaptive continuous query (CQCQ) [49] scheme was proposed to extend Eddies scheme to process CQs, adaptively.

2.5 SUMMARY

In this chapter, we presented the general background on DSMSs, and the specific background on optimizing the processing of ACQs; *Partial Aggregation* and the *Paired Window* processing scheme for optimizing individual ACQs and *Shared*, *Shared Data Shards* and *Intermediate Aggregates* optimizers for multiple ACQs. We also presented our experimental platform and summarized other related work.

3.0 WEAVE SHARE: EXPLOITING WEAVEABILITY TO OPTIMIZE ACQS

In this chapter, we study the interaction of the factors that affect the cost of a shared plan and identify the sharing trade off. The problem is first motivated and formally defined in Sections 3.1 and 3.2, respectively. The proposed solution and a discussion about practical implementation of the proposed optimizer is discussed in Sections 3.5 and 3.6, respectively. Finally, we evaluate our proposed schemes experimentally in Section 3.7.

3.1 MOTIVATION

Examining the *Shared* optimizer (discussed in Section 2.2.3.1) that utilizes *Paired Window* processing scheme (discussed in Section 2.2), it becomes clear that there is a tradeoff involved in the sharing of multiple ACQs. On one hand, partial aggregation is performed only once for all ACQs, as opposed once for each ACQ.

On the other hand, sharing might lead to increasing the number of fragments (and in turn edges) for each individual ACQ. This means that for each ACQ, more partial aggregates are generated at the sub-aggregate level and in turn, more operations are needed at the final-aggregate level. In some cases, the increase in number of final-aggregate operations per ACQ might outnumber the gains from sharing the sub-aggregate operations leading to an overall cost higher than processing each ACQ individually.

In particular, in a shared execution plan of a set of ACQs Q, if $l = \{s_i | s_i \; is \; the \; slide \; of \; q_i \in Q, 1 \leq i \leq |Q|\}$, then during a period of $lcm_i(s_i)$ each query $q_i \in Q$ sees a number of edges equal to M, where M is the number of merged edges in the common slide $CS = lcm_i(s_i)$. For a set of queries Q with slides l, the *edge rate* E per Q is defined as:

Definition 3. Edge Rate *(E) is the rate of sub-aggregate fragments (or equivalently, edges) produced by a shared sub-aggregate operator and is computed as:* $E = \frac{M}{lcm(S)}$.

To illustrate sharing trade off, consider again Example 3 of two ACQs q_a and q_b with ranges $r_a = 12$ and $r_b = 10$, and slides $s_a = 9$ and $s_b = 6$ seconds, respectively. According to the *Paired Window* (Definition 2), the fragments in q_a's slide are of length $g_{a,1} = 3$ and $g_{a,2} = 6$ and for q_b, the fragments are $g_{b,1} = 4$ and $g_{b,2} = 2$.

If q_a and q_b are processed independently, their sub-aggregation operators will produce 2 fragments every 9 and 6 sec, respectively. That is, an *edge rate* (i.e., number of fragments generated per sec) of $E_a = 0.22$ and $E_b = 0.33$ edges per sec. Thus, the total final-aggregation operations performed per sec is 0.55.

Meanwhile, if q_a and q_b share their partial aggregation, then s_a and s_b are integrated into *composite slide* $CS_{a,b} = lcm(s_a, s_b) = 18$ and the union of edges in $CS_{a,b}$ will appear at times (3, 4, 6, 9, 10, 12, 16, 18). Hence, each of q_a and q_b would examine a combined edge rate of $E_{a,b} = 0.44$, resulting in more final-aggregation operations (0.88 per sec). This simple example clearly shows the presence of a trade off in the shared processing of multiple ACQs.

The increase in edge rate, in turn, presents a trade-off in the total cost between shared and unshared processing of ACQs. In particular, for:

- *No Share:* Partial aggregation is performed once of *each* query, whereas the final-aggregate operator for each query q_i receives at most *two* fragments per slide s_i.
- *Sharing:* Partial aggregation is performed once for *all* queries, whereas the final-aggregate operator of each query q_i receives $E \times s_i$ fragments per slide s_i.

From the above, and given an input data stream with arrival rate λ tuples per second, we can compute the total processing cost of *unshared* set of n queries Q in terms of total number of aggregate operations per second as [45]:

$$C_{\textit{No Sharing}} = n\lambda + \sum_i \frac{2}{s_i} \times \lceil \frac{r_i}{s_i} \rceil \tag{3.1}$$

The term $n\lambda$ is the total number of operations required for partial aggregation and the term $\sum_i \frac{2}{s_i} \times \lceil \frac{r_i}{s_i} \rceil$ is the total number of final-aggregate operation. In particular, $\frac{2}{s_i}$ is the number of edges (fragments) per second for ACQ q_i, which is the edge rate E_i of q_i. Each of those edges

participates in the final-aggregate computation of $\lceil \frac{r_i}{s_i} \rceil$ window instances, which is the overlap factor of q_i.

Definition 4. Overlap factor (ω_i) *of ACQ* q_i *is the number of overlapping windows each tuple (and hence each fragment) belongs to and is computed as:* $\omega_i = \lceil \frac{r_i}{s_i} \rceil$. *Similarly, the* Overlap Factor (Ω) *of a set of shared ACQs* Q *is the number of overlapping windows each tuple/fragment belongs to and is computed as the sum of overlap factor of each ACQ in the shared set. That is* $\Omega = \sum_{\forall q_i \in Q} \omega_i$.

Similarly, we can compute the total processing cost of *shared* set (Q) of n ACQs in terms of total number of aggregate operations per second as:

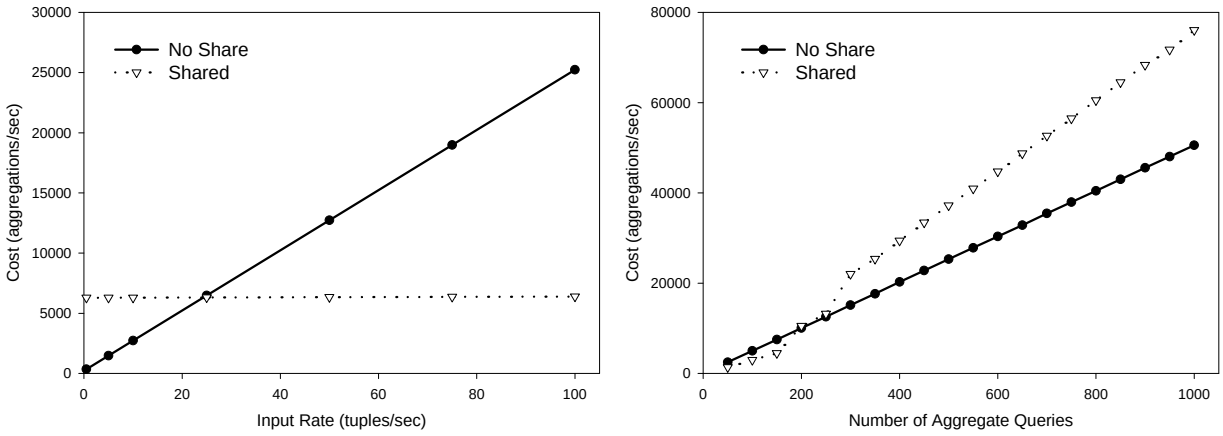
$$C_{Shared} = \lambda + E \times \Omega \tag{3.2}$$

where E is the output edge rate of the sub-aggregate operator. Notice that the cost of partial aggregation under sharing is only λ (as opposed to $n\lambda$ in Equation 3.1). The cost of final-aggregation, however, is computed as: $E \times \Omega$; since E is the number of edges generated per second and each of those edges participates in the final-aggregate computation of Ω window instances.

From Equation 3.2, it is clear that at high input rate λ, a shared processing is beneficial since it avoids repeating the work needed to compute partial aggregates leading to a constant cost of λ operations per second regardless of the number of queries in Q. The more sharing of ACQs, however, might lead to a significant increase in the cost of final-aggregation. This is due to two factors:

1. High Edge Rate: This is especially the case when the queries in Q have very few edges in common resulting in a high E.
2. High Overlap Factor: This is especially the case when the queries in Q exhibit a large number of slides per window resulting in a high $\sum_i \lceil \frac{r_i}{s_i} \rceil$ or equivalently, high Ω.

To further study that conflict, we performed an experiment to compare the costs of the *Shared* scheme [45] vs. *No Share* where queries are executed separately. Please, refer to Section 2.3 for the experimental setup. The results obtained by varying the input rate and by varying the number of ACQs are shown in Figure 6.



(a) Input Rate

(b) Number of Queries

Figure 6: *Share* vs *No Share*

Figure 6(a) shows that for a workload of 250 queries, *Shared* outperforms *No Share* for all input rates above 25 tuples/sec. For instance, at input rate 50 tuples/sec, *Shared* reduces the cost by 50%. This result is consistent with the result in [45]. Figure 6(b), however, shows that at the same input rate of 50 tuples per second, No share consistently outperforms *Shared* as the number of ACQs increase. This is due to the increase in cost of final-aggregates as explained above.

Figures 6(a) and 6(b) show that there is no clear winner between *Shared* and *No Share*. That is, sharing at the sub-aggregate level is sometimes at odds with the amount of processing needed at the final-aggregate level. This conflict depends on several factors such as data input rate, the size of workload (i.e., number of queries), as well as per query specifications (i.e., range and slide). The above observations motivated us to consider a new technique for optimizing multiple ACQs that would use a criterion that considers all the factors that impact the cost of the query plan.

3.2 FORMALIZATION

Our proposed optimizer *Weave Share* aims at reaping the benefits of cost reduction provided by sharing of partial aggregation phase while at the same time minimizing the increase in cost incurred at the final aggregation phase when sharing. This led us to the idea of grouping ACQs in *multiple* execution trees, where each tree contains only those ACQs that *fit* best together.

Under our scheme, a set of ACQs $Q = q_1, q_2, ..., q_n$ are distributed over a set of m trees $t_1, t_2, ..., t_m$ where all ACQs that belong to the same tree t_i are shared. Hence, the cost of each execution tree is the same as Equation 3.2, but is calculated for the set of ACQs in the tree. Thus the total cost of the *Weave Share* query plan is simply the sum of the cost of the individual trees. Formally:

Definition 5. *For a query plan that contains m execution trees, the total cost of the query plan in terms of total number of aggregate operations per second is computed as:*

$$C_{\textit{Weave Share}} = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{3.3}$$

where λ is the data input rate, E_i and Ω_i are the edge rate and overlap factor for tree t_i, respectively.

Equation 3.3 above represents the objective function that we are trying to minimize. The first term of the cost function ($m\lambda$) is the number of operations needed to generate the fragments (i.e., sub-aggregation), wheres the second component is the number of aggregate operations performed on the fragments to produce outputs (i.e., final aggregation).

Notice, that both *Shared* and *No Share* are two special cases of Equation 3.3. In particular, under *Shared*, the number of trees is equal to 1 ($m = 1$), whereas under *No Share*, the number of trees is equal to the number of individual queries n ($m = n$). On one hand, setting $m = 1$ enables Share to minimize the cost of the first component of the objective function (i.e., cost of partial aggregation). On the other hand, setting $m = n$ enables *No Share* to minimize the second component (i.e., final-aggregation).

Our goal in *Weave Share* is to find the balance between the two components of the objective function so that to minimize the total cost of the query plan. In particular, our objective is to find the most beneficial number of trees (i.e., m) as well as the best assignment of queries to each tree in order to provide the lowest execution time and highest throughput.

To this end, finding an optimal solution for ACQ sharing is provably an NP-hard combinatorial optimization problem as was formally shown in [83]. This motivated us to explore solutions based on greedy heuristics as it is the case in [83] and in traditional multiple query optimization and materialized views selection [66, 51].

3.3 WEAVEABILITY

The affinity of ACQs, i.e., their similarity, is an important factor that determines whether it is beneficial to share two ACQs or not. We refer to this affinity as the *weaveability* of ACQs. Specifically, given the paired-window processing scheme, two ACQs are said to be *perfectly weaveable* if the edges of both ACQs are identical. That is, when the two ACQs are shared, the edge rate does not increase for either of the ACQs. If the ACQs are not perfectly weaveable, the more *common* edges between the ACQs in their composite slide, the less the increase in edge rate for the ACQs when shared, hence the more *weaveable* they are. Thus, we define the degree of weaveability as follows.

Definition 6. *Given two ACQs q_a and q_b with slides s_a and s_b, respectively, the degree of Weaveability of q_a and q_b ($\mathcal{WV}_{a,b}$) is the ratio of the number of common edges M_c in the composite slide $CS_{a,b} = lcm(s_a, s_b)$, to the total number of edges ($M_{a,b}$) in $CS_{a,b}$. Specifically,*

$$\mathcal{WV}_{a,b} = \frac{M_c}{M_{a,b}} \tag{3.4}$$

Note that the definition of weaveability is recursively applicable to two groups of shared ACQs, i.e., execution trees.

Thus, if the edges of one ACQ is contained in the other, then all edges of the composite slide are common edges, and they have weaveability degree $WV = 1.0$. This definition is recursively applicable to two groups of shared ACQs, i.e., execution trees. That is the degree of weaveability of two trees is the ratio of the common edges to the total number of edges in their composite slide.

Sharing weaveable ACQs has a minimum impact on increasing the final-aggregation cost since they encounter minimal increase in the edges rate of the shared ACQs. For example, for the two ACQs q_a and q_b (Example 3), the set of edges of the composite slide are (3, 4, 6, 9, 10, 12, 16, 18), while the common edges are (12, 18). Thus, the weaveability $WV_{a,b} = \frac{2}{8} = 0.25$, which is a

weak weaveability and that is why their shared tree encounter a high increase in the edge rate as discussed in Example 3.

3.4 CHALLENGES OF GROUPING MULTIPLE ACQS

Grouping ACQs to multiple trees involves three major challenges. Namely: 1) designing a technique that effectively prunes the combinatorial search space, 2) handling the dynamic addition and deletion of ACQs over time, and 3) efficiently computing the weaveability with minimal overhead.

Towards the first challenge, grouping ACQs could be seen as first determining the optimal number of execution trees and then assigning ACQs to the trees. Thus, we have initially considered mapping our ACQ sharing problem to the generalized task assignment problem which is known to be NP-Hard [26]: the input is a set of heterogeneous machines and a set of tasks, where each task has a certain cost when processed on a certain machine. The output is an assignment of tasks to machines that minimizes the total cost.

This mapping, however, assumes the knowledge of number of machines (i.e., trees), which is not the case. Furthermore, even if we assume the knowledge of the optimal number of trees to use, the increase in processing cost when adding an ACQ to a tree is not constant as it depends on which other ACQs have already been assigned to that tree. This is simply true because the cost function in Equation 3.3 involves the edge rate term, which depends on which ACQs are shared and the degree of weaveability of those ACQs.

Thus, we can not directly use any of the classical algorithms for solving the task assignment problem (e.g., Dynamic Programming) to solve our ACQ sharing problem. This is mainly because an optimal solution for a sub-problem is not necessarily a part of the optimal solution of the whole problem. In other words, there is no optimal substructure property.

Given the problem complexity discussed above, we have explored a suite of alternative algorithms towards the efficient sharing of ACQs. In this paper, we present *Weave Share*, an efficient heuristic that fully considers all cost factors in generating shared plans (Section 3.5).

The second challenge is the need for an online version of the algorithm that handles the addition and deletion of ACQs as time advances. To handle this challenge, we propose *Incremental Weave*

Share, the online version of *Weave Share* that avoids the reconstruction of the query plan every time an ACQ is added or deleted. Both *Weave Share* and its online version are discussed in the following section (Section 3.5).

The third challenge (i.e., computing weaveability) stems from the complexity of counting the number of common edges between two different trees. This is because when merging two trees, there is no closed-form formula that determines the common edges. Specifically, this problem maps to small sieve theory problem which is a hard problem, and whose current solutions mostly deal with approximations and there is no closed formula to solve it [12]. Yet, the degree of weaveability directly determines the amount of increase in total processing cost (if any) when merging. To efficiently consider the weaveability while generating the shared plan, we propose several optimizations for the process of counting the number of common edges (Section 3.6).

3.5 THE *WEAVE SHARE* ALGORITHM

In this section, we describe *Weave Share*, our proposed algorithm for minimizing the execution cost of multiple aggregate continuous queries. Our proposed *Weave Share* exploits weaveability to reap the benefits of cost reduction provided by sharing partial aggregation, while minimizing the increase in cost incurred at the final aggregation. Basically, *Weave Share* tries to group ACQs in multiple execution trees, where each tree contains only ACQs that *weave* best together.

To achieve our goal, *Weave Share* takes a global view of the execution plan as well as the objective function to minimize (i.e., Equation 3.3). In particular, it simultaneously considers both of the cost components (i.e., partial- and final-aggregation) to group ACQs in multiple trees with minimum execution cost.

Weave Share (pseudo-code in Algorithm 1) takes as an input a set of ACQs $q_1, q_2, ..., q_n$ and produces a set of m shared trees where each tree contains one or more ACQs. Initially, the number of trees is equal to the number of individual queries, $m = n$ and each ACQ forms a separate tree, which is equivalent to the case of no sharing.

Weave Share advances towards sharing one step at a time in a greedy manner, where in each iteration two weaveable execution trees are merged, reducing the number of trees by one, until

Algorithm 1 The *Weave Share* Algorithm

1: **Input: A set of** n **ACQs**
2: **Output: Weaved query plan** P **that consists of** m **execution trees**
3: **begin**
4: $P \leftarrow$ *Create an execution tree for each ACQ*
5: $l \leftarrow n$ {current number of trees}
6: $(\textit{max-reduction}, t_1, t_2) \leftarrow (0, -, -)$ {current tree-pair to merge}
7: **repeat**
8: **for** $i = 0$ *to* $l - 1$ **do**
9: **for** $j = i + 1$ *to* l **do**
10: $temp \leftarrow$ cost-reduction-if-merging(t_i, t_j)
11: **if** $temp >$ *max-reduction* **then**
12: $(max - reduction, t_1, t_2) \leftarrow (temp, t_i, t_j)$
13: **end if**
14: **end for**
15: **end for**
16: **if** *max-reduction* > 0 **then**
17: merge(t_1,t_2)
18: $l \leftarrow l - 1$
19: **end if**
20: **until** No merge is done
21: Return P
22: **end**

either no more merging is beneficial or a single tree is reached. In particular, at each iteration, given a set T of l trees: $T = t_1, t_2, ..., t_l$ ($l \leq n$), *Weave Share* estimates the benefits of merging all possible pairs of trees in T and merges the pair of trees that yields the maximum reduction in total cost.

Given Equation 3.2, it is expected that for a pair of trees (t_x and t_y) to qualify for merging, they must satisfy either one or both of the following properties:

1. High degree of weaveability. The higher the degree of weaveability of the merged trees, the less the increase in the combined edge rate $E_{x,y}$ and the less the overall merged tree cost.
2. Low total overlap factor ($\Omega_{x,y} = \Omega_x + \Omega_y$), which is the total number of final-aggregation operations performed on each fragment in the new tree. The less the number of window instances, the less the number of final-aggregate operations performed on each fragment.

   The benefit (i.e., cost reduction) from merging t_x and t_y is:

$$\Delta(C_{x,y}) = \lambda + E_x\Omega_x + E_y\Omega_y - E_{x,y}\Omega_{x,y} \tag{3.5}$$

Note the term λ in Equation 3.5 above denotes the savings at the sub-aggregation level. That is, each tuple is processed once instead of twice. The rest of the terms in the equation represents the savings in the final aggregation level.

Clearly, any two trees that exhibit the two properties above are good candidates for merging as they allow us to exploit the sharing of partial-aggregation while at the same time minimize the increase in final-aggregation. These are the main optimization criteria for *Weave Share*. We demonstrate how *Weave Share* iterations work using the example below.

3.5.1 *Weave Share* by Example

Consider three queries q_a, q_b and q_c with sliding window specifications as shown in Table 3. Additionally, consider an input rate $\lambda = 1.2$ tuples per second. Figure 7 shows the sequence of iterations performed by *Weave Share* as well as the resulting query plans along with the trees weaveability.

Figure 7 shows that initially (first column to the left of the Figure), the number of trees is three, with no sharing, i.e., where each ACQ is to be processed independently. This results in a total cost of 11.6 based on Equation 3.3, as shown in the Figure (the calculations details are omitted for brevity). Next, the algorithm enters the main loop where it tries to merge a pair of trees that would reduce the cost the most.

