Algorithms and Optimizations for Incremental Window-Based Aggregations

by

Anatoli U. Shein

B.S., Duquesne University, 2012

M.S., University of Pittsburgh, 2018

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Anatoli U. Shein

It was defended on

September 3rd 2019

and approved by

Dr. Panos K. Chrysanthis, Department of Computer Science, School of Computing and Information

Dr. Alexandros Labrinidis, Department of Computer Science, School of Computing and Information

Dr. Adam J. Lee, Department of Computer Science, School of Computing and Information

Dr. Vladimir I. Zadorozhny, Department of Informatics and Networked Systems, School of Computing and Information

Dissertation Director: Dr. Panos K. Chrysanthis, Department of Computer Science, School of Computing and Information

Algorithms and Optimizations for Incremental Window-Based Aggregations

Anatoli U. Shein, PhD

University of Pittsburgh, 2019

Online analytics, in most advanced scientific, business, and social media applications, rely heavily on the efficient execution of large numbers of Aggregate Continuous Queries (*ACQs*). *ACQs* continuously aggregate streaming data and periodically produce results such as *max* or *average* over a given window of the latest data.

Incremental Evaluation is widely accepted for processing *ACQs*. It involves storing and reusing results of calculations performed over the unchanged parts of the window, rather than performing the *re-evaluation* of the entire window after each update. Recently proposed *Incremental Evaluation* techniques achieve high throughput and low latency in both single- and multi-query environments. In multi-query environments, these techniques share partial aggregates among all of the registered queries (i.e., all the queries are merged and processed as a single execution tree) to achieve maximum sharing. However, it was shown that maximum sharing does not always offer maximum performance, and selective sharing achieves better results by splitting the query load into multiple execution trees. To strike a balance between non-shared and fully shared query executions, the notion of *Weavability* was proposed, which led to several new *Multi-Query* optimizers.

In this dissertation, we identify that (1) the current *Incremental Evaluation* techniques fail to exploit the semantics of aggregation operations, leaving considerable room for improvement, and (2) the *Weavability*-based *Multi-Query* optimizers target the *Incremental Evaluation* techniques that are agnostic towards the algebraic properties of the operations, preventing them from achieving improved throughput and latency, and perform *Weavability* calculations in an inefficient and resource-intensive way, hindering optimizers' scalability with the increasing *ACQ* load.

Motivated by the above observations, in this dissertation we re-examine how the principle of sharing is applied in *Incremental Evaluation* techniques as well as in the *Multi-Query* optimizers. Our **hypothesis** is that sliding-window aggregation processing can benefit from (1)

improving the performance of *Incremental Evaluation* by exploiting the algebraic properties of *ACQ*'s underlying aggregate operations and (2) developing new *Multi-Query* optimizers that can target multi-node distributed environments and efficiently generate high quality execution plans by exploiting the new *Incremental Evaluation* techniques.

This dissertation research contributes new algorithms for both *Incremental Evaluation* (*FlatFIT* and *SlickDeque* techniques), and *Multi-Query* optimization (*Formula F1*, *Distributed ACQ Optimizers*, and *New Cost Estimation Methods*). We evaluate all our contributions both theoretically in terms of time and space complexities, and experimentally in terms of throughput, latency, cost minimization, and load balancing using both real and synthetic datasets.

Table of Contents

List of Tables

List of Figures

Preface

Above all, I would like to express my appreciation to my academic advisor Dr. Panos Chrysanthis. His profound belief in my work and irreplaceable guidance were essential for the completion of this dissertation. Also, I gratefully acknowledge Dr. Alexandros Labrinidis for his valuable input to my research, and my committee members Dr. Adam J. Lee and Dr. Vladimir Zadorozhny for their insightful feedback.

I would like to thank all my friends and colleagues that I met throughout my PhD journey for their constructive criticism and excellent cooperation, especially Briand Djoko, Cory Thoma, Daniel Petrov, Hany Saleeb, Nickolaos Katsipoulakis, Rakan Alesghayer, and Salim Malakouti. It was always helpful to bounce ideas off of them, and if I ever lost interest, they kept me motivated.

I dedicate this work to my parents Liudmila and Vladimir, and my amazing wife Christine. Their unwavering belief in me kept me going through all of the hardships that I faced along the way. Thank you.

1.0 Introduction

Data stream processing has gained momentum in many applications that require quick responses based on incoming high velocity data flows. A representative example is a stock market application, where multiple clients monitor the price fluctuations of the stocks. In this setting, a system needs to be able to efficiently answer analytical queries (e.g., average stock revenue, profit margin per stock, etc.) for different clients, each one with (potentially) different timing requirements. Efficient data stream processing is also important in monitoring applications and online analytics in the fields of health care, science, social media, and network control.

Data Stream Management Systems (*DSMS*) were proposed in academia [1, 2, 18, 22, 40, 11, 10] and adopted in industry [48, 3, 8, 32, 37, 6, 56, 14] as the most suitable systems for handling such data flows on-the-fly and in real time. Traditional Database Management Systems (*DBMS*s) struggle to meet the strict timing requirements of processing online analytics on such data streams because they need to store the data into the system before processing it, which incurs high I/O and computational costs. Conversely, in *DSMSs* clients register their analytical queries on incoming data streams, which are processed in main memory continuously by aggregating streaming data as it arrives, as such these queries are called *Aggregate Continuous Queries* (*ACQs*). *ACQs* are typically associated with a *range* (or window) (**r**) and a *slide* (**s**) which can be either *tuple count* or *time*-based. A slide denotes the period at which an *ACQ* updates its outcome; a range is the window over which the statistics are calculated. If the range is equal to or smaller than the slide, it is called a *Tumbling Window*, otherwise a *Sliding Window*. In this work we focus on sliding-window aggregations (also known as *SWAG* [45]) since it is the most commonly used case.

Example 1 Consider a stock trading application monitoring average stock prices every 3 seconds for the past 5 seconds. Such an application submits a time-based *ACQ* with *SWAG* specifications of a 5 second range and 3 second slide. ■

An *ACQ* requires the *DSMS* to maintain state over time while performing aggregations. Normally, *DSMSs* only keep the window of the most recent data items, and when new data

arrives, the window slides by discarding the data items that fall outside of the window specification and filling in the new data items. This allows the *ACQ* to execute over the updated window and reflect recent changes. It is clear that the greater the range of an *ACQ*, the higher its cost is to maintain in the system since more data is kept in memory. Similarly, the smaller the slide of an *ACQ*, the higher its computational cost since the query result needs to be evaluated more frequently. It has been shown that in *SWAG* processing it is beneficial to utilize *Incremental Evaluation* (*IE*), which operates by maintaining and reusing calculations performed over the unchanged parts of the window, rather than executing the *re-evaluation* of the entire window after each update [15, 34]. *IE* is also referred to as *Two-Ops* because it is typically implemented as two operators and executes in two phases by (1) running partial aggregations on the data while accumulating it and (2) producing the answer by performing the final aggregation over the partial results [31, 33].

Initially *SWAG* processing assumed a single-node (CPU) processing infrastructure. The advances in multi-core architectures and the high demand in processing huge volumes of *ACQs* [3] led to the deployment of *ACQs* on multi-node processing environments such as multi-core, distributed, multi-tenant cloud, or high performance computing (HPC) infrastructures. In a single-query setting, the focus is on supporting one long-running, high accuracy *ACQ* by re-using its intermediate calculations and parallelizing the aggregation operators [29, 30]. In a *multi-query* (*MQ*) setting, whether single or multi-node, multiple *ACQs* with different window properties are executed simultaneously (which in practice can reach millions of simultaneous queries [21, 23]) and for an extended period of time, until they are explicitly terminated. In this execution environment *ACQs* often calculate similar (or algebraically compatible) aggregation operations on the same stream. It has been shown that such *ACQs* can be selectively combined into execution trees (that form an execution plan) to increase the processing efficiency by sharing partial aggregations [20]. Clearly, it is crucial to be able to generate high quality execution plans quickly. Unfortunately, this has been proven to be NP-hard [57], and currently only approximation algorithms can produce acceptable execution plans. Such approximation algorithms are utilized in the state of the art *MQ* optimizers *WeaveShare* [20] and *TriWeave* [19].

1.1 Problem Statement

We distinguish the open problems of the current *SWAG* processing along the following two dimensions of sliding-window aggregation processing: (1) *Incremental Evaluation* and (2) *Multi-Query Optimization*. In this section we identify their shortcomings, which motivated our contributions.

1.1.1 Shortcomings of Incremental Evaluation

S1. Current IE techniques do not scale with increasing window sizes.
The state-of-the-art *IE* solutions for processing *ACQs* are *FlatFAT* [47], *TwoStacks* [45], and *DABA* [45]. Each of these solutions aims to increase *ACQ* processing throughput while minimizing latency, however they leave substantial room for improvement. *FlatFAT* utilizes a tree structure for reusing calculations, yielding logarithmic time complexity (i.e., $O(log(n))$ where n is the size of the window), which does not scale with increasing window sizes. The other two techniques, *TwoStacks* and *DABA*, both have *constant* time complexities, however *TwoStacks* introduces large latency spikes in *ACQ* processing due to the calculation imbalance between its update operations, and *DABA* has a high constant in its complexity due to its amortization strategy, leading to lower throughputs. That is, the main shortcoming of the existing *IE* techniques is scalability, due to falling short of supporting *ACQs* processing with high throughput and low latency.

One of the main reasons why the above shortcomings exist is that the current *IE* techniques process aggregate operations with different algebraic properties *uniformly*, which rules out their ability to further improve their performance by taking advantage of optimizations targeted at specific algebraic properties (e.g., invertibility).

1.1.2 Shortcomings of Multi-Query Optimization

S2. Current IE techniques are not designed for MQ shared processing.
The state-of-the-art *IE* techniques focus solely on single-query processing where an *ACQ* is reusing its intermediate calculations instead of re-evaluating the entire window after each

slide. However, most of them fail to consider *MQ* environments, where multiple *ACQs* calculating similar aggregations with different ranges and slides can be processed within the same data structure and share partial results with each other, achieving higher efficiency.

Even though some *IE* techniques were developed to operate in *MQ* environments, their processing is always shared among all of the *ACQs* since they all are processed by a single structure (or execution tree), which forces the maximum level of sharing. However, it was shown that maximum sharing does not always offer maximum performance, and *selective sharing* achieves better results by splitting the query load into multiple execution trees selectively and processing them separately. *WeaveShare* [20] and *TriWeave* [19] *MQ* optimizers were proposed to selectively combine *ACQs* into execution trees.

S3. Current *MQ* optimizers do not scale with the increasing number of *ACQs*. State-of-the-art *WeaveShare* and *TriWeave* produce high quality execution plans using the *Weavability* concept, and are theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem [12]. However, they do not scale with the increasing *ACQ* load due to the bottlenecks in their resource-intensive *Weavability* calculations, which currently are performed using an inefficient count-based approach to make collocation decisions.

S4. Current *MQ* optimizers are oblivious to distributed processing capabilities. The state-of-the-art *MQ* optimizers are also targeting only single-node *DSMSs*, failing to exploit the availability of multi-node (multi-core and multi-processor) distributed environments for the generation of cost-effective, high quality execution plans of *ACQs*.

S5. Current *MQ* optimizers do not support state-of-the-art *IE* techniques. Ultimately, the above-mentioned state-of-the-art *MQ* optimizers currently make their query collocation decisions based on the outdated *IE* technique, *Panes*. That is, the cost estimation formulas of the state-of-the-art *MQ* optimizers are not applicable for use with the new *IE* techniques because they perform different (smaller) numbers of final aggregation operations per window slide.

The shortcomings outlined above motivated the research in this dissertation.

1.2 Our Approach

In order to address **S1** and **S2** we investigate how novel (not tree-based) *ACQ* processing structures and algebraic query semantics can be utilized in new *IE* techniques leading to improved scalability and enabled support for *MQ* processing.

We attack **S3** by researching a new way of identifying window specification overlaps in a more efficient (and not count-based) way. Specifically, we develop a formula that computes the number of overlaps mathematically rather than materializing a composite slide and counting them directly.

We examine challenges of **S4** by tailoring new collocation decision algorithms for distributed systems. That is, we attempt to minimize the total execution plan cost (which allows processing more *ACQs*) while also balancing the workload among computation nodes evenly (which prevents the need to over-provision nodes in order to cope with unbalanced workloads).

Finally, we explore the possible solutions for **S5** by scrutinizing the behaviors of new *IE* techniques in *MQ* settings and exploring how they can be used in *MQ* optimization. In order for an *IE* technique to be supported by an *MQ* optimizer, two requirements must be met: (1) the *IE* technique must support *MQ* processing within its structure, and (2) the *MQ* optimizer must be able to estimate calculation costs of multiple *ACQs* processed together using this technique. Thus, the opportunity arises to explore the suitability of new and more efficient *IE* techniques for use in combination with the *MQ* optimizers.

Thereupon, our **hypothesis** is that sliding-window aggregation processing can benefit from (1) improving the performance of *Incremental Evaluation* by exploiting the algebraic properties of *ACQ*'s underlying aggregate operations and (2) developing new *Multi-Query* optimizers that can target multi-node distributed environments and efficiently generate high quality execution plans by exploiting the new *Incremental Evaluation* techniques.

1.3 Contributions

This dissertation contributes new methods and understandings of both dimensions of *Incremental Evaluation* (*IE*) and *Multi-Query* (*MQ*) optimization. These contributions support our hypothesis and address the shortcomings of the current state-of-the-art techniques.

1. A taxonomy of all IE techniques available today and their breakdown in terms of applicability, complexity, and usability in *MQ* environments.

2. A new efficient final aggregation technique *FlatFIT* that allows higher *ACQ* processing throughput (partially addresses **S1**) than *FlatFAT* which was the state-of-the-art technique at the time. *FlatFIT* reduces the number of partials used in computing a final aggregation by dynamically storing the intermediate results and their corresponding pointers in a novel indexing structure. The indexing structure indicates how far ahead *FlatFIT* can skip in each step of its calculation. *FlatFIT* is applicable for *MQ* processing (addresses **S2**). We experimentally show that *FlatFIT* achieves up to a 17x throughput improvement over *FlatFAT* for the same input workload while using less memory [43].

3. Another new final aggregation technique, *SlickDeque*, that maintains both high throughput and low latency in *ACQ* processing by treating *ACQs* differently based on their invertibility property (fully addresses **S1**). The invertible operations are processed using *SlickDeque* (Inv), our new modified *Panes* (Inv) approach. The non-invertible *ACQs* are processed with *SlickDeque* (Non-Inv), our novel deque-based algorithm that intelligently maintains and utilizes intermediate partial aggregates, allowing a greater level of reuse of previously calculated results. It is also applicable for *MQ* processing (addresses **S2**). We show that *SlickDeque* maintains 283% lower latency spikes on average while achieving up to 345% throughput improvement over the state-of-the-art approaches along with requiring up to 5 times less memory [44].

4. A novel closed formula, *F1*, that accelerates all of the *Weavability*-based *Multi-Query* optimizers by replacing the iterative and calculation-heavy *Bit Set* method with a closed formula for *Weavability* calculations (addresses **S3**). We showed that $F1$ can reduce the computation time of any technique that combines partial aggregations within composite

slides of multiple *ACQs* by up to 60,000x, and that it is superior to the current approach [41] in both time and space complexities.

5. A set of novel *Weavability*-based *Multi-Query* optimizers that allow processing *ACQs* in a distributed environment (address **S4**), including *Weave-Group to Nodes* (WG_{TN}) and *Weave-Group Inserted* (WG_I) optimizers, that produce plans of significantly higher quality than the rest of the optimizers by minimizing the total cost (where WG_{TN} is best in 90% cases) and achieving better load balancing (where WG_I is best in 80% cases) [42].
6. A theoretical analysis of all of the available *IE* techniques that determines their average operational cost (Ω) per slide given any set of input *ACQs* (addresses **S5**) and allows estimating their performance on average within 22% of the actual performance.
7. A new *MQ* optimization implementation that incorporates the new *IE* techniques into the into the state-of-the-art *MQ* optimizers *WeaveShare* and *TriWeave* using the theoretical study mentioned in Contribution 6. The new implementations of *WeaveShare* and *TriWeave* reduce execution costs by up to 270,000x compared to the previous implementations (addresses **S5**).

We support all of our contributions by carrying out their extensive experimental evaluation using both synthetic and real data sets. Towards this we develop two experimental testbeds: (1) a C++ based execution platform for measuring the performance of different *IE* techniques, and (2) a Java based *MQ* optimization platform for generating execution plans by selectively combining large numbers of *ACQs* into execution trees.

1.4 Roadmap

In Chapter 2 we summarize the related work, which constitutes the background of our work, and we introduce our taxonomy of *IE* techniques. We also state our assumptions about processing SWAG. We present and evaluate theoretically and experimentally our new *Incremental Evaluation* techniques *FlatFIT* and *SlickDeque* in Chapters 3 and 4, respectively. Our novel formula, *F1*, for accelerating *Weavability*-based *MQ* optimizers is proposed in Chaper 5, our new *MQ* optimizers for distributed environment in Chapter 6, and in Chapter 7

we present our solution that combines the new *IE* techniques with *MQ* optimizers. We conclude and provide an overview of proposed future work in Chapter 8.

2.0 Background & Related Work

In this chapter we review the underlying concepts of our work, which are the *Incremental Evaluation* for sliding-window computation, and *Weavability*-based *Multi-Query* optimizers. We also review other related work.

2.1 Algebraic Properties

One of the important metrics that allows for the evaluation of the difficulty of processing a particular ACQ incrementally is the algebraic properties of the underlying aggregate operation. Based on classification from [17], all aggregate operations are divided into three broad categories: *distributive*, *algebraic*, and *holistic*.

- **Distributive** aggregation means that the aggregation for the set S can be computed from two of the same aggregations of subsets $S1$ and $S2$, where subsets $S1$ and $S2$ were constructed by splitting S in two. For example, if we have a set of 10 numbers and the Sum of the first 7 is 20, and the Sum of the 3 remaining is 15, then we can get the Sum of all 10 numbers by adding 20 and 15. Therefore, Sum is a distributive aggregation.
- **Algebraic** aggregation means that the aggregation can be computed from a number of distributive aggregations, e.g., Average, which is calculated from Sum and Count. The list of common distributive aggregations includes Count, Sum, Sum of Squares, Product, and Max. By combining these distributive aggregations we can calculate some commonly used algebraic aggregations such as: Average (Count and Sum), Standard Deviation (Sum of Squares, Sum, and Count), Geometric Mean (Product and Count), and Range (Max and Min).
- **Holistic** aggregations are neither distributive nor algebraic, e.g., Median, Top-K, Quantile, Collect Distinct. Holistic aggregates are out of the scope for this work since they require specifically tailored algorithms which cannot be generalized [17].

In this dissertation we will focus on optimizing the distributive aggregations; calculating the algebraic aggregations follows trivially. Distributive aggregations can be further classified by their mathematical properties: *associativity*, *invertibility*, and *commutativity*. Below we provide brief definitions of these properties.

- An operation $\oplus$ is *associative* if $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ is true for all x, y, z.
- An operation $\oplus$ is *invertible* if there exists an operation $\ominus$ such that $(x \oplus y) \ominus y = x$ for all x, y, and $\ominus$ is feasibly inexpensive.
  - Note: if operation $\oplus$ is *non-invertible*, then $x \oplus y = z$, where $z \in \{x, y\}$. This is only true for **non-holistic** operations (which we target in this work).
- An operation $\oplus$ is *commutative* if $x \oplus y = y \oplus x$ is true for all x, y.

2.2 Assumptions

In this dissertation we make the following assumptions about processing *SWAG*:

Query Operation Assumptions In terms of query operation generality, all of the compared non-naive *IE* techniques support non-invertible and non-commutative operations while requiring the operations to be associative. In general, all operations that can be executed on a window of values are associative. The common non-associative operations such as subtraction ($x - y - z$), division ($x/y/z$), exponentiation (x^{y^z}), and some binary operations such as $NAND$ and NOR, are generally impractical when executed on sets larger than two.

Window Structure Assumptions In *non-FIFO* window structures, the events of insertion and expiration are not synchronized, which can cause window overflow situations when there are not enough expiring tuples (or partial aggregates) to make room in the window for the insertions. All of the compared approaches, including ours, are able to handle such cases by performing dynamic resize operations. However in this work we are focusing on the *FIFO* window environment which is the most common method of processing sliding-window aggregations in practice.

Arrival Order Assumptions All of the compared *IE* techniques allow updates on multiple

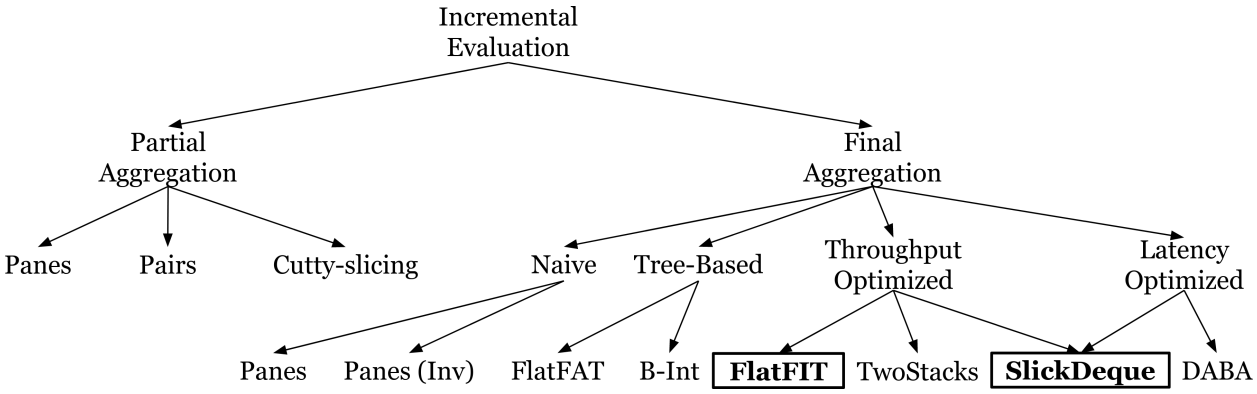

Figure 1: Incremental Evaluation Taxonomy. Our contributions are marked with squares.

partial aggregates already stored within the window. However in this dissertation we focus on the classic streaming scenario when all new partial aggregates are processed by the final aggregator one-by-one as they become available. In such settings the arriving tuples have to be *in-order* or slightly *out-of-order*. As long as the *out-of-order* tuples are within the same partial aggregation, the final result will not be affected. If, however, some tuples fall outside of their partial, inconsistencies in the final result may arise. The mechanism for coping with such situations (e.g., [46]) are outside of the scope of this dissertation.

Result Accuracy Assumptions In this dissertation we focus on *IE* techniques that produce *exact* answers since it is crucial for many applications (e.g., financial, medical, etc.). That is, we do not consider approximate calculation methods, which were proposed to save time and space by sacrificing accuracy [7, 4, 13, 16].

2.3 Incremental Evaluation Taxonomy

In order to provide a better context to our work, we developed a taxonomy of existing *Incremental Evaluation* (*IE*) techniques (illustrated in Figure 1). The *IE* techniques can be broadly divided into *partial aggregation* and *final aggregation* categories. The *final aggregations* can be further distinguished into *Naive*, *Tree-based*, *Throughput Optimized*, and *Latency Optimized* approaches.

Table 1: Partial Aggregation Technique Comparison

Partial Aggregation Algorithm	# of partials per window when $r\%s = 0$	# of partials per window when $r\%s \neq 0$
Panes	r/s	$r/GCD(r,s)$
Pairs	r/s	$2 \cdot \lfloor r/s \rfloor + 1$
Cutty-slicing	r/s	$\lceil r/s \rceil$

2.3.1 Partial aggregation

Partial aggregation can be thought of as the buffering of partial results until the query result needs to be returned by the final aggregation. Since partial aggregation allows buffering results that are later processed by a more expensive final aggregator, each buffered partial aggregate (or simply *partial*) is reused multiple times as part of different final aggregations, alleviating the use of CPU and memory resources. When processing time-based windows, partial aggregation also helps to mask bursty inputs. Clearly, it is beneficial to reduce the number of produced partials in order to minimize the amount of work done by the final aggregator. To this end the following partial aggregation techniques were proposed (summarized in Table 1).

Panes [33] was proposed as the first partial aggregation technique for processing *ACQs* efficiently. The idea behind it is to partition the incoming datastream into "*panes*" (we refer to them as *partials*), and maintain just one aggregate value for each partial. This way every incoming tuple affects the aggregate value for just the current partial, and when the whole aggregate is due to be reported, the answer is assembled by performing the final aggregation over all of the partials in the current window. Therefore, each new partial is reused multiple times for different final aggregations.

For example, in Figure 2 an *ACQ* is processed with a range of 4 partials and a slide of 1. This way each final aggregation assembles a query answer from the 4 most recent

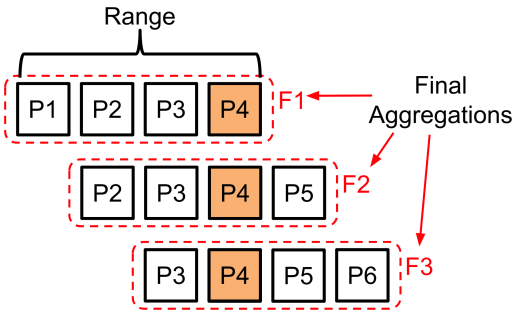

Figure 2: Panes Technique (range=4 and slide=1)

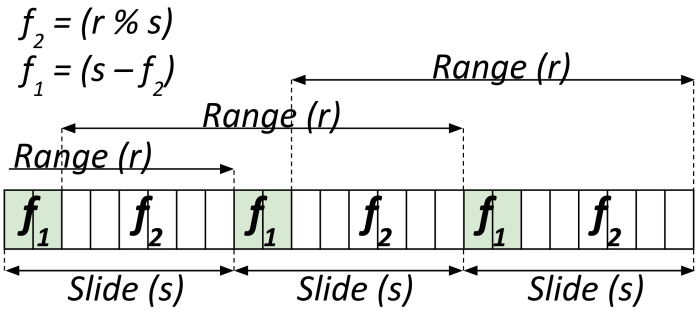

Figure 3: Paired Window Technique (range=14 and slide=8)

partials. Notice that partial *P4* is used 3 times (during 3 consecutive slides) as part of the final aggregations *F1*, *F2*, and *F3*.

The number of partials per window is $range/slide$ if the range is divisible by slide, otherwise it is $range/GCD(range, slide)$, where GCD is the Greatest Common Divisor.

Paired Windows (or simply *Pairs* [31]) was a technique introduced to reduce the number of partials in a window in cases where the range is not divisible by the slide. It works by splitting each slide into two fragments of different lengths as illustrated in Figure 3, where fragment lengths $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$, were calculated as follows: $\boldsymbol{f_1} = range\%slide$ and $\boldsymbol{f_2} = slide - \boldsymbol{f_2}$ (in Figure 3, $\boldsymbol{f_1} = 2$ and $\boldsymbol{f_2} = 6$). This way each window is composed of $2 \cdot \lfloor r/s \rfloor + 1$ partials, which significantly reduces the memory consumption and accelerates final aggregations.