In the first iteration, there are three possible pair-wise merges as illustrated in the second column of Figure 7. Specifically, the possibilities are $< q_a, q_b >$, $< q_a, q_c >$, or $< q_b, q_c >$. Merging the pair $< q_a, q_c >$ leads to the maximal reduction in cost, reducing it to 4.3 aggregations per second

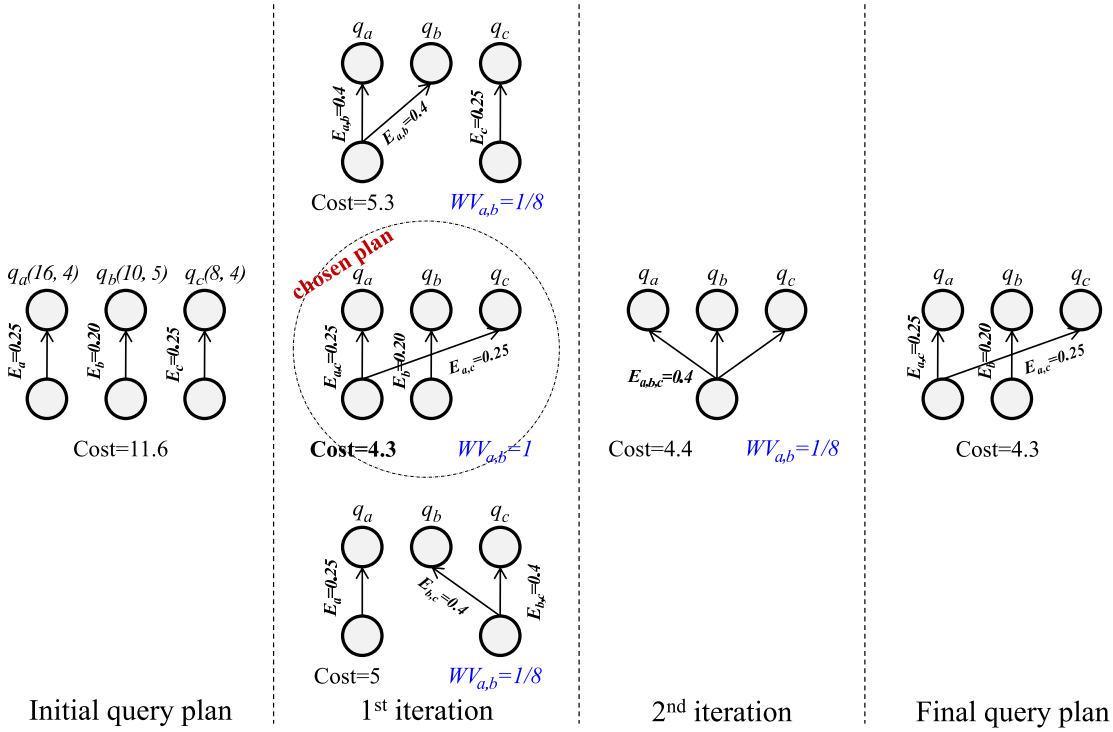


Figure 7: *Weave Share* by example - Iterations of *Weave Share*.

according to Equation 3.5. Thus, the algorithm merges them together into tree $t_{a,c}$ and proceeds to the second iteration.

In the second iteration (the third column in Figure 7, the only possibility is to merge $t_{a,c}$ with q_b. This, however, would lead to an increase in the cost to 4.4 aggregations per second. Since there is no room for improvement, *Weave Share* terminates the loop and returns the query plan it constructed: $t_{a,c}$ and q_b, where q_a and q_c are shared in $t_{a,c}$ and q_b is executed independently.

Note that q_a and q_c *weave* well together, in the sense that all the edges of q_a exist in edges of q_c (i.e., common). This is due to the fact that their slides are equal. This results in no increase in the edge rate when they are merged and in turn, minimizes the overall execution cost.

Table 3: *Weave Share* by example - windows' specifications

ACQ	range (r_i)	slide (s_i)	ω_i
q_a	16	4	4
q_b	10	5	2
q_c	8	4	2

3.5.2 Sharing AVERAGE ACQs

Sharing AVERAGE ACQs is a special case due to the way an average function is rewritten as a sub- and final-aggregation. In particular, the AVERAGE ACQ is rewritten as the division of the SUM of sub-SUMs by the SUM of sub-COUNTs. This is illustrated in the left-most part of Figure 8. Thus the sub-aggregation operator performs 2 operations: SUM and COUNT, and the fragments queue actually holds two fragments per entry. Thus, as illustrated in the middle part of Figure 8, the sub-aggregation is in fact equivalent to two operators, and the intermediate queue is equivalent to two queues. Similarly, the final-aggregation is equivalent to three operators, one that sums the sub-sums, second sums the sub-counts, and the last divides the two.

Computational overhead-wise, the two plans on the left- and the right-most parts of Figure 8 are identical. The only difference is that the right-most plan is more flexible to the scheduler to change order of execution to improve response time. In addition, the right-most plan has the potential to share the sub-aggregation operators with other SUM and COUNT ACQs. Therefore, the AVERAGE ACQs can be shared as follows.

1. Rewrite all AVERAGE ACQs as $\frac{SUM}{COUNT}$
2. Apply Weave Share to all SUM ACQs, including those of the AVERAGE rewriting.
3. Apply Weave Share to all COUNT ACQs, including those of the AVERAGE rewriting.
4. After the above 2 steps, if any AVERAGE function is not beneficial to rewrite, both its SUM and its COUNT won't be shared. Therefore, we re-group these sub-aggregation operators into one operator that performs both SUM and COUNT at same time, and apply Weave Share to these set of operators.
5. the final-aggregation operators of re-grouped sub-aggregation operators are also re-grouped.

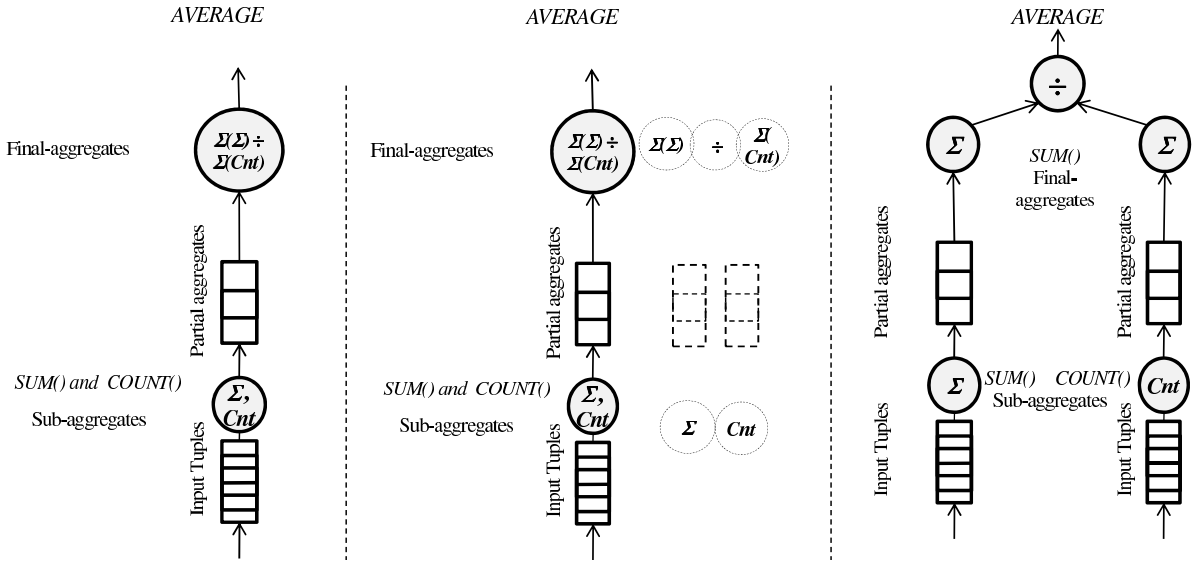


Figure 8: Sharing AVERAGE ACQs.

3.5.3 Varying Predicates and Group-by

Weave Share can easily handle the case when different ACQs have different pre-aggregation filters (i.e., selection operators). For example, one query might monitor the average-volume of stock-trades that are higher than $100, while another monitors the same for trades that are higher than $500. To share the execution of such ACQs, we adopt the *Shared Data Shards* (SDS) technique [45] as follows.

Figure 9 illustrates the weaved (or *Weave Share*) plan, integrated with the *Shared Data Shards* (SDS) scheme to optimize the handle the case of varying window specifications and different predicates (as discussed in Chapter 3, Section 3.5.3). The figure shows that ACQs with predicates defined on the same set of attributes, called *predicate-compatible* ACQs, are weaved separately, each yielding to one or more shared groups. That is, *Weave Share* is applied to each set of predicate-compatible ACQs.

The lower part of Figure 9 shows the augmentation process, where each tuple is evaluated against all predicates and augmented with a lineage, i.e. a signature, to encode which predicates this tuple satisfies. The signature is simply a bitmap vector, where each bit is set to 1 if the tuple satisfies the corresponding predicate. A router uses the signature of each tuple to route, and

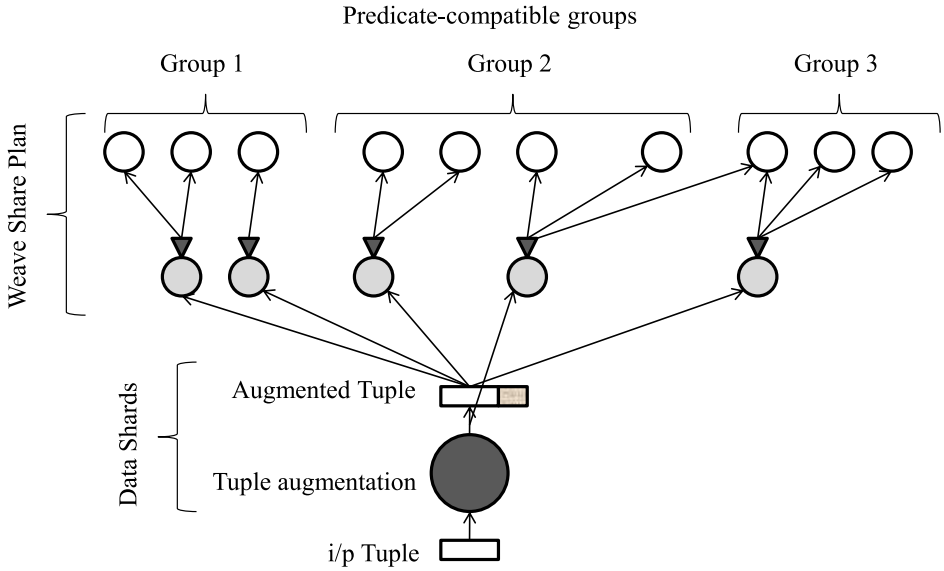


Figure 9: An Instance of a Weaved Plan

possibly duplicate, the tuple to every group for which the tuple satisfies a subset of its predicates.

Further, when different ACQs have different group-by attributes, *Weave Share* can utilize the techniques in [59, 84]. Specifically, each sub-aggregation operator can utilize a hash table based on the values of the union of all group-by attributes. When a fragment is due, proper hash table entries are combined together to form the fragment of each set of queries with identical group by attributes.

3.6 IMPLEMENTATION OPTIMIZATIONS OF THE *WEAVE SHARE* OPTIMIZER

In this section, we propose a set of implementation optimizations to increase the efficiency of the *Weave Share* optimizer in generating the weaved plan. We first analyze the time complexity of the *Weave Share* algorithm to spot potential performance bottlenecks to be optimized.

Given a set of n ACQs, the time complexity of *Weave Share* algorithm is asymptotically $O(n^3)$. The algorithm starts with n trees and in each iteration it reduces the number of trees by one. Thus, in worst case, the algorithm needs n iterations. In each iteration i, $(n-i)^2$ comparisons are needed

to find the pair of trees that yield the maximum benefit. Thus, the total time complexity is $O(n^3)$.

Computing the benefit of merging two trees (say t_x and t_y), requires calculating the new edge rate $E_{x,y}$ (Equation 3.5). Given that there is no closed-form formula that determines the common edges as as discussed in Section 3.4, this is clearly an expensive operation which requires counting the set of common edges between the two trees, t_x and t_y.

Conceptually, to calculate the new edge rate resulting from merging two execution trees t_x and t_y into one execution tree $t_{x,y}$, we need to extend the steps needed for merging two ACQs (described in Section 3.1) as follows:

1. Set the composite slide $CS_{x,y}$ to be the least common multiple of the individual slides of all ACQs in t_x and t_y.
2. The edge count M'_x of the ACQs in t_x within the new composite slide $CS_{x,y}$ is computed as: $M'_x = M_x \frac{CS_{x,y}}{CS_x}$, where the last term is the number of times CS_x has been replicated. Similarly, the edge count M'_y of the ACQs in t_y is computed.
3. The composite edge count $M_{x,y}$ is computed as: $M_{x,y} = M'_x + M'_y - M_c$.

In order to compute the last step, we need to know the number of common edges in the composite slide (M_c) between t_x and t_y. This could be done by checking each edge in each ACQ in t_y to see if it is the same to any edge of any ACQ in t_x. Each one of those checks requires two comparisons. Specifically, to check if edge e of some ACQ in t_y is the same to some edge of ACQ q_i in t_x, we check if e is a multiple of the slide of q_i or a shifted by $g_{i,1}$ multiple of the slide. Formally, e is a common edge iff:

$$e\%s_i = 0 \text{ or } (e - g_{i,1})\%s_i = 0 \tag{3.6}$$

This is illustrated in the following example.

Example 5. *Consider a tree with one query q_x that has slide $s_x = 5$ and fragments $g_{x,1} = 2$ and $g_{x,2} = 3$. Further consider a query q_y which has slide $s_y = 3$ and fragments $g_{y,1} = 0$ and $g_{y,2} = 3$. If q_x and q_y are to be merged, the common slide length is $CS_{x,y} = 15$, the edge counts of stretched q_x and q_y are $M'_x = 6$ and $M'_y = 5$, respectively. Hence, $M_{x,y}$ is 5 plus 6 minus the number of common edges (M_c), which is computed by checking each and every edge of q_y against those of q_x.*

The first edge in q_y is $e = 3$, which is not divisible by the slide of $s_x = 5$ nor is $e - g_{x,1} = 3 - 2 = 1$ divisible by $s_x = 5$. Hence, it is not a common edge and $M_{x,y}$ is kept at 11 edges. The current edge e is then advanced to next edge $e = 6$, and the two comparisons are performed and so on until $e = 12$, where $e - g_{x,1} = 12 - 2 = 10$ is divisible by $s_x = 5$, i.e., it is a common edge and the count is decremented. Similarly, at $e = CS_{x,y} = 15$, e is divisible by s_x and the count is decremented once again.

This naive approach encounters a high overhead given that counting the edges process is repeated many times in the main loop of the algorithm, where, in each iteration, an edge count is needed for each pair of trees. We propose three optimizations that can dramatically minimize this overhead as discussed next.

3.6.1 Optimization I: Cost Lookup.

The first optimization we propose is to memoize the benefit (i.e., reduction in the total cost of the query plan) gained by merging two execution trees. This is very similar to Dynamic Programming approach which avoids repeated computations by memoizing previous computations in a look up table. We utilize a two dimensional array called *Cost Lookup* table to store the merging benefits. Thus, in the main loop of the algorithm, only the first iteration will compute the cost saving for each pair of trees. Next iterations will use the lookup table for all pairs, except those that involve the new merged tree from the previous iteration. Thus, the number of computations in each iteration i is reduced from $(n - i)^2$ to $(n - i)$ computations. This minimizes the number of pairs for which an edge count needs to be performed.

Figure 10 shows a possible instance of the Cost Lookup table. To check if merging two trees t_i and t_j is beneficial or not, we lookup the entry $Cost\_Lookup[i][j]$, which is 101.2 in this instance. This means that merging t_i with t_j would reduce the cost by 101.2 operations per second. Negative values mean that the merge would actually increase the cost. $t_{justmerged}$ is the merged tree in a previous iteration and that is why all its entries are nullified in order to be recomputed.

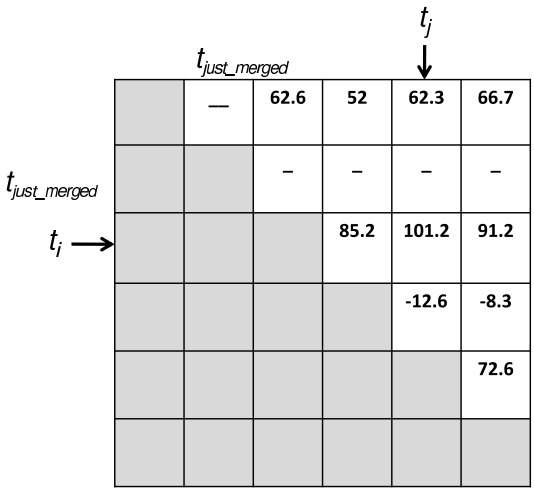


Figure 10: Cost Lookup Table

3.6.2 Optimization II: Edges Bitmap.

The second optimization is to use a bitmap vector that acts as a hash table to represent the edges. The top part of Figure 11 shows the bitmap vector for an ACQ q_x with $s_x = 5$ and edges at locations 2 and 5 (i.e., fragments $g_{x,1} = 2$ and $g_{x,2} = 3$). Given the *Edges Bitmap* structure, finding the common edges between two trees requires to simply traverse the edges of one of the Edges Bitmap to probe the other, i.e., check if they exist in the other bitmap. This requires a number of probes equal to the number of edges in one of the trees, regardless of the number of ACQs in the other tree. Effectively, this optimization pre-computes and materializes the results of finding the common edges described in Example 4.

The Edges Bitmap is maintained as follows. When the tree has one query at most two edges are hashed into the bitmap. When adding a query to a tree, 1) new bitmap is created with length equal to the new composite slide, 2) the old bitmap is replicated in this new bitmap and the previous count of edges is updated accordingly, and 3) the edges of the new query are hashed into the new bitmap, incrementing the edge counter only if no collision occurs.

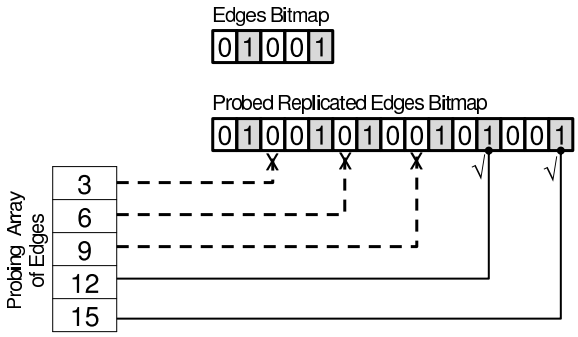


Figure 11: Edges Bitmap and Probing Process

3.6.3 Optimization III: Probing Reorder.

Clearly, given the Edges Bitmap structure, the overall complexity of the algorithm will be affected by the choice of which bitmap to probe when counting common edges. Similar to join optimization, which uses the relation with fewer blocks to probe the other, we propose to use the tree with fewer edges (i.e., smaller edge rate) to probe the other. (this is illustrated in the lower part of Figure 11, where we used q_y which has 5 edges to probe q_x which has 6 edges). Specifically, the bitmap of the probed tree is replicated to the new composite slide, while the bitmap of the probing tree is used to generate an array of edges in the new composite slide. Edges in the array are then hashed into the bitmap of the probed tree, and if collision occurs, then the checked edge is common.

3.7 EVALUATION

Using the simulation platform introduced in Section 2.3 we evaluated the quality *Weave Share* plans (discussed in Section 3.7.1), as well as evaluating the performance of the *Weave Share* optimizer (discussed in Section 3.6).

Before presenting our results, let us review the the algorithms used in our evaluation.

- *Random*: it initializes by creating a tree for an arbitrary ACQ. Then it proceeds for each ACQ,

in random order, to either add the ACQ to the last tree, or to create a new tree for it by flipping an even coin, i.e., equal probability to both decisions.

- *Exhaustive search* simply tries all possible grouping of ACQs. It worth mentioning that in the few simple cases (input rates of 200, 300 and 400 tuples/sec, each with 5, 10 and 15 ACQs), that we were able to get results for exhaustive search (after running the simulator for days-week), *Weave Share* generated the same result as exhaustive search.
- We implemented *Shared* (where all ACQs are merged in one single execution tree [45]), *No Share* (as a base line, where each ACQ is executed separately) and an adapted version of *Local Search* (LS) (see Appendix A).

Recall (Table 2) that we changed the number of ACQs, the input rate, slide skewness, maximum overlap factor and the initial state and the steps bound of *LS*.

3.7.1 Quality of Weave Share Plans

In this section, we present the evaluation and sensitivity analysis of the quality of *Weave Share* plans.

3.7.1.1 Number of ACQs (Fig. 12 to 14) Figures 12 and 13 show the cost of the *Weave Share* plan as the number of ACQs increases from 50 to 1000, for low (50 tuples/sec) and medium (300 tuple/sec) input rates, respectively. In both plots, the maximum overlap factor is set to 50, and the slide skewness is 0.6. As shown in the figures, *Weave Share* always outperforms the best of all other algorithms. For instance, for 1000 ACQs, *Weave Share* outperforms *Insert-then-Weave* and *Shared* by three and four orders of magnitude, at low and medium input rates, respectively.

Among the different versions of *Local Search* we ran, we plot the results of the best version, which starts from *No Share* state, and proceeds for a maximum of 10 times the steps that *Weave Share* needed for the same workload instance.