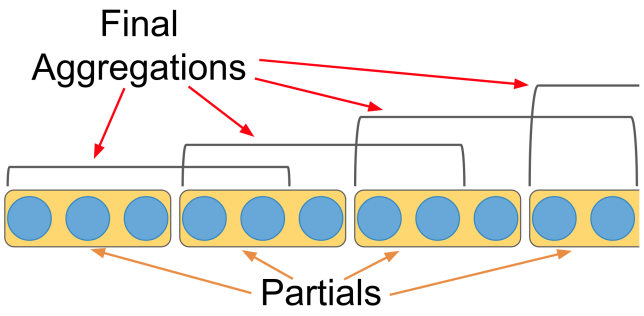

Figure 4: Cutty-slicing Technique (range=5 and slide=3)

Cutty-slicing was proposed as part of the *Cutty* optimizer [9]. The advantage of *Cutty-slicing* is that it starts each new partial only at positions that signify the beginning of new windows. This way the final aggregation can execute in the middle of the partial aggregation calculation by accessing the current value in the partial. An example of this is shown in Figure 4, where partials of size 3 tuples are maintained, and the final aggregator assembles the query answer from the current partial value (of 2 tuples) and the previous partial aggregate (of 3 tuples). This reduces the number of partials per window to $\lceil r/s \rceil$ in cases where the range is not divisible by the slide at the cost of requiring a more complicated implementation.

2.3.2 Final Aggregation

The goal of final aggregation is to produce ACQ results by assembling them from the partials. In this section we describe all of the available final aggregation techniques, and provide our analysis of their time and space complexities (summarized in Table 2).

Complexity Evaluation We evaluate each algorithm's time complexity in terms of the number of aggregate operations it performs per slide to return all query answers given a window size of n partial aggregates. This metric was chosen because the aggregate operations are (1) applied directly to the input data, (2) constitute the the bulk of all performed operations, and (3) their number correlates best with the actual query performance. In

order to cover the entire complexity space, we calculate **amortized** complexities as well as **worst-case** complexities. Amortized complexities are important to us because they correlate with ACQ processing throughputs, while worst-case ones reflect possible latency spikes.

In addition to providing calculations for a **single query** environment (where only one query covering the entire window is executed each slide), we also evaluate an MQ environment with the maximum number of queries (which we refer to as a **max-multi-query** environment). This way, a single query environment can be thought of as a **lower bound** of complexity per slide, while a max-multi-query environment (which executes all queries covering all possible ranges from 1 to the window length (n) each slide), can be thought of as the **upper bound**. It is clear that in most cases the complexity of the general case (with any other numbers of queries) lays between these bounds.

Panes [33] (which we consider to be *Naive* in this work) works by simply iterating over the partials and constructing the answer. The example in Figure 2 performs a final aggregation *F2* by iterating over partials *P2*, *P3*, *P4*, and *P5*. Naturally, such a solution quickly became outdated due to the increasing workloads that created bottlenecks in the final aggregator.

Panes has an exact time complexity (with matching amortized and worst cases) because

Table 2: Final Aggregation Complexities.

Algorithm	Time			Space	
	Single Query		Max-Multi	Single	Max-Multi
	Amort	Worst	Query	Query	Query
Panes	n	n	n^2	n	n
Panes(Inv)	2	2	—	n	—
FlatFAT	$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
B-Int	$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
TwoStacks	3	n	—	$2n$	—
DABA	5	8	—	$2n$	—

**true only when n is a power of 2, otherwise $3n$.

it always executes the same number of operations per slide. In a single query environment, its complexity is $n-1$ (asymptotically n) because it simply iterates over all n partials and aggregates them.

In a max-multi-query environment, *Panes* needs to return n answers each slide for ranges from 1 to n, yielding 0 to $n-1$ operations, respectively. By summing this arithmetic sequence we get $\frac{n^2}{2} - \frac{n}{2}$ (asymptotically n^2).

Panes has the space complexity of n since it stores partials only once and does not keep any additional structures. This complexity stands despite the number of registered queries, since additional queries do not require any additional structures.

Panes (Inv) [33] (or Panes for Invertible/Differential Aggregate Queries) was proposed at the same time as *Panes* to efficiently process invertible aggregates. In our taxonomy this is the only technique that does not allow processing non-invertible aggregations. It works by maintaining a running aggregate (e.g., running *sum*), and invoking the inverse operation (e.g., *subtract*) on every expiring tuple. This algorithm (with minor differences) was also proposed as *R-Int* [5] and *Subtract-on-Evict* [45].

Panes (Inv) has an exact time complexity of 2 operations per slide, since after each arrival of the new partial aggregate, the query answer is updated twice: once by executing an aggregate operation with the incoming partial, and once by executing the inverse operation with the expiring partial. This technique's space complexity is n, because it stores partials only once similarly to the *Panes* technique. Despite being very effective, *Panes (Inv)* is only applicable for invertible operations, and does not allow *MQ* processing.

FlatFAT [47] (or Flat Fixed-sized Aggregator) is a final aggregation technique which stores tuples in a pre-allocated, pointer-less, tree-based data structure (Figure 5). Originally, *FlatFAT* allowed only one tuple per leaf, however it was later extended [9] to perform partial aggregation by allowing it to store partial aggregates as tree leaves. Each internal node of the tree contains an aggregate of its two children. The root node has the result of the entire range allowed by the tree. In our experiments we compare our contributions to the improved version of *FlatFAT* [9].

New partials are inserted into the leaves of the binary tree left-to-right. The leaves form a circular array, meaning that after inserting a value to the rightmost leaf, the next insert

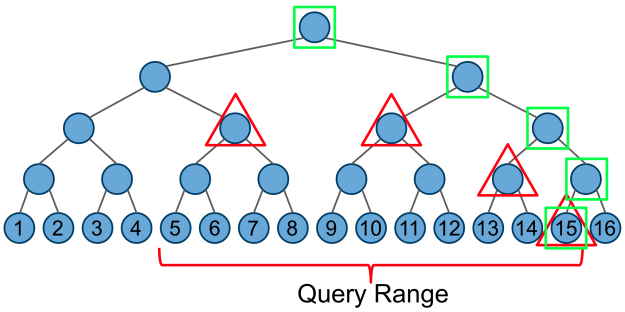

Figure 5: FlatFAT Technique

goes into the leftmost one. Each insert triggers the update procedure, which is performed by walking the tree bottom-up and updating all internal nodes with new aggregate values. An example of an update operation on leaf 15 is illustrated with green squares in Figure 5. The look-up of the answer in *FlatFAT* is performed by returning the root node value if a query requires the result for the maximum window, or by aggregating a minimum set of internal nodes that covers the required range of leaf nodes. The example of answering a query with a range of 11 partials starting from leaf 15 is shown with red triangles in Figure 5.

FlatFAT has an exact time complexity of $log_2(n)$ in a single query environment since each new partial updates the binary tree in a bottom-up fashion from the leaf to the root. Since the number of levels in a binary tree is $log_2(n) + 1$, *FlatFAT* needs exactly $log_2(n)$ operations to calculate the query answer. In a max-multi-query environment it is intuitive that the upper bound of the time complexity is $n \cdot log_2(n)$, since *FlatFAT* needs to iterate over n different query ranges at each slide and each range would require $log_2(n)$ operations at most to return the result. The exact complexity per slide can be produced by iterating over all possible ranges and summing their required numbers of operations, which equates to: $n \cdot log_2(n) - \frac{3n}{2} + \frac{5log_2(n)}{2} + \frac{5}{2}$. For simplicity, we use the asymptotic equivalent of this complexity: $n \cdot log(n)$.

FlatFAT has a space complexity of $2^{\lceil log(n) \rceil + 1}$. Due to its binary nature, it is more space efficient when the window size is a power of two, in which case it consumes $2n$ of memory: n for all leaf nodes and $n - 1$ for all tree nodes above the leaves. The first position within a

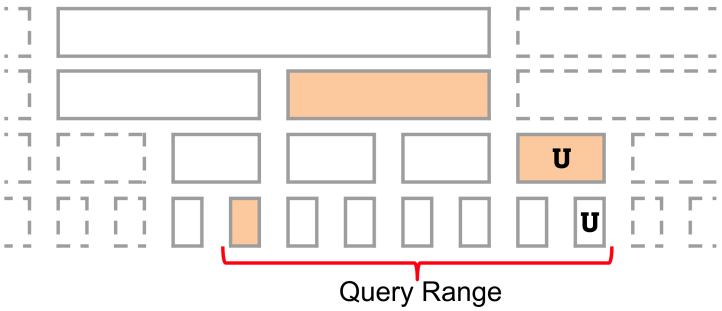

Figure 6: B-Int Technique

flat array normally remains unused in order to simplify the addressing of nodes within the tree. In cases where the window size is not a power of two, *FlatFAT* rounds it up to the closest power of two, which is mathematically expressed as: $2^{\lceil log(n) \rceil}$. Therefore, the space complexity of this algorithm yields $2^{\lceil log(n) \rceil + 1}$. The window rounding manifests the worst-case space complexity of $3n$.

B-Int [5] (or Base Intervals) is another final aggregation technique that uses a multi-level data structure that consists of dyadic intervals of different lengths. On the bottom level the interval length is one partial, on the next level the interval length is two partials, on the third level the length is four partials, and so on until we reach the top level that just has one interval of the maximum supported range length. The whole data structure is organized in a circular fashion so that the rightmost interval on any level is followed by the leftmost interval from the same level (Figure 6). The binary nature of this data structure makes it similar to *FlatFAT*, and like *FlatFAT*, when producing the final aggregate *B-Int* also determines the minimum number of intervals needed to represent the desired range and aggregates them. For example, in Figure 6 *B-Int* aggregates all marked intervals to get the answer for the specified query range. The algorithms for updates and look-ups are slightly different. During insertions, unlike *FlatFAT*, *B-Int* only updates the intervals that end with the inserted value instead of updating the entire structure bottom up until reaching the top layer. This, however, slows down look-ups since more intervals need to be aggregated to get the result.

B-Int, similarly to *FlatFAT*, is of a binary nature, and is only different in how it handles updates and look-ups. In [47] *B-Int* has been shown to have the same asymptotic time complexity as *FlatFAT*, with *B-Int* being slower by a constant factor, which we confirm in this work as well. Also, *B-Int* and *FlatFAT* have the same space complexity of $2^{\lceil log(n) \rceil + 1}$.

TwoStacks [45] was shown to also achieve high throughput by using an old trick from functional programming to implement a queue with two stacks, F (front) and B (back), where all insertions push a value *val* and an aggregation *agg* of everything below it onto B, and evictions pop from F. When F is empty, the algorithm flips B onto F, making it a calculation-heavy step that introduces latency spikes. To produce the final aggregation, the tops of both F and B stacks are aggregated.

TwoStacks executes different numbers of operations for different slides. During insertions, when each new partial is added to the B stack, one aggregate operation is performed to determine the new aggregate value of the entire stack B. After that, another operation is performed using the top values of both the F and B stacks to return the query answer, which makes the complexity of insertions 2 operations. The majority of evictions are free since they are done by just popping the node from the F stack. When F becomes empty, however, B is flipped onto F by popping values one-by-one from B and inserting them into F while performing one aggregate operation per insertion (to populate *agg* values on F). The flip procedure (n operations) clearly constitutes the worst-case complexity per slide. To calculate the amortized complexity we add all operations per one full iteration of the algorithm: n insertions (1 operation each), n queries (1 operation each), and one eviction that causes a stack flip procedure (n operations), totalling $3n$ operations per n slides. Thus, the amortized complexity of the algorithm is constant and equals 3 operations per slide. *TwoStacks* does not currently allow *MQ* processing, however it might be possible to extend it in the future to allow such functionality.

Since *TwoStacks* uses stack structures with nodes containing two values, and both stacks combined can never have more than n nodes total by the nature of the algorithm, its space complexity can be identified as $2n$.

DABA [45] (or De-Amortized Bankers Algorithm) was proposed as an alternative to *TwoStacks* that reduces the latency spikes while maintaining high throughput. The algo-

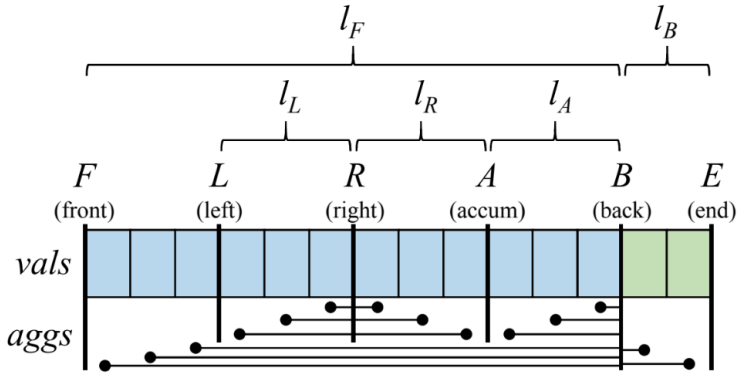

Figure 7: DABA Technique

rithm uses a principle of the Functional Okasaki Aggregator to de-amortize the *TwoStacks* algorithm. *DABA* uses two queues, *vals* and *aggs*, as shown in Figure 7 implemented as chunked-array queues with six ordered pointers which make up the F and B stacks similarly to *TwoStacks*. However after each insertion and eviction event, a function *fixup* is called which re-balances pointers and fixes the consistency of the *aggs* queue.

DABA has constant worst-case time complexity (though it still performs different numbers of operations each slide). To achieve that, *DABA* sacrificed its amortized time complexity (and consequently its throughput). Per one full window iteration, *DABA* executes 2 flip actions, n shift actions, and n evict actions (which all cost 0 operations), n shrink actions (costing 3 operations each), and also n insert actions and n answer look-up actions (cost 1 operation apiece), totalling $5n$ operations per n slides, which yields the amortized complexity of 5 operations. *DABA's* worst-case complexity can be attributed to a step that performs the following sequence of actions: Evict, Flip, Shrink, Insert, Shrink, Query, which costs 8 operations in total. *DABA* also does not currently support *MQ* processing, however it will also be interesting to see if it can be extended for this purpose, and what performance it would have.

Similarly to *TwoStacks*, *DABA* maintains the front and back stacks with nodes consisting of both values and aggregates, however it is implemented on top of the doubly linked list of chunks. The space complexity of *DABA* depends on the number of underlying chunks,

specifically, having less chunks that are bigger in size saves space on pointers (left and right), but wastes space on overallocations (periodically window slides between chunks during the execution leaving up to two chunks' worth of space wasted). If the window is split into k chunks, then *DABA's* space complexity is: $2n+4k+4n/k$. If we take a derivative with respect to k, equate it to zero, and solve for k, we conclude that the minimum space complexity for *DABA* is achieved by setting k to $\sqrt{n}$, and it equals $2n + 4\sqrt{n}$ (asymptotically $2n$).

2.4 Multi-Query Optimization

The general objective of *Multi-Query* (*MQ*) optimization is to reduce (or eliminate) the repeated processing of overlapping operations across multiple *ACQs* [39]. This repetition happens due to the processing of the same data items by different queries which exhibit an overlap in at least one of the following features: (1) predicate conditions, (2) group-by attributes, or (3) window specification. In this work we focus on optimizers targeting the window specification overlaps.

2.4.1 Shared Processing of ACQs

Since the *ACQs* are executed periodically (unlike one-shot, ad hoc queries), several processing schemes, as well as *ACQ* optimizers, take advantage of the shared processing of *ACQs* [31, 20, 9], which reduces the long-term overall processing costs by sharing partial results. To show the benefits of sharing in such scenarios, consider the following example:

Example 2 (Figure 8) Assume two *ACQs* monitor the *max* stock value over the same data stream. The first *ACQ* has a slide of 2 tuples and a range of 6 tuples, the second one has a slide of 4 tuples and a range of 8 tuples. That is, the first *ACQ* is computing partial aggregates every 2 tuples, and the second is computing the same partial aggregates every 4 tuples. Clearly the calculation producing partial aggregates only needs to be performed once every 2 tuples, and both *ACQs* can use these partial aggregates for their corresponding

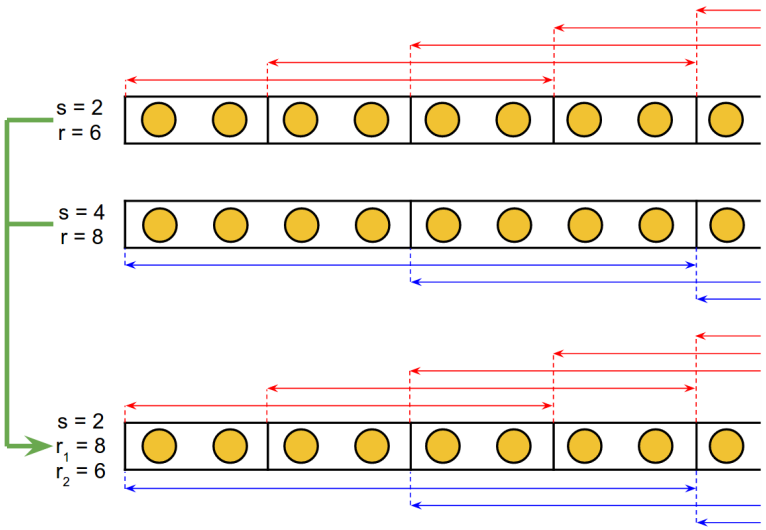

Figure 8: Shared Processing

final aggregations. The first ACQ then only needs to run each final aggregation over the last three partials, and the second over the last 4. ■

Partial results sharing is applicable for all matching aggregate operations, (e.g., three different $ACQs$ all calculating *max* can be processed in a single execution tree), and for different but compatible aggregate operations (e.g., three different $ACQs$ calculating *sum*, *count*, and *average* can be processed in two execution trees calculating *sum* and *count* by treating *average* as $\frac{sum}{count}$).

To determine how many partial aggregates are needed after combining n $ACQs$ into a shared execution plan, we first find the length of the new composite slide, which is the *Least Common Multiple* (LCM) of the slides of the combined $ACQs$ (in Example 2 it is four). Each slide is then repeated $LCM/slide$ times to fit the length of the composite slide, and all slide multiples are marked within the composite slide as *edges*. If slides consist of several fragments due to partial aggregation, all fragments are also marked within the composite slide as edges. If two or more $ACQs$ mark the same location, it means that location is a *common edge*. The more common edges that are present in the composite slide, the more partial aggregation sharing that can be performed.

Originally, the *Bit Set* technique [20] was used to determine how many partial aggregations (*edges*) are scheduled within the composite slide. This technique performs the counting of edges by traversing the entire composite slide, and thus is very inefficient. Later we proposed a more efficient mathematical solution to this problem, *Formula F1* [41] (described in Chapter 5).

2.4.2 Weavability

Out of all the *IE* techniques mentioned in Section 2.3, only *Panes*, *FlatFAT*, *B-Int*, *FlatFIT*, and *SlickDeque* are known to support *MQ* execution. These techniques share partial aggregates among all of the registered queries (i.e., all the queries are merged and processed as a single execution tree), thus achieving maximum sharing. However, it was shown that this is not always beneficial, and *selective sharing* achieves better performance by splitting the query load into multiple execution trees carefully.

Weavability [20] is a metric that measures the benefit of sharing partial aggregations between any number of *ACQs*. If it is beneficial to share computations between these *ACQs*, then these *ACQs* are known to *weave* well together and are combined into the same shared execution tree. Intuitively, two *ACQs weave* perfectly when their *LCM* contains only *common edges*.

The following formula can be used to calculate the cost (C) of the execution plan before and after combining *ACQs* into shared trees so that the difference between these costs tells us if the combination is beneficial:

$$C = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{2.1}$$

where m is the number of the trees in the plan, λ is input rate in tuples per second, E_i is *Edge rate* of tree i (the number of partial aggregations performed per second), and Ω_i is the total number of final-aggregation operations performed per edge of tree i.

The *WeaveShare* [20] and *TriWeave* [19] *MQ* optimizers both utilize the concept of *Weavability* to produce execution plans for sets of input *ACQs*. The *TriWeave* optimizer was proposed as a part of a more general state-of-the-art *TriOps* [19] optimizer, which besides targeting window specifications (using *TriWeave*), also targets predicate conditions and group-by

attributes. As pointed out above, the predicate and group-by optimizations are considered in this work as being orthogonal, although we proposed an improvement in the predicate optimization by intelligently generating fragment-signature pairs as part of our initial investigations [26].

Both *WeaveShare* and *TriWeave* optimizers selectively partition *ACQs* into multiple disjointed execution trees (i.e., groups), resulting in a dramatic reduction in the total query plan processing cost, and are theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem [12]. Both *WeaveShare* and *TriWeave* start with a no-share plan, where each *ACQ* has its own execution tree. Then they iteratively consider all possible pairs of execution trees and combine those that reduce the total plan cost the most into a single tree, and produce final execution plans consisting of multiple disjointed execution trees when they cannot find another pair that would reduce the total plan cost further. The difference between *WeaveShare* and *TriWeave* is that the former assumes separate partial aggregation processing on each execution tree, while the latter assumes combined partial aggregation processing using a large composite slide that passes ready partials to the execution trees.

2.5 Other Related Work

Work similar to sliding-window aggregation existed in *Temporal Database Systems* long before *DSMSs* came around. Such systems store the entire stream of tuples and allow aggregations over any continuous segments of the stream which are called *Historical Windows*. Conversely, *DSMSs* (which we focus on in this work) generally only support *Suffix Windows*, which end at or near the most recent results. In the context of Temporal Databases, [35] utilized red-black trees for aggregations and [54] used SB-trees, which incorporate features from both segment-trees and B-trees. Due to the tree-based nature of these algorithms, their update complexities are $O(log(s))$, where s is the size of the entire stream history over which they build their structures. Additionally, they do not allow non-invertible aggregations, which significantly restricts their operation generality.

In order to save time and space by sacrificing accuracy, the following approximate calculation approaches were proposed: [7, 4, 13, 16]. Our approach focuses solely on computing exact answers since it is crucial for many applications (i.e., financial, medical, etc.).

Under the *MQ* optimization techniques, the general principle is to minimize (or eliminate) the repeated processing of overlapping operations across multiple aggregate queries. This repetition occurs as a result of processing the same data by different queries, which exhibit an overlap in at least one of the following specifications: 1) predicate conditions, 2) group-by attributes, or 3) window settings.

Techniques leveraging the overlaps in predicate conditions and group-by attributes across different *ACQs* are similar to classical multi-query optimization [38] that detects common subexpressions. Techniques leveraging shared processing of overlapping windows across different *ACQs* emerged with the paradigm shift for handling continuous queries. The *shared time slices* technique [31], for example, has been proposed to share the processing of multiple continuous aggregates with varying windows. It has also been extended into *shared data shards* in order to share the processing of varying predicates, in addition to varying windows. Orthogonally, [36] extends classical, subsumption-based, multi-query optimization techniques towards sharing the processing of multiple *ACQs* with varying group-by attributes and similar windows.

2.6 Summary

In this chapter we reviewed the related work in *SWAG* processing and proposed a taxonomy of all available *IE* techniques that can be broken down into into partial and final aggregation techniques. Certainly, it is crucial to perform both partial and final aggregations efficiently.

For partial aggregation we conclude that in the setting where query ranges are divisible by their corresponding slides, all three existing partial aggregation techniques perform the same, otherwise the *Cutty-slicing* technique achieves the best results. The comparison of the partial aggregation techniques is summarized in Table 1.

We break down the existing final aggregation techniques further into *Naive*, *Tree-based*, *Throughput Optimized*, and *Latency Optimized* approaches, and analyze their operational complexities (summarized in Table 2). Since there is clearly room for improvement in final aggregation, in this dissertation we contribute two new techniques: *FlatFIT* (Chapter 3) and *SlickDeque* (Chapter 4).

Additionally, we summarized the related work in *MQ* optimization since it is the next logical step for further improving *SWAG*. We described the cost formula (Equation 7.1), which is the foundation of our improvements to *MQ* optimization in Chapters 5, 6, and 7.

3.0 FlatFIT: Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics

At the time of writing this chapter the state-of-the-art *Incremental Evaluation* (*IE*) technique was *FlatFAT*, which executes *ACQs* with high efficiency, but does not scale well with the increasing workloads. In this chapter we present our novel algorithm, *FlatFIT*, that accelerates such calculations by intelligently maintaining index structures, leading to higher reuse of intermediate calculations and thus improved scalability in systems with heavy workloads.

In the next section we outline the problems with the state-of-the-art approach at the time. In Section 3.2 we introduce our new technique, *FlatFIT* for the final aggregation calculations. The complexity analysis of *FlatFIT* and compared algorithms is presented in Section 3.3. We discuss the evaluation platform and the experiments in Section 3.4 and conclude in Section 3.5.

3.1 Introduction

Efficient handling of aggregate operations that are *non-invertible* and *non-commutative* proved to be essential in calculation heavy domains such as finance and science. Examples include Max, Min, Concatenate, First N, Last N, CountDistinct, CollectDistinct, ArgMax, and ArgMin.

This chapter focuses on such non-invertible or non-commutative operations that are heavily used in practical *ACQs*. We consider both *Single Query* environments where each *ACQ* executes in isolation, for example for privacy reasons, and *Multi-Query* (*MQ*) environments, where a large number of *ACQs* with different periodic properties (accuracies) are operating on the same data stream, calculating similar aggregate operations.

The Reactive Aggregator framework was proposed to efficiently processing these kinds of workloads. The framework was implemented using the *Flat Fixed-sized Aggregator* (also

known as *FlatFAT*) [47]. *FlatFAT* is able to achieve high throughput by utilizing a pre-allocated memory circular tree-based data structure, however it does not scale well with heavy workloads. Additionally, a new system, *Cutty* [9], was proposed that utilizes *FlatFAT* in a *MQ* environment and contributes a novel slicing technique (referred to as *Cutty-slicing* in this dissertation) for partitioning the incoming tuples. However it does not improve the main query processing technique which is *FlatFAT*.

To address the aforementioned shortcomings, in this chapter we propose a novel solution named *Flat and Fast Index Traverser*, or simply *FlatFIT*, which accelerates the processing of *ACQs* by significantly speeding up the final aggregation operation of incremental sliding-window evaluation techniques. *FlatFIT* achieves this acceleration by maintaining intermediate aggregates in intelligent indexing structures that reduce the number of partials used in performing a final aggregation and allows a greater level of reuse of previously calculated results. We show both theoretically and experimentally that our approach allows better scalability in terms of window size, and it becomes advantageous to utilize *FlatFIT* over *FlatFAT* starting with windows of a size as small as eight tuples (or partials in cases when partial aggregation techniques are used).

3.2 FlatFIT Operation

In this section we describe our new algorithm, *FlatFIT*, that significantly speeds up the final aggregation calculations in a sliding-window environment.

3.2.1 The FlatFIT Algorithm

In this subsection we provide the algorithm and implementation details for our approach followed by two clarifying examples. We target single query and multi-query (*MQ*) environments, though single query can be considered a special case of *MQ* processing.

Intuition and Data Structures The *FlatFIT* algorithm works by dynamically storing the intermediate results and their corresponding pointers indicating how far ahead *FlatFIT* can skip in its calculation. It uses two circular arrays *Pointers* and *Partials* interconnected

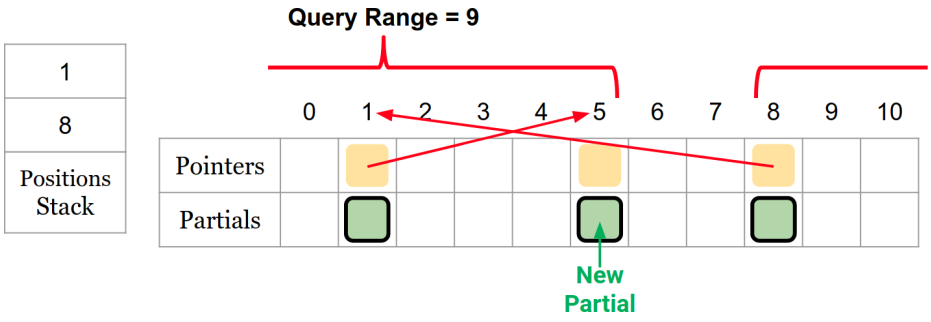

Figure 9: FlatFIT Technique

with their indices and stack *Positions*, which is used to store the indices that are currently processed. The *Pointers* and *Partials* arrays can be thought of as a single weighted jump table that allows *FlatFIT* to skip to the position stored in the *Pointers* array while adding the corresponding value from the *Partials* array to the running aggregate value.

A simple example of update and look-up operations at position (or index) 5 is illustrated in Figure 9. To process a query with a range of 9 partials at this position, *FlatFIT* follows the *Pointers* from position 8 to the starting position 5, and pushes visited positions (8 and 1) on the *Positions* stack. Once position 5 is reached, all the *Partials* from the stored *Positions* are aggregated to return the final answer. This way, the FlatFIT algorithm avoids costly and unnecessary (re)computations and enables a higher reuse of the intermediate results than previous methods. The full pseudocode for the *FlatFIT* algorithm is depicted in Algorithm 1 and consists of the *Preparation* and *Execution* phases.

The Preparation Phase given a set of queries Q and one of the partial aggregation techniques discussed in Section 2.3.1 (i.e., *Pairs*) as an input, starts by building a shared execution plan by executing the *BuildSharedPlan* function (line 5). The *sharedPlan* is constructed as discussed in Section 2.4.1, and it includes a full list of partials (or edges) augmented with their lengths and lists of queries that need to be evaluated at each partial. The *BuildSharedPlan* function identifies the query with the longest range in terms of the number of partials, and saves this range as the member *wSize* of the produced *sharedPlan*. *wSize* signifies the necessary window length needed to process all input queries.

Algorithm 1 FlatFIT Pseudocode

```
1: Input: A set of aggregate continuous queries Q, aggregate operation ⊕, the initial value for ⊕ initVal,
   and partial aggregation technique PAT
2: Output: Continuous answers to queries in Q according to their specifications.
3:
4:                    Phase 1 (Preparation)
5: sharedPlan = BuildSharedPlan(Q, PAT)
6: wSize = sharedPlan.wSize
7: Partials = new array[wSize]
8: Pointers = new array[wSize]
9: Positions = new stack()
10: for i=0 to wSize do
11:     Partials[i] = initVal
12:     Pointers[i] = i + 1
13: end for
14: Pointers[wSize - 1] = 0
15: currInd = 0
16: prevInd = wSize - 1
17:
18:                   Phase 2 (Execution)
19: while results are expected do
20:     length = sharedPlan.getNextPartialsLength()
21:     newPartial = PartialAggregator.aggregate(length, PAT)
22:     Partials[prevInd] = newPartial
23:     Pointers[prevInd] = currInd
24:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
25:     for each query q in queriesToAnswer do
26:         startInd = currInd - q.range
27:         if startInd < 0 then
28:             startInd += wSize
29:         end if
30:         do
31:             Positions.push(startInd)
32:             startInd = Pointers[startInd]
33:         while startInd != currInd
34:         end do while
35:         answer = Partials[Positions.pop()]
36:         while Positions.size() > 1 do
37:             tempInd = Positions.pop()
38:             answer = answer ⊕ Partials[tempInd]
39:             Partials[tempInd] = answer
40:             Pointers[tempInd] = currInd
41:         end while
42:         tempInd = Positions.pop()
43:         answer = answer ⊕ Partials[tempInd]
44:         send (answer)
45:     end for
46:     prevInd = currInd
47:     currInd++
48:     if currInd == windowSize then
49:         currInd = 0
50:     end if
51: end while
```

After generating the *sharedPlan*, *FlatFIT* initializes the data structures (lines 7-14). The two circular arrays are both initialized to a length equal to $wSize$. The *Positions* stack is initialized empty and can expand up to $wSize$ – however normally it is much less (refer to Section 3.2.2). The *Partials* array is initially filled with the initial value *initVal* for the query operation $\oplus$ supplied as input. For example, *initVal* is 0 for the Sum operation or $-\infty$ for the Max operation. Each value in the *Pointers* array is initialized to point to the next consecutive value in it (i.e., $Pointers[2]$ is 3, and $Pointers[wSize-1]$ is 0, since it is a circular array).

The *currInd* variable signifies the current position within the two arrays (line 15). It starts at 0 initially and increases to $wSize-1$ during execution, after which it wraps back to 0. The arriving partial aggregates will be inserted into the *Partials* array always at the index previous to the *currInd*, referred to as *prevInd* (line 16).

The Execution Phase is implemented as a loop that continuously returns all the query results while they are expected. At the beginning of the loop (lines 20-24), *FlatFIT* gets the next partial's length from the *sharedPlan*, and supplies it to our *Partial Aggregator* which uses the provided *PAT* technique to produce the *newPartial* value. The *newParial* is then inserted into the *Partials* array at *prevInd*, and $Pointers[prevInd]$ is updated to point to the *currInd*. Now, the answers to all queries scheduled at this position need to be produced.

After receiving the *queriesToAnswer* from the *sharedPlan* (which is a subset of Q), *FlatFIT* loops over these queries to answer them. The loop starts by identifying the start index *startInd* for each query q (lines 26-29) within the two arrays from which it will start aggregating values. *startInd* is identified by rewinding *currInd* back by q's range length.

Once the *startInd* of q has been determined, our algorithm traverses the *Pointers* array while pushing all visited indices onto the *Positions* stack in a do-while loop until it reaches the *currInd* again (lines 30-34). Then, in order to construct the final aggregation *FlatFIT* needs to access the *Partials* array at all these indices, and at the same time update the values in the *Partials* array to be reused in the future.

Towards this (lines 35-44), *FlatFIT* first initializes the *answer* variable to the value found in the *Partials* array at the index popped from the top of the *Positions* stack. It

then continues by popping all the indices except for the last one from the $Position$ stack in a loop and saving them as a $tempInd$. The values found at the $tempInd$ indices in the $Partials$ array are aggregated with the $answer$ variable using the aggregate operation $\oplus$ supplied as an input. Each time a new partial is aggregated, $FlatFIT$ also writes the current value of the $answer$ into the $Partials$ array at $tempInd$, and copies the $currInd$ into the $Pointers$ array also at $tempInd$. This technique allows $FlatFIT$ to later skip from $tempInd$ to $currInd$ by doing just one aggregate operation. The last index popped from the $Positions$ stack is also used to retrieve the corresponding partial from the $Partials$ array and is aggregated to the $answer$, however it does not need to update the two arrays because it will be overwritten in the next iteration of the execution phase with the new partial.

Observations. Notice that the more queries with different ranges that are registered on the datastream, the more result reusing is performed by $FlatFIT$. In cases where the number of queries registered on the datastream is small, large parts of the $Pointers$ and $Partitions$ arrays might be visited and updated by $FlatFIT$ on certain slides (not more frequently than once per $wSize$), which enables fast calculations on the rest of the window.

The least amount of calculation reuse for the $FlatFIT$ algorithm happens in a single query environment, since once per $wSize + 1$ all indices are visited and pushed onto the $Positions$ stack, which then causes an update on almost the entire window. In this work, we refer to this event as $wReset$. $wReset$ also happens as the first iteration of the execution phase in any environment regardless of how many queries are registered on the datastream. Even though a single query environment turns out to require the most computation for $FlatFIT$, it still significantly outperforms all competitors including the $FlatFAT$ technique. This stands because despite $wReset$ being a heavy calculation part, it only happens once per $wSize + 1$ and it enables $FlatFIT$ to reuse calculated partials efficiently during the rest of the execution.

The following Examples 3 and 4 (illustrated in Figure 10) should clarify the above algorithm. In order to make the explanation more intuitive we execute the two queries $Q1$ and $Q2$ on the same incoming datastream using two algorithms: $Panes$ and $FlatFIT$, and we illustrate each step of their calculations side-by-side.

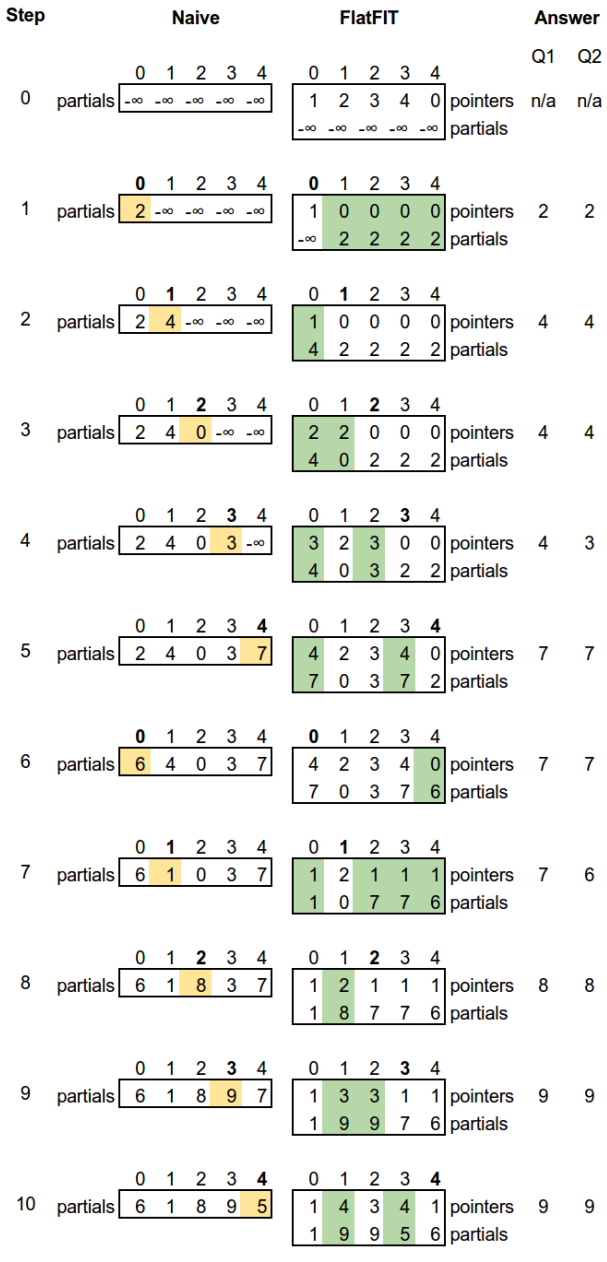

Figure 10: Example of Panes and FlatFIT algorithms working in a Single Query Environment (processing just Q1) and in a Multi-Query Environment (processing both Q1 and Q2)

Example 3 (Single query environment). Assume we have just one query $Q1$ which is seeking the Max value over the range of 5 tuples with a slide of 1 tuple. The slide size is set to one tuple in this example for simplicity, which means that there is no partial aggregation and the answer to Max needs to be calculated after every new tuple arrival. A shared execution plan is not needed in this example since we only have one query, which makes our window size ($wSize$) equal to the range of $Q1$ (5 tuples).

In both the *Panes* and *FlatFIT* representations we mark the positions that have been modified by the algorithms in each step. The *Positions* stack involved in the *FlatFIT* calculation is not illustrated here, however its contents in each step are clear since we know that all indices that are modified in that step were pushed onto the *Positions* stack and then popped back off. The current index ($currInd$) at each step is bolded in Figure 10 for convenience. The tuples enter the system in the order: 2, 4, 0, 3, 7, 6, 1, 8, 9, 5.

After the initialization in Step 0, in Step 1 the first tuple, 2, arrives. The *Panes* algorithm stores the new tuple at the current index in its own *Partials* array, and it executes a full iteration over the entire array in order to find the Max value, which in this case is 2.

FlatFIT writes the first tuple, 2, to the *Partials* array at the previous-to-the-current index, which in this case is 4 (we refer to this index as $prevInd$). Now the algorithms have to make a full circle over the *Pointers* array because in a single query environment, the start index ($startInd$) for the query is always equal to the $currInd$. By the nature of the *FlatFIT* operation discussed above, Step 1 always triggers the $wReset$ event (the update of the whole window except for the current index) because the *Pointers* at each index are pointing to the next index after the initialization, and *FlatFIT* is unable to skip any positions while producing the result. This way, all indices are pushed onto the *Positions* stack and subsequently popped to construct the answer from the partials at those indices, while also updating the arrays for future use. Thus, all indices (except the $currInd$) are pointing now to index 0 and their corresponding values in the *Partials* array are set to 2.

In Step 2, the *Panes* algorithm places the new partial, 4, into the current index and iterates again over the whole window comparing every value in order to get the Max value (which now is 4). Our *FlatFIT* algorithm is able to provide the answer to $Q1$ here with just one Max comparison. From the start index, 1, it skips to index 0 (since $Pointers[1]$ is

0) and then again to index 1 since $Pointers[0]$ is 1. The answer is then computed by taking the Max of $Partials[1]$ and $Partials[0]$, which is 4, and it is then stored in $Partials[0]$.

In Step 3 $FlatFIT$ updates index 1 with the new tuple, 0, and it is able to make a full circle from the $currInd$, 2, back to itself by visiting intermediate indices 0 and 1, after which just index 0 was updated for future use.

In Steps 4, and 5 (and later 9) $FlatFIT$ is able to get the answers in just two Max comparisons similarly to Step 3, and in Steps 6 and 8 it takes just one comparison similarly to Step 2, while $Panes$ did 4 comparisons at each and every step. Step 7 forced $FlatFIT$ to execute 4 comparisons similarly to $Panes$ because the $wReset$ event happens at this step.

In a single query environment the $wReset$ happens on the first inserted partial and then repeats periodically every $wSize + 1$ slides. Since the period is greater than $wSize$ by one, the start position of the $wReset$ operation keeps shifting right by one every cycle. ■

Notice that this small example highlights the benefit of using $FlatFIT$ over $Panes$ by showing that $Panes$ had to execute 40 Max comparisons total to process $Q1$, while $FlatFIT$ executed just 21.

Example 4 (Multi-query environment). In this example we illustrate how $FlatFIT$ works in a MQ environment by augmenting Example 3 with one more query, $Q2$. The new query, $Q2$, is also seeking the Max value and has a slide of 1 tuple, however its range is 2 tuples. Thus, $Panes$ and $FlatFIT$ will need to answer both queries at every step. Since the range of $Q1$ is 5, which is greater than the range of $Q2$, and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a $wSize$ of 5 tuples.

The $Panes$ algorithm in this case does a full loop over the entire array in order to answer $Q1$ each time, and then iterates over the most recent two partials to produce the answer for $Q2$, and this process is repeated at every step.

Conversely, $FlatFIT$, after iterating over the whole structure in Step 1 to produce the answer for query $Q1$, is able to generate the answer for $Q2$ with 0 comparisons, just by calculating the start index, $startInd$, for $Q2$ (which is 3) and reading the answer from the $Partials$ array at this index (since the $Pointers$ array at this index points us directly back at the $currInd$). Similar behavior for calculating the answer for $Q2$ in 0 comparisons can be also found in Steps 3, 7, and 9.

In Step 2, our $FlatFIT$ algorithm calculates the answer for $Q1$ just by doing one comparison (explained in Example 3), and produces the answer for $Q2$ by executing also just one Max comparison (of $Partials[4]$ and $Partials[0]$). Similarly to this step, $FlatFIT$ calculated the answers for query, $Q2$, in just one comparison also in Steps 4, 5, 6, 8, and 10. ■

Notice that even for query, $Q2$, with range as small as 2 tuples, $FlatFIT$ needed just 6 comparisons for the entire example, while $Panes$ had to perform 10. It is intuitive that with increasing query numbers and their ranges, $FlatFIT$ allows much better scalability. Later in this chapter this intuition is backed up by both theoretical analysis (Section 3.3) and experimental evaluation (Section 3.4).

3.2.2 Optimization

In order to reduce memory consumption by the $Positions$ stack in a single query environment we made the following observation: the stack fills up to $wSize - 1$ only during the $wReset$ event, otherwise it can hold up to 2 values at most. In fact, the usage of the $Positions$ stack repeats with period $wSize + 1$ and it always contains $wSize - 1$ entries at the first step of each cycle, one entry at the second and the last entries of the cycle, and two entries in the rest of the $wSize - 2$ steps. This means that the amount of memory consumed by the $Positions$ stack can be reduced from $wSize - 1$ to 2 by implementing the $wReset$ operation manually without using the stack.

In our optimized $wReset$ function, we initialize the $answer$ variable to the initial value, $initVal$, for the query operation $\oplus$, and iterate over both the $Partials$ and $Pointers$ arrays of $FlatFIT$ backwards from $prevInd$ until $currInd$ is reached. At each iteration the value from the $Partials$ array is aggregated into the $answer$ variable, and the current value of the $answer$ variable is written back to the $Partials$ array. The $Pointers$ array is updated to point to the $currInd$ at each iteration. After the traversal is finished, the value from the $answer$ variable is returned, and both arrays of $FlatFIT$ are updated and ready to continue executing the main algorithm.

This manual $wReset$ function is triggered periodically every $wSize + 1$ slides in a single query environment, and triggered just once at the beginning of the execution phase of multi-

Table 3: Final Aggregation Complexities (Complexities of the existing techniques are derived in Section 2.3.2).

Algorithm	Time			Space	
	Single Query		Max-Multi Query	Single Query	Max-Multi Query
	Amort	Worst			
Panes	n	n	n^2	n	n
FlatFAT	$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
B-Int	$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
FlatFIT	3	n	$3n$	$2n$	$2n$

**true only when n is a power of 2, otherwise $3n$.

query environments. The full implications of this optimization on an algorithm's space complexity can be found in Section 3.3.2.

3.3 Complexity Analysis

In this section, we calculate the time and space complexities of *FlatFIT* and summarized them in Table 3 in comparison with other general final aggregation techniques available at that time. The theoretical time complexities of *FlatFIT* and the compared algorithms (Section 2.3.2) are illustrated in Figures 11 & 13, theoretical throughputs in Figures 12 & 14, and theoretical memory consumptions in Figure 15.

3.3.1 Time Complexity of FlatFIT

When executed in a single query environment *FlatFIT* can be observed to execute different numbers of operations for different slides to produce the answer, however the numbers of operations follow a certain cyclical pattern which repeats every $wSize + 1$ slides.

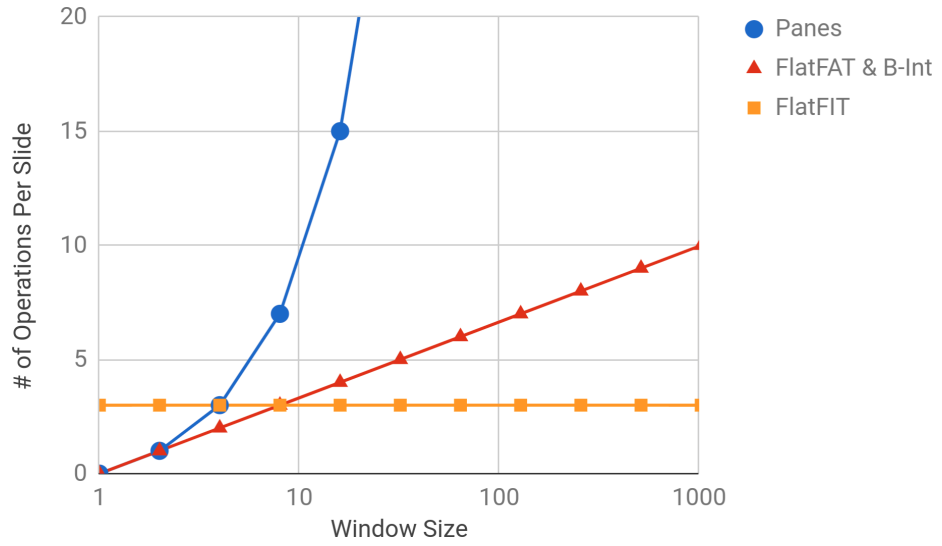

Figure 11: Theoretical operations per slide in a single query environment

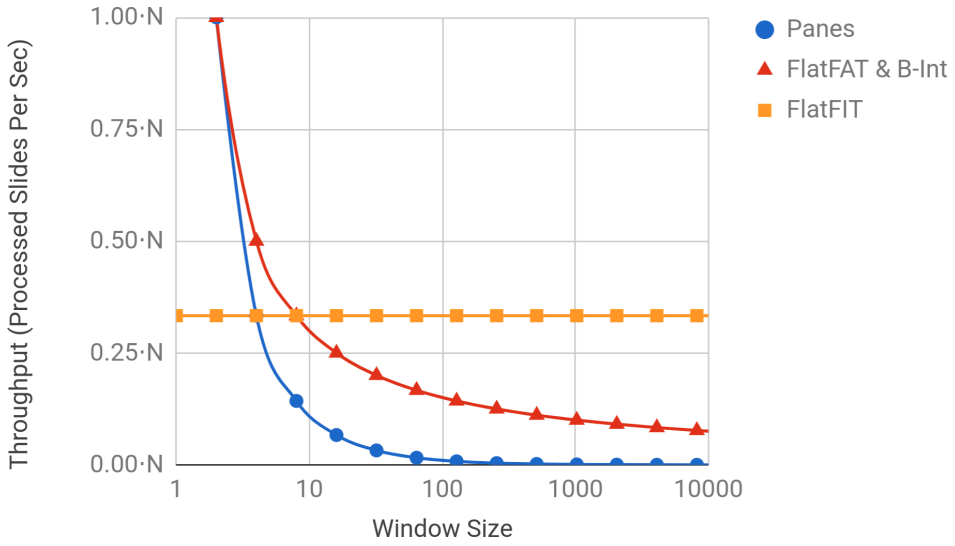

Figure 12: Theoretical throughput in a single query environment running N operations per second

In a single query environment the *wReset* event happens once per period. Its operational complexity with or without the optimization explained in Section 3.2.2 is $n-1$ operations. *wReset* is surrounded by two slides that require just 1 operation, and the rest of the slides

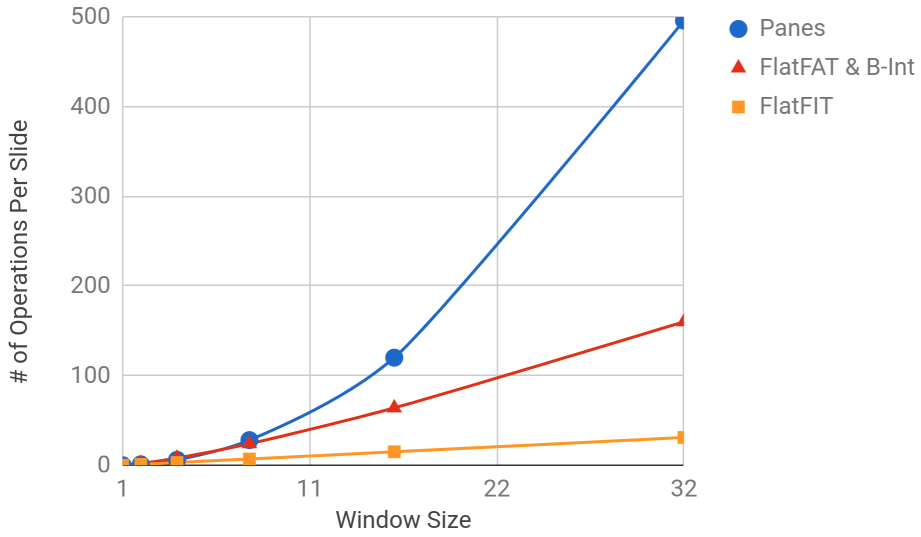

Figure 13: Theoretical operations per slide in a max-multi-query environment

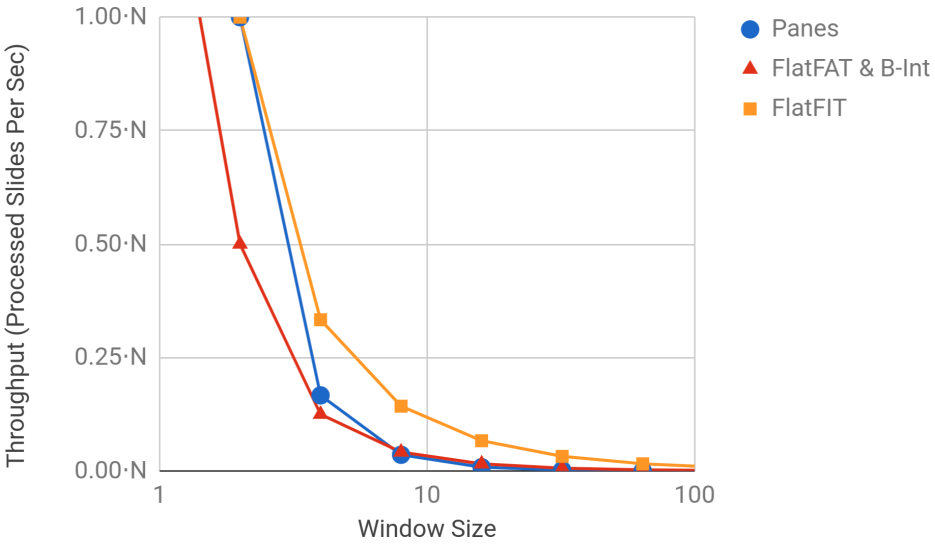

Figure 14: Theoretical throughput in a max-multi-query environment running N operations per second

$(n-1)$ in a period require two operations each. Therefore, by summing everything, we have the complexity for the natural period of $FlatFIT$: $(n-1)+2(n-2)+2 = 3(n-1)$. Since the above complexity is calculated for a segment of $n+1$ slides, for a fair comparison with

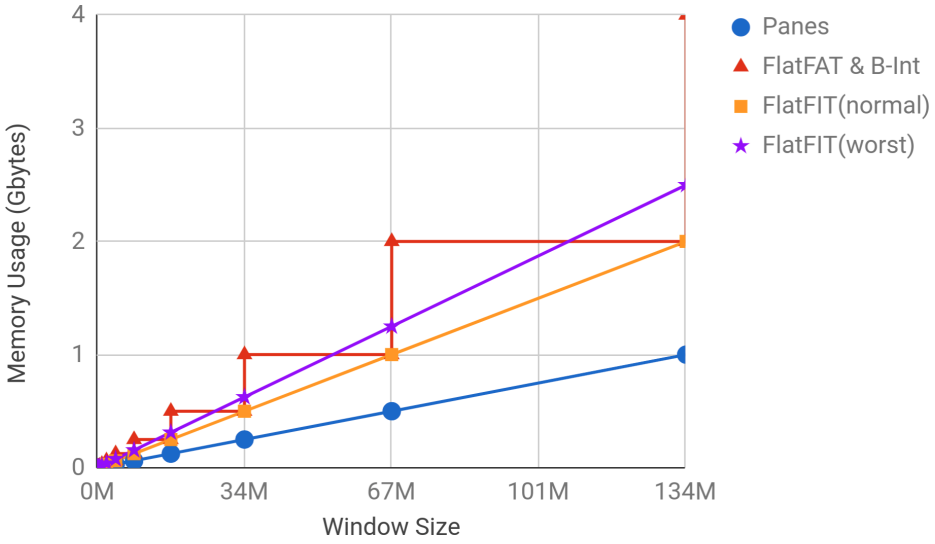

Figure 15: Theoretical Memory Usage in GB increments

other approaches, we need to convert this complexity to the period of length n. To do that we multiply the above equation by n and divide by $(n-1)$, which results in $3n$ operations for the segment of n slides, which in turn makes our complexity equal to just 3 operations per slide and is asymptotically *constant*.