We note that *No Share* and *Random* generate the most expensive plans in both cases. *Local Search* and *Insert-then-Weave*, on the other hand, performs better than *Shared* at low input rate (50 tuples/sec), while *Shared* outperforms both at medium input rate (300 tuples/sec). We also

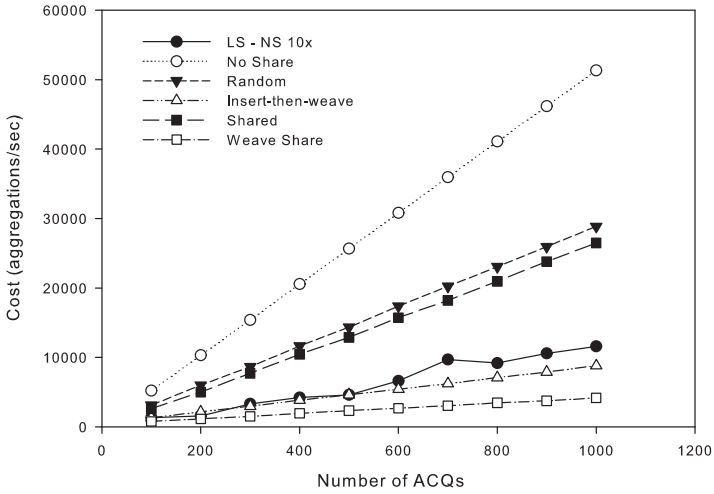


Figure 12: Impact of #ACQs: Low input rate (50 tuples/sec)

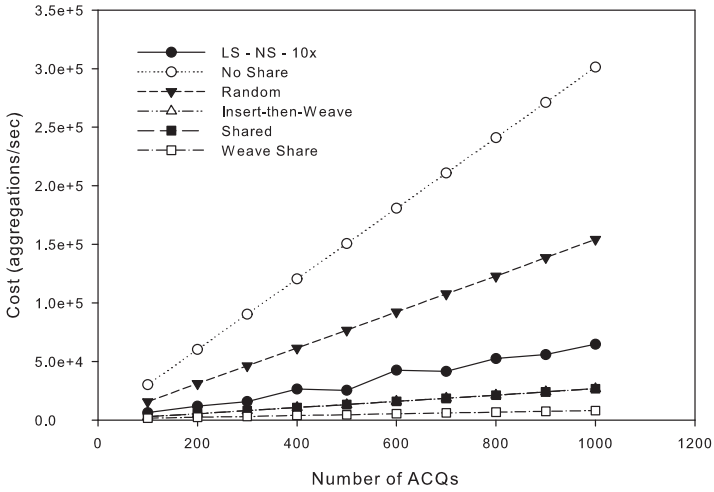


Figure 13: Impact of #ACQs: Medium input rate (300 tuples/sec)

repeated the experiment for high input rate (10K tuples/sec) and the relative behavior of different algorithms is similar to the medium input case.

In Figure 14, we zoom into the performance of *Weave Share* compared to *Shared* as the number

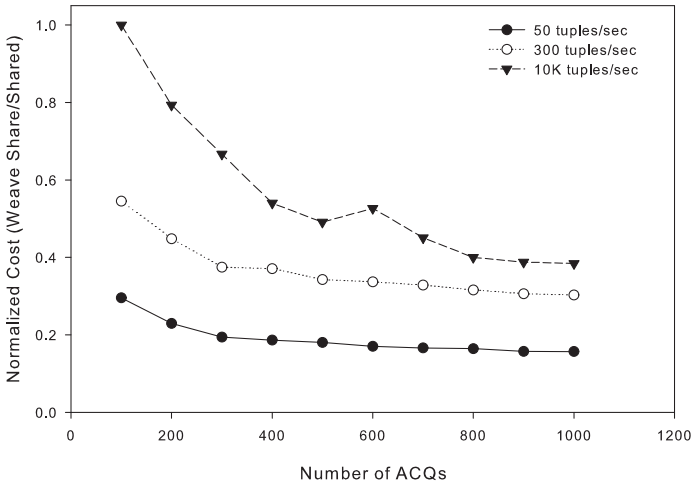


Figure 14: Impact of #ACQs: low, medium and high input rates

of ACQs increases, for the low, medium and high input rates. Specifically, we plot the normalized cost of *Weave Share* plan to the cost of *Shared*, for the three input rates. The figure shows that as the number of ACQs increases, the gain of *Weave Share* increases. It also shows that even for high input rate (10K tuples/sec), as the number of ACQs increase, *Weave Share* outperforms *Shared*. For instance, at 1000 ACQs, for input rate of 10K tuples/sec, *Weave Share* achieves a gain of 62%.

The improvement of *Weave Share* over the best of other algorithms increases as the number of ACQs increases. This is because, the more ACQs, *Weave Share* selectively merges together those ACQs that weave well together, limiting the increment in edge rate (E) and overlap factor (Ω) per tree, while gaining the benefits of shared sub-aggregation. *Local Search* seemed to need more than 10 times the steps to reach a better plan, while incurring a very high overhead. LS-NS 10x took thousand times the time needed by *Weave Share*. It worth mentioning that for an instance of 10 ACQs, *Local Search* generated a plan that is 13% cheaper than that of *Weave Share*. However, this small size does not reflect the commonality of ACQs in monitoring applications.

Figure 15 shows the number of execution trees that were generated by *Weave Share* for the same settings as in Figure 14. As expected, the number of trees increases as the number of ACQs increases, while it decreases as the input rate increases. It also shows that for high input rate of

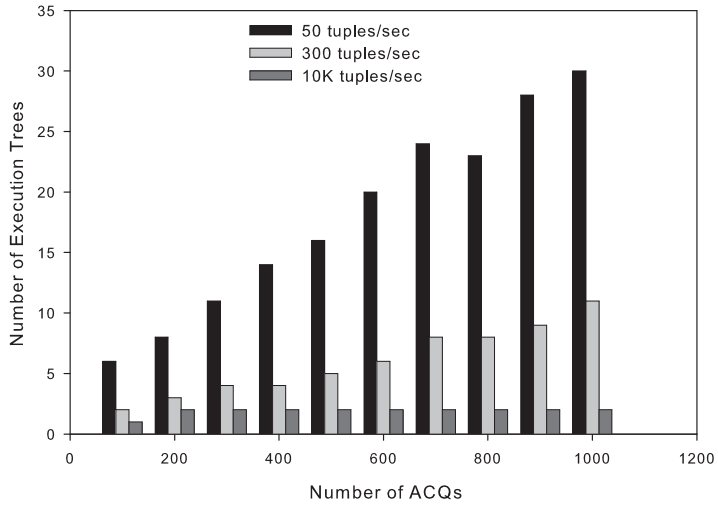


Figure 15: Number of Execution Trees

10K tuples/sec, *Weave Share* still generates more than one tree for more than 100 ACQs. This confirms our observation that the properties of the installed ACQs are as important as the input rate in determining the sharing decision.

Finally, we also tested the performance of *Weave Share* with the workload used to study the sharing tradeoff in Section 3.1 (Figure 6(b)). For all data points, *Weave Share* outperforms the best of the *Shared* and *No Share* by orders of magnitude. For instance, in Figure 6(b), at 1000 ACQs *Weave Share* reduces the cost by 20 times compared to *No Share* and by 30 times compared to *Shared*.

3.7.1.2 Input Rate (Fig. 16) In this experiment we study sensitivity of *Weave Share* to the input rate. We report the normalized cost of *Weave Share* to that of *Shared*, in all the experiments hereafter, as *Shared* is the best alternative (in each experiment).

We plot the normalized cost for different values of number of ACQs in Figure 16. The results in this plot are for workload with Ω_{max} of 50 and slide skewness of 0.6. Similar to the previous experiment, as the input rate increases, the gain of *Weave Share* decreases. For instance, for 250 ACQs, the gain of *Weave Share* starts at 80% at input rate of 50 tuples/sec, and reaches 24% and

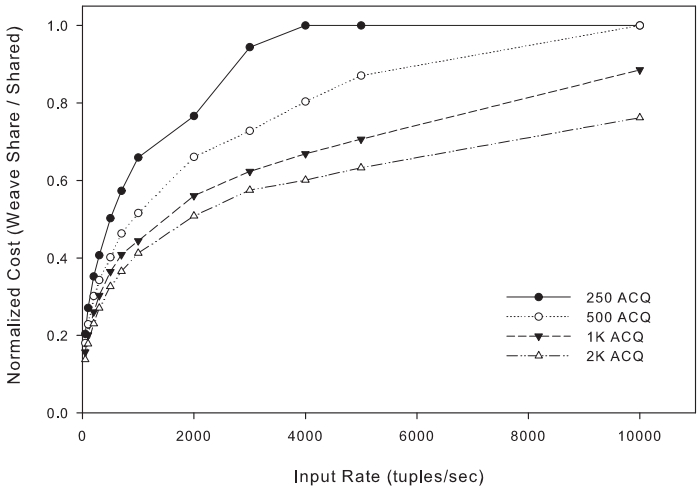


Figure 16: Impact of Input Rate - different # of ACQs

6% at input rates of 2K and 3K tuples/sec, respectively. For this small number of ACQs (only 250), *Weave Share* converges to generate one shared tree only at input rate of 4K tuples/sec. Moreover, even for high input rate (10K tuples/sec), *Weave Share* achieves a gain of 12% and 24% for 1000 and 2000 ACQs (which confirms the results of the first experiment).

We observed that *Insert-then-Weave* converges to *Shared* very fast, i.e., at low input rate values. For instance, for input rate of 300 tuples/sec, *Insert-then-Weave* generates one tree, while *Weave Share* generates a plan that is three orders of magnitude better than *Shared*. The reason is that the initial insert phase generates much fewer trees than the *No Share* case, and thus, the weaving phase has higher potential to generate one tree (because it merges fewer trees).

Weave Share also outperforms *No Share*, *Local Search*, and *Random* by orders of magnitude. For instance, at input rate of 10K tuples/sec, *Weave Share* generates a plan that is more than 100 times better than the best of them.

3.7.1.3 Maximum Overlap Factor (Fig. 17) In this experiment, we vary the maximum overlap factor (Ω_{max}) for different input rate values. Specifically, we set the input rate to 100, 1K, 10K, 100K and 1M tuples/sec. For all cases, the slide skewness was 0.6, and number of ACQs was 2000

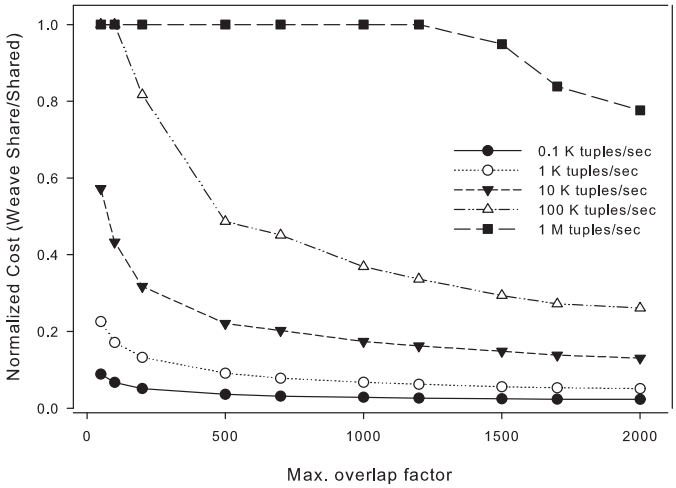


Figure 17: Impact of Ω_{max}: different rates

ACQs. Recall that the overlap factor is the ratio between an ACQ's range and its slide. Hence, increasing the overlap factor increases the number of final-aggregations but it has no effect on the sub-aggregation.

In Figure 17 we plot the normalized cost of *Weave Share* to *Shared*. As expected, as the maximum overlap factor increases from 50 to 2000, the gain of *Weave Share* increases. For instance, for 1M tuples/sec input rate, *Weave Share* achieves a gain of 23% at overlap factor of 2000, while it achieves a gain of 98% at input rate of 100 tuples/sec.

We also observe that all algorithms exhibit an increment in the cost as the maximum overlap factor increases, reflecting the fact that the the overlap factor is multiplied by the edge rate in Equation3.2, which is the cost of final-aggregation. The increment in *Weave Share* however, is much slower than that of *Shared*. This is because *Weave Share* generates plans that consist of more than one tree, keeping the maximum value of $E\Omega$ as small as possible. This enables Weave Share to outperform all other algorithms for most cases, or performs similar to the best (*Shared*) in the remaining cases.

3.7.1.4 Slide Skewness (Fig. 18)

In this experiment, we examined the slide distribution skewness parameter. By increasing the skewness, the query workload will contain more large-slide queries as generated by the Zipf distribution. Figure 18 shows the normalized cost of *Weave Share* to *Shared* for different number of ACQs, at input arrival rate of 100 tuple/second and maximum overlap factor of 10.

For all number of ACQs, we see that as the skewness increases, the relative gain provided by *Weave Share* increases. This continues until a global maximum is reached, where the gain starts to diminish until *Weave Share* performs similar to *Shared* (i.e., share all ACQs). The reason is that initially, as the skewness increases the more large-slide ACQs we have, and hence the higher the *penalty* of sharing them with small-slide ACQs which are not weaveable to them. *Weave Share* avoids this by selectively sharing ACQs that weave well together.

As the Zipf distribution becomes very skewed towards large-slides, most of the ACQs are large-slide ones, whereas small-slide ACQs gradually disappear. This means that grouping all in a single tree is the right choice. In which case, *Weave Share* captures this phenomenon and does generate a single execution tree, sharing all ACQs. Figure 18 also shows that the more ACQs are in the system, the larger the maximum gain of *Weave Share* is. This is consistent with the previous results shown in Figures 12 and 13.

3.7.2 Theoretical Lower Bound

Finding a theoretical lower bound is interesting and challenging, and it is one of our ongoing efforts. As in traditional multi-query optimization, our goal is to avoid ”worst-case” query plans and indeed it could be easily shown that *Weave Share* always avoids the poor plans that might be generated by either *Shared* or *No Share*. We also experimentally investigate and demonstrate the competitiveness of *Weave Share* by comparing it to *Exhaustive Search* (OPT) and *Local Search* (LS).

The *OPT* experiments (with different settings, some ran for a month and some ran for over 200 days) showed that *Weave Share* generates mostly optimal plans. Specifically, for input rates of 200, 300 and 400, with number of ACQs 5, 10 and 15, respectively, *Weave Share* generated the optimal plan. In only one case, *Weave Share* generated the optimal number of execution trees, but

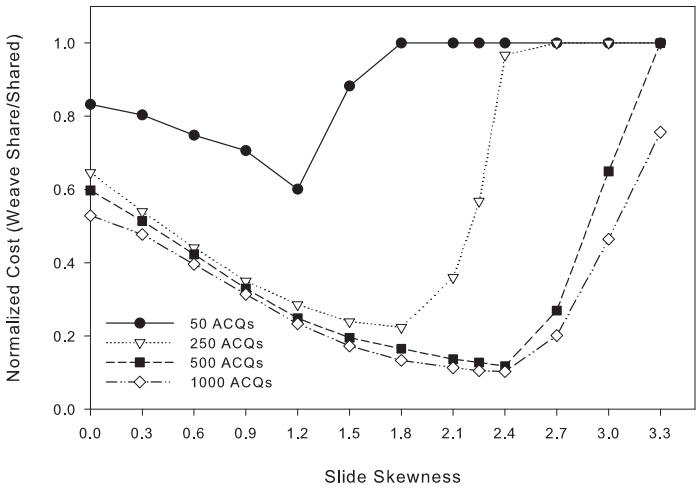


Figure 18: Impact of Slide Skewness

with 3% higher cost, due to a different grouping of ACQs. In this specific case, *Shared* plan was 32% more costly compared to the optimal plan.

LS is a near-optimal technique that utilize backtracking to avoid local optima. LS didn't find a better plan than those generated by *Weave Share*, while incurring a very high overhead. Specifically, LS-NS-10x took thousand times the time needed by *Weave Share* and didn't generate a better plan. The reason is that an iteration of LS moves a single ACQ from a tree to another, while an iteration of *Weave Share* merges two trees, i.e., moves a group of ACQs at once. Thus, Weave Share reaches a reasonable sub-optimal solution much faster than LS.

3.7.3 Impact of Optimizations

The impact of the above optimizations can be seen in Figure 19. The figure shows the overhead for a setting of 250 ACQs, input rate of 100 tuple/second, a slide skewness of 0.7 and a maximum overlap factor 10. The figure shows the overhead of the naive *Weave Share*, where no optimization is used, compared to the three optimization variants. In the first variant, only cost lookup is used. In the second variant, both cost lookup and edge bitmap are used and finally, in the third variant all three optimizations are used.

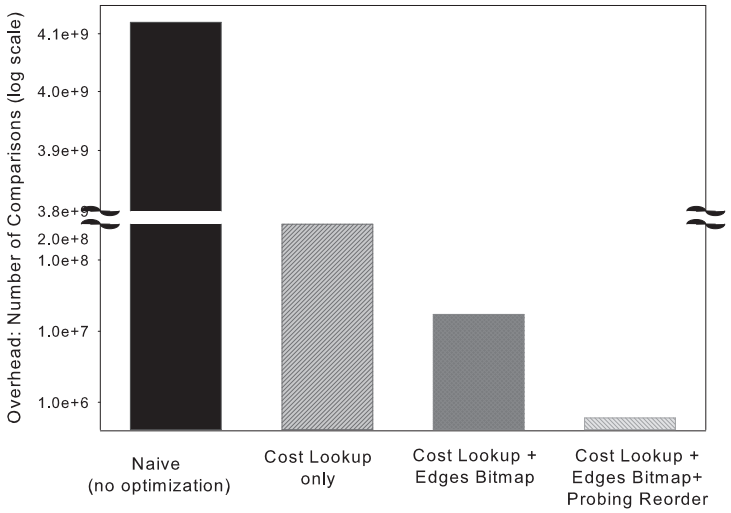


Figure 19: Optimizations' Benefits

Figure 19 shows orders of magnitude reduction in the overhead with the addition of each of the proposed optimization techniques (notice the log scale for the Y-axis). This leads to a total overall reduction of 99% compared to the naive approach. We obtained similar results for different workload settings.

3.8 SUMMARY

In this chapter, we studied the factors that affect the sharing decision of multiple similar ACQs, assuming they have the same pre-aggregation predicates and same group-by attributes, but different window specifications. We introduced the concept of *Weaveability* to capture the affinity of ACQs and the potential gains of sharing their processing. We also proposed the *Weave Share* optimizer, which is a cost-based multiple ACQs optimizer that utilizes *Weaveability* to group ACQs into multiple execution trees to minimize the total cost of the query plan. We also proposed several optimizations for the *Weave Share* optimizer that dramatically improves its efficiency, compared to the naive implementation. We experimentally demonstrated the quality of the generated *Weave*

Share plans which are up to four orders of magnitude better than the best alternative. We also demonstrated the impact of the implementation optimizations.

4.0 INCREMENTAL Weave Share

In the previous chapter, we described the basic (offline) *Weave Share*, which constructs a query execution plan from scratch. In this chapter, we consider the online case where newly submitted ACQs are weaved into an existing weave share query plan, as well as the case of re-weaving existing trees after the deletion of some ACQs.

4.1 ADDING NEW ACQS

Reconstructing the *Weave Share* query plan from scratch is one possible solution to handle the submission of a new set of ACQs into the system. In that solution, given an already existing set of ACQs Q in a *Weave Share* plan P and a set of new ACQs Q', *Weave Share* is invoked to generate a new weave share plan P' which includes the ACQs $Q \cup Q'$. This solution, however, has two drawbacks: 1) it incurs a large overhead since the algorithm is re-invoked to run from scratch whenever new ACQs are added, and 2) it might often lead to an unnecessary reconstruction since, in many cases, the new plan P' can be directly achieved from the current plan P.

To address the above drawbacks, we develop *Incremental Weave Share* which takes a more *lazy* approach for maintaining the weaved plan. This involves the following two steps:

1. Immediately incorporating new ACQs into the existing plan.
2. Reconstruct the query plan from scratch *only when needed*.

In this incremental version of *Weave Share*, a new tree t_{new} is created for each new ACQ q_{new} that is added to the system and *Weave Share* is invoked to merge t_{new} with the trees in the current plan P to generate a new incremental plan P''. Thus, among the existing trees, t_{new} will be

Algorithm 2 The *Incremental Weaved Share* Algorithm

Input: A new query q **and current query plan** P
Input: Offline slope and tolerance factor ϵ
Output: Updated weaved query plan P'
BEGIN
$t \leftarrow$ *Create a new execution tree for* q
$P' \leftarrow P \cup t$
repeat
  $l \leftarrow$ *current number of trees*
  $maximumsave \leftarrow 0$
  for $i = 0$ *to* $l - 1$ **do**
    for $j = i + 1$ *to* l **do**
      calculate the save if trees t_i and t_j are merged
      update maximum save info
    end for
  end for
  if a pair found **then**
    merge the trees that would lead to maximum save
    $l \leftarrow l - 1$
  end if
until No merge is done
if $\frac{cost(P')}{cost(\textit{offline weaved plan})} > \epsilon$ **then**
  $P' \leftarrow Call(\textit{Weaved Share(set of all queries)})$
  Update learned $offlineslope$
end if
Return P'
END

merged with the one tree with which it weaves the best. The newly merged tree might be further merged with other trees in the plan if this is beneficial. This process continues until no further improvements are attainable.

The cost of the incremental plan P'' might, however, be worse than the plan P' which would be generated by the offline *Weave Share*. In order to detect the magnitude of that degradation, *Incremental Weave Share* maintains the *performance slope* of the *plan-cost curve*. This curve is basically a plot of the offline-generated plan cost vs. the number of ACQs. The points on the curve are obtained when a plan P' is generated from scratch.

As new ACQs are submitted to the system, the cost of P'' is compared with the extrapolated cost using the performance slope. If the difference percentage is more than a certain *deviation tolerance* threshold, which is a system parameter, a reconstruction phase is triggered and performed asynchronously. Specifically, for a *deviation tolerance* of ϵ, a reconstruction is triggered iff:

$$\frac{cost(P'')}{\textit{extrapolated cost using performance slope}} - 1 > \epsilon \tag{4.1}$$

As such, the deviation tolerance value acts as a *knob* to control the reconstruction behavior. For instance, setting the tolerance to zero, resembles reconstructing the weaved plan whenever a new ACQ is added, whereas setting the tolerance to ∞ is equivalent to the case where no reconstruction is ever performed.

Finally, it worth mentioning that if a reconstruction is triggered, the actual cost of the offline query plan is compared to the online one and the better is deployed. This is to avoid the case when the extrapolated cost is misleading.

4.2 DELETING ACQS

We handle the deletion of existing ACQs similarly to the addition of new ACQs. Specifically, deleted ACQs are first removed from their respective execution trees. Then the benefit of merging each of those updated trees with each of all the other trees (updated and not updated) in the weaved plan need to be computed. This is similar to *Weave Share* iterations, where the just merged execution tree entries in the cost-lookup table are updated. This process is repeated until no more

improvements are attainable. Similarly to adding ACQs, given the performance slope and a tolerance factor, a reconstruction phase may be triggered depending on the degradation from the extrapolated cost.

4.3 WEAVED PLANS SWITCHING

In this section we describe how *Incremental Weave Share* can switch to new weaved plans without interrupting in progress data processing. Specifically, the updated weaved plan contains three types of execution trees: deleted, new and updated trees. Below we describe how to handle each of them.

First, the final-aggregation operators of deleted ACQs are marked in the current executing weaved plan, and stop executing. Second, *Incremental Weave Share* generates the updated Weaved plan while the current plan continues executing. Once the new weaved plan is ready, the new trees are added to the running plan and starts execution. Finally, the updated trees are handled as follows.

The current window edge due by each final-aggregation operator is marked both in the current and updated plans. Input tuples and fragments (sub-aggregations) needed to generate the current window are fed to both plans. The current plan is allowed to continue execution until each ACQ produces the current window aggregate result, at which point, the corresponding final-aggregation operator in the new plan starts executing. Once all ACQs in the current plan produces its output, the tree is removed from current plan. Eventually, all updated trees will be replaced by the new plan trees.

4.4 FREQUENCY OF ACQS ADDITIONS AND DELETIONS

The frequency of how often ACQs are being added or deleted has a direct impact on the performance of *Incremental Weave Share*. The more frequent the additions and deletions are, the more benefits *Incremental Weave Share* has, because it avoids frequent expensive reconstructions. In a typical monitoring application, there are several phases of popularity of the application which

is reflected on the frequency of addition of ACQs. The first phase is the setup phase, in which the application, or the phenomenon to be monitored, is not popular yet. In that phase, ACQs are added sporadically. The peak phase of an application is when it becomes very popular, during which ACQs are massively added. Then a calm down phase follows, when saturation of number of ACQs is reached: a saturation phase. Finally, occasionally, some external events might trigger new interest in the application, which leads a new epoch of ACQs to be added (epoch phase). Another possibility is the periodic interest, such as monitoring events that trigger interest periodically. For instance, sales of flu shots is of interest during the Fall season. ACQs that monitor flu shots sales is expected to be registered periodically, and be deleted after the season passes. Deleting ACQs in general is not very common, by definition of the ACQ being a continuous query, i.e., a query that runs continuously.

Example 6. *In a financial market monitoring application, when a new start-up company launches (setup phase), people starts slowly monitoring its index performance. Once it becomes popular (peak phase), hundreds of ACQs will be registered to monitor this company, until the saturation phase is reached. Once a while, some global, technical, economical or political events may trigger new interest in this company, which lead to a the epoch phase*

4.5 ADAPTING TO CHANGES IN INPUT RATE

The data stream input rate affects the cost of the Weaved plan. It is known that input rates typically fluctuate. To adopt to every single change in the input rate will incur a huge unnecessarily overhead, especially that the weaved plan won't change for a small change in the input rate. Dramatic changes, however, might lead to a change in the weaved plan. By dramatic changes we mean when the change of the input rate exceeds the edges rate. Vise versa, when the change of the input rate goes below the edge rate, it is also a dramatic change. A dramatic change essentially changes the dominating factors of the cost function of the weaved plan, and hence might change the sharing decision.

To handle the changes in the input rate, we propose to segment the expected input rate range into low, medium and high ranges. A weaved plan is then generated for each range. Thus, for

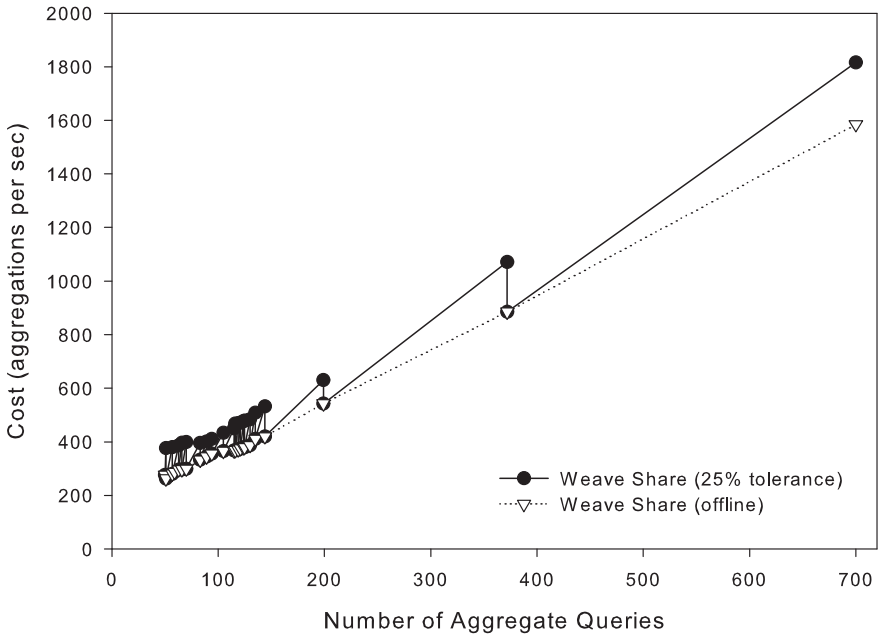


Figure 20: Incremental vs offline *Weave Share* - Deviation

changes between these ranges no computations are needs to be done. If the input rate changes to a value not contained in the existing ranges, e.g., a burst arrival that exceeds the high range, the nearest range can be utilized while a reconstruction takes place in offline. The plan for this new input rate range is memoized to be utilized later if needed.

To handle both changes in ACQs (i.e., addition and deletion) and changes in input rate, whenever a reconstruction phase is triggered by *Incremental Weave Share*, it computes a plan for each of the input ranges, starting with the current input rate range.

4.6 EVALUATION

In this section we study the performance of the *Incremental Weave Share* optimizer. In the first experiment shown in Figure 20, we plot the cost of the weaved plans that are incrementally generated by *Incremental Weave Share* as ACQs are added to the system. The tolerance factor for the

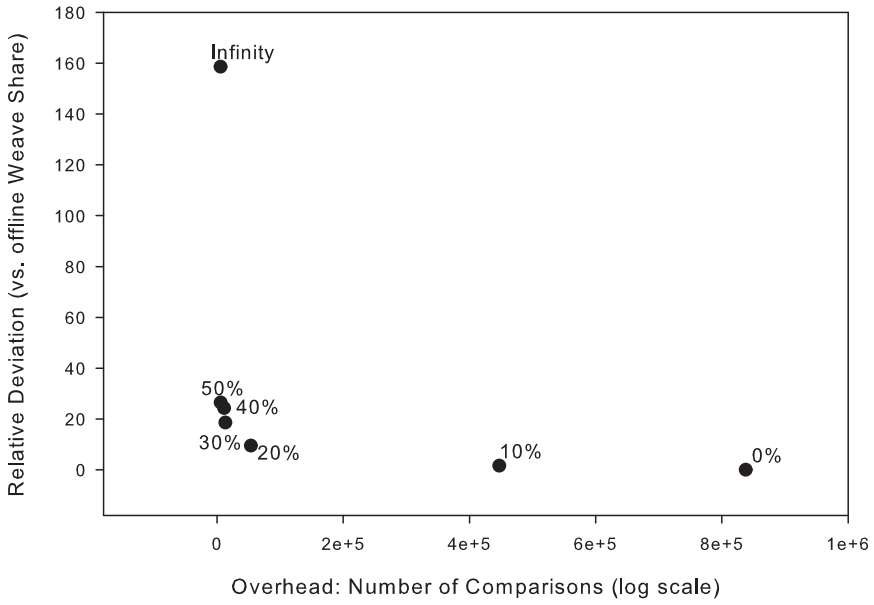


Figure 21: *Incremental Weave Share* - Overhead

results in this figure was 25%. We also plot, in the same figure, the cost of the weaved plans that are generated by the offline *Weave Share*.

Recall that, for *Incremental Weave Share*, the tolerance factor is used to determine when to issue a reconstruction phase. That is, a reconstruction phase is triggered if the ratio of the current execution plan cost to the extrapolated offline cost, given the learned offline slope, exceeds the tolerance factor. As such, as the figure shows, with adding more ACQs, *Incremental Weave Share* deviates from the offline version until the deviation exceeds the tolerance of 25% that is when reconstruction is performed and the online and offline performances become the same. The figure also shows that the rate of reconstruction decreases with increasing the number of ACQs. This is because the more ACQs, there is a higher chance for a new ACQ to find an existing tree that it weaves well with.

In Figure 21 we plot the overhead as number of comparisons on the X-axis, versus the average relative error between the plan generated by *Incremental Weave Share* and the plan generated by offline *Weave Share* on the Y-axis, for different tolerance factor values (the points' labels). For

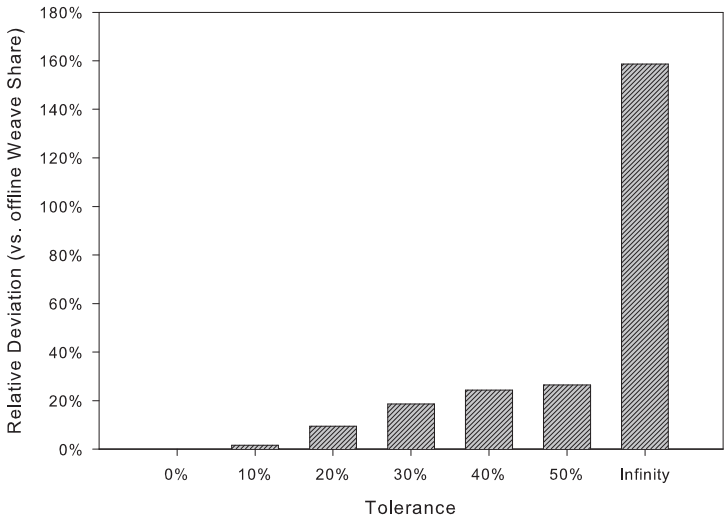


Figure 22: *Incremental Weave Share* - Deviation

instance, the point labeled as Infinity shows the online performance when no reconstruction is issued at all (tolerance = ∞). As expected, the Figure shows that as the tolerance factor increases, the relative error increases while the overhead decreases. It also shows that the relative error is always less then or equal to the tolerance factor. From the above results, we conclude that a tolerance factor of 20% or 30% achieves a good balance between performance and overhead.

In Figure 22 we plot the average deviation between the plan generated by *Incremental Weave Share* and the plan generated by offline*Weave Share*. The corresponding overhead of *Incremental Weave Share* is plotted in Figure 23. The last column in the two plots labeled as `Infinity` shows the online performance when no reconstruction is issued at all (tolerance = ∞). As expected, the figures show that as the tolerance factor increases, the relative error increases while the overhead decreases. It also shows that the relative error is always less then or equal to the tolerance factor. From the above results, we conclude that a tolerance factor of 20% or 30% achieves a good balance between performance and overhead.

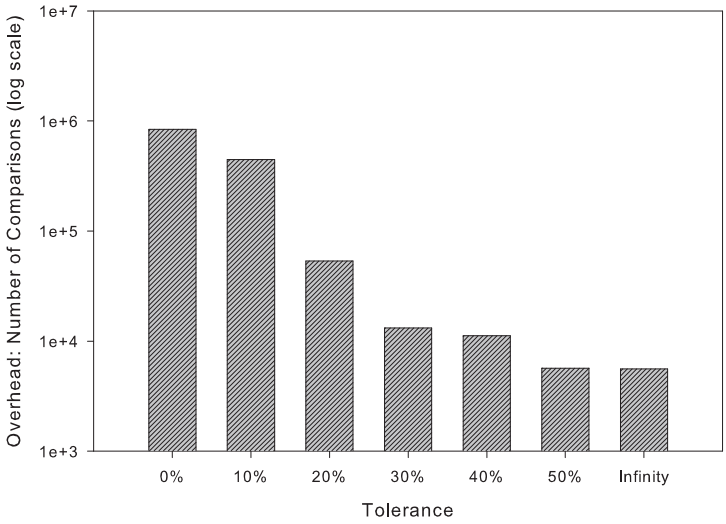


Figure 23: *Incremental Weave Share* - Overhead

4.7 SUMMARY

In this chapter, we proposed the *Incremental Weave Share* optimizer which handles the addition and deletion of ACQs by incrementally update the weaved plan. *Incremental Weave Share* utilizes a tolerance factor, which is a system parameter, to control how often a reconstruction from scratch phase is triggered. We experimentally demonstrated the performance of *Incremental Weave Share* and showed that a tolerance factor of 20% or 30% achieves a good balance between the quality of the weaved plan and the overhead.

5.0 TRIOPS: THREE-LEVEL PROCESSING MODEL

In this chapter, we propose a novel processing model for ACQs, called *TriOps*, with the goal of minimizing the repetition of operator execution at the sub-aggregation level. We also present *TriWeave*, a *TriOps*-aware multi-query optimizer built on the same principles as *Weave Share*. We analytically and experimentally demonstrate the performance gains of our proposed schemes which shows their superiority over alternative schemes. Finally, we generalize *TriWeave* to incorporate the classical subsumptions-based multi-query optimization techniques.

5.1 MOTIVATION

The ACQ processing model under the *Paired Window* technique is a two-level (i.e., two operators) query execution plan,as discussed in Chapter 2 (Section 2.2.2). The *Weave Share* optimizer [35, 34] adopted the two-level processing model, under which model, partitioning of ACQs into multiple execution trees requires duplicating the sub-aggregation operator across the different disjoint trees (i.e., one sub-aggregation operator for each tree). Naturally, *Weave Share* considers that duplicated cost in its optimization objective and tries to minimize the number of generated trees to minimize the overall cost.

While *Weave Share* tries to balance the tradeoff between sharing and no sharing, it still suffer another sharing tradeoff by using the tow-operators processing model. On one hand, fewer execution trees (i.e., sharing) means fewer sub-aggregation operators, which means less cost at the sub-aggregation level. On the other hand, more execution trees (i.e., no sharing) means smaller edge rate for each ACQ, which reduces the cost at the final-aggregation level.

As mentioned in Chapter 1, in order to fully reap the benefits of the new *Weave Share* multi-

query optimizer, a new underlying aggregate operator implementation is needed that minimizes or eliminates the effect of replication of sub-aggregation operators. This implementation should allow more flexibility in the data flow between the sub-aggregation and final-aggregation levels so that partial aggregate results are easily pipelined to different final-aggregate operators, or equivalently, to different trees of operators as in the case of *Weave Share*.

5.2 *TRIOPS* AND *TRIWEAVE*

5.2.1 *TriOps* Processing Model

TriOps is a new aggregate operator implementation that works in synergy with the new *Weave Share optimizer* to minimize the total cost of processing multiple ACQs. *TriOps* employs a three-level data processing model that minimizes the repetition of operations at the sub-aggregate level.

Consider first the case when similar ACQs have varying window specifications, but same predicates and same group-by attributes (the cases with different predicate and group-by attributes are discussed next in Sections 5.3 and 5.4, respectively). As with all *Partial Aggregation* based processing models, *TriOps* uses a sub-aggregation operator to aggregate input tuples once, generating a stream of fragments. In *TriOps*, a single sub-aggregation operator is shared among all ACQs. Instead of directly rolling up into the final-aggregation operators, however, *TriOps* introduces a new intermediate level of aggregation.

The *intercede-aggregation* operator is introduced to the query plan between sub- and final-aggregation levels. This new level of aggregation is made aware of the weaved plan and its ACQs partitions. In particular, it behaves for each group of ACQs that are shared in one execution tree (which we refer to, hereafter, as *partition group*) as its unshared sub-aggregation operators in the case of the two-operator model under *Weave Share*.

In this way, *TriOps* avoids the disadvantages of replicating the sub-aggregation operator for each *partition group* and the disadvantages of using a single sub-aggregation operator shared by all ACQs. By utilizing a single sub-aggregation, *TriOps* avoids processing input tuples multiple times, and by making the *intercede-aggregation* operator *partition group* aware, it avoids the increase in

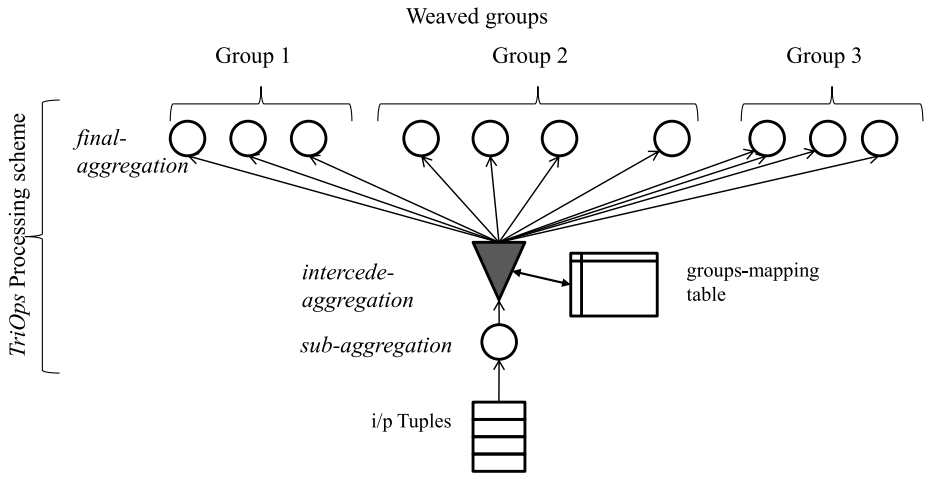


Figure 24: *TriOps* Shared Processing Scheme

the processing overhead, i.e., the number of aggregate operations, needed at the final-aggregation level.

The *intercede-aggregation* performs the following tasks.

1. It buffers all the fragments generated by the sub-aggregation for all partition groups and keep them until they are rolled up into all *partition groups* of ACQs that use them.
2. When an edge of a certain *partition group* is reached, *Intercede-aggregation* aggregate all relevant (smaller) fragments that together form the fragment that this group expects and pass it to the group's final-aggregation operators.

Being *partition group*-aware, *intercede-aggregation* achieves the last step by coalescing, for each group, the smaller fragments generated by the single shared sub-aggregation operator into the stream of fragments that this group would have seen if it had its own sub-aggregation operator. This is done only once for each group of ACQs, when a window edge is due for one of the ACQs in that group. Thus, each fragment is aggregated once per group, instead of once per window instance, as the case with the two-operator model. To illustrate the idea of *intercede-aggregation*, consider the following example.

Example 7. *Consider the first two ACQs of our running example, namely,* $q_a(8, 5)$ *and* $q_b(5, 4)$.

Let us assume that the weaved plan decides to not share execution. Thus, q_a *has the following sequence of edges timestamps:* $3, 5, 8, 10, 13, ...$ *On the other hand,* q_b *has edges at timestamps* $1, 4, 5, 8, 9, 12, ...$ *Under* TriOps*, the shared sub-aggregate operator would produce fragments with the timestamps sequence of* $1, 3, 4, 5, 8, 9, 10, 12, 13, ..$*, that is the union of the two sequences of edges. When the edge at timestamp* 3 *(of* q_a*) is reached, for instance, the* intercede-aggregation *will aggregate the fragments* 1 *and* 3 *to produce the fragment that* q_a *is expecting, and route this fragment to the input buffer of* q_a*. For another instance, when edge* 4 *(of* a_b*) is reached, the* intercede-aggregation *will aggregate fragments* 3 *and* 4 *to generate the fragment that* q_b *is expecting. This can be easily generalized to groups of ACQs where every edge belongs to a certain group, instead of a single ACQ, and the* intercede-aggregation *computes the fragment that this group expects to see.*

A weaved plan using the *TriOps* processing model is illustrated in Figure 24. The figure shows the introduced new level of aggregation, that is the *intercede-aggregation* operator. As illustrated in the figure, *intercede-aggregation* uses a group-mapping lookup table to generate the proper fragments for each group. This table is generated and maintained by the multiple ACQs optimizer (*Weave Share* in this case) as will be explained in Section 5.2.3.

5.2.2 *TriOps* Cost and Advantages

In this section, we analyze the cost function of a weaved plan using the *TriOps* processing model and discuss its advantages. Recall from Chapter 3, Section 3.2 that the total cost of a weaved plan that is consisted of m trees is computed as:

$$C_{\textit{m-trees, 2-operator}} = m\lambda + \sum_{i=1}^{m} E_i\Omega_i \tag{5.1}$$

Note that the first term of Equation 5.1 is the cost at the sub-aggregation level, whereas the second term is the cost at the final-aggregation level.

Given *TriOps* new processing scheme, however, the total cost of a weaved plan in Equation 5.1 changes to:

$$C_{\textit{m-trees, TriOps}} = \lambda + m.E + \sum_{i=1}^{m} E_i\Omega_i \tag{5.2}$$

where E represent the edge rate of the shared sub-aggregation, and E_i is the edge rate of fragments each *partition group* sees from the *intercede-aggregation* operator. The term $m.E$ represents the cost of the *intercede-aggregation*, where each fragment is aggregated once for each group.

Comparing the cost function of *TriOps* (Equation 5.2) to that of the two-operator model (Equation 5.1), the new processing model reduces the cost of *Partial Aggregation*, which is the cost of the sub-aggregation level in case of two-operator model, from $m\lambda$ to $\lambda + m.E$, which is the cost of the sub-aggregation plus that of the *intercede-aggregation* operators in case of the *TriOps*. Since the edge rate (E) is typically much smaller than λ, the *TriOps* scheme typically reduces the cost by a factor proportional to $\frac{E}{\lambda}$. The only exception is the hypothetical case when the input rate is one tuple per time unit and the sub-aggregation is generating one fragment per time unit (i.e., $E = \lambda = 1$). In this case, the cost of using two-operator model will be less expensive by exactly the value of the input rate λ ($= 1$ extra aggregations per time unit).