In a max-multi-query environment, $FlatFIT$ updates all indices at each slide by answering queries of all possible ranges, which allows it to keep the data structure maximally updated. In this scenario the $wReset$ event happens only once at the beginning of the execution phase and is never triggered again, since the algorithm keeps all of the indices updated at all times. Due to this, at each slide $FlatFIT$ is still able to return answers to all queries in just 3 operations on average, making its operational complexity $3n$ operations per slide.

To summarize, $FlatFIT$ is superior in time complexity in comparison with the algorithms existing at the time of developing $FlatFIT$ (See Figures 11 - 14 and Table 3).

3.3.2 Space Complexity of FlatFIT

FlatFIT needs two pre-allocated arrays of size n to operate and a stack that can grow up to n in size, however after introducing the optimization (in Section 3.2.2) in a single query

environment, it cannot contain more than two values. In the max-multi-query environment the stack can contain even less: just one value at max without regard to the size of n. This makes asymptotic space complexity of $FlatFIT$ $2n$. However, in terms of space complexity, single query and max-multi-query environments do not bound $FlatFIT$. In a general case where we have more than one query and less than maximum queries registered, the stack might have to store up to $n/2$ values at most, in the case with just two queries. However, each additional query (of a different range) after that cuts the maximum stack memory consumption in half by enabling higher reuse of calculations. Therefore, if the number of queries is q, the space complexity of $FlatFIT$ becomes $2n$ for $q = 1$ and $q = n$, and $2n + \frac{n}{2^{q-1}}$ for the rest of the possible values of q.

To summarize, the existing $Panes$ algorithm is superior to $FlatFIT$ in space complexity, however it is clearly not feasible for heavy workloads. $FlatFIT$ offers the next best space complexity while being the most scalable solution in terms of time complexity out of all the algorithms existing at the time of developing $FlatFIT$ (See Figure 15 and Table 3).

3.4 Experimental Evaluation

In this section, we present our experimental evaluation that confirms that the theoretical advantage of $FlatFIT$ stands true in practice compared to other final aggregation approaches.

3.4.1 Experimental Testbed

Platform In order to test the performance of our sliding-window aggregation technique, we we built an experimental platform in C++ (compiled with G++5.4.1). Specifically, we implemented a stand-alone stream aggregator platform and programmed all of the compared *IE* algorithms within the same codebase, sharing data structures and function calls to enable a fair comparison. Although all of these algorithms can be easily ported to any commercial general purpose stream processing system, we chose to go with a stand-alone platform to

carry out our evaluation in an isolated environment in order to avoid any potential system interference and overheads.

Dataset We utilized the DEBS12 Grand Challenge Dataset [24], which is widely utilized in the workload-based evaluations like ours [27, 28, 25, 9]. The dataset contains events generated by sensors of large hi-tech manufacturing equipment. Each tuple in this dataset incorporates 3 energy readings (stored as 32 bit integers) and 51 (predominantly boolean) values signifying various sensor states. The records were sampled at the rate of 100Hz, and the whole dataset includes ~33 million events, which we separated into 3 datasets by copying one energy reading per tuple into a separate file while discarding the sensor state values. For each of the experiments we loaded each dataset into main memory before running aggregations on it. In cases when all the values in the set were processed, but we still needed to continue execution, we continued processing from the beginning of the set, i.e., circling back.

Workload Clearly, the performance of the final aggregation techniques heavily depends on the window size, i.e., the larger the window size the longer it takes to process updates to it. Thus, we used tuple-based windows where we varied the window size from 1 to 134 million tuples. Given that the goal of our evaluation is just to compare different final aggregation techniques, we eliminated any side effects (i.e., overheads or benefits) induced by partial aggregation by setting all query slides to one tuple.

Evaluation Metrics We chose to compare the algorithms using throughput and memory requirement. *Throughput* is measured as the number of query results returned per second in a single query environment, while in a multi-query environment it is measured as the number of slides of a shared execution plan processed per second. We calculated it by running each compared algorithm on each dataset for 10 minutes at the fastest possible rate while computing the total numbers of returned results and processed slides, and at the end divided them by 600 to get the results per second. *Memory Requirement* is measured by the maximum resident set size of processes running the corresponding techniques, which we calculated using Linux's */usr/bin/time* utility.

For our aggregations we chose a *distributive* operation, Max, as opposed to an *algebraic* operation like Mean (which is decomposed into Count and Sum for processing) in order to

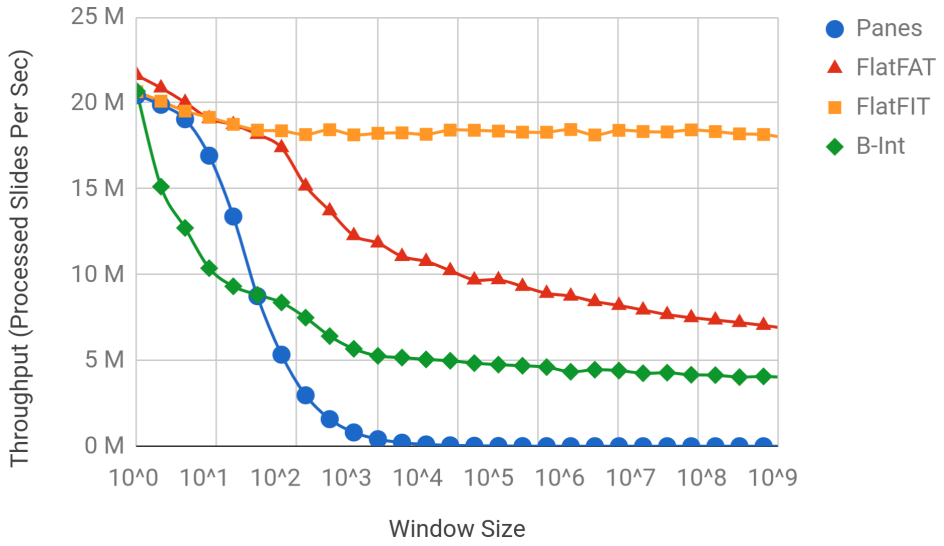

Figure 16: Throughput in processed slides per second in single query environment

benchmark the algorithms more accurately. Additionally, Max is a non-invertible operation that illustrates generality of the algorithms.

System We ran our experiments on an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz machine with 16 GB of RAM. For robustness, all experimental results are taken as averages of three independent runs of each experiment aggregating three different energy readings from the DEBS12 dataset.

3.4.2 Experimental Results

3.4.2.1 Exp 1: Single Query Throughput (Figure 16)

In this test we varied the window size from 1 tuple to 134 million tuples where each window is a power of two, and ran a query calculating Max over the entire window after each new tuple arrival. Clearly, increasing window size increases the amount of required calculations causing lower throughputs for all four algorithms. The results are depicted in Figure 16. Notice that the rates at which throughput decreases are very similar to what we expected from the theoretical analysis of the algorithms (Figure 12).

Our statistical calculations show that *FlatFIT*'s throughputs are on average 1.8 times

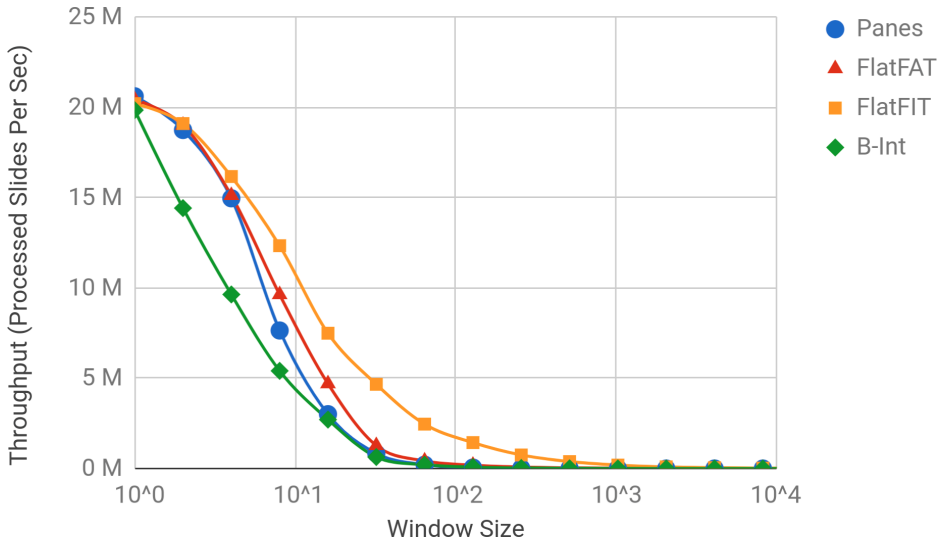

Figure 17: Throughput in processed slides per second in max-multi-query environment

higher than throughputs of $FlatFAT$ with a maximum of 2.6 times. We also observed that $FlatFIT$ starts outperforming $FlatFAT$ on windows as small as 8 tuples and increases its gain on the rest of the algorithms rapidly. $FlatFAT$ showed to be more beneficial than $FlatFIT$ only on window sizes from 1 to 4 tuples, however this benefit is negligible (4.4% at max).

The advantage of our $FlatFIT$ algorithm for bigger windows comes from the fact that it is able to reuse calculations more efficiently. In contrast, in small windows the overhead of maintaining the complex structures outweighs the benefit of reuse.

3.4.2.2 Exp 2: Max-Multi-Query Throughput (Figures 17 and 18)

In this test we again varied the window size from 1 to 134 million tuples, however we ran a maximum number of queries calculating Max value over the ranges from 1 to the window size after each new tuple arrives. In this environment, increasing window size decreases throughputs for all four algorithms much faster, because we are processing many queries per each slide, which makes the number of slides processed per second decrease quickly. The results of processing up to a window size of 1000 are depicted in Figure 17). Similarly to

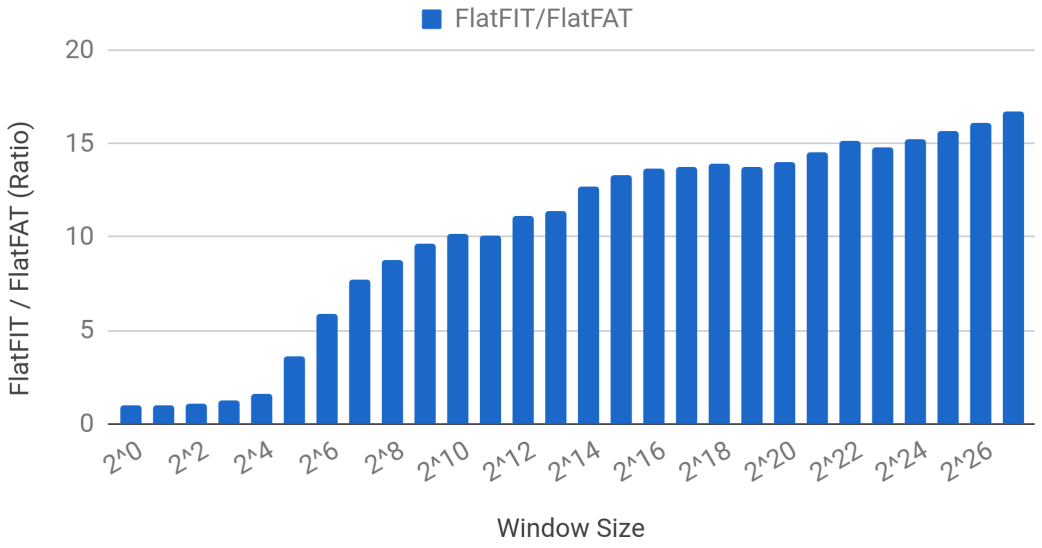

Figure 18: FlatFIT / FlatFAT throughput ratio

the previous experiment, the rates at which throughput decreases are very similar to what we expected from the theoretical analysis of the algorithms (Figure 14). The improvement of $FlatFIT$ over $FlatFAT$ is depicted separately in Figure 18. Our approach demonstrated superior scalability again by yielding throughputs that are on average 10 times higher than throughputs of the $FlatFAT$ technique with a maximum of 17 times. Notice that in this setting $FlatFIT$ performs the best on all window sizes from 2 to 134 million tuples (and only underperforms compared to $Panes$ and $FlatFAT$ on window size 1 by 2% and 1%, respectively).

Notice that the advantage of the $FlatFIT$ algorithm for the large windows becomes even more clear in the multi-query environment due to the fact that $FlatFIT$'s calculation reuse increases with the increasing number of queries, which is not the case for the other compared algorithms. On small window sizes (between 1 and 4 tuples) $Panes$ and $FlatFAT$ slightly outperformed $FlatFIT$, which is consistent with the previous experiment with a single ACQ. In such scenarios, the overhead of maintaining a complicated structure of $FlatFIT$ outweighs the benefit of using it since the updates to the structure itself prevail the useful operation.

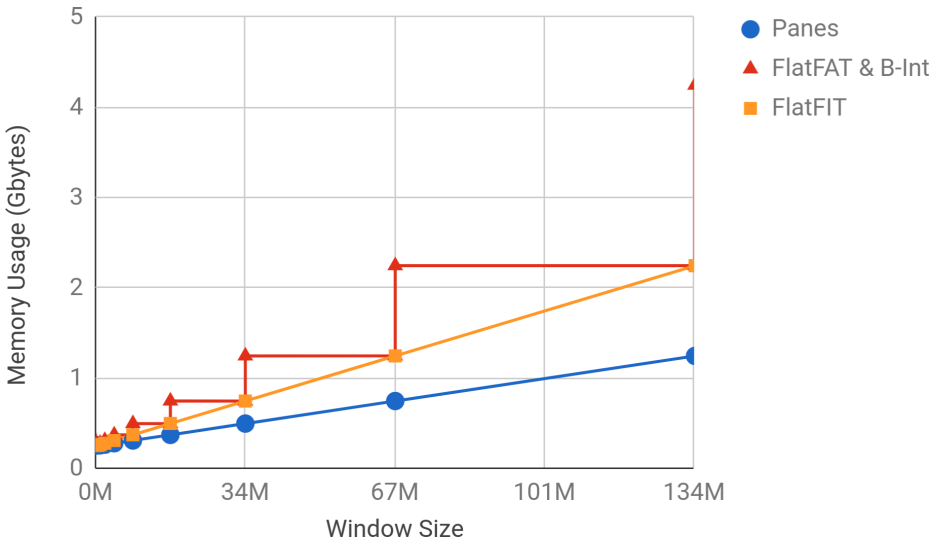

Figure 19: Experimental Memory Usage in GB increments

3.4.2.3 Exp 3: Memory Consumption (Figure 19)

In this test we again varied the window size from 1 to 134 million tuples and included window sizes that are not powers of two, and we executed a query calculating Max value over the whole window size incrementally. We measured the maximum resident set size of the processes for all runs. The results of this test are depicted in Figure 19). The increasing window size increases the space requirement of the algorithms in addition to increasing the processing cost. The rates at which memory increases are almost identical to what we expected from the theoretical analysis of the space complexities (Figure 15), with only a constant difference between any two corresponding data points of all algorithms. We believe that this difference is caused by the buffering of the incoming tuples which is performed by our platform and not accounted for in the theoretical analysis.

In this experiment $FlatFIT$ demonstrated favorable scalability again by consuming on average 1.4 times less memory than the $FlatFAT$ with a maximum of 1.9 times. This advantage is because $FlatFIT$'s memory requirement increases linearly with the increasing window size, while $FlatFAT$'s memory requirement doubles every time the window size crosses a power of two.

3.5 Summary

The main contribution of this chapter is a novel technique, $FlatFIT$, for incremental $SWAG$ processing. It works by intelligently maintaining and reusing calculated partial aggregations in an index structure, it supports both non-invertible and non-commutative aggregate operations, and it is applicable for both single query and MQ environments.

In this chapter, we theoretically showed that $FlatFIT$ significantly decreases the number of operations required for a continuous query to produce the answer while reducing the algorithm's space consumption and supporting generality in query operations. It achieves a time complexity of $\boldsymbol{O(1)}$ (compared to $\boldsymbol{log(n)}$ complexity of the the state-of-the-art at that time $FlatFAT$ approach) and a space complexity of $\boldsymbol{2n}$ (compared to $\boldsymbol{2^{\lceil log(n) \rceil + 1}}$ complexity of $FlatFAT$).

We also showed experimentally that, with the exception of very small windows, $FlatFIT$ achieves up to 2.6 times higher throughputs in a single query environment and up to 17 times in a multi-query environment compared to $FlatFAT$, while also reducing memory consumption by up to 1.9 times. As far as we know, $FlatFIT$ is the first IE technique that achieved constant amortized time complexity. In the next chapter we will present the $SlickDeque$ technique that further improves IE processing, and resulted from our experience in designing and evaluating $FlatFIT$.

4.0 SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation

The current state-of-the-art *Incremental Evaluation* techniques *FlatFIT* and *TwoStacks* aim to increase throughput, and *DABA* to minimize latency, while all process invertible and non-invertible aggregates uniformly. In this chapter, we propose a novel algorithm, *SlickDeque*, that distinguishes the execution between invertible and non-invertible aggregates and offers better throughput and latency for both types. In addition, our method requires less memory and efficiently supports *multi-query* processing.

In the next section we outline the problems with the state-of-the-art *IE* approaches. We introduce our new technique, *SlickDeque* for the final aggregation calculations in Section 4.2. The complexity analysis of *SlickDeque* is presented in Section 4.3. We summarize the experimental evaluation in Section 4.4 and conclude in Section 4.5.

4.1 Introduction

Handling of aggregate operations that are both *invertible* and *non-invertible* proved to be essential in domains such as finance and science. *Invertible* operations include Sum, Product, Count, Average, and Standard Deviation, while *non-invertible* operations include Max, Min, Range, Alphabetical Max (for strings), ArgMax of Cosine, and ArgMin of x^2. It was shown previously that *invertible* operations can be processed efficiently by maintaining a running Sum (or other aggregation), and invoking the inverse operation (such as Subtract) on every expiring tuple, however *non-invertible* operations require more effort to be processed efficiently and remain a challenge.

The state-of-the-art solutions for processing *ACQs*, *FlatFIT* [43] and *TwoStacks* [45], aim to increase throughput and *DABA* [45], to minimize latency. These solutions process invertible and non-invertible aggregates uniformly, which negatively affects their performance with increasing workloads. To address the aforementioned shortcomings, in this chapter we

propose a novel solution named *SlickDeque*, which handles aggregate operations differently based on their invertibility property. The invertible operations are processed using *SlickDeque* (Inv), our new modified *Panes* (Inv) approach, while non-invertible *ACQs* are processed with *SlickDeque* (Non-Inv), our novel deque-based algorithm that intelligently maintains and utilizes intermediate partial aggregates allowing a greater level of reuse of previously calculated results. The separation based on invertibility leads to exceptional throughput and latency for both invertible and non-invertible operations in systems with heavy workloads. Additionally, in this work we consider *Multi-Query* environments, where large numbers of *ACQs* with different ranges and slides operate on the same data stream, calculating similar aggregations.

4.2 SlickDeque Operation

In this section we describe our new algorithm, *SlickDeque*, that significantly speeds up the final aggregation calculations in a sliding-window environment by employing different processing schemes for invertible and non-invertible aggregations.

4.2.1 The SlickDeque Algorithm

In this subsection we provide the algorithm and implementation details for our approach followed by the clarifying examples. We break down our algorithm description based on invertibility of the aggregate operator.

4.2.1.1 SlickDeque for Invertible Aggregates

For processing invertible aggregates we propose *SlickDeque* (Inv), a modified *Panes* (Inv) approach which allows multi-query processing by maintaining running aggregates for each unique range in a hashmap. Pseudocode for it is depicted in Algorithm 2. The algorithm consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase given a set of queries, Q, and one of the partial aggregation

Algorithm 2 SlickDeque (Inv) Pseudocode

```
1: Input: A set of aggregate continuous queries Q, invertible aggregate operation ⊕, the initial
   value for ⊕ initVal, the inverse operation ⊖, and partial aggregation technique PAT
2: Output: Continuous answers to queries in Q according to their specifications.
3:                    Phase 1 (Preparation)
4: sharedPlan = buildSharedPlan(Q, PAT)
5: wSize = sharedPlan.wSize
6: partials = new array[wSize]
7: answers = new map(queryRange → answer)
8: for i=0 to wSize do
9:     partials[i] = initVal
10: end for
11: for each query q Q do
12:     answers.insert(q.range, initVal)
13: end for
14: currPos = 0
15:                   Phase 2 (Execution)
16: while results are expected do
17:     length = sharedPlan.getNextPartialsLength()
18:     newPartial = partialAggregator.aggregate(length, PAT)
19:     for each (qR → ans) pair in answers do
20:         startPos = currPos - qR
21:         if startPos < 0 then
22:             startPos += wSize
23:         end if
24:         ans = ans ⊕ newPartial ⊖ partials[startPos]
25:     end for
26:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
27:     for each query q in queriesToAnswer do
28:         send answers.getVal(q.range) as answer to q
29:     end for
30:     partials[currPos] = newPartial
31:     currPos++
32:     if currPos == wSize then
33:         currPos = 0
34:     end if
35: end while
```

techniques (*PAT*) discussed in Section 2.3.1 (e.g., *Pairs*) as an input, *SlickDeque* (Inv) builds a shared execution plan by executing the *buildSharedPlan* function (line 4). The *sharedPlan* is constructed as discussed in Section 2.4.1, and includes a full list of partials (or edges) augmented with their lengths and lists of queries to be evaluated for each partial. The *buildSharedPlan* function identifies the query with the longest range in terms of the number

of partials, and saves the range as the member *wSize* of the *sharedPlan* (line 5). *wSize* signifies the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Inv) initializes its data structures: a circular array, *partials*, (line 6) and a map, *answers*, (line 7). The *partials* array is initialized to a length equal to *wSize*, and is used to store partial aggregates. The *answers* map maintains the mappings of all queries with unique ranges to their current answers. Queries operating over the same range can share results even if they have different slides. Both the *partials* array and the values of the *answers* map are initialized (lines 8-13) with the initial value for the operation $\oplus$, *initVal*, supplied as input. For example, *initVal* is $-\infty$ for the Max operation.

The *currPos* variable signifies the current position within the *partials* array (line 14). It starts at 0 initially and increases to $wSize - 1$ during execution, after which it wraps back to 0. The arriving partial aggregates will be inserted into the *partials* array always at the *currPos*.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected. At the beginning of the loop (lines 17-18), *SlickDeque* (Inv) gets the next partial's length from the *sharedPlan*, and passes it to the *newPartial Aggregator* which uses the provided *PAT* technique to produce the *newPartial* value.

Next, *SlickDeque* (Inv) loops over all range-to-answer mappings ($qR \rightarrow ans$) in the *answers* map (lines 19-25). The loop starts by identifying the start position, *startPos*, for each mapping within the *partials* array from which the values need to be aggregated. *startPos* is identified by rewinding *currPos* back by query range, qR, length.

Since *SlickDeque* (Inv) only works for the invertible queries, it utilizes both the aggregate operation $\oplus$ (e.g., Sum if query is seeking Sum), and an inverse operation $\ominus$ (e.g., Subtract if the original operation is Sum). This way each answer, *ans*, is updated by executing the aggregate operation $\oplus$ with the newly calculated *newPartial* value and the inverse operation $\ominus$ with expiring *partials[startPos]* value (line 24).

Next, the answers to all queries scheduled at the current position need to be produced (lines 26-29). After receiving the *queriesToAnswer* (a subset of Q) from the *sharedPlan*, *SlickDeque* (Inv) loops over them while sending back the corresponding answers pulled from

the *answers* map. Then, the *Parial* value is inserted into the circular *partials* array at *currPos*, and *currPos* is moved one position forward (lines 30-34).

The following Example 5 (illustrated in Figure 20) should clarify the above algorithm. In order to make the explanation more intuitive we execute the two queries, $Q1$ and $Q2$, on the same incoming datastream using two algorithms: *Panes* and *SlickDeque* (Inv), and we illustrate each step of their calculations side-by-side.

Example 5 Assume we have queries $Q1$ and $Q2$, which are seeking the Sum over the ranges of 3 and 5 tuples, respectively, both with a slide of 1 tuple. The slide size is set to one tuple in this example for simplicity, which means that there is no partial aggregation and the answers to both queries need to be calculated after every new tuple arrival. Since the range of $Q2$ is 5, which is greater than the range of $Q1$, and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a *wSize* of 5 tuples.

Both *Panes* and *SlickDeque* (Inv) algorithms use the *partials* array in order to maintain incoming partial aggregates (in this case just tuples). The difference is that *Panes* produces answers to queries by iterating over this array, while *SlickDeque* (Inv) utilizes the additional *answers* map (Introduced above).

In the *partials* array we mark the positions that have been modified by the algorithm in each step. The current position (*currPos*) at each step is bolded in Figure 20 for convenience. The tuples enter the system in the order: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization in Step 0, in Step 1 the first tuple, 6, arrives. Both algorithms store the new tuple at the *currPos* in the *partials* array, and *Panes* iterates over indexes 3, 4, and 0 in order to answer $Q1$, and iterates over the entire array to answer $Q2$. Both answers in this case are 6.

SlickDeque (Inv) on the other hand in step 1 just updates all answers in the *answers* map by executing the operation $\oplus$ (in this example it is Sum) with the newly arrived tuple 6 and the inverse operation $\ominus$ (in this example it is Subtract) with values at indexes 2 and 0 in the *partials* array, which both are zeros. The updated answers are stored in the *answers* map.

In Step 2, the new partial, 5, arrives, and *Panes* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and sums up all of the values that were visited. The *SlickDeque* (Inv) algorithm on the other hand, is able to provide answers

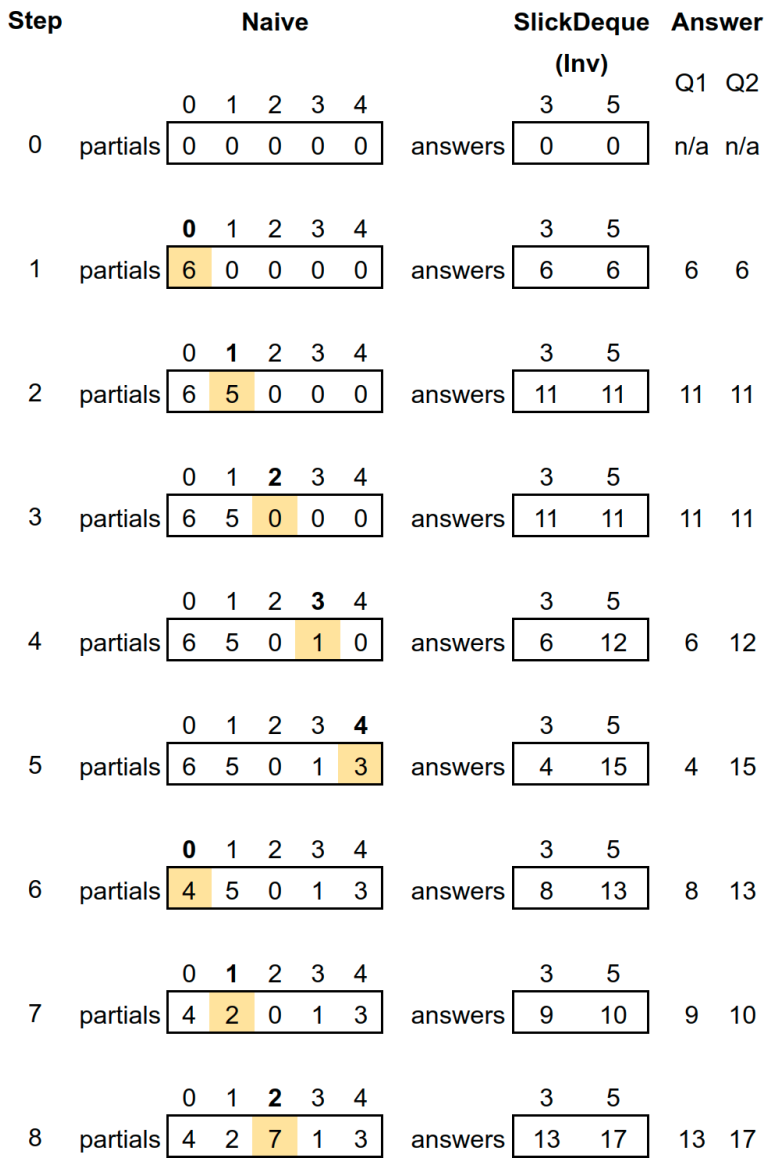

Figure 20: Example 5 processing of invertible aggregate queries Q1 and Q2 using Panes and SlickDeque (Inv) algorithms.

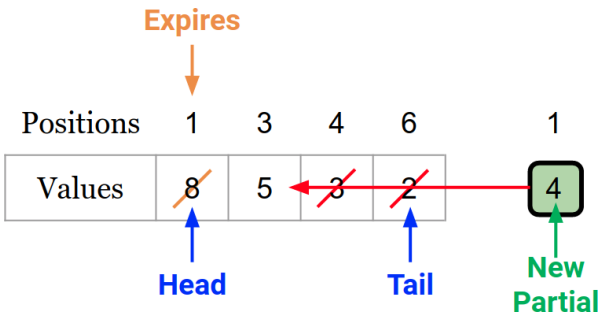

Figure 21: SlickDeque (Non-Inv) Technique

to both queries with just two operations each. It adds 5 and subtracts 0 from both answers in the map, making both 11.

Skipping ahead, in Step 4 *SlickDeque* (Inv) adds the new tuple, 1, to both answers, subtracts 6 from the answer to $Q1$ (since it is now out of range of $Q1$), and then subtracts 0 from the answer to $Q2$ (since 0 was in *partials[3]* in the previous step), returning 6 and 12 as answers to $Q1$ and $Q2$ respectively.

Skipping further, in Step 7 *SlickDeque* (Inv) adds 2 to both answers, and subtracts 1 from $Q1$'s answer (since it is now out of range for $Q1$) making it 9, and subtracts 5 from $Q2$'s answer (since 5 was in *partials[1]* in the previous step) making it 10. ■

Notice that in this example *Panes* had to execute a total of 48 Sum operations, while *SlickDeque* (Inv) executed a total of 32 operations (Sum and Subtract).

4.2.1.2 SlickDeque for Non-Invertible Aggregates

For processing non-invertible aggregates we propose a novel algorithm, *SlickDeque* (Non-Inv), which accelerates the processing of *ACQs* by intelligently maintaining and utilizing a deque data structure consisting of nodes allocated in chunks interconnected with pointers. For simplicity of explanation we assume that each node is allocated on a separate chunk. The benefits of allocating multiple nodes per chunk are explained in Section 4.3.2.

The **intuition** behind the *SlickDeque* (Non-Inv) algorithm can be seen in in Figure 21, which illustrates an update operation (insert partial 4 with sequential position 1) performed on the deque structure. The look-up of the answer (max value) is performed by returning

the head node value if a query requires the result for the maximum window (in this example value 5), or otherwise by looking up the correct node using its *Position* value. Notice that value 8 expires in this example, and the new partial 4 removes existing partials 2 and 3 since it is greater.

The full pseudocode for *SlickDeck* (Non-Inv) is depicted in Algorithm 3, and similarly to *SlickDeque* (Inv) it consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase Similarly to *SlickDeque* (Inv), the execution starts by building a *sharedPlan* by executing the function *buildSharedPlan* (line 4). It is constructed using one of the partial aggregation techniques as discussed in Section 2.3.1, and it includes a full list of partials augmented with their lengths and lists of queries that need to be evaluated for each partial. The query with the longest range in terms of the number of partials is identified and saved as the member *wSize* of the *sharedPlan*, signifying the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Non-Inv) defines node, $Node$, structure that has members *pos* and *val*, and initializes deque, d, composed of nodes, $Node$, (lines 6-7). *SlickDeque* utilizes the *currPos* variable to signify the sequential number of the current partial aggregate. It starts at 0 initially and increases to $wSize - 1$ during execution, after which it wraps back to 0.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected, and identically to *SlickDeque* (Inv), it begins by aggregating a *newPartial*. The if-statement on line 13 is removing the expired node (if present) from the head of the deque, d. The while-loop after that (line 16) is executing operation $\oplus$ on two values: the value of the tail node and of the new partial. If the new partial is returned by the operation, the tail node is removed from the deque (it will never be a query answer), and the next one is tested, otherwise the loop stops. The new node is then added to the deque with *currPos* as the position and *newPartial* as the value (line 19).

Next, set *queriesToAnswer* (a subset of Q scheduled at this position) is accessed from the *sharedPlan*, and the answers for its queries are produced in the for-loop below. Naturally, when the *sharedPlan* was constructed, all queries in each *queriesToAnswer* set were ordered descendingly by their range. We utilize this ordering to answer all queries by looping over the

Algorithm 3 SlickDeque (Non-Inv) Pseudocode

```
1: Input: A set of aggregate continuous queries Q, non-invertible aggregate operation ⊕, and partial
   aggregation technique PAT
2: Output: Continuous answers to queries in Q according to their specifications.
3:                    Phase 1 (Preparation)
4: sharedPlan = buildSharedPlan(Q, PAT)
5: wSize = sharedPlan.wSize
6: Node with members pos and val
7: Deque d composed of nodes of type Node
8: currPos = 0
9:                    Phase 2 (Execution)
10: while results are expected do
11:     length = sharedPlan.getNextPartialsLength()
12:     newPartial = partialAggregator.aggregate(length, PAT)
13:     if d.size > 0 AND d.front.pos == currPos then
14:         d.pop_front()
15:     end if
16:     while d.size>0 AND d.back.val⊕newPartial == newPartial do
17:         d.pop_back()
18:     end while
19:     d.push_back(new Node(currPos, newPartial))
20:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
21:     i = d.firstNode
22:     for each query q in queriesToAnswer do
23:         startPos = currPos - q.range
24:         boundaryCrossed = false
25:         if startPos < 0 then
26:             startPos += wSize
27:             boundaryCrossed = true
28:         end if
29:         if boundaryCrossed == false then
30:             //Answer Loop 1
31:             while i.pos < startPos OR i.pos > currPos do
32:                 i = i.nextNode
33:             end while
34:         else
35:             //Answer Loop 2
36:             while i.pos < startPos AND i.pos > currPos do
37:                 i = i.nextNode
38:             end while
39:         end if
40:         send i.val as answer to q
41:     end for
42:     currPos++
43:     if currPos == wSize then
44:         currPos = 0
45:     end if

46: end while
```

deque only once, since the larger ranges always correspond to the deque nodes closest to the head. Therefore, the position i within the deque is defined outside the loop and initialized to the head of the deque (line 21).

The loop starts by identifying the *startPos* of the aggregation for each query, q, by subtracting q's range from *currPos* (line 23). If *startPos* is negative it means that this range crosses a boundary between two windows, and thus the boolean *boundaryCrossed* is set to true and *startPos* is increased by the *wSize*. Otherwise *boundaryCrossed* is set to false.

Then, based on whether the current range crosses the window boundary or not, one of the two subsequent *Answer Loops* is executed (lines 29-39), iterating over nodes from the current position i until the answer node is identified based on the *pos* member of each node, and returned as an answer to the query, q. The next iteration (to answer the next query) will continue working from the position i forward, until all queries are processed. After returning all required answers the *currPos* is moved one position forward (lines 42-45).

The following Example 6 (illustrated in Figure 22) should clarify the above algorithm. To make the explanation more intuitive we again execute the two queries $Q1$ and $Q2$ on the same incoming datastream using *Panes* and *SlickDeque* (Non-Inv), and illustrate each step of their processing side-by-side.

Example 6 Assume we have queries $Q1$ and $Q2$, which are seeking Max over the ranges of 3 and 5 tuples respectively, both with a slide of 1 tuple. The slide size is again set to one tuple for simplicity, which means that there is no partial aggregation and the answers to both queries need to be calculated after every new tuple arrival. As before, the range of $Q2$ (5) is greater than the range of $Q1$ (3), and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a $wSize$ of 5 tuples.

While *Panes* uses the circular *partials* array to maintain the incoming partials (in this case just tuples), *SlickDeque* (Non-Inv) only utilizes deque in its operation. In both *partials* and deque we mark the positions modified in each step. The tuples enter the system in the same order as in Example 5: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization Step, in Step 1 the first tuple, 6, arrives. *Panes* stores it at the *currPos* in the *partials* array, and iterates over the last 3 indexes (3, 4, and 0) to answer $Q1$, and over the entire array to answer $Q2$. Both answers in this case are 6.

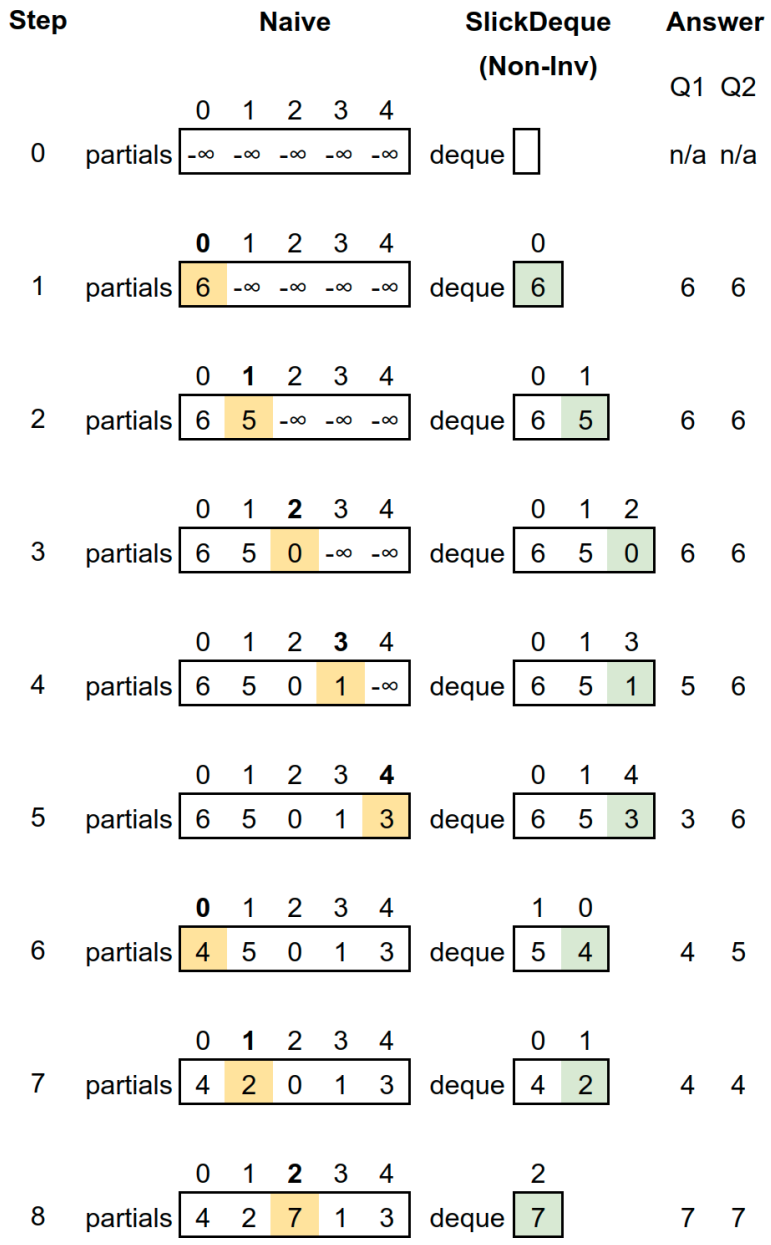

Figure 22: Example 6 processing of non-invertible aggregate queries Q1 and Q2 using Panes and SlickDeque algorithms.

SlickDeque (Non-Inv) places a new node with $pos = 0$ (which is *currPos*) and $val = 6$, at the head of the deque, and since its *pos* value is both within the last 3 and 5 positions from *currPos*, its *val* is returned as the answer to both $Q1$ and $Q2$.

In Step 2, the new partial, 5, is placed into the *currPos*, and *Panes* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and returns the Max value from all values visited, which is 6. Our algorithm on the other hand, places the new tuple 5 as a *val* of the new node (with $pos = 1$) at the end of the deque, and returns 6 (the *val* of the head node of the deque) as an answer to both queries.

Skipping ahead, in Step 4 *SlickDeque* (Non-Inv) removes the tail node of the deque since the newly arrived tuple, 1, is greater than 0, which is the *val* of the tail node, and adds the new node with $pos = 3$ and $val = 1$ at the end of the deque. Since $Q2$ has a larger range, it is scheduled to be processed first. Its *startPos* is identified: $3 - 5 = -2$, and since -2 is negative, the window boundary is crossed. Therefore *startPos* is moved to $-2 + 5 = 3$, and the *Answer Loop 2* is executed returning the *val* of the head node, 6. The *startPos* of $Q1$ is $3 - 3 = 0$, and since 0 is not negative, the window boundary is not crossed. Thus, the answer is produced by iterating using *Answer Loop 1*, which returned 5, the *val* of the second node from the head.

Skipping further, in Step 6 *SlickDeque* (Non-Inv) removes the head node of the deque (with $pos = 0$ and $val = 6$) which expires at this step since the *currPos* is 0. Also, since the newly arrived tuple, 4, is greater than 3, the last node of the deque is removed, and the new node with $pos = 0$ and $val = 4$ is added at the end of the deque. $Q2$ and $Q1$ are then both processed by executing the *Answer Loop 2* and returning 5 and 4 respectively. ■

Note that this example also shows the advantage of *SlickDeque* (Non-Inv) over *Panes* by showing that *Panes* had to execute 48 Max operations total, while *SlickDeque* (Non-Inv) executed 11.

Table 4: Final Aggregation Complexities. Our contributions are bolded. Complexities of the existing techniques are derived in Section 2.3.2.

Algorithm		Time			Space	
		Single Query		Max-Multi Query	Single Query	Max-Multi Query
		Amort	Worst			
Panes		n	n	n^2	n	n
Panes(Inv)		2	2	—	n	—
FlatFAT		$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
B-Int		$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
FlatFIT		3	n	$3n$	$2n$	$2n$
TwoStacks		3	n	—	$2n$	—
DABA		5	8	—	$2n$	—
Slick Deque	**Inv**	2	2	$2n$	n	$2n$
	Non-Inv	<2	n*	n	2 to $2n$*	2 to $2n$*

*the probability of these cases is negligible: 1 in n!.

**true only when n is a power of 2, otherwise $3n$.

4.3 Complexity Analysis

In this section, we calculate the time and space complexities of *SlickDeque* (summarized in Table 4).

4.3.1 Time Complexity of SlickDeque

SlickDeque for Invertible Operations has an exact time complexity of just 2 operations per slide in a single query environment, since after each arrival of the new partial aggregate, the query answer is updated twice: once by executing an aggregate operation with the incoming partial, and once by executing the inverse operation with the expiring partial. In a max-multi-query environment *SlickDeque* (Inv) has to perform $2n$ operations, since one

aggregate operation and one inverse operation need to be executed on each of the answers to n queries, which makes the algorithm's exact time complexity $2n$.

SlickDeque for Non-Invertible Operations executes variable numbers of operations per slide. As opposed to *FlatFIT*, *TwoStacks*, and *DABA* which are input agnostic and have their worst-case steps executed periodically, *SlickDeque* (Non-Inv) depends on the input, and the probability of ever executing its worst-case step is minuscule as we point out below.

Intuitively, in the long-running environment with a non-infinite window, each partial can causes at most two operations: one when it is inserted (invokes its comparison with the tail of the deque), and one when it is deleted by another incoming partial (invokes comparison of the incoming partial with the next item on deque). Clearly, the only two situations when a partial performs less than two operations in its lifetime are:

1. When the partial becomes the first element of the deque after its insertion (either by removing all other partials or by being inserted into an empty deque).
2. When the partial expires before being removed by another partial.

If both situations happen to the same partial it will be involved in 0 operations in its lifetime. Also, it is impossible to execute a full window iteration without hitting one of the two situations by one of the partials at least once, since we cannot have an element in a deque that would both not get removed by another incoming partial as well as not expired after a full window iteration. Thus, the amortized complexity of this algorithm depends on the input, however it is always less than 2 operations.

The worst time complexity of this algorithm happens when the input (except the last partial of the window) is ordered in the opposite way of the aggregate operator order, e.g., if Max is processed and the entire input is ordered descendingly, forcing the deque to fill up, after which the next input partial has the largest value so far. This causes the new element to perform n operations while deleting all nodes on the deque. Fortunately, such a situation is highly unlikely on most inputs (1 in $n!$ chance in the uniform case). Consider the state-of-the art *DABA* algorithm that we showed to have a worst-case complexity of 8 operations. In order for *SlickDeque* (Non-Inv) to have a step with the same complexity there should be at least 9 ordered partials in the input. The probability of receiving 9 values ordered in a

specific way in a row is 1 out of 9! (equals 362880), which is unlikely. Yet, in long running high velocity systems it is still possible, and thus *DABA* might occasionally have a lower latency while processing a particular aggregation.

In a max-multi-query environment, to process all queries scheduled at a slide, the deque is traversed from the head while answering each query. Clearly, if the number of nodes in the deque is smaller than the number of different queries to answer, some nodes will have answers to multiple queries. Thus, the worst case would again be when the input forced the deque to completely fill up, for which the probability is again 1 in $n!$. In such a case, iterating over the entire deque at each step will take n operations (and at worst 2 operations per step as shown in the single query environment), so the complexity of the worst-case becomes $2n$. In the best case, the deque would have only one node each slide that would answer all queries, which would make complexity just 2 operations total.

To summarize, the differentiated processing of invertible and non-invertible operations allows *SlickDeque* to utilize optimizations tailored towards each type that are not available in the general case. Thus, *SlickDeque* is superior in the time complexity for both invertible and non-invertible cases compared to all other algorithms (See Table 4). However, in the worst-case complexity per slide, theoretically *SlickDeque* has a small possibility (1 in 362880 based on the input) to be outperformed by *DABA*.

4.3.2 Space Complexity of SlickDeque

SlickDeque for invertible operations stores partial aggregates similarly to *Panes*. In addition, it stores the answer for each query with a unique range, making its single query space complexity $n + 1$, and max-multi-query $2n$.

SlickDeque for non-invertible operations performs node allocations in chunks to reduce the space required by pointers similarly to *DABA*, causing an overallocation of up to two chunks' worth of space (at the beginning and at the end of the deque). The space complexity of *SlickDeque* (Non-Inv) does not depend on the number of registered queries, but depends on the input. In the worst-case, the input forces the deque to become full. In such a case, having n nodes with two values each, and k chunks with two pointers each, the space consumption

becomes $2n + 4k + 4n/k$. By taking a derivative with respect to k, equating it to zero, and solving for k, we conclude that k should be set to $\sqrt{n}$ to minimize the worst-case complexity, which becomes $2n + 4\sqrt{n}$ (asymptotically $2n$). Similarly to the time complexity, the chance of the worst-case happening in normal conditions is very low: just 1 in n!. In the best case, however, each incoming partial forces the deque to eliminate all of its nodes, making the space complexity constant (2).

To summarize, *SlickDeque* shows a clear advantage over the rest of the algorithms in terms of space complexity (See Table 4). *SlickDeque* (Inv) shares the space complexity of n with *Panes*, while the rest of the algorithms have a complexity of at least $2n$, and the complexity of *SlickDeque* (Non-Inv) is always less or equal than $2n$ (based on the input). This means that only *Panes* can possibly outperform it, however the probability of that happening is low (just 1 in $n!/2$), and even then, *Panes* is still not a feasible solution because of its high time complexity.

4.4 Experimental Evaluation

In this section, we present our experimental evaluation that confirms the theoretical superiority of *SlickDeque* in practice, by comparing it to the *Panes*, *FlatFAT*, *B-Int*, *FlatFIT*, *TwoStacks*, and *DABA* approaches.

In this evaluation we used the same experimental testbed as we used in Section 3.4.1, with the difference that in addition to measuring *Throughput* and *Memory Requirement*, we also measure *Latency*. *Latency* is measured in terms of the total wall clock time it took to calculate and return the answer to each query. Also, in this evaluation, in addition to Max operation, we also test with Sum operation to measure the performance of processing both non-invertible and invertible aggregations respectively.

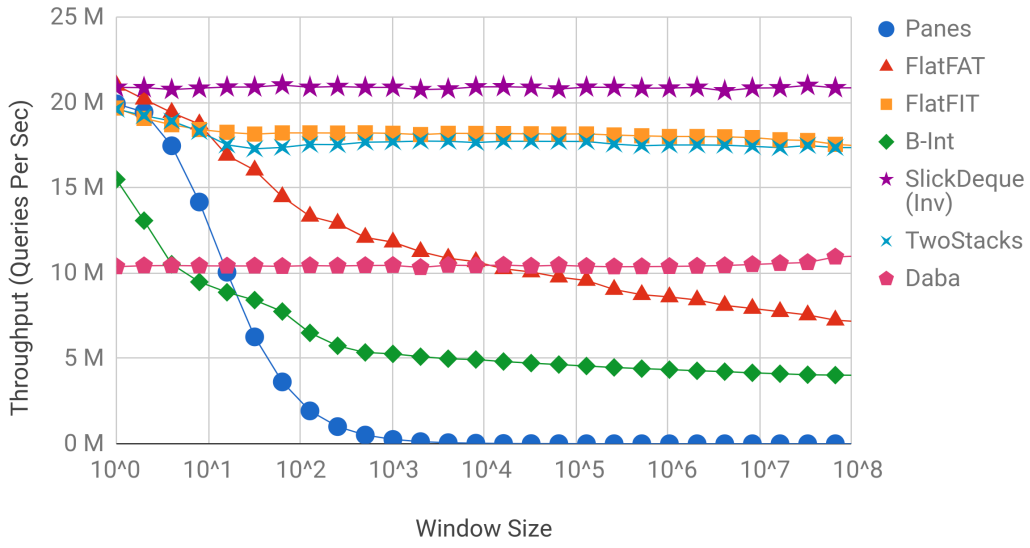

Figure 23: Throughput in processed queries per second in single query environment (Sum)

4.4.1 Experimental Results

4.4.1.1 Exp 1: Single Query Throughput (Figures 23 & 24)

Exp1(a) Invertible Aggregates (Figure 23) In this experiment we varied the window size from 1 to 134 million tuples where each window is a power of two, and ran a query calculating the invertible aggregation Sum over the entire window after each new tuple arrival. From the results in Figure 23 we clearly see that there are two groups of algorithms based on their behavior with increasing window size: (1) with constant throughput (*SlickDeque*, *FlatFIT*, *TwoStacks*, and *DABA*), and (2) with steadily degrading throughput (*FlatFAT*, *B-Int*, and *Panes*). Notice that the throughput rates are similar to what we expected from the theoretical analysis of the algorithms in Section 4.3.

Figure 23 shows that *SlickDeque*'s throughput is on average 15% higher than the throughput of the second best algorithm (*FlatFAT* on windows 1 through 16, and *FlatFIT* on the rest) with a maximum of 19%. We also observed that *SlickDeque* starts outperforming other algorithms on windows as small as 4 tuples and increases its gain rapidly. *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 4 tuples, where the overhead of *SlickDeque* is not amortized. However this benefit of *FlatFAT* is negligible (1%

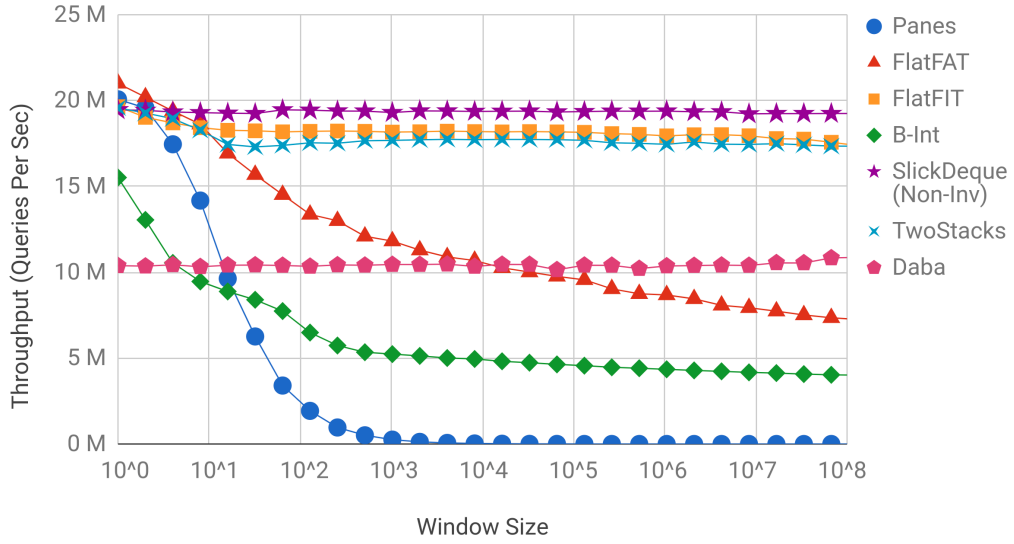

Figure 24: Throughput in processed queries per second in single query environment (Max)

at max). In all other cases, including large windows, the advantage of *SlickDeque* (Inv) can be attributed to its focus on processing solely invertible aggregations unlike the other techniques, enabling it to use of the inverse operations to speed up processing.

Exp1(b) Non-Invertible Aggregates (Figure 24) In this experiment we replaced the calculation of Sum with the non-invertible aggregation Max, that again runs over the entire window after each tuple arrival. Similarly to Exp1(a), we see that the throughput of some algorithms is practically unaffected by the increasing window size. The results are depicted in Figure 24. Once again, the throughput rates correspond to what we expected from the theoretical analysis of the algorithms.

In this experiment *SlickDeque*'s throughput is on average 7% higher than the throughput of the second best algorithm with a maximum of 10%, and *SlickDeque* starts outperforming all other algorithms on windows as small as 16 tuples. Consistent with the previous experiment Exp1(a), *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 8 tuples with an advantage of 7% at max. Also, the benef of using *SlickDeque* (Non-Inv) is due to its focus on processing solely non-invertible aggregations, which allows it to discard a significant portion of inputs and thus achieve a more efficient reuse of interim calculations.