In addition to reducing the cost of the weaved plan, *TriOps* processing model offers several other performance advantages, namely, efficient adaptivity, smaller operator invocation overhead and less memory overhead.

Adaptivity to changes in the workload characteristics becomes more efficient, as mentioned earlier, because of the fixed physical query plan across the *partition groups*. If the input rate changes, for instance, the new plan might group the ACQs differently. Yet, the physical plan (i.e., the set of operators) will still utilize a single shared sub-aggregation operator, a single *intercede-aggregation* and the same set of final-aggregation operators, one for each ACQ. The only change in the plan is the group-mapping table. Further, the addition and deletion of ACQs becomes as simple as adding or dropping a final-aggregation operator, and updating the group-mapping table.

In terms of operator invocation overhead, *TriOps* replaces m sub-aggregation operators of paired-windows scheme, by exactly two operators; one shared sub-aggregation and one *intercede-aggregation*. The fewer number of operators means fewer context switching, which means less overhead. Finally, given that the *TriOps* processing scheme uses a single sub-aggregation operator, input tuples are buffered until they are consumed only once, as opposed to be buffered until they are consumed m times, once per *partition group*, as in the *Paired Window* case. While the *intercede-aggregation* requires extra buffering of the fragments, the savings from shorter buffering of the input tuples surpasses this overhead. Specifically, instead of buffering λ tuples/second until they

are consumed by all m sub-aggregate operators, the λ tuples per time unit are buffered until they are consumed once, and E fragments per time unit are buffered until they are consumed by the m groups.

5.2.3 *TriWeave* Optimizer

The fact that our new processing model reduces the cost of *Partial Aggregation* suggests that a selective grouping of ACQs based on *TriOps*'s cost model would result to more partition groups and lead to better performance. This led us to develop *TriWeave*, which is a new *TriOps*-aware multiple ACQs optimizer.

The *TriWeave* optimizer works similar to the *Weave Share* optimizer, trying to selectively weave together in shared *partition groups* the ACQs that weave well. That is, to group ACQs in a way that minimizes the total plan cost as per Equation 5.2. The steps of the *TriWeave* optimizer are shown in Algorithm 3 and can be summarized as follows.

- Initialize the plan by creating a group for each ACQ, i.e., no sharing at all.
- While beneficial, i.e., reducing the total cost of the plan, find the pair of groups that yields the maximum reduction in the plan cost when shared.
- Merge the pair of groups found in the previous step and update the plan.
- When no such pair of groups is found, generate the group-mappings table and return the current plan as the *TriWeave* Plan.

Notice that upon changes of the workload, such as addition or deletion of ACQs or major changes in the input rate, *TriWeave* needs to regenerate the group-mapping table to replace the current one.

We experimentally demonstrate the performance gains of *TriWeave* in Section 5.6. The results confirm our hypothesis that *TriWeave* generates better quality weaved plans with more partition groups compared to *Weave Share*.

Algorithm 3 The *TriWeave* Algorithm

Input: A set of n **ACQs**
Output: *TriWeave* query plan P
Begin
$P \leftarrow$ *Create an execution tree for each ACQ*
$l \leftarrow n$ {current number of trees}
(*max-reduction*, t_1, t_2) $\leftarrow (0, -, -)$ {current tree-pair to merge}
repeat
  for $i = 0$ *to* $l - 1$ **do**
    for $j = i + 1$ *to* l **do**
      $temp \leftarrow$ cost-reduction-if-merging(t_i, t_j)
      if $temp >$ *max-reduction* **then**
        $(max - reduction, t_1, t_2) \leftarrow (temp, t_i, t_j)$
      end if
    end for
  end for
  if *max-reduction* > 0 **then**
    merge(t_1,t_2)
    $l \leftarrow l - 1$
  end if
until No merge is done
group-mapping $\leftarrow$ Generate-Mapping-Table(P)
Return P
End

5.3 *TRIOPS*: WINDOWS AND PREDICATES

In this section, we study the case when ACQs have varying window specifications as well as different predicates. We first discuss the drawbacks of the adopting *Shared Data Shards* (SDS) technique that handles the case when ACQs have different predicates in Section 5.3.1. We provide the details

of the Inverted Predicate-signature (*IPS*) structure, which is how *TriOps* efficiently adopts *SDS* to process ACQs with different predicates and varying window specifications, in Section 5.3.2.

5.3.1 Drawbacks of Integrating Shared Data Shards Technique with *Weave Share*

As discussed in details in Section 2.2.3.2, the *Shared Data Shards* (SDS) [45] technique was proposed to handle the case when ACQs have the same window specifications but different predicates. The assumption is that complex predicates over the same data stream may overlap. *SDS* can be integrated with the *Weave Share* to handle ACQs with different window specifications and different predicates, as discussed in Chapter 3, Section 3.5.3. This integration is achieved by introducing an operator before the sub-aggregation operator that pre-processes the input tuples, augmenting them with a signature that determines which predicates this tuple satisfies (Figure 9). Then, in the sub-aggregation operator, each set of tuples with the same signature are aggregated together producing shards of fragments. Finally, each shard is routed to the proper final aggregate operator to produce the results.

There are two drawbacks of the *SDS* scheme that the *TriOps* processing model addresses. The first drawback is the transient memory overhead involved in replicating the fine grained fragments in the input buffer. That is, given a set of l predicates, a signature of length l is augmented to each tuple, yielding 2^l different possible signatures. This means that each fragment is split into possibly 2^l fragment-signature pairs. Replicating these fragments in the input buffers of each and every ACQ, exponentially increases the memory overhead.

Directly related to this issue is the the second drawback, which is the increase in the processing overhead. That is, the final aggregation operator of each ACQ needs to perform 2^l aggregations per fragment, for every window instance. *TriOps* overcomes these two drawbacks through the *intercede-aggregation* level and by fusing the tuple-augmentation with the sub-aggregation level as we discuss next. Another drawback of *SDS*, when utilized by *Weave Share*, is the need to store tuple signatures due to the multiple sub-aggregation operators. Under the *TriOps* model, there is never a need to store the signatures, since it uses a single sub-aggregation operator.

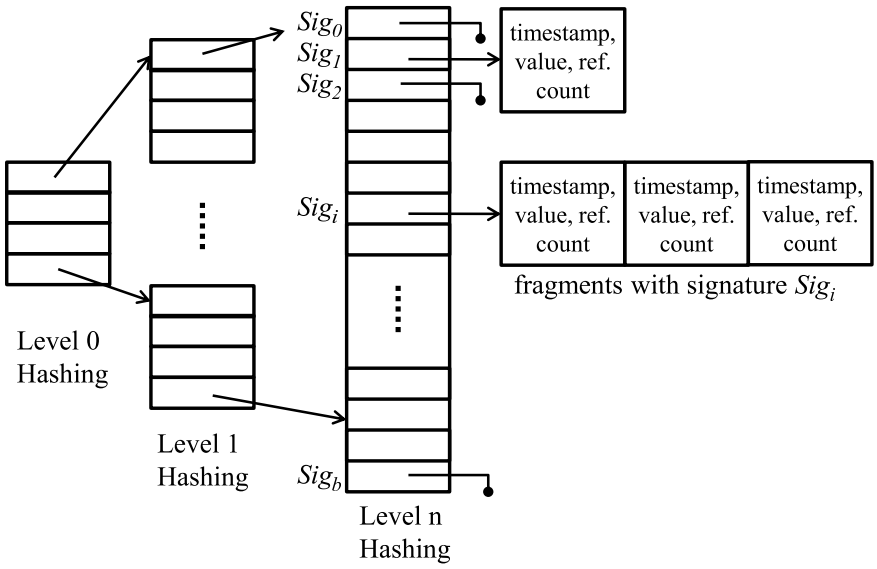


Figure 25: Inverted Predicate Signatures Structure

5.3.2 *TriOps*: Handling Different Predicates

TriOps efficiently adopts the *SDS* scheme to process ACQs with different predicates as well as varying window specifications. To do so, *TriOps* first fuses the tuple-augmentation with the sub-aggregation phase. The goal of this merge of tasks is to remedy the need to store the signature of each tuple, or fragment. Further, it utilizes an inverted-predicate signatures (*IPS*) index, which is essentially a multi-level hash-based shared buffer between the sub-aggregation and the *intercede-aggregation* operators. The sub-aggregation operator uses *IPS* to aggregate the different fragments. Each entry of *IPS* is a list of fragments that have the same signature of that entry, for the different timestamps, i.e., different partitions of the input data. Thus, the signatures need not be augmented to the fragments, nor to the input tuples, but are instead embedded in the *IPS* structure.

Figure 25 illustrates the *IPS* data structure using multi-level hashing. Every node in the linked lists is a fragment of a certain edge, plus a reference count which indicates how many groups shall read this fragment, so that once the reference counter drops to zero, the fragment is discarded. Given this structure, the group-mapping table becomes a lookup table, where for each group, a set of fixed pointers to entries in *IPS* indicate the set of fragments that satisfies this group's predicates.

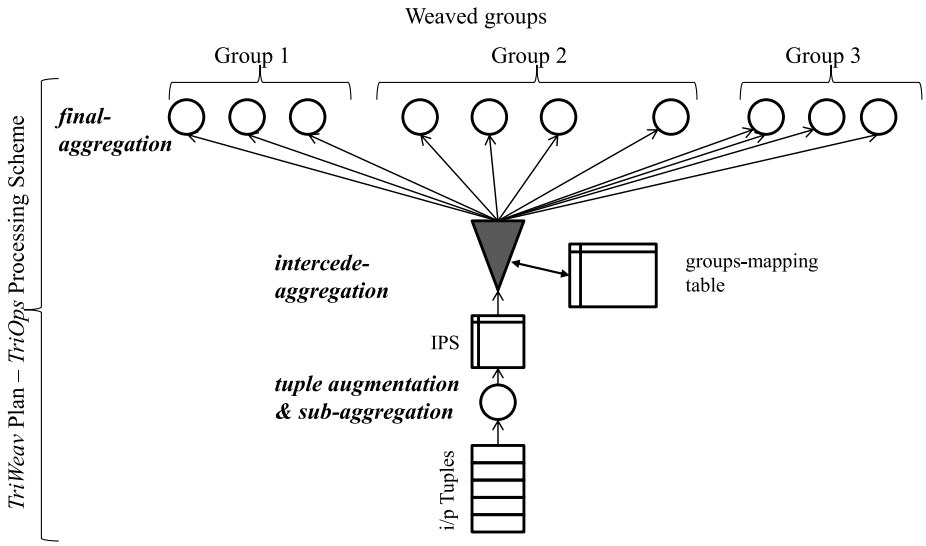


Figure 26: *TriOps* - Windows and Predicates

The second optimization that the *TriOps* model offers is the reduction of the memory overhead. Specifically, the fragment-signature pairs are no longer replicated in the input buffers of each ACQ. Rather, they are maintained in the *IPS* index until the *intercede-aggregation* operator aggregates them and pushes them to the shared buffer of each *partition group* of ACQs, observing the relevance to groups as encoded in the implicit signatures.

Figure 26 shows the *TriWeave* plan using the *TriOps* model for handling different window specifications and predicates. Given such plan, the execution proceeds as follows:

1. The sub-aggregation operator processes each input tuple and incrementally evaluates all the predicates (e.g., using predicate indexes and group filters [49]) for this tuple. The results of these predicate evaluations are used to locate the entry in the *IPS* index to aggregate the tuple.
2. The group-mapping table is modified by adding, for each group, a list of pointers to *IPS* entries that represent the set of fragments that belong to this group, i.e., satisfies the predicates of at least one ACQ in this group. When an edge is due for a certain group, the *intercede-aggregation* looks up the group-mapping table to directly collect the different fragments that belong to this group, aggregates them and produces the fragments of this group. This is illustrated in the example below.

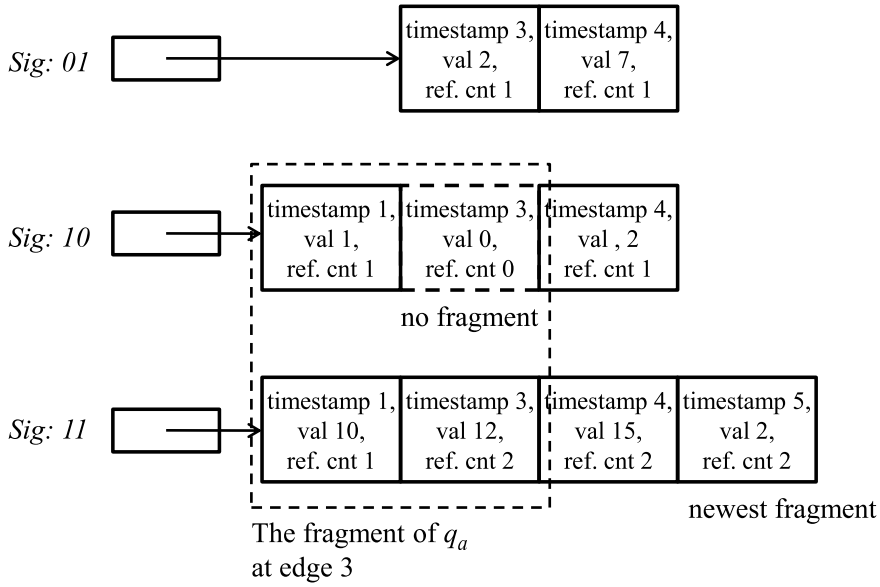


Figure 27: Fragment-signature pairs that belong to the same fragment

3. Finally, each final-aggregation operator aggregates the augmented-fragments that satisfy its predicate to generate the final results.

We further illustrate these steps with the following example.

Example 8. *Consider ACQs q_a and q_b of our running example, where the predicate c_a is different from predicate c_b. In this case, the signature has two bits, and there are three possible signature values:* $01, 10$ *and* 11. *Figure 27 shows a snapshot of the* IPS *for these two ACQs. The figure shows the fragments that together constitute the fragment due at edge 3 for q_a, assuming that the most significant bit in the signatures represents predicate c_a. Thus, the* intercede-aggregation *will aggregate these fragments and push them to the input buffer of q_a.*

Figure 27 also shows interesting possible scenarios. For the 01 signature entry, the fragment at edge 1 was already consumed by q_b and was therefore deleted. Also, the fragment at edge 3 in the row of 10 signature does not exist, because no tuples with this signature were inserted during this fragment time span. Finally, in the 11 signature row, the Figure shows that some tuples arrived with this signature and were aggregated to form a new fragment for edge 5.

To efficiently handle addition of new ACQs, the new predicate is represented by adding a

most significant bit in the signature. Thus, all previous hash entries remain valid and new entries are hashed properly. Deletion of ACQs needs a careful handling. If the deleted ACQ results in deleting the predicate represented by the most significant bit, then the *IPS* table can be reduced to half, or the top most level hashing if multi-level hashing is utilized. If that is not the case, then a new *IPS* with half the size is instantiated for new entries. The two *IPS* work simultaneously until no entries exist in the old IPS at which point it is to be discarded. It worth mentioning that if the two-levels processing model adopts the *IPS* technique, it mainly converts into the *TriOps* model.

Finally, it worth mentining that *IPS* can be utilized by the two-level processing scheme, which will essentially convert it into a three-levels processing scheme.

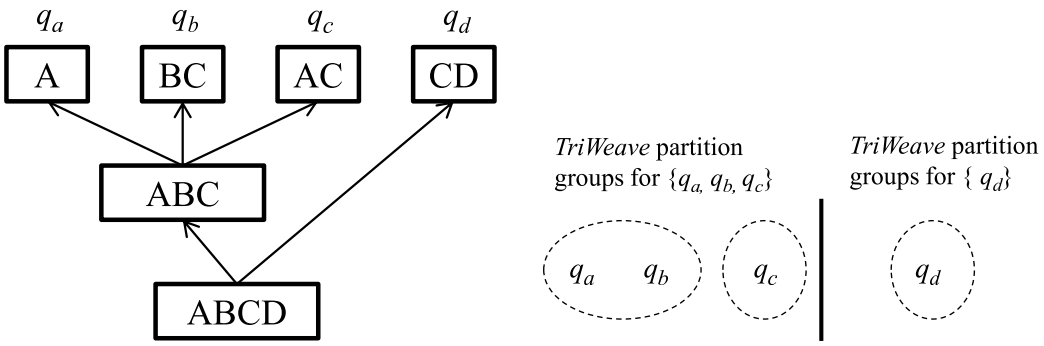
5.4 *TRIOPS*: WINDOWS, PREDICATES AND GROUP-BY

In this section, we demonstrate how *TriOps* can efficiently optimize the processing of multiple ACQs with varying window specifications, predicates and group-by attributes. We first consider the case when all ACQs have the same predicate, but varying window specifications and different group-by attributes in Section 5.4.1. Then we consider the general case when window specifications, predicates and group-by attributes are all different in Section 5.4.2.

5.4.1 Windows and Group-by

In order to optimize the shared processing of multiple ACQs that have varying window specifications, as well as different group-by attributes, we utilize the *intermediate-aggregates* optimizer [59]. As discussed in details in Chapter 2, *Intermediate-aggregates* handles the case of same window specifications, same predicates but different group-by attributes. We utilize it in the following manner.

1. We apply the *intermediate-aggregates* optimizer as if all windows are identical to generate the group-by tree for this case.
2. Given the group-by tree, each first level node (i.e., a node that is a child of the root) represents a set of ACQs that can share their processing, given their different group-by attributes, but



(a) The group-by tree

(b) *TriWeave* Weaving

Figure 28: An Instance of Four ACQs

assuming same windows. We then apply *TriWeave* for each of these sets, independently to generate their *partition groups*.

3. Finally, we integrate the ACQs *partition groups* with the group-by tree using the *TriOps* processing model.

The last step is achieved by replacing each first level node in the group-by tree with a *intercede-aggregation* that is aware of the *partition groups*, and also performs a group-by aggregation using the set of attributes of that group-by tree node. To illustrate these steps, consider our running example (Figures 28 and 29).

Example 9. *Assume we have four ACQs* q_a, q_b, q_c *and* a_d *with windows' specifications:* $(8, 5)$, $(5, 4)$, $(10, 1)$ *and* $(5, 4)$, *respectively. Assume also that the ACQs has group-by attributes:* A, BC, AC *and* CD, *respectively. We first generate the group-by tree for the four ACQs, using the* intermediate-aggregate *scheme, which is shown in Figure 28(a). The label of each node represent the set of group-by attributes used by this node. That is, each node represent an aggregation operator that performs a group-by aggregation using this set of attributes. The group-by tree in this case has one internal node labeled* ABC. *Thus, the set of leaf nodes (i.e., ACQs) of the sub-tree for which* ABC *is the root, represent a set of ACQs that share their processing given their different group-by attributes, but assuming they have the same window specifications. Specifically,*

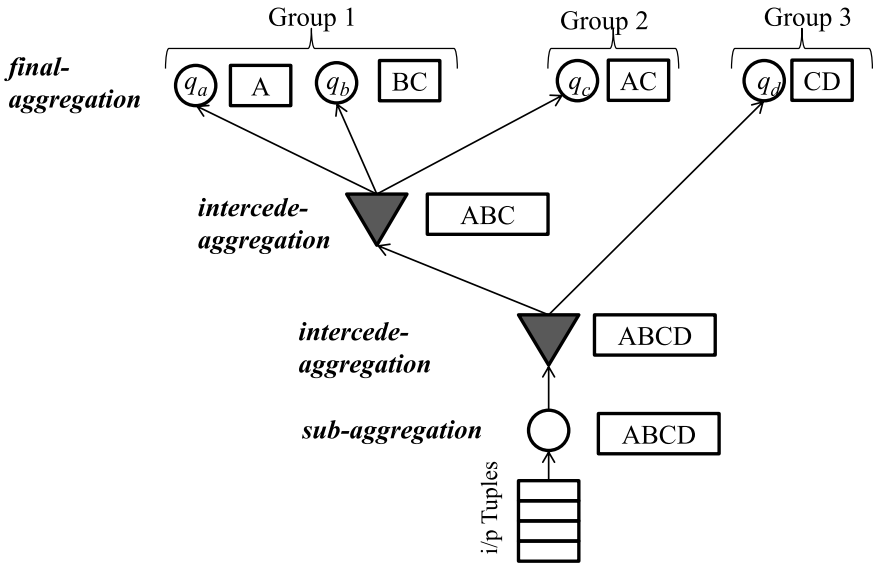


Figure 29: Integrating *TriWeave* Plan with Intermediate-aggregates Tree

q_a, q_b *and* q_c *are shared together, while* q_c *is shared with the sub-aggregation of the other three ACQs. Thus, we have two set of ACQs, set* $T_1 = \{q_a, q_b, a_c\}$ *and set* $T_2 = \{q_d\}$*. We proceed by generating the weaved plan for each set, i.e., apply* TriWeave *on* T_1 *then* T_2*. Figure 28(b) shows the output of this step which weaves* T_1 *into two groups, one that shares* q_a *and* a_b*, while the other has* q_c *by itself. The weaved plan of* T_2 *is trivial as it has one ACQ. The last step is to integrate the results of the first two steps together into a* TriOps *plan. Figure 29 shows the integrated* TriOps *plan. Simply, the root of group-by tree is mapped to the sub-aggregation operator, while each internal node is mapped into a* intercede-aggregation *operator, whenever possible.*

The rational behind this procedure is to follow a conservative approach towards sharing. Specifically, two ACQs are shared only if they are shared under both the group-by tree and the weaved plan. For instance, while q_b and q_d have identical window specification, they were not shared in the group-by tree, so we do not share them. Notice that the group-by tree might has multi-levels of nodes, for further processing optimization. For example, in Example 9 above, q_a and q_c could have a common parent node labeled AC, which performs a group-by aggregation using the attributes set AC and is a child of the node ABC. Mapping such internal nodes to the *TriOps* plan depends

on the weave plan of the set of ACQs of this internal node, following the same conservative rational. Specifically, if the weaved plan shares the ACQs of the internal node, then this internal node is mapped into another *intercede-aggregation* operator. Otherwise, it is just dropped from the integrated plan.

Another remark on the integration process is that in case the group-by tree has the input stream as its root, we still use the first level nodes to determine the sets of ACQs to be weaved separately. Finally, it worth mentioning that if the *Paired Window* scheme is utilized, each node except leaf ones, can be mapped to a sub-aggregation, and the leaf nodes mapped to final-aggregation. This however does not allow any optimization for the varying window specifications. Using the *TriOps* processing scheme, specifically the *intercede-aggregation* allows such optimization being weaving-aware. Further, it enables efficient handling of predicates using the *IPS* as we discuss in the following Section.

5.4.2 Windows, Predicates and Group-by

In order to optimize the processing of multiple ACQs with varying windows specifications, predicates and group-by attributes, we first follow the same integration procedure discussed in Section 5.4.1 above, which optimizes the plan for varying window specifications and different group-by attributes. Then, to support different predicates, we augment the plan with *IPS* structures before each and every *intercede-aggregation*. Figure 30 shows such augmented plan for the ACQs of Example 9.

5.5 GENERALIZED *TRIWEAVE* OPTIMIZER

In this section we put everything together into the generalized *TriWeave*, the weaveability based optimizer that optimizes the plans to process ACQs with varying window specifications, different predicates and different group-by attributes. Generalized *TriWeave* follows the steps discussed in Section 5.4 to handle different group-bys. In Section 5.5.1 we discuss the impact of predicates on the optimization process and then present the generalized *TriWeave* optimizer in Section 5.5.2.

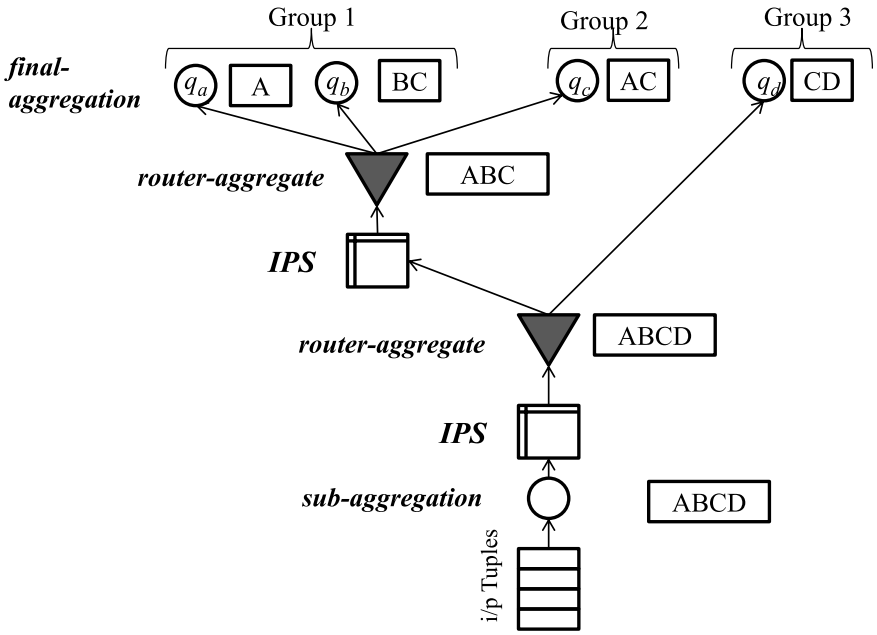


Figure 30: *TriWeave* Plan - Varying Windows, Predicates, and Group-by

5.5.1 Impact of Predicates on Weaving

Assume there are n ACQs which are weaved into m groups, and each group has n_i ACQs, s.t. $\sum_i n_i = n$. If each ACQ has its unique predicate, then the more groups (i.e., large m) the more buffering (i.e., memory overhead) and aggregate operations needed by the *intercede-aggregation* operator. The number of unique signature fragments per edge for group i is $1 \leq p_i \leq 2^{n_i}$. Using the *Paired Window* processing scheme, this leads to an increment of p_i aggregations at the final-aggregation level, per fragment for every edge. However, using the *TriOps* processing scheme, this increment on cost is only reflected at the *intercede-aggregation* operator, which means it is to be done once per group. Specifically, for each group, each fragment is computed by aggregating the fragment-signature pairs that satisfy its predicates only once, and routed to the proper final-aggregation operators.

Clearly, if two ACQs have two predicates that are disjoint, or are identical, then they are best shared since this minimizes the number of fragment-signature pairs per fragment. On the other hand, if the two predicates overlap, this leads to the maximum increase in the cost, depending

on data distribution, which is not known a priori. If a predicate is contained in another, this will be the average case of in terms of number of fragment-signature pairs per fragment. Finally, if the predicates are orthogonal, i.e., defined on different attributes, it is even harder to estimate the expected number of fragment-signature pairs per fragment. Now, consider the following cases:

1. Given two ACQs that are not are not weaveable (i.e., should not be shared given their window specifications), the relationship between their predicates does not impact their weaveability. In particular, if the two predicates are disjoint or identical (i.e., least number of fragment-signature pairs), it is still not beneficial to group these two ACQs together, given that they are not weaveable. Because their sharing leads to an increase in the cost without any gain.
2. Given two ACQs that are perfectly weaveable (e.g., they have the same window specifications), the predicates relationship might impact the sharing decision. If the predicates are disjoint or identical (i.e., best to share) and the two ACQs are already shared, then nothing changes. On the other hand, if the two ACQs has overlapping predicates, it might be beneficial to not share them, to reduce the number of fragment-signature pairs to be aggregated per fragment. This however, depends on the *edge rate* (the higher the edge rate, the more beneficial it is to not share them) and the number of fragment-signature pairs per fragment, which depends on the data distribution and the predicates' constants.

However, given that it is not feasible to have accurate estimates of the data distribution for data streams, where data characteristics change over time, and given that it is expensive to analyze predicate containment, especially since ACQs are added and deleted over time, and given that using the *TriOps* processing scheme, the cost overhead is at the intermediate level, it is more better to ignore the impact of predicates on the sharing decision, for the sake of efficiency.

5.5.2 The Algorithm

The steps of the generalized *TriWeave* are shown in Algorithm 4 and summarized as follows:

- Generate the group-by tree.
- Generate the *partition groups* for each set of ACQs that are represented as a root-child node in the tree. The output of this step is the group-mapping table.

- Integrate the group-by tree and the weaved plan.
- Augment the plan with *IPS* structures to generate the final *TriWeave* plan that utilizes *TriOps*.

Notice that upon changes of the workload that lead to changes in the weaved plan only, such as addition or deletion of ACQs with no new group-by attributes, or major change in the input rate, only the group-mapping table that is used by the *intercede-aggregation* needs to be updated. However, if the group-by attributes are modified, then the whole *TriWeave* need to be re-applied. If the updated plan can be applied using an existing plan by changing the group-mapping table, then the new plan can be deployed immediately. Otherwise, a more careful plan switching needs to be done.

It worth mentioning that we chose to first generate the *intermediate-aggregates* tree, then use it to determine partitions of ACQs to be used as input to *TriWeave* since it sounds more natural, and efficient mapping to a *TriOps* plan. It is interesting however to examine the perforamnce and overhead of the reverse ordering, i.e., apply *TriWeave* first, then use group-paritions as input for *intermediate=aggregates*. We leave that to our future work.

5.6 EVALUATION

Using the simulation platform introduced in Section 2.3 we evaluated the performance of *TriOps* processing scheme and *TriWeave* optimizer. We briefly highlight the additions in Section 5.6.1, and discuss the experimental results in Section 5.6.2.

5.6.1 Experimental Platform

We used the simulation platform discussed in Section 2.3 to evaluate *TriOps* and *TriWeave*. Below we highlight the specific additions used here.

ACQs: In these experiments, we used overlap factor (ω_i) with a maximum value $\Omega_{max} = 50$. Recall that $r_i = s_i \times \omega_i$.

Performance Metrics: We measured the quality of *TriWeave* plans in terms of their cost computed as the number of aggregate operations per second (which also indicates the throughput). We chose

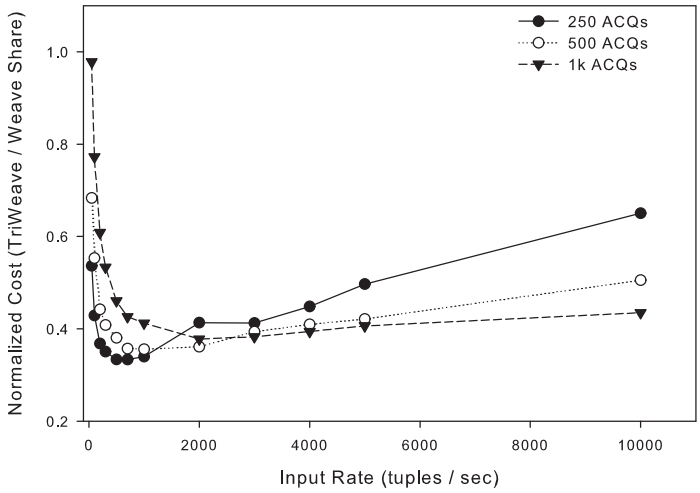


Figure 31: *TriWeave* performance gain - Impact of Input Rate

this metric because it provides an accurate and fair measure of the performance, regardless of the platform used to conduct the experiments. To quantify the performance gains of *TriOps*, we compare different weaved plans, each using *TriOps* as the underlying processing scheme versus using paired-windows.

Algorithms: We used *Weave Share* [35] and *Shared* [45] as the base case algorithms for our comparisons. Recall that *Shared* is the optimizer that shares the sub-aggregation operator among all ACQs. That is, the weaved plan has exactly one group. *Weave Share* is the optimizer that assumes the two-level processing model in selectively partitioning the ACQs into one or more partition groups based on their weaveability. We also tried different combinations of optimizers and processing models. For instance, we generated *Weave Share* and *Shared* plans, assuming the two-level model, but then ran the plan using the *TriOps* model. The goal is to get better insight and understanding of the behavior of *TriWeave* and *TriOps*.

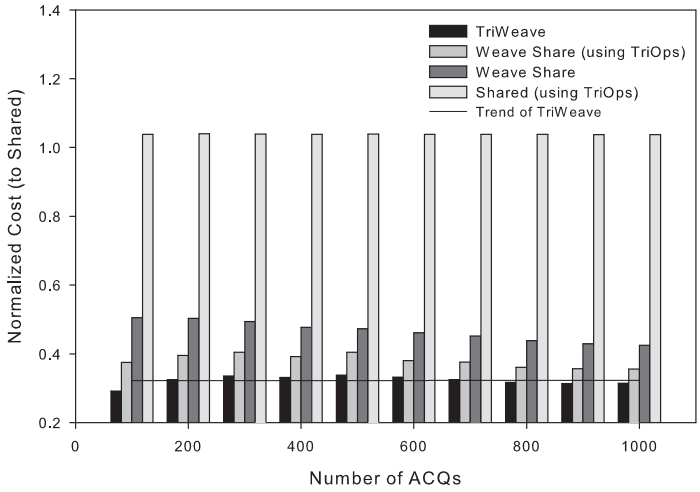


Figure 32: Using *TriOps* processing for different plans (50 tuples/sec)

5.6.2 *TriOps* Performance

In the first set of experiments, we measured the performance gains of *TriOps*-varying windows. Specifically, we compare the quality of the weave-plan using *TriOps* to that using Data Slices. In Figure 31 we plot the normalized cost (to reflect the gains) of weaved-plans using *TriOps* compared to that using Data Slices, as the input rate increases, for different numbers of ACQs. The figure shows that for the low input rates, the edge rate is the dominating factor of the cost. This is revealed by the small improvement over *Weave Share* (less than 40%). Figure 31 also shows that at low input rate, the gain reduces as the number of ACQs increases.

Interestingly, at very low input rates, the role of the number of ACQs is reverted. That is, the more ACQs, the less the performance gains of *TriOps*. This is because when the input rate is very low, the cost at the final-aggregation becomes the dominating factor of the total cost. Thus, the more ACQs, the more overlapping operations needed and the less the gains *TriOps* achieves.

In Figure 33 we show the performance gains of *TriWeave* compared to *Weave Share*. We see that *TriWeave* achieved a further 63% improvement over the *Weave Share* optimizer. We observe the same phenomena of reverting the impact of ACQs for different input rates.

In the next set of experiments, we take a further look into the performance of *TriOps*. Specif-

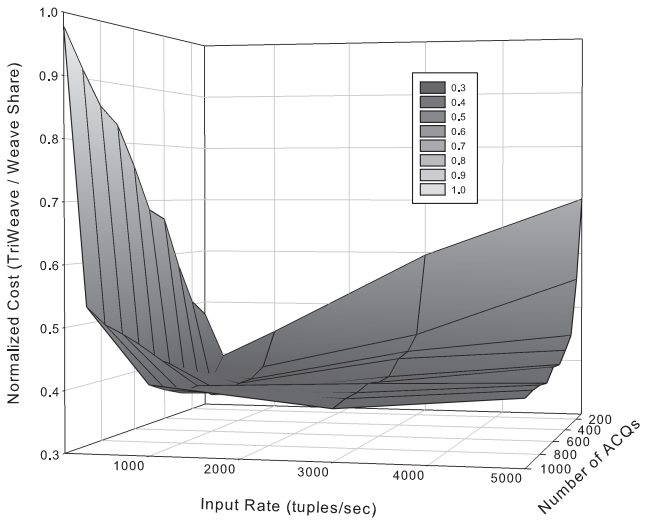


Figure 33: *TriWeave* - Impact of Input Rate and No. of ACQs

ically, we generated query plans using different optimizers (namely, *Weave Share*, *Shared*, and *TriWeave*), then we measured the cost of each plan when using the Data Slices vs when using the *TriOps* scheme and report the normalized cost of different alternatives, normalized to the cost of the Shared plan using Data Slices.

Figures 32, 34 and 35 show the performance gains of the *TriOps* processing scheme for low (50), medium (300) and high (10K tuples/second) input rates, respectively. We plot the normalized cost as the number of ACQs increases. We also highlight the trend of the *TriWeave* plan in each plot. All three Figures show that for each plan, utilizing *TriOps* achieves gains over Data Slices, except for the *Shared* case. The reason is simply because when there is only one group, the *intercede-aggregation* adds an overhead with no benefit. On the other hand, when there are at least two groups, utilizing *TriOps* achieves between 40 and 60% gain.

All three Figures also show that the gain of *TriOps* increases as the number of ACQs increases. This is mainly due to the fact that the more ACQs, the more chances for selective sharing. While Data Slices will be less aggressive to generate more groups (due to its cost function, i.e, the sharing trade-off), *TriOps* on the other hand is able to take full advantage of such opportunity. We confirmed that by checking the number of groups each scheme produces, and we found that *Tri-*

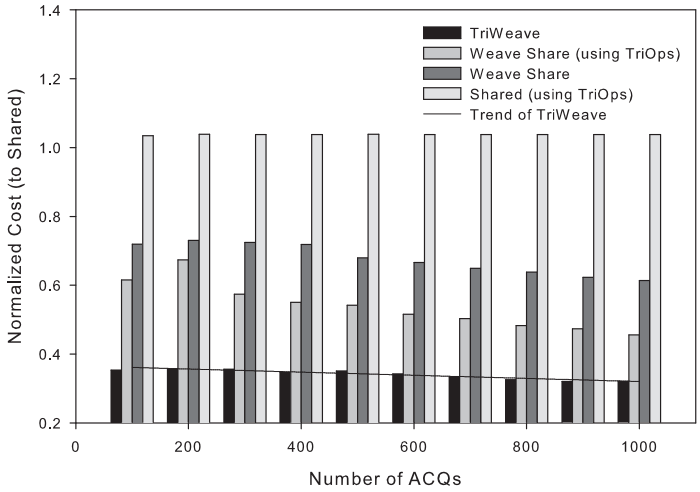


Figure 34: Using *TriOps* processing for different plans (300 tuples/sec)

Ops consistently generate plans with much larger number of groups. This is also seen in Figure 35, when *Weave Share* generates one shared group, while *TriOps* generates multiple groups and achieves up to 40% gain.

Finally, the rate with which the gain of *TriOps* increases as the number of ACQs increases is faster for higher input rates. The reason is clear by comparing the cost functions of Data Slices (Equation 5.1) and that of *TriOps* (Equation 5.2). Simply, that the higher input rate, *TriOps* achieves larger reduction by replacing the multiple sub-aggregation operators by one sub-aggregation and one *intercede-aggregation*.

5.7 SUMMARY

In this chapter, we questioned the effectiveness of the widely accepted two-level or two-operator implementation of aggregate continuous queries (ACQs) and proposed a new three-level processing model, called *TriOps*, in the context of Weaved Plans which selectively group ACQs into multiple query execution trees (partition groups). We illustrated that the proposed *intercede-aggregation*

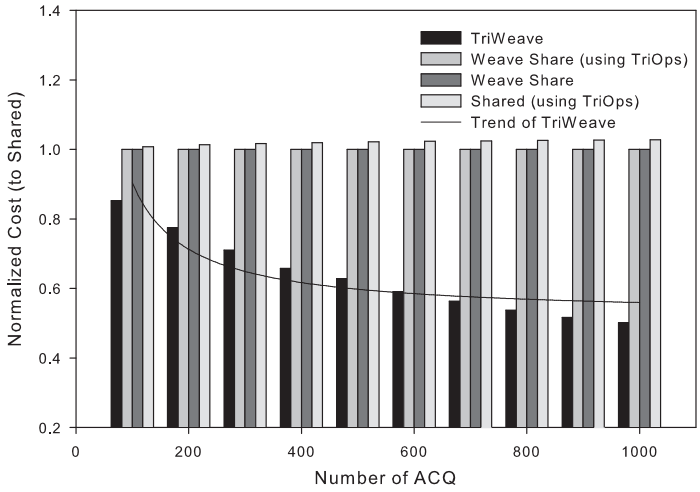


Figure 35: Using *TriOps* processing for different plans (10K tuples/sec)

operator in *TriOps* minimizes the total cost of processing multiple ACQs by allowing sharing of the sub-aggregation across all partition groups and performing partial final-aggregation shared by the ACQs of a given partition group. Further, we illustrated how *intercede-aggregation* operator can efficiently support the processing of multiple ACQs with different predicates and group-by attributes in addition to varying window specifications. Finally, we developed *TriWeave*, a *TriOps*-aware multiple query optimizer along the lines of Weave Share optimizer and generalized *TriWeave* to integrate the classical subsumption-based multi-query optimization techniques. We evaluated the effectiveness of *TriOps* and the quality of the query plans produced by *TriWeave* using simulation. Our experimental results demonstrated the performance gains and superiority of our proposed schemes to other alternatives.

Algorithm 4 Generalized *TriWeave* Optimizer

1: **Input: A set of** n **ACQs**
2: **Output: *TriWeave* query plan** P
3: **Begin**
4: $T \leftarrow$ Generate the group-by tree
5: $WP \leftarrow \emptyset$ {weaved plan}
6: **for** For every node t_i that is child of $root(T)$ **do**
7: $C_i \leftarrow ACQs(t_i)$
8: $P_i \leftarrow$ *Create an execution tree for each ACQ* {Initialize to no sharing plan}
9: $l \leftarrow n_i$ {current number of partition groups}
10: ($max\text{-}reduction, t_1, t_2$) $\leftarrow (0, -, -)$ {current tree-pair to merge}
11: **repeat**
12: **for** $i = 0$ *to* $l - 1$ **do**
13: **for** $j = i + 1$ *to* l **do**
14: $temp \leftarrow$ cost-reduction-if-merging(t_i, t_j)
15: **if** $temp >$ *max-reduction* **then**
16: $(max - reduction, t_1, t_2) \leftarrow (temp, t_i, t_j)$
17: **end if**
18: **end for**
19: **end for**
20: **if** *max-reduction* > 0 **then**
21: merge(t_1,t_2)
22: $l \leftarrow l - 1$
23: **end if**
24: **until** No merge is done
25: $WP \leftarrow P_i \cup WP$
26: **end for**
27: $P \leftarrow$ integrate(T, WP)
28: $P \leftarrow$ augment_IPS(P)
29: Return P
30: **End**

6.0 AQSIOS 3.0: REALIZATION OF WEAVE SHARE

In this chapter we describe the challenges and the final design decisions in realizing our proposed weave-based algorithms to optimize the shared processing of ACQs in a real system. Specifically, we implement *Weave Share* in AQSIOS 3.0 [6]. We first overview the AQSIOS prototype in Section 6.1. Then, we describe the challenges involved in implementing the *Paired-window* processing scheme as well as the *Weave Share* optimizer in AQSIOS and how we address them in Section 6.2. We finally provide performance results for our implementations in Section 6.3.