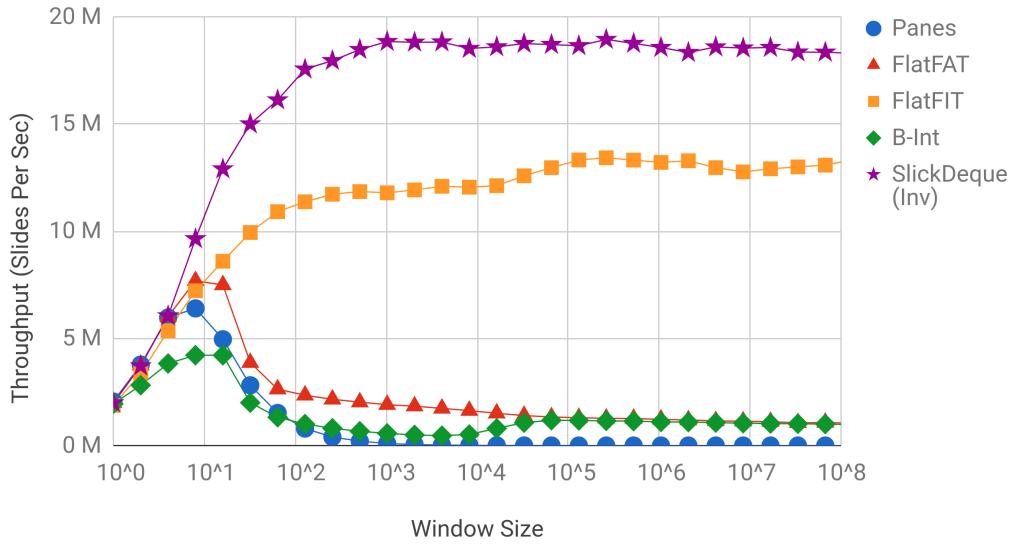

Figure 25: Throughput in processed slides per second in multi-query environment (Sum)

4.4.1.2 Exp 2: Max-Multi-Query Throughput (Figures 25 & 26)

Exp2(a) Invertible Aggregates (Figure 25) In this experiment we ran a maximum number of queries calculating Sum value over the ranges from 1 to the window size after each new tuple arrives. In this context increasing the window also increases the number of queries that are processed after each slide, enabling higher reuse of unchanged partial results among them. Thus, in Figure 25 we see that the throughput gradually increases until the moment when the overhead of dealing with the large window outweighs the benefit of sharing between queries.

In this setting, our approach demonstrated superior scalability yet again by yielding throughput that is on average 45% higher than the throughput of the second best technique with a maximum of 60%. Notice that *SlickDeque* performs the best on window sizes from 4 tuples to 134 million tuples and only underperforms compared to other algorithms on window sizes 1 and 2 by 3% and 2%, respectively. The observations here are similarly to what we saw in Exp1(a).

Exp2(b) Non-Invertible Aggregates (Figure 26) In this experiment we ran the maximum number of queries calculating Max over all ranges from 1 to the entire window after each

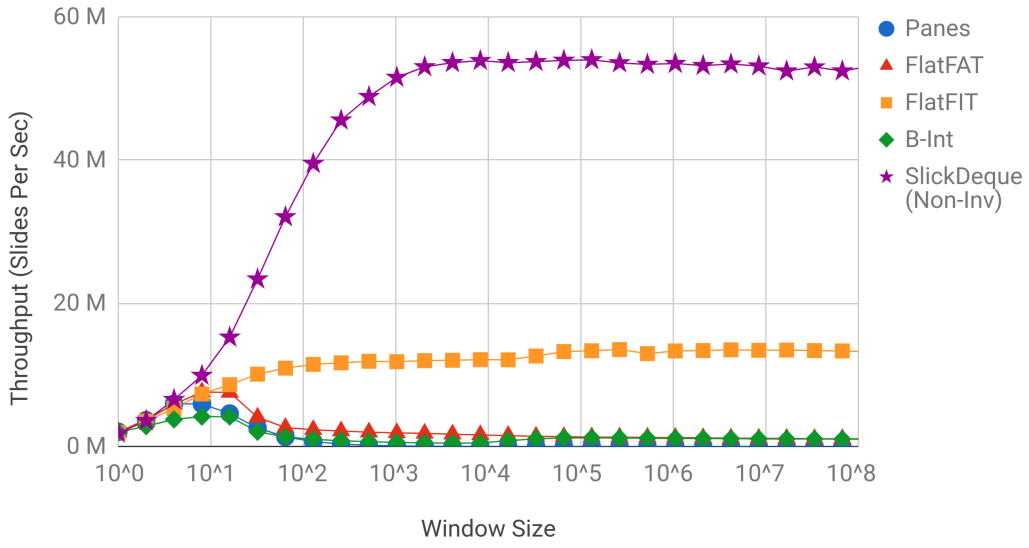

Figure 26: Throughput in processed slides per second in multi-query environment (Max)

tuple arrival. The results are depicted in Figure 25, and are close to our results in experiment Exp2(a).

In this setting *SlickDeque* yielded throughput on average 266% higher than the throughput of the second best technique with a maximum of 345%. *SlickDeque* showed to perform the best on windows from 4 tuples to 134 million tuples while falling behind *Panes* and *FlatFAT* on windows 1 and 2 by 7% on average. The observations here are similarly to what we saw in Exp1(b).

Conclusions In all throughput experiments *SlickDeque* exhibits the best results, while being slightly outperformed on small window sizes (between 1 and 8 tuples) when the overhead of maintaining its structure outweighed the benefit of using it.

4.4.1.3 Exp 3: Query Processing Latency (Figure 27)

In this experiment we fixed our window size at 1024 tuples and ran all algorithms on the first million tuples of the DEBS data set while recording how long it took to return an answer to each query. We executed a single query processing Sum (invertible) in the first test, and Max (non-invertible) in the second test. We dropped the highest 0.005% latencies

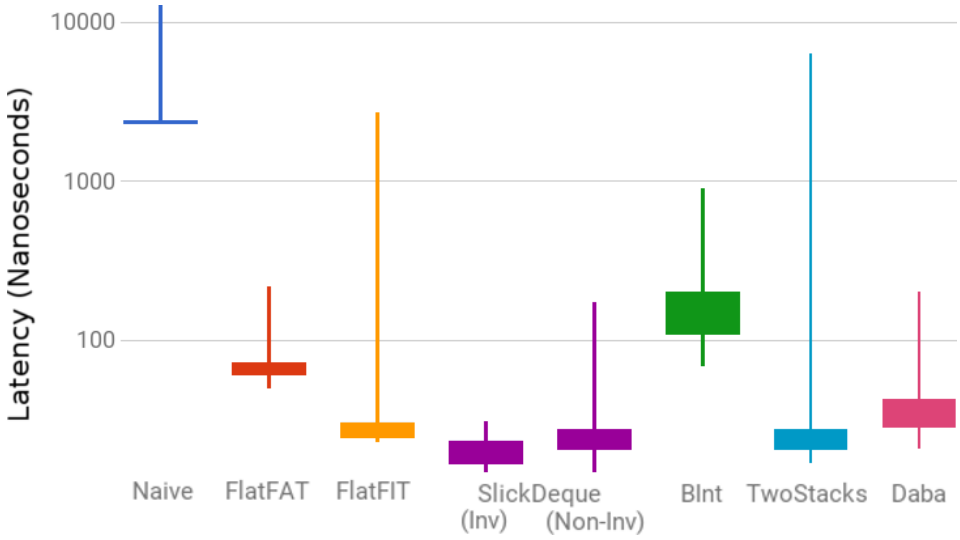

Figure 27: Latency in nanoseconds per query answer

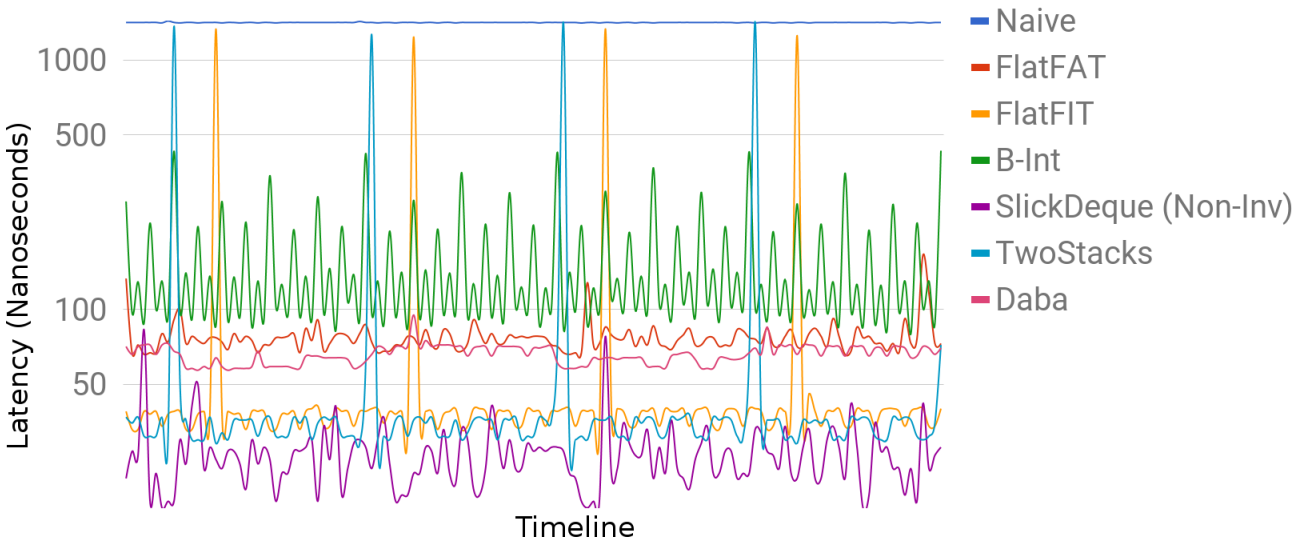

Figure 28: Latency spikes in nanoseconds per query answer

from all algorithms as outliers. The latency results of both tests were nearly identical for all algorithms except *SlickDeque*, thus we combined them in Figure 27, where only *SlickDeque* has separate entries for invertible and non-invertible cases.

Figure 27 shows that both invertible and non-invertible *SlickDeque* versions exhibited the

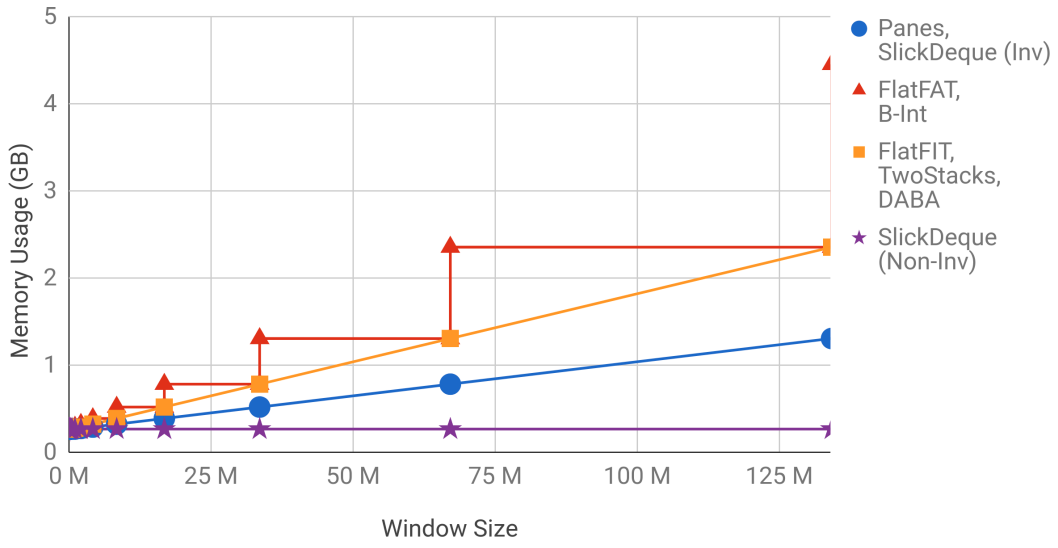

Figure 29: Experimental Memory Usage in Gigabyte increments

lowest latency in all the following categories: Min, Max, Average, Median, 25th Percentile, and 75th Percentile. Across all of the abovementioned categories, *SlickDeque* outperformed the second best algorithm by 8% on average and 17% at most (for the non-invertible version), and by 75% average and 548% at most (for the invertible version). Also, *SlickDeque* outperformed the second best *DABA* algorithm by 283% on average in terms of the lowest max latency spike. Yet, as exhibited in a snapshot of execution of this experiment in Figure 28, *SlickDeque* (Non-Inv) occasionally has latency spikes that are higher than the latency of *DABA* algorithm at that point due to its worst case latency being linear. However under no circumstances *SlickDeque* exhibited a spike higher than the highest spike of *DABA*. Also, during the bulk of the execution *DABA*'s latencies are considerably higher.

Similarly to Experiments 1 and 2, the latency improvements when using the *SlickDeque* algorithm can be credited to its ability to process invertible and non-invertible aggregations differently, and thus being able to use more efficient structures for each type.

4.4.1.4 Exp 4: Memory Requirement (Figure 29)

In this experiment we again varied the window size from 1 tuple to 134 million tuples (but

also included window sizes that are not powers of two). We executed a query calculating the invertible Sum aggregation in the first experiment, and the non-invertible Max aggregation in the second. We measured the maximum resident set size (RSS) of the processes for all runs. The results of this test are depicted in Figure 29. On this graph, we combined the results of both invertible and non-invertible runs of all algorithms since their space requirements were identical in both Sum and Max cases except for *SlickDeque*, which we plotted separately for each case. Notice that due to the great similarity of space requirement for several algorithms, we plotted: *FlatFAT* together with *B-Int*, *FlatFIT* together with *TwoStacks* and *DABA*, *Panes* together with *SlickDeque* (Inv), and *SlickDeque* (Non-Inv) was plotted separately. The memory requirement rates correspond to what we predicted from the theoretical analysis in Section 4.3. *SlickDeque* demonstrated excellent scalability by matching the space usage of *Panes* for the invertible case due to storing the input partials only once, and for the non-invertible case outperforming the second best algorithm (*Panes*) by 2 times on average with a maximum of 5 times due to being able to discard incoming partials dynamically before they expire.

4.5 Summary

The key contribution of this chapter is *SlickDeque*, a novel technique for incremental sliding-window final aggregation processing for single and *MQ* environments. Its power is the differentiated handling of aggregate operations based on their invertibility, which allows *SlickDeque* to use optimizations tailored towards each type and that are not available in the general case.

We theoretically show that *SlickDeque* significantly decreases the number of operations required for a continuous query to return results while reducing its space requirement. As far as we know, there are no prior algorithms that can achieve the same time and space complexities without loss of query generality.

We experimentally evaluate *SlickDeque* based on a real dataset and show that it significantly outperforms state-of-the-art techniques in all tested scenarios by increasing the

ACQ throughput by up to 19% in a single query environment and by up to 345% in an *MQ* environment, while maintaining 283% lower latency spikes on average and reducing memory consumption by up to 5 times. We also show that our approach becomes superior to the state-of-the-art approaches at window sizes as small as eight tuples with its benefits increasing rapidly as window sizes increase, making *SlickDeque* widely applicable for processing *ACQs* in a variety of *DSMSs*. In addition, *SlickDeque* is the most effective *IE* technique when used in *MQ* optimizers as we will show in Chapter 7.

It is important to note that in some cases it is still more beneficial to use other techniques instead of *SlickDeque*. For example, when our query workload consists of very small windows it would be beneficial to utilize the naive *Panes* technique, since with other techniques the overhead of maintaining complex structures outweighs the benefit of utilizing them. Also, in scenarios that have strict deadlines for the *ACQ* result arrivals and where the occasional spikes are completely unacceptable, the *DABA* technique should be used at the expense of more calculation intensive processing.

5.0 F1: Accelerating the Optimization of Aggregate Continuous Queries

In the previous two chapters we focused on single-query processing where an *ACQ* is reusing its intermediate calculations instead of re-evaluating the entire window after each slide. In *MQ* environments, multiple *ACQs* calculate similar aggregations with different ranges and slides can be processed within the same data structure and share partial results with each other, achieving higher efficiency. State-of-the-art *WeaveShare* and *TriWeave* produce high quality execution plans using the *Weavability* concept.

In this chapter we propose a novel closed formula, $F1$, that accelerates *Weavability* calculations, and thus allows *WeaveShare* (and *TriWeave*) *MQ* optimizers to achieve exceptional scalability in systems with heavy workloads. In general, $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple *ACQs*.

In the next section we outline the problems with existing *Bit Set* approach. We introduce our new formula, $F1$, for the *Weavability* calculation and its additional optimization in Section 5.2. The complexity analysis on it is presented in Section 5.3. The evaluation platform and the experiments are discussed in Section 5.4. We conclude in Section 5.5.

5.1 Introduction

The state-of-the-art *WeaveShare* algorithm produces very high quality execution plans by utilizing the *Weavability* concept [20], which is used to decide which *ACQs* are similar enough to be combined. *WeaveShare* is theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem, which was shown to be more than an order of magnitude improvement over the best existing alternatives [12]. However, when we tried to implement it in a multiple-tenant *DSMS*, we observed that the current approach of calculating *Weavability* using *Bit Set* is very computationally expensive.

This motivated us to explore a more efficient algorithm to accelerate the calculation

process in order to make the *WeaveShare* algorithm more scalable for systems with heavy workloads. Towards this, in this chapter we propose a mathematical solution *Formula 1* (or $F1$), which reduces the number of operations needed to produce the efficient execution plan and by doing so speeds up the plan generation time. $F1$ also eliminates concerns over the amount of system memory as it does not need to store any large data during its operation. In fact, $F1$ acceleration has enabled us to explore additional cost savings that can be achieved by utilizing the distributed nature of the Cloud Infrastructure by intelligently colocating $ACQs$ on different computing nodes [42]. In general, $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple $ACQs$.

5.2 Formula 1 (F1)

In this section, we overview the current *Bit Set* approach, and then describe our new formula $F1$ that significantly speeds up the edge rate calculation in a composite slide. We target two scenarios for $ACQs$ with matching or compatible aggregate operations: 1) when all ACQ slides are factors of their corresponding ranges, and 2) when some of the ranges are not multiples of their corresponding slides.

5.2.1 Bit Set Approach

In order to show how the *Bit Set* approach works, consider the following example:

Example 7 There are two $ACQs$ that perform the *count* aggregate operation on the same data stream. The first ACQ has a slide of 2 sec and a range of 6 sec, the second one has a slide of 4 sec and a range of 8 sec. Therefore, the first ACQ is computing partial aggregates every 2 sec, and the second is computing the same partial aggregates every 4 sec.

Clearly, the calculation producing partial aggregates only needs to be performed once every 2 sec, and both $ACQs$ can use these partial aggregates for their corresponding final aggregations. The first ACQ then will run each final aggregation over the last three partial aggregates, and the second ACQ will run each final aggregation over the last 4 partial aggregates. ■

The procedure to determine how many partial aggregations is needed after combining n *ACQs* using a *Bit Set* is formalized as follows:

- Find the length of the new combined (composite) slide, which is the *Least Common Multiple* (LCM) of all the slides of the combined *ACQs*.
- Each slide is then repeated $LCM/slide$ times to fit the length of the new composite slide. All partial aggregations happening within each slide are also repeated and marked in the composite slide as *edges*.
- If the location is already marked, it cannot be marked again. If two *ACQs* mark the same location, it means that location is a common *edge*.

The most complex part of the calculation occurs when the system is scheduling each partial aggregation operation (*edge*) and is tracking these operations using a *Bit Set*. The size of the *Bit Set* increases rapidly if the *ACQs*' time properties differ. For each *ACQ* added to the execution tree, *WeaveShare* needs to traverse the whole *Bit Set* to make sure that all partial aggregations necessary for this *ACQ* are marked in a *Bit Set* for future execution. Since the size of a *Bit Set* increases exponentially with the increase of the input size, traversing it becomes prohibitively time-consuming as we show in Section 5.3. Additionally, the exponential increase of the size of the *Bit Set* puts a hard limit on system's capabilities, based on the amount of memory available.

5.2.2 Case with NO Fragments

In the case when all of the ranges of the *ACQs* that are installed onto the *DSMS* are divisible by their corresponding slides, we can store partial aggregates at every slide. For example, if we have an *ACQ* with a slide of 3 sec and a range of 9 sec, we can store partial results every 3 sec, and perform the final aggregations on the 3 last saved partial aggregations to get the answer for the last 9 sec. In order to calculate the edge rate after *weaving* together n *ACQs*, we need a *Bit Set* of the length equal to the LCM of all n slides. At first, the *Bit Set* is populated with zeros. For each one of n *ACQs* we traverse the whole *Bit Set* and mark all bits whose indexes are divisible by the corresponding *ACQ*'s slide with ones. If the bit was already marked, the algorithm does nothing and just moves to the next bit.

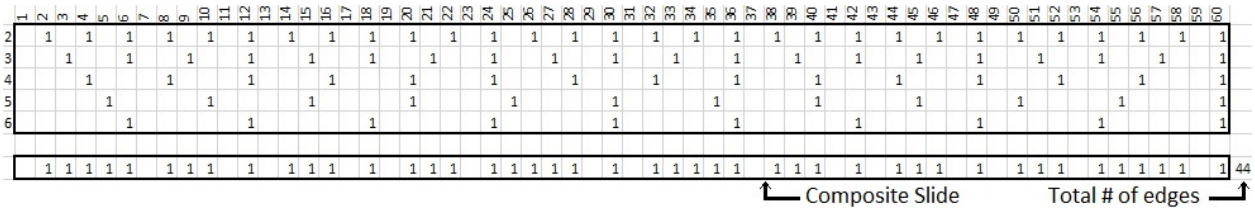

Figure 30: Marking edges produced by five different *ACQs* with NO fragments in the composite slide, represented by a Bit Set

Example 8 Consider five stock monitoring *ACQs* with the following slides: 2, 3, 4, 5, and 6. Their *LCM* is 60, therefore we need a *Bit Set* of size 60. First, we traverse the *Bit Set* and mark all indexes divisible by 2 (all even numbers up to 60). Now the *Bit Set* has 30 bits marked. Next we mark all indexes that are divisible by 3. The *Bit Set* already has 40 bits marked (10 overlapped with already marked ones). Next we mark all indexes that are divisible by 4. The *Bit Set* still has 40 bits marked since all of the bits we were trying to mark were already marked by a slide of 2. After repeating the same for slides of 5 and 6, we calculate how many bits we have in our *Bit Set*, and the answer is 44. This method is illustrated in the Figure 30. ■

To accelerate this calculation process we propose the *Formula 1* (or $\boldsymbol{F1}$):

$$LCM_n \sum_{i=1}^{n} [(-1)^{i+1} G_1(n,i)] \tag{5.1}$$

Where $LCM_n = LCM(s_1, s_2, \ldots, s_n)$, and function $G_1(n,i)$ is a sum of the inversed LCMs of all possible groups of slides of size i from a set of size n. For example:

$$G_1(3,2) = \frac{1}{LCM(s_1,s_2)} + \frac{1}{LCM(s_1,s_3)} + \frac{1}{LCM(s_2,s_3)} \tag{5.2}$$

$F1$ can be expanded as follows:

$$LCM_n[G_1(n,1) - G_1(n,2) + \ldots \pm G_1(n,n-1) \mp G_1(n,n)] \tag{5.3}$$

Equation 5.3 is composed of an alternating series of function G_1 multiplied by the LCM_n. $LCM_n \cdot G_1(n,1)$ and represents the number of all edges produced by all *ACQs* and therefore

includes all overlapping edges. The goal of the calculation is to count every edge only once, even if it overlaps multiple times in different $ACQs$. Therefore, the rest of the elements of the series will eliminate all of the overlapping edges from the current result. $LCM_n \cdot G_1(n, 2)$ represents the number of edges that overlap in all different pairs of slides and after subtracting it, we get a smaller number than the number we are looking for, because there are potentially some edges where more than two slides overlap at the same time. For example, if slides a, b, and c overlap at some specific edge e, we add it three times: for pairs (a, b), (a, c), and (b, c). The following element $LCM_n \cdot G_1(n, 3)$ compensates for these cases by adding back all edges that overlap in each set of three slides. After adding it, we have again a larger number than the sought-after number, because we might have four or more slides overlapping at the same edge. Therefore, each element compensates for the previous ones' inaccuracies up to the point when we add/subtract the final edge of the composite slide, which clearly occurs only once, since $LCM_n \cdot G_1(n, n) = \frac{LCM_n}{LCM_n} = 1$. The last added/subtracted edge has an index equal to the LCM_n.

Equation 5.3 is an alternating series and we know in advance that the number of elements is always equal to the number of $ACQs$ in the execution tree and is a finite number. Therefore, by definition, the sequence always converges.

The following is an example of using $F1$ to calculate the number of edges:

Example 9 Consider the same set of stock monitoring $ACQs$ as we had in Example 8: slides are $2, 3, 4, 5$, and 6. As a first step of our algorithm we calculate the LCM_n of the whole set of slides. $LCM_n = LCM(2, 3, 4, 5, 6) = 60$. Next we substitute our values into Equation 5.15:

$$60 \cdot G_1(5, 1) - 60 \cdot G_1(5, 2) + 60 \cdot G_1(5, 3) - 60 \cdot G_1(5, 4) + 1 \tag{5.4}$$

Every element is expanded as shown above. For example, the expansion of element $60 \cdot G_1(5, 2)$ is as follows. (Note that LCM_{ab} denotes $LCM(a, b)$).

$$\begin{aligned} 60 \cdot G_1(5, 2) = &\frac{60}{LCM_{23}} + \frac{60}{LCM_{24}} + \frac{60}{LCM_{25}} + \\ &+ \frac{60}{LCM_{26}} + \frac{60}{LCM_{34}} + \frac{60}{LCM_{35}} + \frac{60}{LCM_{36}} + \\ &+ \frac{60}{LCM_{45}} + \frac{60}{LCM_{46}} + \frac{60}{LCM_{56}} = 70 \end{aligned} \tag{5.5}$$

Figure 31: $F1$ converging to the solution for 20 $ACQs$ in 20 steps

Finally we have: $87 - 70 + 36 - 10 + 1 = 44$. This answer matches the solution from Example 8. ■

Notice that the elements of the alternating series are interchangeably increasing and decreasing the solution as we approach the end of the calculation. For 20 different $ACQs$, the calculation of overlapping edges using $F1$ consists of 20 addition operations, causing the total number to change as depicted in Figure 31.

5.2.3 Case WITH Fragments

In case some of the ranges of the $ACQs$ that are being installed onto the $DSMS$ are not divisible by their corresponding slides, according to the *Paired Window* approach, the slides should be broken into fragments. This enables us to store partial aggregates for every fragment. For example, if we have an ACQ with slide 5 sec and range 7 sec, the slide is split into two fragments: $f_2 = 7 \pmod 5 = 2$ and $f_1 = 5 - 2 = 3$. Now we can store partial results for first 3 sec, then for following 2 sec, then again for the following 3 sec and so on.

In the original *WeaveShare* [20], in order to calculate the edge rate after *weaving* together n $ACQs$ with fragments, we again need to work with a *Bit Set* of the length equal to the LCM

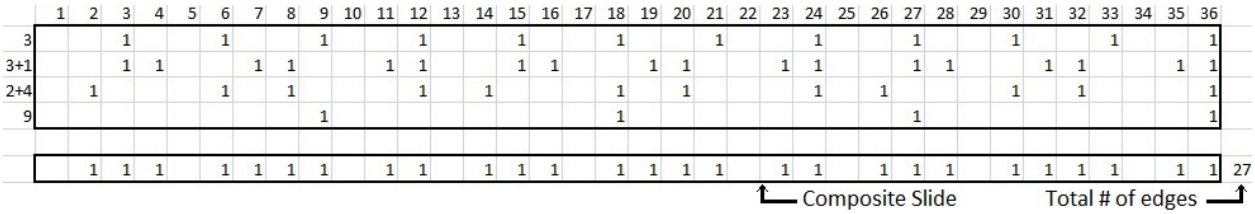

Figure 32: Marking edges produced by four different *ACQs* WITH fragments in the composite slide, represented by a Bit Set

of all n slides. The Bit Set is pre-populated with zeros again. For each ACQ we traverse the whole Bit Set and mark bits corresponding to the times when partial aggregations will happen with ones. If the bit was already marked, the algorithm does nothing and just moves to the next location.

Example 10 Consider four stock monitoring *ACQs* with the following slides: $3, 4, 6$, and 9. *ACQs* with slides of 4 and 6 consist of fragments $(3, 1)$ and $(2, 4)$ respectively. *ACQs* 3 and 9 do not have fragments. The overall LCM of all slides together is 36, therefore we need a *Bit Set* of size 36. First, we traverse the *Bit Set* and mark all indexes divisible by 3 for the ACQ with a slide of 3 and no fragments. Now the *Bit Set* has 12 bits marked. Next consider an ACQ with a slide of 4 and fragments $(3, 1)$. We traverse the *Bit Set* by starting from 0 and adding fragment 3 followed by fragment 1 repeatedly. Thus, the *Bit Set* will be marked at indexes $3, 4, 7, 8, 11, 12$, etc. Now there are 24 marked bits. Next we continue to an ACQ with a slide of 6 and fragments $(2, 4)$. Again, we start at 0 and by adding 2 and 4 repeatedly we mark the following bits: $2, 6, 8, 12, 14$, etc., marking 27 bits in total. For the last ACQ with a slide of 9 and no fragments, we traverse the *Bit Set* at increments of size 9 and mark each 9th bit with one. The total number of set bits stays 27, because the last ACQ did not add any new bits, therefore our answer is 27. This method is illustrated in Figure 32. ■

To **generalize** $F1$ for both cases (if we do have *ACQs* with fragments and if we do not) we introduce the notion of *shifts*. Each ACQ that does not have fragments has a *shift* of zero. Each ACQ that does have fragments must be presented as two *ACQs* with the same slides, but different *shifts*. First one has a shift of zero, and the second one has a *shift* equal

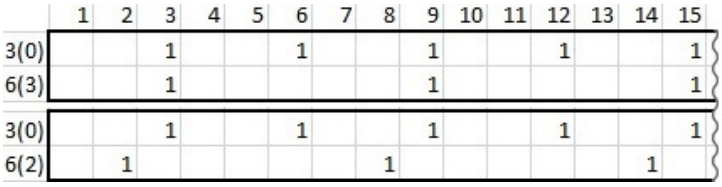

Figure 33: (slide 3, shift 0) and (slide 6, shift 3) **DO overlap**, but (slide 3, shift 0) and (slide 6, shift 2) **DO NOT**

to the first fragment of the original ACQ. When counting overlapping edges of $ACQs$, and when at least one of their shifts is not zero, we can encounter two different cases:

- $ACQs$ overlap, and the number of common edges is the same, as it would be if all of the $ACQs$' shifts were zeros.
- $ACQs$ do not overlap at all. Since the shifts are not compatible, the number of common edges is zero.

Example 11 Assume two $ACQs$ with slides of 3 and 6. If their corresponding shifts are 0 and 3, there is an overlapping edge every 6 time units. However, if the corresponding shifts are 0 and 2, there are no overlapping edges. This is illustrated in Figure 33. ■

To decide whether two $ACQs$ q_1 and q_2 (if at least one of them has non-zero shift) will overlap, we propose the following **Overlap Check Formula** based on GCD (*Greatest Common Divisor*):

$$|q_1.shift - q_2.shift| \mod GCD(q_1.slide, q_2.slide) \tag{5.6}$$

- If the *Overlap Check Formula* resolves to *zero* then the $ACQs$ **DO** overlap
- *Otherwise* the $ACQs$ **DO NOT** overlap

Proof of the **Overlap Check Formula** (by contradiction) Assume that we have two $ACQs$ q_1 and q_2, with corresponding slides s_1 and s_2, and shift difference h. Assume further that h *mod* $GCD(s_1, s_2) \neq 0$ and (for the sake of contradiction) the $ACQs$ *DO* overlap. Let us denote all edges produced by q_1 as $\{e_{1-1}, e_{1-2}, \ldots, e_{1-n}\}$, and edges of q_2 as

$\{e_{2-1}, e_{2-2}, \ldots, e_{2-n}\}$. Let us first look at the two *ACQs* separately. Since every edge produced by q_1 is divisible by s_1, and every edge produced by q_2 is divisible by s_2, and both s_1 and s_2 are divisible by $GCD(s_1, s_2)$ (by definition of GCD), every edge produced by q_1 and q_2 is divisible by $GCD(s_1, s_2)$. Therefore, all edges that are not divisible by $GCD(s_1, s_2)$ cannot possibly overlap any of the edges produced by either q_1 or q_2. Without loss of generality, let us consider q_2 from the standpoint of q_1. Then, all edges of q_2 are shifted by h with respect to edges of q_1, and they can be written as follows: $\{e_{2-1} + h, e_{2-2} + h, \ldots, e_{2-n} + h\}$. These edges should be divisible by $GCD(s_1, s_2)$ in order for them to overlap the edges of q_1: $\{e_{1-1}, e_{1-2}, \ldots, e_{1-n}\}$. We know that the edges $\{e_{2-1}, e_{2-2}, \ldots, e_{2-n}\}$ are divisible by $GCD(s_1, s_2)$. However, by the initial assumption, the shift h that is being added to them is not divisible by $GCD(s_1, s_2)$. Thus, edges $\{e_{2-1} + h, e_{2-2} + h, \ldots, e_{2-n} + h\}$ cannot possibly be divisible by $GCD(s_1, s_2)$. Therefore, none of the edges of q_2 can possibly overlap with the edges of q_1. In the case that h would actually be divisible by $GCD(s_1, s_2)$, all shifted edges of q_2 that are divisible by s_1 would overlap with the edges of q_1. However, the initial assumption states that h is not divisible by $GCD(s_1, s_2)$, which leads us to the conclusion that the *ACQs* q_1 and q_2 do not overlap, which is a contradiction. Hence, the initial formula is correct. ■

Next we generalize our formula $F1$ for use in cases when *ACQs* have fragments, and cases when none of the *ACQs* have fragments. The **general F1** is:

$$LCM_n \sum_{i=1}^{n} [(-1)^{i+1} G_2(n, i)] \tag{5.7}$$

Where again $LCM_n = LCM(s_1, s_2, \ldots, s_n)$, and function G_2 is the same as function G_1, however all elements produced by G_2 have to be checked with the *Overlap Check Formula* for redundancy as described below. Prior to using this formula, for each *ACQ* that has fragments, we create two new *ACQs*: one of them has a shift of zero, another one has a shift equal to the first fragment of the original *ACQ*. All new *ACQs* are added back to the set of the original *ACQs* replacing the originals. To calculate each G_2 we find all possible groups of size x from the new set of *ACQs* just like in the case with no fragments. Some of these groups are redundant because they do not have overlapping edges (because of the shifts). To remove all redundant groups, we check all possible pairs within each group using

the *Overlap Check Formula*, and if any of the pairs return a non-zero value, then the whole group is discarded. Otherwise, G_2 is calculated and used the same way as in the case with *NO* fragments. The generalized formula $F1$ still converges, which can be proven using the same strategy as in the case with no fragments.

Equation 5.7 expands into an alternating series likewise:

$$LCM_n[G_2(n,1) - G_2(n,2) + \ldots \pm G_2(n,n-1) \mp G_2(n,n)] \tag{5.8}$$

We show how Equation 5.8 works with the following example.

Example 12 Assume the same set of stock monitoring *ACQs* as in Example 10: slides are 3, 4, 6, and 9, and *ACQs* with slides of 4 and 6 consist of fragments $(3,1)$ and $(2,4)$ respectively. As a first step of our algorithm we calculate the LCM_n of the whole set of slides. $LCM_n = LCM(3,4,6,9) = 36$. Next we replace the *ACQs* that have fragments with the *ACQs* that have corresponding shifts. In our set we now have two *ACQs* with a slide of 4 (shifts 0 and 3), and two *ACQs* with a slide of 6 (shift 0 and shift 2). The rest of the *ACQs* stay the same. We can substitute our values into the generalized formula $F1$:

$$36 \cdot G_2(6,1) - 36 \cdot G_1(6,2) + 36 \cdot G_1(6,3) - 36 \cdot G_1(6,4) \tag{5.9}$$

The calculation is almost identical to the case with no fragments, except every group produced by G_2 has to be checked with the *Overlap Check Formula* to see if it is redundant or not. For example, the expansion of the second group is shown below. Note that $3_0 4_3$ denotes a group of *ACQs* with slides of 3 and 4 and shifts of 0 and 3, respectively. The fractions that have been crossed out did not pass the test with the *Overlap Check Formula*.

$$\begin{aligned} 36 \cdot G_1(6,2) = {} & \frac{36}{LCM_{3_0 4_3}} + \frac{36}{\cancel{LCM_{3_0 6_2}}} + \frac{36}{LCM_{3_0 9_0}} + \\ & \frac{36}{LCM_{3_0 4_0}} + \frac{36}{LCM_{3_0 6_0}} + \frac{36}{\cancel{LCM_{4_3 6_2}}} + \frac{36}{LCM_{4_3 9_0}} + \\ & \frac{36}{\cancel{LCM_{4_3 4_0}}} + \frac{36}{\cancel{LCM_{4_3 6_0}}} + \frac{36}{\cancel{LCM_{6_2 9_0}}} + \frac{36}{LCM_{6_2 4_0}} + \\ & \frac{36}{\cancel{LCM_{6_2 6_0}}} + \frac{36}{LCM_{9_0 4_0}} + \frac{36}{LCM_{9_0 6_0}} + \frac{36}{LCM_{4_0 6_0}} = 26 \end{aligned} \tag{5.10}$$

Finally we have: $46 - 26 + 8 - 1 = 27$. This answer matches the solution from Example 10.

■

5.2.4 F1 Optimization

Since we are using the Euclidean GCD algorithm for all of our LCM calculations, we found that we can achieve a significant additional speed up by utilizing the technique of memorization. We adopted this technique by preloading a table of GCDs into main memory before the execution begins. If the user is willing to allocate b bytes of memory to store the GCD table and each GCD takes g bytes of memory, we can store in memory GCDs of all the possible pairs of numbers up to $\sqrt{2b/g}$. In our implementation we are using 8 byte numbers of the Long type for calculations, so if we want to allocate 4 GB of main memory to store $GCDs$, we can fit GCDs of all the pairs of numbers up to $32,768$. If we calculate the GCD for numbers that are larger than the above limit, the GCD table still save us some time by taking advantage of the recursive nature of the Euclidean algorithm. The effects of the optimization are shown in Section 5.4.

5.3 Complexity Analysis

In this section, we calculate the difference between the complexities of *Bit Set* calculation and our $F1$ method.

Time Complexities To compare the time complexities we start by identifying the initial calculations needed by both algorithms. We denote the number of $ACQs$ as n, and the max slide as max. The following steps need to be done at the beginning of both algorithms.

- Remove all duplicate slides, since the same slides produce the same edges, and we do not want to repeat the same calculation for every duplicate. This is done by sorting slides with duplicate removal in $n \cdot logn$ time.
- Precalculate the LCM_n, which is the LCM of all slides and store it in the main memory. This operation takes $(n-1) \cdot log(max)$ at worst, since we need to perform LCM operation pairwise $n-1$ times, and each $LCM(a,b)$ needs at worst $log(min(a,b))$ operations [51].
- Remove all slides that are multiples of other slides included in the set. We do this because all edges produced by such slides are already produced by their factors. This can be done

in $n \cdot (n-1)/2$ operations since our slide set is sorted, and for each subsequent slide we need to do a number of comparisons that equals the number of comparisons performed by the previous slide minus one.

Therefore, precalculation takes $n \cdot logn + (n-1) \cdot log(max) + n \cdot (n-1)/2$ operations, however since it is performed by both algorithms, we can ignore it for the matter of comparison. After completing the initial computation, we now have LCM_n stored in main memory, and a set of slides which does not contain any duplicates or multiples. Therefore, the set can now only have either prime numbers or numbers for which their multiples do not appear in the set.

To better illustrate differences in complexity we utilize two different sets of slides:

- Working Set (S_w) is a set of slides that includes only prime numbers and numbers that do not have their multiples in this set. This is a set produced after the initial preparation and it is used in real working scenarios. An example of a valid working set: $S_w = 3, 5, 8, 14, \ldots, max$. $|Sw| = n$, and the maximum element is denoted as max.
- Auxiliary set (S_a) is a set of slides, that consists of sequential numbers. $Sa = 1, 2, \ldots, n$. $|Sa| = n$, and the maximum value max equals n. Note that this set contains all natural numbers from one to n, including multiples of other numbers from this set. It is still a valid set for our computation, and can be obtained by skipping the preliminary optimization that removes all multiples.

Next we show that the lower bound of the *Bit Set* calculation is higher than the upper bound of the $F1$ computation.

The complexity of the *Bit Set* calculation is:

$$\sum_{i=1}^{n} \frac{LCM_n}{s_i} \tag{5.11}$$

Where $LCM_n = LCM(s_1, s_2, \ldots, s_n)$. The complexity holds since for each of n $ACQs$ we would need to traverse the whole *Bit Set*, whose length is equal to the LCM of all slides of all $ACQs$, with a step equal to each ACQ's slide. We can expand the Equation 5.11 to the following:

$$LCM_n \cdot (\frac{1}{s_1} + \frac{1}{s_2} + \ldots + \frac{1}{s_n}) \tag{5.12}$$

First, we perform complexity analysis using the Auxiliary set S_a. Let us focus on the first part of the product in Equation 5.12: LCM_n. We know that the LCM of all numbers in this set is the product of the highest prime powers occurring in the set. The log of the LCM is therefore the sum of the $logs$ of the prime powers in the set:

$$log(LCM_n) = \sum_{i=2}^{n} f(i),\ where\ f(i) = \begin{cases} log(p) & if\ i = p^m,\ where\ m \geq 1\ \&\ p\ is\ prime \\ 0 & Otherwise \end{cases} \tag{5.13}$$

This sum has significance in the *Prime Number Theorem* and it is well known to be asymptotically equal to e^n [53].

Next we focus on the second part of the product in Equation 5.12: $(\frac{1}{s_1}+\frac{1}{s_2}+\ldots+\frac{1}{s_n})$. Since our set only has sequential numbers from one to n, this part will look like: $(\frac{1}{1}+\frac{1}{2}+\ldots+\frac{1}{n})$, which is a classic example of a diverging harmonic series $\sum_{n=1}^{\infty}\frac{1}{n}$. For any n, this series can be calculated as follows: $\sum_{n=1}^{\infty}\frac{1}{n} = ln(k)+\gamma+\epsilon_k$, where γ is the Euler-Mascheroni constant ($\gamma \approx 0.577$) and $\epsilon_k \sim \frac{1}{2k}$, which approaches zero as k goes to infinity[52]. For our purposes, since γ is a constant and ϵ_k is negligible, they are ignored. Also, we can say that time complexity $e^n \cdot ln(n)$ is asymptotically equal to e^n, therefore we can assume asymptotical time complexity of the *Bit Set* computation for set S_a is e^n.

When we use set S_w, the time complexity for LCM_n is larger than when using set S_a. This is true because we replace n sequential numbers that start from 1 with n non-duplicate primes and non-multiples, which are larger and have a larger total LCM. The time complexity for the part $\sum_{i=1}^{n}\frac{1}{s_i}$ becomes smaller, because we are increasing numbers in the denominator, however it is still insignificant, because even in the worst case we can lower bound it with $ln(max)-ln(max-n)$. Thus, the time complexity of the *Bit Set* computation for the working set S_w is at least e^n.

To calculate the complexity of $F1$, we need to determine the number of operations that need to be performed based on the size of the input. First, we know that the number of elements in our alternating series is equal to the number of $ACQs$ in the set. Let us take

Equation 5.3 and expand LCM_n into the parentheses:

$$\begin{aligned} &LCM_n \cdot G_1(n,1) - LCM_n \cdot G_1(n,2) + \ldots \\ &\pm LCM_n \cdot G_1(n,n-1) \mp LCM_n \cdot G_1(n,n) \end{aligned} \tag{5.14}$$

As we previously mentioned, $LCM_n \cdot G_1(n,n) = 1$, therefore:

$$\begin{aligned} &LCM_n \cdot G_1(n,1) - LCM_n \cdot G_1(n,2) + \ldots \\ &\pm LCM_n \cdot G_1(n,n-1) \mp 1 \end{aligned} \tag{5.15}$$

To determine how many groups will be produced by each one of these elements we use binomial coefficients. Each element of type $LCM_n \cdot G_1(n,k)$ therefore produces $\binom{n}{k}$ of distinct k-element groups of type $\frac{LCM_n}{LCM_k}$, where $LCM_k = LCM(s_1, s_2, \ldots s_k)$. Therefore, the total number of all of these groups is: $\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n-1} + \binom{n}{n}$. By the additive property of binomial coefficients, this sum equals $2^n - 1$. Next we determine how many calculations are performed in each group. The numerator of all groups is LCM_n and since it is kept in main memory, we do not need to recalculate it every time. The denominator is LCM_k, and it is determined by calculating the LCM of the first two elements, and then iteratively calculating the $LCMs$ of the resulting number with the rest of the elements in the group. Therefore, for each group we need to perform $k-1$ LCM calculations, and one calculation to add the group to the total number, which makes k calculations total. Since each group with k elements needs k calculations, the total number of calculations needed for all groups becomes: $1\binom{n}{1} + 2\binom{n}{2} + 3\binom{n}{3} + \ldots + (n-1)\binom{n}{n-1} + n\binom{n}{n}$. This resolves to $2^{n-1} \cdot n$, which can be calculated by taking the generalization of binomial series: $(1+x)^a = \sum_{k=0}^{\infty} \binom{a}{k} \cdot x^k$ and differentiating it with respect to x and then substituting $x = 1$ [50]. Due to the use of the Euclidean algorithm to calculate the LCM, the complexity of each LCM calculation is $log(min(a,b))$ at most [51]. Therefore, at worst $F1$ has a time complexity of $2^{n-1} \cdot n \cdot log(max)$, which asymptotically equals 2^n.

Thus, we have determined that the *Bit Set* calculation has a time complexity of **at least e^n**, and $F1$ has a time complexity of **at worst 2^n**. Clearly, when n goes to infinity, it is increasingly beneficial to use $F1$ versus *Bit Set*.

Additionally, since we have calculated the formulas for determining the exact number of operations done by both *Bit Set* and $F1$, we can compare the increase in the amount of

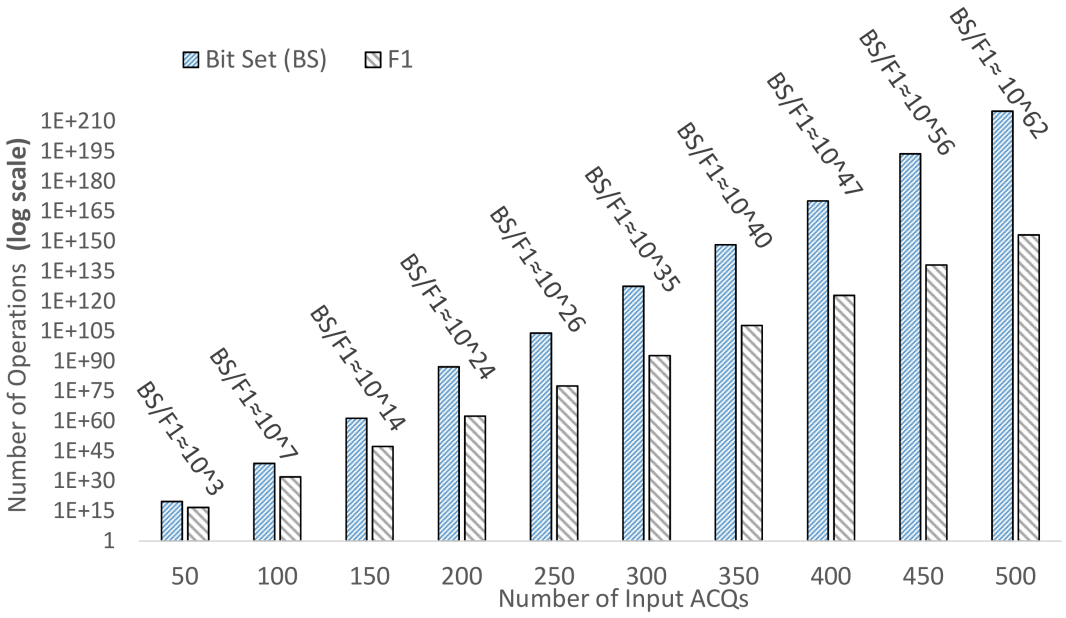

Figure 34: Number of operations needed by Bit Set and $F1$ for plan generation. Top labels show BitSet/$F1$ ratio

operations performed by *Bit Set* and $F1$ with the increase of the number of input *ACQs*. The comparison is shown in Figure 34. *ACQs* for this comparison were sequentially drawn from the Auxiliary set (S_a) introduced above. Note that since the difference between *Bit Set* and $F1$ operation numbers is drastic and grows exponentially we had to use a logarithmic scale to still see the operations of $F1$. This comparison shows that $F1$ is much more scalable than *Bit Set* in terms of the number of operations required.

Space Complexities The space complexity of the *Bit Set* calculation is LCM_n, since we have already shown that the *Bit Set* grows at the rate of $\boldsymbol{e^n}$. The space complexity of the $F1$ calculation is $\boldsymbol{O(1)}$ (*constant*) since it does not require storing edges. Edge overlaps are calculated strictly mathematically. Since $F1$ expands into a sum, we only need to keep one number in memory, which is increased or decreased by the elements of the alternating series sequentially. The improvement in space complexity is extremely important for the *WeaveShare* algorithm, since the leading cause of its failures with large workloads is "out of memory" errors.

5.4 Experimental Evaluation

In this section, we summarize the results of our experimental evaluation of the scalability of $F1$ in terms of the size of the input set of the $ACQs$, the diversity of their time properties, and the input rate of the data stream.

5.4.1 Experimental Testbed

In order to show the significance of our *Weavability* calculation optimization we built an experimental platform in Java. Specifically, we implemented the *WeaveShare* optimizer as described in [20] with different options for calculating *Weavability*. Our **workload** is composed of a number of $ACQs$ with different characteristics. We are generating our workload synthetically in order to be able to fine-tune system parameters and get a more detailed sensitivity analysis of the optimizer's performance. Moreover, it allows us to target possible real-life scenarios and analyze them.

Our system's **experimental parameters** are:

[$\boldsymbol{Algorithm}$] specifies which technique is used for *Weavability* calculations. The available techniques are: (a) Bit Set (BS), (b) Formula 1 ($F1$), and (c) Formula 1 + Optimization ($F1 + Opt$). The $F1 + Opt$ technique uses a 4 GB table for keeping GCDs in main memory.

[$\boldsymbol{Q_{num}}$] Number of $ACQs$. We assume that all $ACQs$ are installed on the same data stream and their aggregate functions allow them to share partial aggregations among them. The actual function does not have any effect on performance other than the ability to share partial aggregations.

[$\boldsymbol{S_{max}}$] Maximum slide length, which provides an upper bound on how large slides of our $ACQs$ can be. The minimum slide allowed by the system always equals one.

[$\boldsymbol{\lambda}$] The input rate, which describes how fast tuples arrive through the input stream in our system.

[$\boldsymbol{Z_{skew}}$] Zipf distribution skew, which depicts the popularity of each slide length in the final set of $ACQs$. A Zipf skew of zero produces uniform distribution, and a greater Zipf skew is skewed towards large slides (for more realistic examples).

Table 5: Experiment Parameters

#	Q_{num}	S_{max}	λ	Z_{skew}	O_{max}	Gen
1	*100 - 1M*	1K	0.002	0.5	10	Nrm
2	1K	*100-10*K	1	3	100	Div
3	1K	100	*100-1M*	1	1K	Nrm
4	50	500	0.1	*0-100*	100	Nrm
5	1K	600	1	3	*100-1*	Div

[$\boldsymbol{O_{max}}$] Maximum overlap factor, which defines the upper bound for the overlap factor. The overlap factor of each ACQ is drawn from a uniform distribution between one and the maximum overlap factor.

[***Gen***] Generator type, which defines whether the workload is normal (Nrm), which includes any slides or diverse (Div), which includes only slides of a length that is a prime number. When the slides are prime, their LCM is equal to their product, which makes it more difficult to share partial aggregations.

We ran all our experiments on a dual processor 8 core Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz server with 96 GB of RAM available. All results are taken as averages of running each experiment five times.

5.4.2 Experimental Results

To test the scalability of our approaches $F1$ and $F1 + Opt$ versus BS in terms of the parameters Q_{num}, S_{max}, λ, Z_{skew}, and O_{max}, we ran five experiments, where we varied each one of these parameters while keeping the rest of them fixed. The parameters were selected separately for each experiment in a way that would highlight the differences in the scalabilities of the three approaches the best. The experimental parameters are specified in the Table 5. Please note that since $F1$ and $F1 + Opt$ showed to have significantly smaller runtimes compared to BS, we had to use **logarithmic scale** to be able to display all techniques' performances in the same graphs.