6.1 THE AQSIOS DSMS PROTOTYPE

Advanced Query System Infrastructure On Streams (AQSIOS) [6] is a prototype DSMS developed by the ADMT Lab (Advanced Data Management and Technology Laboratory) of the University of Pittsburgh. When AQSIOS was first developed, it was an effort to prototype the new generation DSMS, whose design had equal emphasis on optimizing performance and enhancing functionality. The goal was that these new generation DSMSs will simplify the development of a wide range of monitoring applications, with diverse requirements.

The AQSIOS project reexamined all four critical components of a DSMS, namely: the *Query Scheduler*, the *Load Shedder*, the *Query processor*, and the *Data Dissemination* modules. The two key innovations of this project are:

1. it formalizes QoS/QoD metrics for DSMSs and develops algorithms designed to optimize these metrics.
2. it looks at how the four DSMS modules mentioned above, i.e., *Query Scheduler*, the *Load*

Shedder, the *Query processor* and the *Data Dissemination*, can be integrated to work in synergy, instead of making isolated decisions that may have a significant negative impact on the overall performance; and

3. its plans included the analytical and experimental evaluation of the proposed algorithms and also the implementation and evaluation of a prototype system.

AQSIOS 1.0 prototype is implemented starting from STREAM 0.6.0 code. AQSIOS 1.0 implements *Highest Rate* (HR) [71], a priority-based scheduler, in addition to the basic *Round Robin* (RR) scheduler. *HR* scheduler prioritizes operators based on the their output rate, and executes the operator with the highest priority in order to minimize the response time. Furthermore, AQSIOS 1.0 implements a simple yet complete load manager that monitors the system workload at run time and automatically decides the appropriate amount of random shedding from the input data when the system is overloaded.

The second release, AQSIOS 2.0, incorporates the second version of the *ALoMa* [64] load manager and supports priority-classes scheduling incorporating the Continuous Query Class scheduler (*CQC*) [53]. *CQC* is a two-level scheduler which combines weighted *RR* and *HR* to effectively handle different ranks of CQ classes. In addition, AQSIOS 2.0 runs *DILoS* [65], a complete synergy between *ALoMa* and *CQC*. *DILoS* is an integrated approach that exploits the synergy between scheduling (*CQC*) and load shedding (*ALoMa*) to effectively handle different ranks of CQ classes, each associated with different QoS and QoD requirements, in a a multi-tenant environment. Users of AQSIOS 2.0 can specify which class a query belongs to and the priorities of the classes, which will be honored by *DILoS*.

The new release, AQSIOS 3.0 contains our implementation of the *Weave Share* query optimizer. We describe the challenges we faced to realize *Weave Share* next.

6.2 CHALLENGES

AQSIOS inherited the basic query processing and optimizer from STREAMS code, which was mostly based on traditional DBMS techniques. As such, in order to realize *Weave Share* in AQSIOS, we need to first implement the *Paired Window* processing scheme (Section 2.2.2). This, in

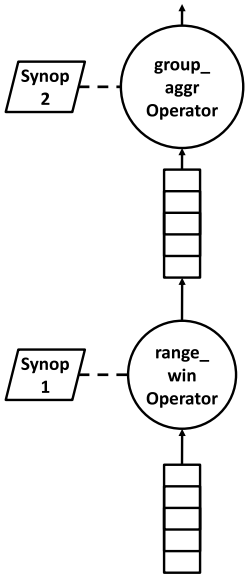


Figure 36: Current ACQ query plan

turn, involved the full implementation of sliding windows. Another challenge was the modification of the current optimizer to recognize similar ACQs and invoke the *Weave Share* optimizer to generate their execution plan.

To better illustrate the first challenge, Figure 36 shows how an Aggregate Continuous Query (ACQ) plan would look like in the current release of AQSIOS (version 2.0) [6]. The window operator exists only if there is a window specification clause in the CQL [8]. If no window is specified in the query, the system applies the aggregation on each input tuple, which is equivalent to a tuple based window of size 1 and slide 1. AQSIOS supports both time- and tuple-based windows, which is inherited from STREAMS code. The window operators, however, do not support sliding windows. More precisely, the current support for sliding windows in AQSIOS is a slide of one tuple. This is regardless of whether the window is a range (i.e., time) or row (i.e., tuple) based window for the range specification. This implementation assumes the slide to be always a row-based slide of length 1. That is for every input tuple, an aggregation result is due. Thus, we needed to add support for the sliding windows.

Not only is support for sliding windows needed, but also a whole set of new operators, with

their semantics, need to be added to AQSIOS in order to support the *Paired Window* processing scheme. Specifically, a sub-aggregation, a slice-manager and a final-aggregation operators need to be implemented.

We chose to implement the slice manager as part of the sub-aggregation operator to reduce context inter-operator communication and context switching.

To support sliding windows, there are three choices. Either to implement it in: (1) the Window operator, (2) the Synopsis, which is the buffer area associated with the operator to maintain its state, or (3) the new slice manager operator. The third choice might be the easiest, but this means that only ACQs will support sliding windows. On the other hand, the first and second choices mean that all CQs can support the sliding windows. Given that the ACQ query plan does not need any window operators, since the slice manager is effectively the window operator, it does not make sense to extend window operators to support sliding windows. We therefore chose to support sliding windows in the implementation of the slice manager. Thus, the query plan would look as the instance in Figure 37.

In summary, we performed the following implementation tasks.

1. Supporting Sliding windows. This involves:
   a. CQL parsing to get the slide.
   b. Processing the Slide window. For this, we have three different options that shall be discussed next.
2. Optimizing multiple ACQs
   a. Allow a CQL directive to instantiate the multiple-ACQs optimizer.
   b. Bypass the current optimizer when this directive exists.
3. Implementation of New Operators.
   a. Slide Manager, which could potentially be merged with the sub-aggregation operator.
   b. Sub-Aggregation operator.
   c. Final-Aggregation operator.
4. Modifying current implementation.
   a. Range Window operator to be slide-aware.
   b. Synopsis operator to be slide-aware.
   c. Group Aggregation operator to be slide-aware.

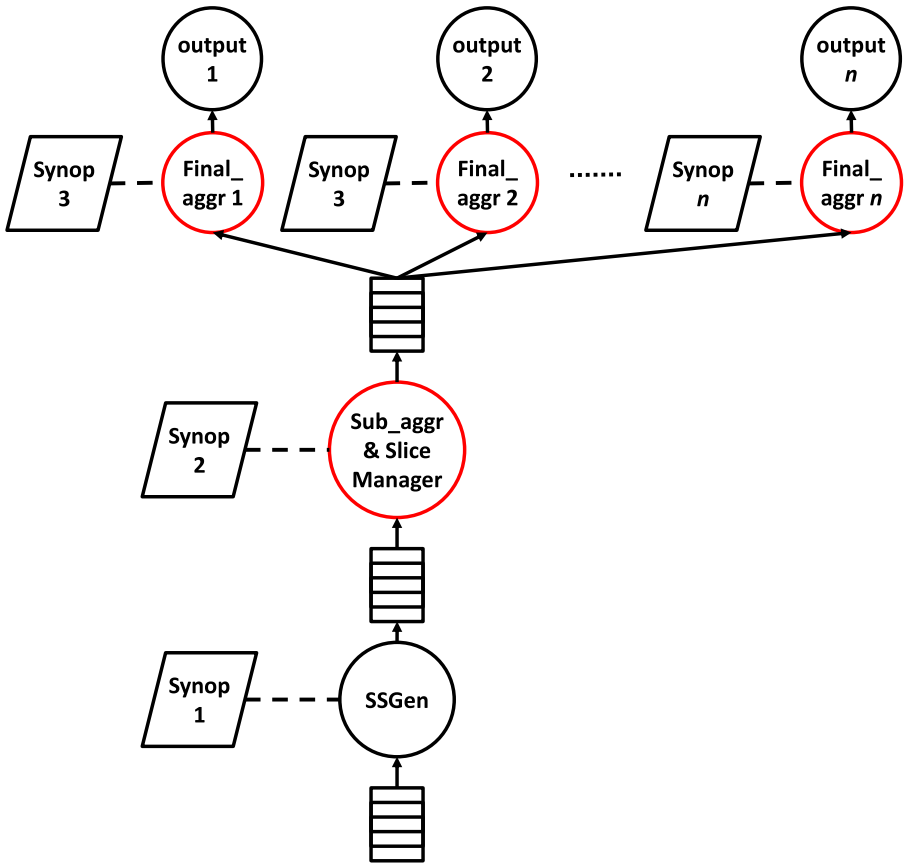


Figure 37: The Weaved Query Plan

To enable the *Weave Share* optimizer, we had to to re-do many steps in converting the logical plan into a physical plan, as mentioned before. That was mainly due to the lack of modularity of the STREAMS basic optimizer. In order to focus on ACQs, we assumed that all continuous queries in the system are ACQs that uses the following template.

```
SELECT <aggregate_function>(<attribute>)
FROM <stream_source> [RANGE <window_range> SLIDE <window_slide>];
```

To invoke the *Weave Share* optimizer, a configuration file parameter need to best properly to request the query optimizer to utilize *Weave Share*. Our implementation of the sub- and final-aggregation operators do support group-by attributes. However, AQSIOS 3.0 does not support different group-by attributes, nor different predicates.

6.3 EVALUATION

In this section we provide the results of *Weave Share* implementation in AQSIOS. In addition, we implemented *Shared* and *No Share*. *Shared* is the optimizer that generates a single execution tree, while *No Share* is the one that generates an execution tree for each ACQ. All other optimizers utilize the same implementation of sub- and final-aggregation operators of *Weave Share*.

In these experiments, we used synthetic data to regenerate the settings of our simulation-based experiments. In all experiments, the data has a Poisson arrival rate of 50 or 300 tuples/sec. We generated a workload of 2 minutes. The ACQs are generated using a slide length following a Zipf distribution with skewness of 0.6, and a maximum overlap factor of 50, as in our simulation-based experiments.

We utilized both the simple *RR* scheduler and the *HR* scheduler. We disabled the load shedder in order to study the performance of the query optimizer with minimal effect from the other modules, i.e., scheduler and load shedder. Reported results are the average of 10 runs.

6.3.1 Performance Under *RR*

As mentioned earlier, the Continuous Query Language (CQL) [8] does not support window slide. The default window operator performs a slide by one tuple by default. We focus in our experiments on *Weave Share* versus *Shared* to confirm the observed simulation-based performance gains of *Weave Share*.

In Figures 38 and 39 we plot the cost in terms of number of aggregation operations per second of *Weave Share* and *Shared* for 50 and 300 tuples/sec, respectively, as the number of ACQs increases. For 300 tuples/sec, *Weave Share* achieved 3 orders of magnitude improvement over *Share*. Also, for 50 tuples/sec, *Weave Share* always reduces the cost over *Shared*, with reduction between 22% and 62%. These results are similar to the simulation results discussed in Chapter 3 (which showed three orders of magnitude improvement compared to *Shared* for 500 ACQs.) We replicate here a snapshot of this simulation results in Figure 40.

We also compared the response time of *Weave Share* to that of *Shared* and *No Share*. In Figure 41 we plot the average response time (in micro-seconds) per window, i.e., per ACQ output

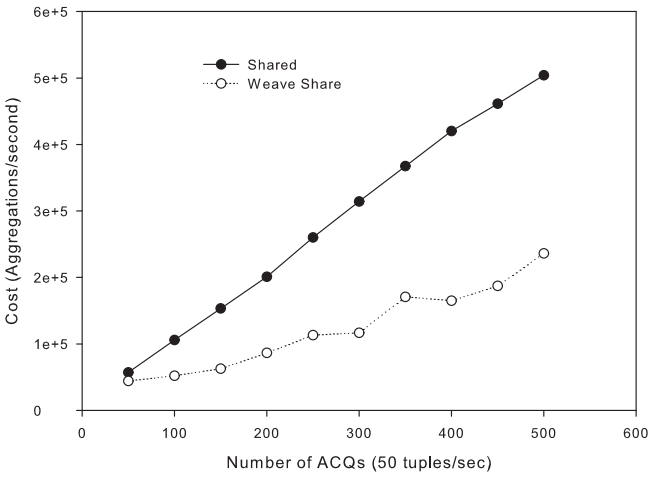


Figure 38: Cost - 50 tuples/sec

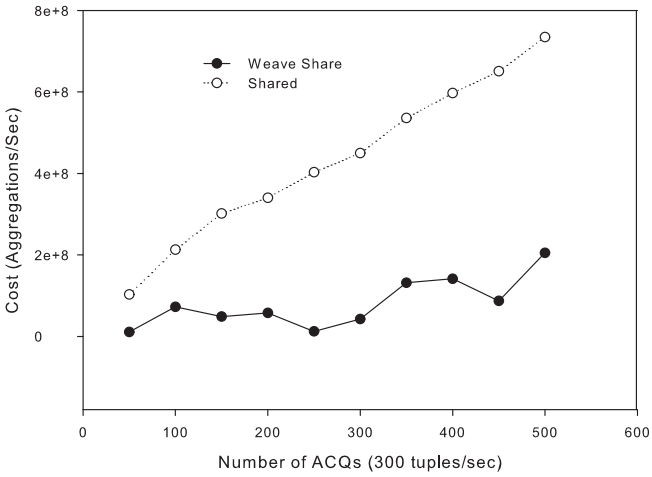


Figure 39: Cost - 300 tuples/sec

tuple. The average was taken across all ACQs, across multiple runs. We see that *Weave Share* reduces the average response time by between 43% to 67%.

We also noticed in Figure 41 that *No Share* performs better than *Shared*, which is and indicator that the system is under-loaded since *No Share* outperforms *Shared* when the system is under-

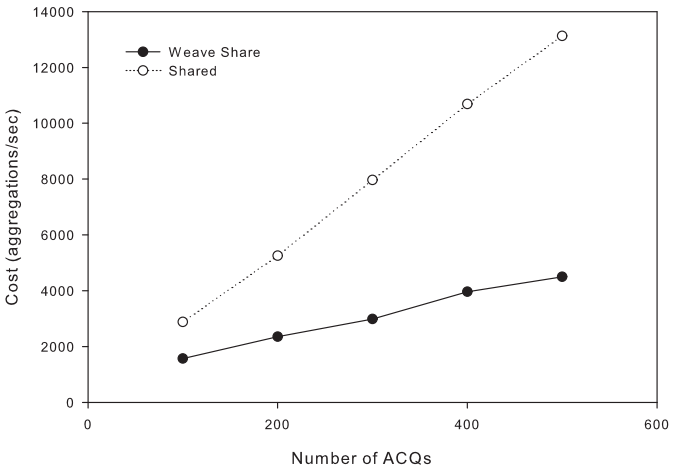


Figure 40: Simulation results for 300 tuples/sec

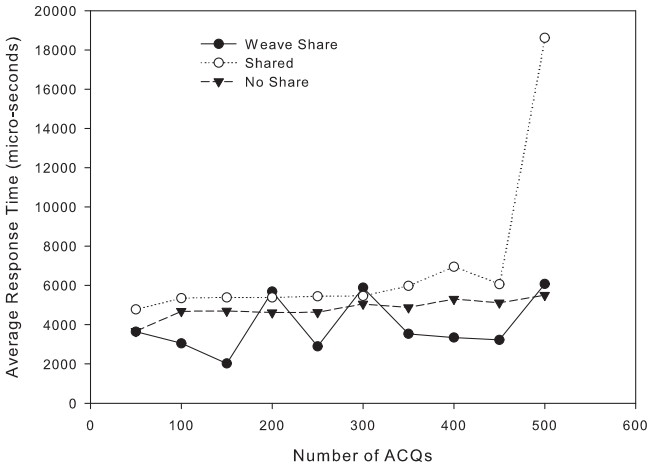


Figure 41: Average Response Time - 300 tuples/sec

loaded. We repeated the experiment with higher input rate of 50K tuples/seconds. The results of this experiment are shown in Figure 42. We first notice that the response time jumps from micro seconds to few seconds. Performance gains of *Weave Share* are also clearer for this high input rate. Specifically, *Weave Share* reduces the response time between 75% and 99%, compared to *Shared*,

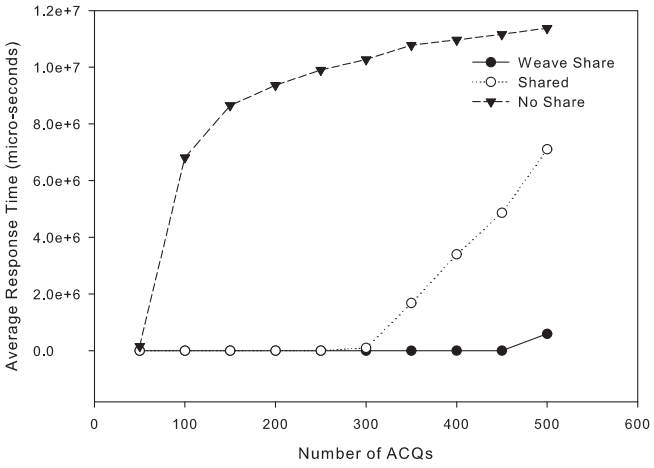


Figure 42: Average Response Time - 50K tuples/sec

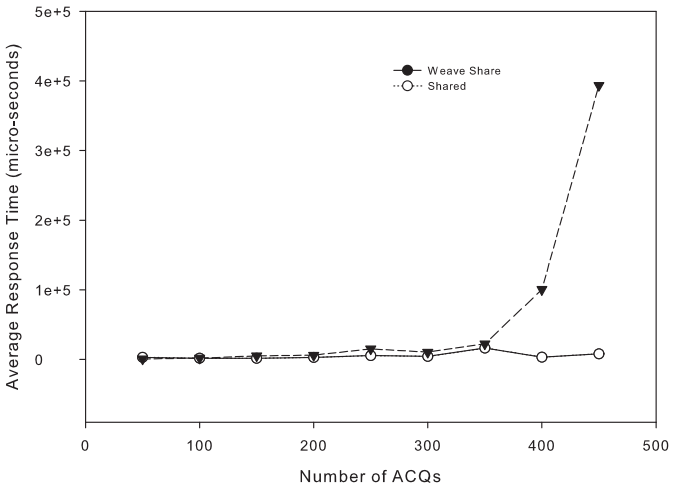


Figure 43: Average Response Time - *HR* scheduler - 50K tuples/sec

for more than 100 ACQs. For 50 and 100 ACQs, where the response time is in few milliseconds, *Weave Share* response time was similar to *Shared*. *No Share*, on the other hand, suffers a relatively huge response time for 100 or more ACQs.

6.3.2 Performance Under *HR*

HR is better than *RR* as a cost-based scheduler, although unfair and hence susceptible to starvation. We expect that utilizing a smarter scheduler, further gains, in terms of response time, can be achieved. In order to minimize the average response time, *HR* gives higher priority for execution to the operator that have a higher production rate. We had to instrument the sub- and final-aggregation operators to measure and report their production rate.

In Figure 43 we plot the average response time (in micro-seconds) f or the *Weave Share* and *Shared* optimizers, for input rate of 50K tuples/second, using the *HR* [71] scheduler. The results however did not show the expected further gains for *Weave Share*. Specifically, *Weave Share* response time is less than that of *Shared* by 53% and 97%, as opposed to 75% and 99% in the case of using the round-robin scheduler. The reason is that sub-aggregation operators typically has higher production rate (which is the edge rate) than that of the final-aggregation (which is one output every window instance). Thus, under *HR*, sub-aggregation operators receives higher priority until they process all the tuples currently in the input buffer, or until the output buffer is full. At this point, the final-aggregation operators gets chance to process their input fragments, which increases their priority and gives them a chance to catch-up with the response time.

The performance gains of *HR* was not seen when utilizing the *Paired Window* processing scheme. This shows that careful choice of the scheduler is needed. In fact, this confirms our motivation for our suggested future work (discussed in Chapter 7) that a study of the synergy between the query optimizer and the scheduler is needed to discover best practice strategies. Otherwise, while optimizing each component independently, one optimized solution for one module might cancel the optimization of the other module and vice-versa.

6.3.3 The Optimizer Performance

In Figure 44 we plot the number of generated execution trees by the *Weave Share* optimizer, for two input rate values; 50 and 300 tuples/sec. As expected, the higher the input rate the less the number of execution trees. On the other hand, for each input rate, the more the number of ACQs, the more execution trees *Weave Share* generates, which also confirms our expectations.

In Figure 45 we plot the optimization time needed by *Weave Share* to generate the weaved plan.

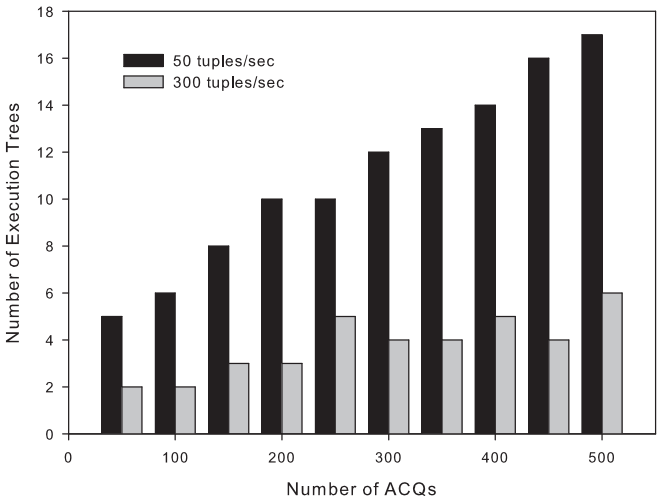


Figure 44: Number of Execution-Trees of *Weave Share*

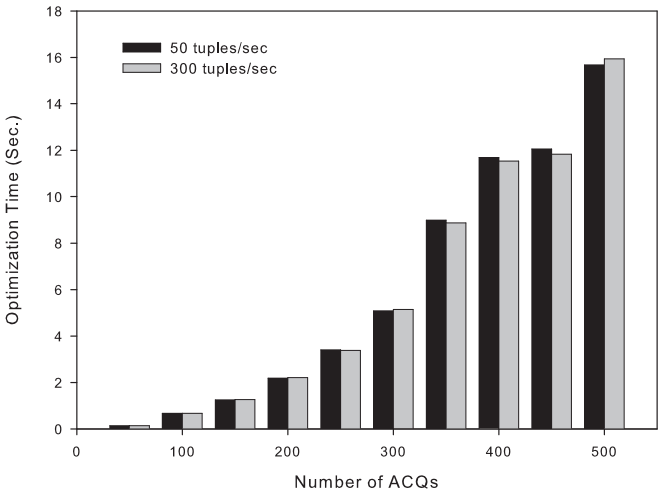


Figure 45: *Weave Share* Optimization Time

The figures show that the *Weave Share* takes up to 16 seconds for 500 ACQs to generate orders of magnitude better plans. This confirms that utilizing the *Weave Share* is practical. As expected, we also see that the more ACQs the longer it takes *Weave Share* to generate the plan.

6.4 SUMMARY

In this chapter, we presented our implementation of the *Weave Share* optimizer in the AQSIOS DSMS prototype. We documented the design and code changes. We also presented the results of *Weave Share* implementation and compared them with those obtained using simulation. We compared *Weave Share* to *Shared* and *No Share*, using synthetic data. The results confirmed our simulation-based results of orders of magnitude improvement over *Shared*. We also measured the response time, and *Weave Share* achieved up to 50% less response time. It remains for our future work to use benchmarks to evaluate our implementation. Another future extension to this implementation is to implement the *TriOps* processing scheme and the generalized *TriOps* optimizer, to handle different group-by attributes and different predicates.

7.0 CONCLUSIONS AND FUTURE WORK

7.1 SUMMARY OF CONTRIBUTIONS

Optimizing the processing of Aggregate Continuous Queries is imperative for Data Stream Management Systems (DSMSs) to reach their full potential in supporting (critical) monitoring applications. Towards this, Shared Processing and Scheduling has been utilized in the literature.

This dissertation provides a new perspective of how multiple ACQs should be shared in order for DSMSs to achieve the desired scalability. It formalizes the ACQs properties that determine their affinity to be shared, proposes a new selective sharing optimizer, and proposes a novel ACQs processing model.

Specifically, in this dissertation we have initially aimed at addressing the following four fundamental questions related to multiple ACQs optimization.

Q1. In addition to the data streams input rate, what other factors of the workload characteristics and ACQs properties affect the cost of a shared query plan? And more importantly, how do these factors interact with each other to affect the cost of a query plan?

Q2. Given our understanding of how the factors that affect the cost of the shared plan interact, can we design a multiple ACQs optimizer that considers all these factors while making the sharing decision? Could this new optimizer comprehensively handle all three cases of variability in the ACQs specifications (i.e., windows, predicates and group-by attributes)?

Q3. Given that ACQs are added to, and deleted from, the DSMS over time, and given that input rates also fluctuates, what is the best adaptive sharing strategy? In other words, when the workload characteristics changes, should the query plan be recomputed or be incrementally updated?

Q4. Is the currently widely-accepted *Partial Aggregation* technique the best continuous aggregation operator implementation for the shared processing of multiple ACQs?

During the course of our experimentation with our simulator, a fifth question motivated the last part of our dissertation.

Q5. Does different optimization techniques for the different DSMS modules integrate well together, in the sense that the integrated system achieves an aggregated performance gain of the gains of the individual techniques, or do different techniques negatively impact each other?

In this dissertation, we identified the *Weaveability* property of Aggregate Continuous Queries, which quantifies their potential to benefit from sharing their processing. We demonstrated how utilizing the Weaveability in optimizing the shared plan of multiple ACQs can yield up to orders of magnitude better plans using the *Weave Share* optimizer. We also proposed *Incremental Weave Share*, to handle the addition and deletion of ACQs.

We questioned the effectiveness of the widely accepted two-level or two-operator implementation of aggregate continuous queries (ACQs) and proposed a new three-level processing model, called *TriOps*, in the context of Weaved Plans which selectively group ACQs into multiple query execution trees (partition groups). We illustrated that the proposed the *intercede-aggregation* operator in *TriOps* minimizes the total cost of processing multiple ACQs by allowing sharing of the sub-aggregation across all partition groups and performing partial final-aggregation shared by the ACQs of a given partition group. Further, we illustrated how the *intercede-aggregation* operator can efficiently support the processing of multiple ACQs with different predicates and group-by attributes in addition to varying window specifications. Finally, we developed *TriWeave*, a *TriOps*-aware multiple query optimizer along the lines of the *Weave Share* optimizer and generalized *TriWeave* to integrate the classical subsumption-based multi-query optimization techniques.

We analytically and experimentally demonstrate the performance gains of our proposed optimization techniques and processing schemes. Finally, we realized *Weave Share* and the *Paired Window* processing scheme in the AQSIOS prototype. We demonstrated *Weave Share* performance through experiments using synthetic data sets and showed the impact of different schedulers on the performance of the query optimizer.

7.2 IMPACT OF THIS DISSERTATION

This dissertation provides a better understanding of how characteristics of ACQs and properties of the workload affect the cost of a shared plan. The concept of *Weaveability* captures the interaction of ACQs characteristics and they affect the cost of a shared plan, in one metric. *Weaveability* is a new powerful tool, which this dissertation introduces, that can, and should, be utilized by a multiple ACQs optimizer to generate better quality shared plans. This opens the door for a new dimension in multiple ACQs optimization.

To share (share all) or not to share (share nothing) among ACQs has been the focus of the data management research community in optimizing multiple ACQs with different windows and different predicates. This dissertation brings in the option of selective sharing, or grouping, of multiple ACQs in order to generate even better shared plans. The *Weave Share* optimizer, proposed in this dissertation, efficiently utilizes *weaveability* and selectively partitions the ACQs into shared groups. Achieving up to four orders of magnitude improvement over the best other alternative, *Weave Share* institutes the corner stone step towards scalability for DSMSs in serving monitoring applications.

This dissertation also proposes solutions that solves the two problems that have been studied orthogonally in the literature. Namely, the optimization of multiple ACQs with different window specifications and/or predicates, and the optimization of ACQs with different group-by attributes. This dissertation proposes the generalized *TriWeave* optimizer, which efficiently integrates the selective sharing of *Weave Share* with the subsumption-based solutions that handle the overlap of different group-by attributes. *TriWeave* is the first, in the literature, general multiple ACQs optimizer that solves the general case, which is the real world case.

This dissertation further developed a new processing model for the ACQs. *TriOps* is a sharing-aware processing model that this dissertation proposes to fully reap the advantages of selective sharing. In fact, *TriOps* is what enables *TriWeave* to handle all possible cases of variability in ACQs specifications.

AQSIOS 3.0, which is the implementation of *Weave Share* in the AQSIOS DSMS prototype and is available online, is the first step towards implementing *TriWeave*. This contribution of the dissertation sets the stage for the data management research community to further study the

problem of optimizing the processing of multiple similar ACQs and to study the synergy between the query optimizer and other modules in the DSMS, using real system implementation.

7.3 FUTURE WORK

Our future work is mainly to generalize the proposed *TriWeave* Optimizer to handle general complex continuous queries. That is, when the system has Aggregate Continuous Queries as well as regular SPJ (i.e., Select, Project and Join) continuous queries. We also consider the case when ACQs are sub-queries of a more complex continuous query. Then, given our realization of *Weave Share* in AQSIOS, we can study the synergy between the two-levels-based or *TriOps*-based ACQs query plan and the scheduler and load shedder, given that these processing schemes involves multiple operators which can be potentially scheduled independently in an arbitrary order. modules in the DSMS so that we can integrate our weave-based optimizers with these other modules. These challenges form the major two possible future extensions to the work in this dissertation discussed below. Another extension include to examine the reverse ordering of the generalized *TriWeave* optimizer steps, as discussed in Section 5.5.2.

7.3.1 Generalization: Optimizing Complex CQs with ACQs

Currently, we consider only (simple) ACQs consisting of only one aggregation, which is the typical case for monitoring applications. However, the more general case is when some of the aggregations are actually sub-queries of more complex CQs. We need to generalize our query optimizer so that it handles the general case. In particular, having non-ACQs might affect the sharing decision. For example, when the CQ has a `HAVING` clause, the results of the aggregation are fed to a filter (selection) operator. In this case, sharing might not be feasible because the continuous query is not reporting the aggregate value, it is actually reporting other attributes of the input schema based on the aggregate value. Further, when the output of an ACQ is fed to different continuous queries with different rates, sharing might not always be the best thing to do. An investigation of how to generalize the proposed heuristics in order to handle such cases is needed.

7.3.2 Synergy with other Modules

We need to first upgrade our implementation of *Weave Share* to *TriWeave*. That is to implement the *TriOps* processing scheme, so that we can perform our studies with the more general *TriWeave* optimizer. Given this implementation, we can utilize AQSIOS to study the synergy of the *TriWeave* optimizer with other modules of the DSMSs. This will lead to designing an integrated query optimizer and best practices recommendations.

For instance, Admission Control is utilized to avoid overloading situations. In [52] we follow a Game Theoretic [63] approach and propose an auction-based admission control mechanisms to host continuous queries on the cloud. Our proposed admission control mechanisms exploit sharing in making the admission decision. We shall investigate how our proposed admission control mechanisms can benefit from the knowledge about our optimizer. For instance, a very expensive ACQ that weaves perfectly with other ACQs might be admitted, and vice versa. Thus, we can develop admission mechanisms that exploit weaveability and shared processing of ACQs.

Similarly, Scheduling goes hand in hand with query optimization to optimize for QoS and QoD metrics. Studying which scheduler performs better for a given shared processing scheme might lead to new discoveries or at least best practice recommendations. Typically, the DSMS scheduler utilizes information about the query plan, input rates, operators' costs and selectivities, and queue status in order to decide which operator to execute next. In our exploration, we shall investigate how to improve the scheduler's performance minimizing the response time given *TriWeave*. The intuition is that our proposed web-transactions scheduler *ASETS* [36, 72], can be modified to schedule ACQs in a way to adaptively minimize the response time.

APPENDIX A

ADAPTING LOCAL SEARCH TECHNIQUE

Finding a theoretical lower bound is interesting and challenging, and it is one of our ongoing efforts. As in traditional multi-query optimization, our goal is to avoid *worst-case* query plans and indeed it could be easily shown that *Weave Share* always avoids the poor plans that might be generated by either Shared or NoShare. We also experimentally investigated and demonstrated the competitiveness of *Weave Share* by comparing it to exhaustive search (optimal) and Local Search (LS) which is near-optimal by supporting backtracking to avoid local optima.

Local search (LS) is a an approach that can be adapted to exploit weaveability to further explore the solution space and better evaluate our proposed approach. LS is a "meta-heuristic" state space search approach [1]. It is used for solving computationally hard combinatorial optimization problems, that can be formulated as finding a solution that minimizes (or maximizes) a certain cost function.

The Local Search approach start by randomly selecting state (i.e., a valid solution) and proceeds towards a local minimum one step at a time, where a step is a single small change in the current solution. The resolution of the step should be small enough in order to guarantee exploring all possible states that are adjacent to the current state.

In each iteration, LS checks all adjacent states and moves to the one that minimizes the cost function until no more adjacent steps are better (i.e., a local minimum is reached) or a time, or number of steps bound is reached. Examples of problems where LS has been applied are the traveling salesman, vertex cover and boolean satisfiability problems.

In order to adapt Local Search to exploit weaveability of ACQs, we chose the step to be a single

move of one ACQ from one execution tree to another existing one, or to a new one. This allows for backtracking and considering all possible adjacent states. As for the initial state, we considered two possible simple initial states. The first is the No Share state, where each tree contains one ACQ only. The second is a randomly generated grouping of ACQs.

APPENDIX B

AQSIOS 3.0 RELATED CODE

We modified the existing code, and added new code. In Table 4, we summarize both modified and added code files and the purpose of each addition or modification. Below, we briefly describe the major changes in the code. We categorize the changes in three tasks: (1) adding the weave share optimizer, (2) adding the new operators, and (3) adding support for the new optimizer and the new operators.

B.0.3 Adding *Weave Share* Optimizer

The Query Optimization in AQSIOS 2.0 is not modular and does not execute at a certain part of the code. However, while the client reads in the queries, each query is registered, at which point a logical plan is generated and converted into a physical plan. While generating the physical plan, some optimizations take place. Then, after all the queries are registered, a simple global optimization phase is instantiated. In order to add the *Weave Share* optimizer, we had to redo all these steps; i.e., generating the logical representation of the plan, converting it into a physical one, and adding auxiliary structures to the physical plan.

Related code: weaveshare.h, server_impl.cc, plan_mgr.cc,

plan_mgr_impl.cc, plan_mgr.h and plan_mgr_impl.h

Table 4: Summary of New and Modified Code

File Name	New/Modified	Description
dsms/src/execution/operators/final_aggr.cc (& .h)	New	the finall-aggregation operator
dsms/src/execution/operators/sub_aggr.cc (&.h)	New	the slice manager and the sub-aggregation operator
dsms/include/metadata/phy_op.h	Modified	added metadata about the new physical operators
/dsms/src/metadata/inst_aggr.cc	Modified	instantiate the new operators
dsms/include/server/params.h, dsms/src/server/config_file_reader.cc (&.h)	Modified	add the Optimizer specification tag and input rate
dsms/src/common/include/parser/nodes.h, dsms/src/common/include/parser/nodes_debug.h, dsms/include/parser/nodes_debug.h, dsms/src/parser/nodes.cc (& .h)	Modified	added window slide specifications to the parser and for the logical plan representation
dsms/include/querygen/query.h	Modified	added slide specification for the query representation
dsms/include/metadata/weaveshare.h	New	definitions of Data structures needed by *Weave Share*
dsms/src/server/server_impl.cc (& .h), dsms/src/metadata/plan_mgr_impl.cc (&.h)	Modified	added the *Weave Share* optimizer

B.0.4 Adding New Operators

We implemented the final-aggregation as well as the sub-aggregation operators. The implementation of the sub-aggregation operator also also includes the slice manager. The support for the sliding window was also implicitly implemented in both new operators. Thus, we didn't need to modify the existing window operators. The new sub- and final-aggregation operators both allow

group-by attributes.

Related code: final_aggr.cc, final_aggr.h, sub_aggr.cc, sub_aggr.h, phy_op.h, and inst_aggr.cc

B.0.5 Adding Support for Sliding Windows and *Weave Share*

In order to support sliding windows, we needed to modify the CQL, and hence the parser, to read in the slide specifications. This required changes in the internal logical representations of the query and the logical query plan. To support *Weave Share* optimizer we needed to add two tags in the configuration file. Namely, a tag to specify if a *Weave Share* optimizer is to be used or not, and another to specify the initial input rate that *Weave Share* shall use to generate the initial weaved plan. AQSIOS has mechanisms that monitor the input rate, and hence, the plan could be changed accordingly. This would, however, require to implement a mechanism for dynamic plan switching. This was left for future work.

Related code: params.h, config_file_reader.cc, config_file_reader.h,
nodes.h, nodes_debug.h, nodes_debug.h, nodes.cc, nodes.h and query.h

BIBLIOGRAPHY

[1] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimizarion*. Princeton University Press, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. C. etintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.

[4] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.

[5] M. H. Ali, M. F. Mokbel, and W. G. Aref. Phenomenon-aware stream query processing. In *MDM*, pages 8–15, 2007.

[6] AQSIOS, http://db.cs.pitt.edu/aqsios, 2011.

[7] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager (demonstration description). In *SIGMOD*. ACM, 2003.

[8] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDBJ*, 15(2):121–142, 2006.

[9] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD, pages 261–272, New York, NY, USA, 2000. ACM.

[10] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29:261–272, May 2000.

[11] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 253–264. ACM, 2003.

[12] E. Bach and K. Pruhs. Personal communications, June 2010.

[13] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Load management and high availability in the medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 929–930, New York, NY, USA, 2004. ACM.

[14] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel. An approach to adaptive memory management in data stream systems. In *ICDE*, 2006.

[15] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668. ACM, 2003.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[18] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[19] P. K. Chrysanthis. Aqsios - next generation data stream management system. *CONET Newsletter*, June 2010.

[20] C. Chung. *Evolutionary Solutions and Internet Applications for Algorithmic Game Theory*. PhD thesis, U. of Pittsburgh, Pittsburgh, PA, Aug. 2009.

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, pages 1115–1118. ACM, 2010.

[22] http://www.coral8.com/, 2004.

[23] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.

[24] C. D. Cranor, T. Johnson, and O. Spatscheck. *Streams Book*, chapter Data Stream Processing Techniques for Network Management. November 2006.

[25] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Multi-query optimization for sketch-based estimation. *Inf. Syst.*, 34(2), 2009.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. WH.Freeman & Co., New York, NY, USA, 1990.

[27] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *CIKM '05*, 2005.

[28] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3–8, 2006.

[29] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE TKDE*, 19(1):57–72, 2007.

[30] L. Golab, K. G. Bijay, and M. T. Ozsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, pages 844–845, 2006.

[31] L. Golab, T. Johnson, and O. Spatscheck. Prefilter: predicate pushdown at streaming speeds. In *Proceedings of the 2nd international workshop on Scalable stream processing system*, SSPS '08, pages 29–37, New York, NY, USA, 2008. ACM.

[32] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

[33] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. DMKD*, 1(1):29–53, 1997.

[34] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Exploiting weaveability to optimize the processing of multiple aggregate continuous queries. Technical Report TR-11-177, University of Pittsburgh, 2010.

[35] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.

[36] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, pages 357–368. IEEE Computer Society, 2009.

[37] R. Gupta and K. Ramamritham. Query planning for continuous aggregation queries over a network of data aggregators. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1, 2011.

[38] M. Hammad, M. Franklin, and W. Aref. Scheduling for shared window joins over data streams, 2003.

[39] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Y. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[40] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, pages 485–496, 2007.

[41] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 271–278, New York, NY, USA, 2003. ACM.

[42] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To share or not to share? In *VLDB*, 2007.

[43] A. Kementsietsidis, F. Neven, D. Van de Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 1(1):16–27, 2008.

[44] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, pages 1081–1092. ACM, 2010.

[45] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634. ACM, 2006.

[46] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[47] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322. ACM, 2005.

[48] L. Ma, Q. Zhang, K. Wang, X. Li, and H. Wang. Semantic load shedding over real-time data streams. *International Symposium on Computational Intelligence and Design*, 1:465–468, 2008.

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

[50] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670. ACM, 2004.

[51] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318. ACM, 2001.

[52] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs. Admission control mechanisms for continuous queries in the cloud. In *ICDE '10: Proc. of the 26th International Conference on Data Engineering*. IEEE Computer Society, March 2010.

[53] L. A. Moakar, T. N. Pham, P. Neophytou, P. K. Chrysanthis, A. Labrinidis, and M. A. Sharaf. Class-based continuous query scheduling for data streams. pages pp. 1–6, August 2009.

[54] M. F. Mokbel and W. G. Aref. Place: A scalable location-aware database server for spatio-temporal data streams. *IEEE Data Engineering Bulletin.*, 28(3):3–10, 2005.

[55] M. F. Mokbel and W. G. Aref. Sole: scalable on-line execution of continuous queries on spatio-temporal data streams. *VLDB J.*, 17(5):971–995, 2008.

[56] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad. Place: A query processor for handling real-time spatio-temporal data streams. In *VLDB*, pages 1377–1380, 2004.

[57] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. In *STDBM*, pages 57–64, 2004.

[58] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. *GeoInformatica*, 9(4):343–365, 2005.

[59] K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.

[60] Nasdaq. nastraq: North american securities tracking and quantifying system. http://www.nastraq.com/description.htm.

[61] P. Neophytou, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Power-aware operator placement and broadcasting of continuous query results. In *MobiDE*, pages 49–56, 2010.

[62] P. Neophytou, J. Szwedko, P. K. Chrysanthis, A. Labrinidis, and M. A. Sharaf. Optimizing the energy consumption of continuous query processing with mobile clients. pages pp. 1–6, June 2011.

[63] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, editors. *Algorithmic Game Theory*. 2007.

[64] T. Pham, P. Chrysanthis, and A. Labrinid. An adaptive load manager for the aqsios stream engine. *Technical Report*, 2010.

[65] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. pages 1–6, April 2011.

[66] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[67] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *PODS*, pages 238–248. ACM, 1997.

[68] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[69] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Tuning qod in stream processing engines. In *ADC*, pages 103–112, 2010.

[70] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, pages 511–522. VLDB Endowment, 2006.

[71] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems*, 33(1):1–44, 2008.

[72] M. A. Sharaf, S. Guirguis, A. Labrinidis, K. Pruhs, and P. K. Chrysanthis. Asets: A self-managing transaction scheduler. In *SMDB Workshop at ICDE*, 2008.

[73] Streambase: http://www.streambase.com, 2006.

[74] System S, http://domino.research.ibm.com/, 2008.

[75] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB Conf*, 2003.

[76] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 625–636, New York, NY, USA, 2011. ACM.

[77] http://www.truviso.com, 2005.

[78] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB, pages 787–798. VLDB Endowment, 2006.

[79] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

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

[81] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *VLDB*, 2006.

[82] G. Xue, Q. Pan, and M. Li. A new semantic-based query processing architecture. *Parallel Processing Workshops, International Conference on*, page 63, 2007.

[83] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310. ACM, 2005.

[84] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *VLDBJ*, 19(4):557–583, 2010.

[85] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544. ACM, 2007.