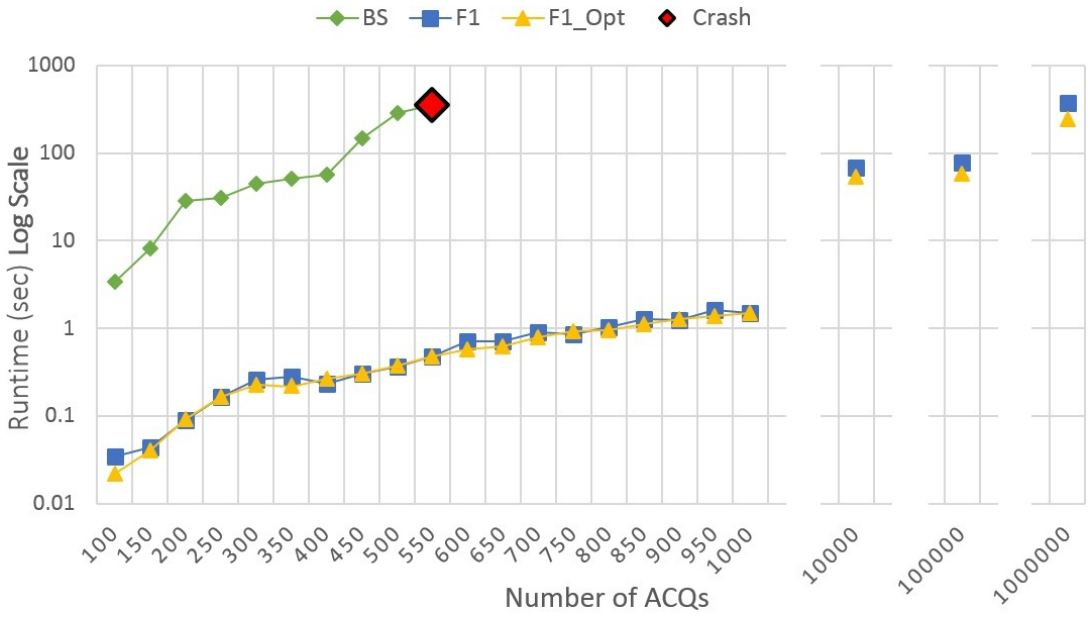

Figure 35: Scalability of the number of ACQs

5.4.2.1 Exp 1: Number of ACQs Scalability (Figure 35). In this test we varied the Q_{num} from 100 to 1,000,000 (given that simultaneously processing such large numbers of queries is becoming increasingly crucial [21, 23]). Clearly, increasing the Q_{num} also increases the amount of required calculations, causing higher runtimes for all three algorithms. The results are depicted in Figure 35. The *Bit Set* approach did not finish execution, because after we crossed Q_{num} of 550 it started running out of memory (on a 96GB RAM machine) and eventually crashed (on all runs). Otherwise, the growth rates of these techniques are similar to what we expected from the theoretical analysis of the time complexities of their underlying algorithms. The statistics show that our techniques' runtimes are on average 350 times faster than runtimes of BS with a maximum of 790 times, and our techniques are able to scale up to 1,000,000 $ACQs$ on this setting without running out of memory. Also, the $F1 + Opt$ plan outperformed the $F1$ plan by approximately 28% on average, validating our optimization expectations.

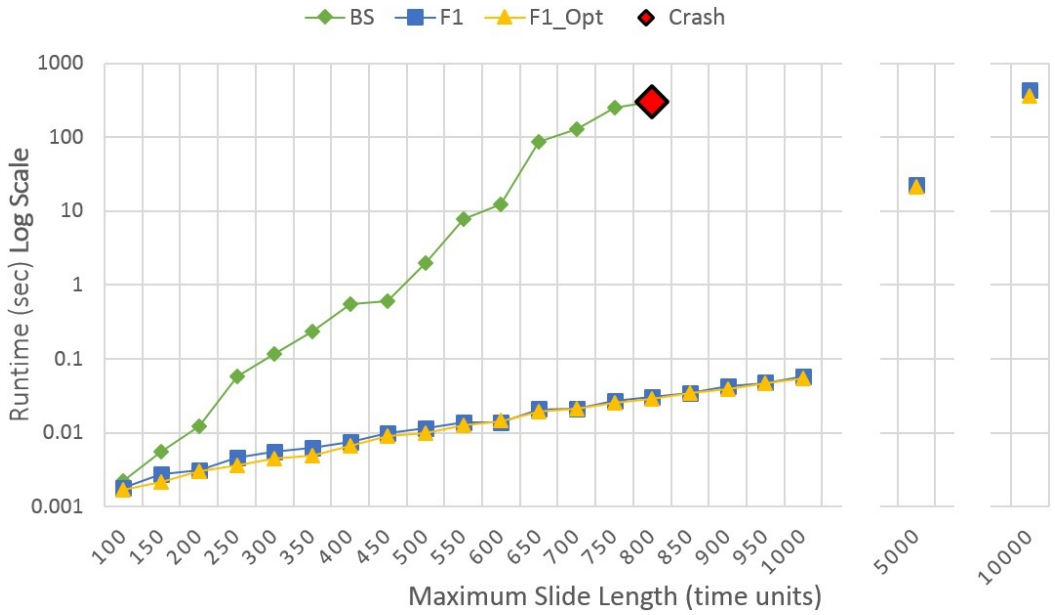

Figure 36: Scalability of the maximum slide length

5.4.2.2 Exp 2: Max Slide Scalability (Figure 36). In this test we varied the S_{max} from 100 to 10,000. Similarly to Exp 1, increasing the S_{max} also increases the amount of required calculations. This happens because with a higher max slide parameter, the generated $ACQs$ have longer slides, which results in higher $LCMs$ and fewer overlapping edges. In Figure 36 we see that the BS approach did not finish execution again after we crossed S_{max} of 800 because of the "out of memory" error. The growth rates of our techniques however are again similar to what we expected from our theoretical analysis of their time complexities. Our proposed techniques' runtimes in this experiment are on average 2,200 times faster than runtimes of BS with a maximum of 10,000 times, and our techniques are able to scale up to the S_{max} of 10,000 $ACQs$ on this setting and finish the plan generation successfully. The $F1 + Opt$ plan outperformed the $F1$ plan by an average of 18%.

5.4.2.3 Exp 3: Input Rate Scalability (Figure 37). In this test we varied λ from 100 to 100,000. Increasing λ also increases the amount of required calculations because with higher input rates, according to the Equation 7.1, it becomes more beneficial to combine

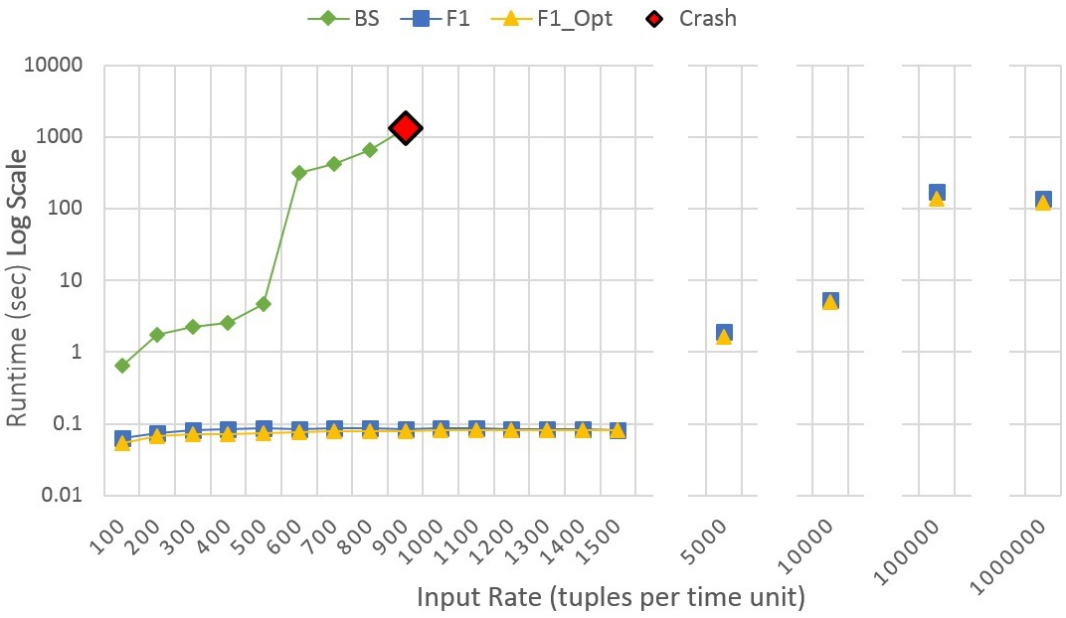

Figure 37: Scalability of the input rate

more execution trees. This forces *WeaveShare* to combine some trees with different time properties that would not have been combined if the input rate was lower. Thus, increasing λ leads to higher LCMs and higher runtimes. The average number of execution trees formed at the end of the plan generation is 71 when $\lambda = 100$, 29 when $\lambda = 400$, 15 when $\lambda = 900$, 4 when $\lambda = 10,000$ and 1 when $\lambda = 100,000$ or 1,000,000. The fact that at some λ all trees get merged into one explains why runtimes stop increasing after this λ. In this setting it happened at $\lambda = 100,000$ (see Figure 37). In this experiment the BS approach crashed when λ reached 900, and the average number of trees at that point was 15. Our approaches demonstrated good scalability again and were able to increase input rate to the point where all trees are *Weaved* into one. On average our techniques ran 3,800 times faster than BS with a maximum of 16,000. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 19%.

5.4.2.4 Exp 4: Slide Skew Sensitivity (Figure 38). In this test we varied the Z_{skew} from 0 to 100. This experiment is similar to the max slide scalability experiment, because

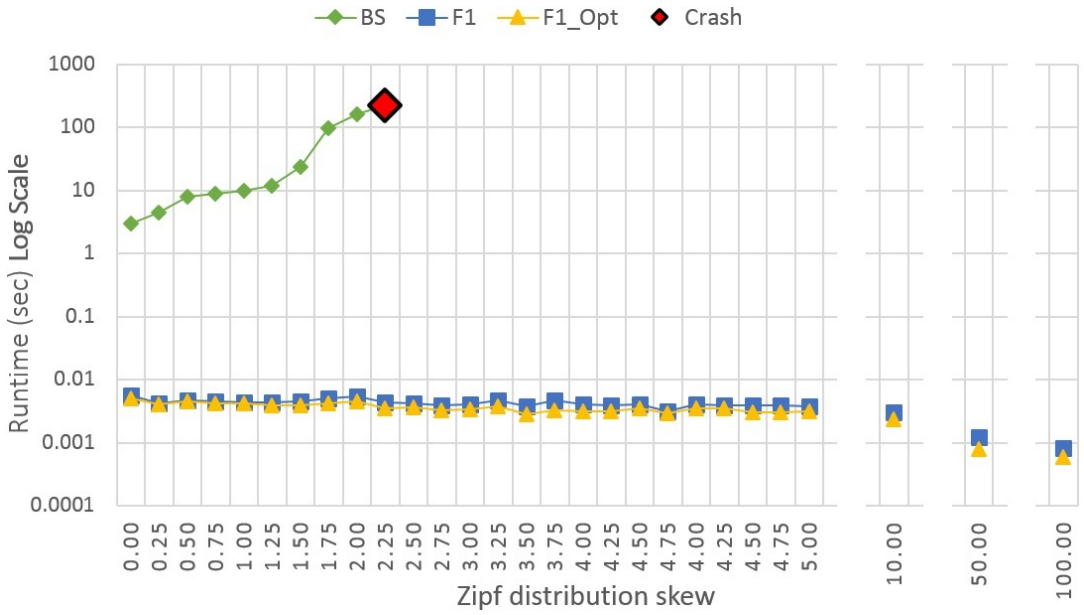

Figure 38: Sensitivity to the zipf distribution skew

in both experiments we are gradually increasing the amount of $ACQs$ with large slides and therefore increasing the amount of required calculations. The difference is that, when skewing all slides drawn from the same set to the larger side, at some point they start repeating, which then reduces the amount of the required calculations. In our experiment (see Figure 38) we first observe the initial increase in the amount of computation, which leads the BS approach to crash with an "out of memory" error (at $Z_{skew} = 2.25$), and then we see gradual decrease in computation, because there are many repeating slides in the input set. In this setting our proposed techniques' runtimes are on average 14,000 times faster than runtimes of BS with a maximum of 60,000 times, and our techniques are able to scale up to the Z_{skew} of 100 and finish the plan generation successfully. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 18% again.

5.4.2.5 Exp 5: Overlap Factor Sensitivity (Figure 39). In this test we varied the O_{max} from 100 to 1. We did it in reverse order since its value is inversely proportional to the amount of computation required to generate an execution plan using *WeaveShare*. Based

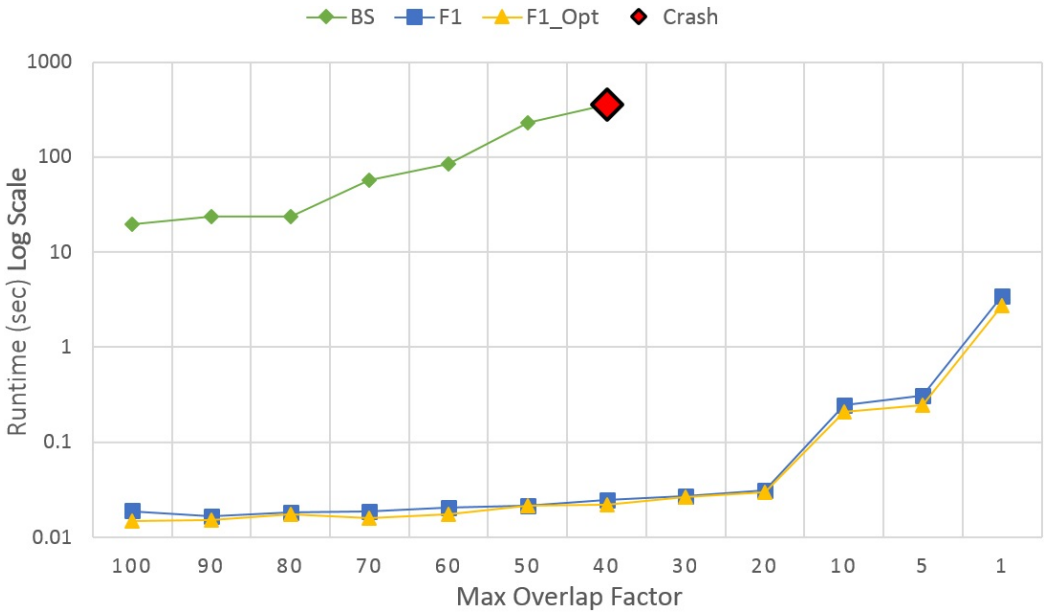

Figure 39: Sensitivity to the maximum overlap factor

on Equation 7.1 we can see that smaller O_{max} benefits the total cost if the corresponding *ACQs* are combined to fewer execution trees, which causes *WeaveShare* to *Weave* more trees with different time properties together. In our experiment (see Figure 39) the BS approach crashed when O_{max} reached 40. Our approaches again demonstrated good scalability and were able to finish the plan generation successfully even with the minimum value of $O_{max} = 1$. On average our techniques ran 5,600 times faster than BS with a maximum of 16,000. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 26%.

5.4.2.6 Experimental Results Summary

Clearly, the above experimental results show that our techniques $F1$ and $F1 + Opt$ deliver the best performance in terms of plan generation runtimes and scalability, while producing same high quality execution plans as the original *WeaveShare* optimizer. These drastic improvements result from substituting the expensive count-based approach with a more efficient compute-based one. The results of our experiments are summarized in Table 6.

Table 6: Experimental Results

Experiment		Best Achieved		Runtime: $BS/F1$		$F1$ vs
#	Param	BS	$F1$	Avg	Max	$F1 + Opt$
1	Q_{num}	550*	1M	350	790	28%
2	S_{max}	800*	10K	2,200	10,000	18%
3	λ	900*	1M	3,800	16,000	19%
4	Z_{skew}	2.25*	100	14,000	60,000	18%
5	O_{max}	40*	1	5,600	16,000	26%

*Execution stopped with "out of memory" exception

5.5 Summary

The main contribution of this chapter is a novel closed formula, $F1$, for accelerating *Weavability* calculations required for determining the best execution plans for sharing partial aggregations of $ACQs$. Our approach replaces the counting of the edges within a *Bit Set* with mathematical computation and is applicable for all cases (with and without fragments).

We theoretically evaluated the state-of-the-art *Weavability* calculation approach against $F1$ and provided a mathematical proof that $F1$ improves the time complexity from *at least* $\boldsymbol{e^n}$ to *at most* $\boldsymbol{2^n}$, and the space complexity correspondingly from *at least* $\boldsymbol{e^n}$ to **constant**. Thus, $F1$ significantly decreases the number of operations required for the plan generation while reducing the algorithm's space consumption to the bare minimum.

We showed experimentally that the $F1$ approach achieves up to 60,000 times faster plan generation compared to the current state of the art, and is able to achieve much better scalability in terms of the number of input $ACQs$, their diversity, and the input rate of the data stream. We showed that $F1$ is able to successfully process 1,000,000 $ACQs$ whereas the limit of the current technique is 550.

It should be noted that $F1$ can reduce the computation time of any optimization technique that requires scheduling partial aggregations within composite slides of multiple $ACQs$. In the next chapter we will show this in the context of MQ optimization in a distributed environment.

6.0 Processing of Aggregate Continuous Queries in a Distributed Environment

In this chapter, we study the problem of generating high quality execution plans of *ACQs* in *DSMSs* deployed on multi-node (multi-core and multi-processor) distributed environments. Towards this goal, we classify optimizers based on how they partition the workload among computing nodes and on their usage of the concept of *Weavability*, which is utilized by the state-of-the-art *WeaveShare* optimizer to selectively combine *ACQs* and produce low cost execution plans for single-node environments. For each category, we propose an optimizer, which either adopts an existing strategy or develops a new one for assigning and grouping *ACQs* to computing nodes. We implement and compare all of our proposed optimizers in terms of (1) keeping the total cost of the *ACQs* execution plan low and (2) balancing the load among the computing nodes.

In the next section, we provide the rationale for this work. We describe the challenges of producing execution plans for distributed processing environment and define the new performance metrics in Section 6.2. The categorization and various multi-node optimizers are presented in Section 6.3 and their descriptions in Secs. 6.4 and 6.5. The evaluation platform and the quality of produced multi-node plans are discussed in Section 6.6. We conclude in Section 6.7.

6.1 Introduction

The state-of-the-art *WeaveShare* optimizer is a cost-based *ACQ* optimizer that produces low cost execution plans by utilizing the concept of *Weavability* [20]. Since *WeaveShare* is targeting single-node *DSMSs*, it is oblivious to distributed processing capabilities, and as our experiments have revealed, *WeaveShare* cannot produce *ACQ* execution plans of equivalent cost that can be assigned to the various computing nodes. This motivated us to address the problem of generating high quality execution plans of *ACQs* in *DSMSs* deployed on multi-node (multi-core and multi-processor) distributed environments with a *Weavability*-based

optimizer. Formally, given a set $\mathcal{Q}$ of all *ACQs* submitted by all clients and a set $\mathcal{N}$ of all available computing nodes in the distributed *DSMS*, our goal is to find an execution plan $\mathcal{P}(\mathcal{Q}, \mathcal{N}, \mathcal{T})$ that maps $\mathcal{Q}$ to $\mathcal{N}$ ($\mathcal{Q} \rightarrow \mathcal{N}$) and generates a set $\mathcal{T}$ of local *ACQ* execution trees per node, such that the total cost of the *ACQs* execution is low and the load among the computing nodes is balanced.

The rationale behind these two optimization criteria is (Section 6.2):

- *Minimizing the total cost* of the execution plan allows the system to support more *ACQs*. In the case of the Cloud, since Cloud providers charge money for the computation resources, satisfying more client requests using the same resources results in less costly client requests.
- *Balancing the workload* among computation nodes saves energy while still meeting the requirements of the installed *ACQs*, which directly translates to monetary savings for the distributed infrastructure providers. Additionally, it is advantageous for the providers to maintain load balancing, because it prevents the need to over-provision in order to cope with unbalanced workloads.

6.2 System Model and Execution Plan Quality

In this chapter, we assume a typical *DSMS* deployed over a set of servers (i.e., computing nodes). These servers can be a local cluster or on the Cloud and are capable of executing any *ACQs* using partial aggregation. Submitted *ACQs* are assumed to be independent of each other and have no affinity to any server. Furthermore, without a loss of generality, we target *ACQs* that perform similar aggregations on the same data stream.

In a single node system, the main metric defining the quality of an execution plan is the *Cost* of the plan. The *Cost* of the plan is measured in operations per second. That is, if the plan cost is X, then we would need a server that can perform at least X operations per second in order to execute this plan and satisfy all users by returning the results of their *ACQs* according to their specified range and slide.

In the context of the distributed environment, we have to split our workload between the

available nodes. Since our workload consists of *ACQs*, we can assign them to the available computing nodes in the system and group them into execution trees within these nodes. Thus, in any distributed environment, the *Total Cost* of a plan P is calculated as a sum of all costs C_i (according to the Equation 7.1) of all n nodes in the system:

$$TotalCost(P) = \sum_{i=1}^{n} C_i \tag{6.1}$$

This metric is important for the Cloud environment, because lowering the total cost T allows *DSMSs* to handle larger numbers of different *ACQs* on the same hardware, which in turn can potentially lower the monetary cost of each *ACQ* for the clients.

Another important metric in a distributed environment is the *Maximum node cost* of all computational nodes. The maximum node cost of a plan P is calculated by finding the highest cost C_i of all n nodes in the system:

$$MaxCost(P) = Max_i^n C_i \tag{6.2}$$

Minimizing the *Max Cost* is vital for distributed *DSMSs* with heavy workloads. In such a case, if we optimize our execution plans purely for the *Total Cost*, due to the heavy workload, the *Max Cost* can become higher than the computational capacity of the highest capacity node in the system, and the system will not be able to accommodate this execution plan. Furthermore, it is advantageous for the providers to maintain load balancing, because it prevents the need for over-provisioning in order to cope with unbalanced workloads.

Additionally, good load balancing could enable power management that executes *ACQs* at lower CPU frequency. This could lead to significant energy savings, ergo monetary savings, given that the energy consumption is at least a quadratic function of CPU frequency [55].

Table 7: Optimizer Categories

		Optimizers				
		Non-Cost-based		Cost-based		
		Random	Round Robin	to Lowest	to Nodes	inserted
Categories	Group Only	G_{RAND}	G_{RR}	G_{TL}	-	-
	Weave Only	W_{RAND}	W_{RR}	W_{TL}	W_{TN}	W_I
	Weave + Group	WG_{RAND}	WG_{RR}	WG_{TL}	WG_{TN}	WG_I

6.3 Taxonomy of Optimizers

As mentioned in the Introduction, in order to structure our search for a suitable multi-query optimizer for a distributed *DSMS* in a systematic way, we categorize possible *ACQ* optimizers based on how they utilize the concept of *Weavability* for both non-cost-based and cost-based optimization. This taxonomy is shown in Table 7. Below, we highlight the underlying strategy of each category.

Group Only This category allows for the grouping of *ACQs* on different computation nodes. No sharing of final or partial aggregations between *ACQs* is allowed. Optimizers in this category are expected to be effective in environments where sharing partial aggregates is counter productive, for example, when there are no similarities between periodic properties of *ACQs*. Even though there is no sharing between *ACQs* in this category, it is still essential to maintain the load balance between computation nodes in a distributed environment. Since node costs in this case are calculated trivially by adding together separate costs of *ACQs* running on this node, there can be many analogies (such as CPU scheduling in OS) to optimizers from this category.

Weave Only This category allows the sharing of final and partial aggregations between *ACQs*. The *Weavability* concept is used in this category to generate the number of execution trees matching the number of available nodes. As a result, only one execution tree can

be present on each computation node in the resulting plan. Optimizers in this category are expected to be effective in the environments where partial result sharing is highly advantageous, for example, if the submitted *ACQs* all have similar periodic properties (*ACQ* slides are the same or multiples of each other).

Weave and Group This category allows both the sharing of aggregations between *ACQs* within execution trees and the grouping of them on different computation nodes. Thus, multiple execution trees can be present on any node. Optimizers in this category are attempting to be adaptive to any environment and produce high quality execution plans in different settings by collocating and grouping *ACQs* in an intelligent way.

6.4 Non-Cost-based Optimizers

In this section, we provide the details on the *Non-Cost-based* optimizers, which we further classified as *Random* and *Round Robin* optimizers. Random and Round Robin optimizers iterate through a set of input *ACQs*, selecting a node for each *ACQ* in a random or round robin fashion respectively.

Depending on the way *ACQs* on a node are woven,

- G_{RAND} & G_{RR} ($GroupOnly$) add the *ACQs* to the selected node as a separate tree.
- W_{RAND} & W_{RR} ($WeaveOnly$) *weave* the *ACQs* into a single, shared tree on the node.
- WG_{RAND} & WG_{RR} ($WeaveAndGroup$) choose (in random or round robin fashion) whether to add this *ACQ* as a separate tree, or to *weave* it with one of the available trees on this node.

6.5 Cost-based Optimizers

In this section, we provide the details on the second class of optimizers: *Cost-based* optimizers (Table 7), which includes three categories: "To Lowest", "To Nodes", and "Inserted".

Note that no representatives for the "Group Only – Insert" and "Group Only – To Nodes" categories are listed in Table 7 because in both cases the representative is effectively G_{TL} without weaving. In all optimizers, we consider the initial cost of each node to be zero.

6.5.1 Category "To Lowest"

Optimizers in this category follow the "To Lowest" algorithm shown in Algorithm 4.

Algorithm 4 The *"To Lowest"* Algorithm

Input: A set of Q Aggregate Continuous Queries, N computation nodes, and *Category*
Output: Execution plan P
Create an execution tree $(t_1, t_2, \ldots, t_Q)$ for each query
Calculate costs for all execution trees $(c_1, c_2, \ldots, c_Q)$
Sort all execution trees from expensive to cheap
Assign N most expensive trees to N nodes $(n_1, n_2, \ldots, n_N)$ ▷ assign one tree per node
$T \leftarrow Q - N$ ▷ T is the number of remaining trees to be grouped/weaved
for $i = 0$ *to* T **do** ▷ iterate over the trees until all are grouped/weaved to nodes
  $MinNode \leftarrow findMinNode()$ ▷ determine the node with the current smallest cost
  switch *Category* **do**
    case *GroupOnly* ▷ each node can have multiple trees
      $group(t_i, MinNode)$ ▷ group t_i as a separate tree to $MinNode$
    case $WeaveOnly$ ▷ each node can have only one tree
      $weave(t_i, MinNode)$ ▷ weave t_i to the tree in $MinNode$
    case $WeaveAndGroup$ ▷ each node can have multiple trees
      $Cost_1 \leftarrow group(t_i, MinNode)$ ▷ new cost of $MinNode$ if t_i is grouped to $MinNode$
      $MinTree \leftarrow findMinTree(MinNode)$ ▷ minimal costing tree in $MinNode$
      $Cost_2 \leftarrow weave(t_i, MinNode)$ ▷ new cost of $MinNode$ if t_i is weaved to $MinTree$
      if $Cost_1 < Cost_2$ **then**
        $group(t_i, MinNode)$ ▷ group t_i as a separate tree to $MinNode$
      else
        $weave(t_i, MinNode)$ ▷ weave t_i to $MinTree$
      end if
  end switch
end for
end (Return P)

Group to Lowest (G_{TL}) This optimizer is a balanced version of a *No Share* generator, which assigns each ACQ to run as a separate tree and that are then assigned to available nodes in a cost-balanced fashion.

Algorithm: The trees are first sorted by their costs, then, starting from the most expensive one, each tree is assigned to the node that currently has the lowest total cost.

Discussion: Since this optimizer does not perform any partial result sharing, it is only useful in cases when sharing is not beneficial (when none of the slides have any similarities in their periodic features).

Weave To Lowest (W_{TL}) This optimizer builds on the G_{TL} algorithm and weaves all *ACQs* on a node into a single, shared tree.

Algorithm: After sorting *ACQs* by cost (as in G_{TL}), W_{TL} assigns each *ACQ* to a node with the current lowest total cost and weaves it into the shared tree on the node.

Discussion: The W_{TL} optimizer executes *Weavability* calculation only once per input which makes it more expensive to run than G_{TL}. Additionally, by limiting to a single shared tree and not considering the compatibility of existing *ACQs* with new ones, it produces plans with high *Total Cost*, and, consequently, high *Max Cost*, even though it performs rudimentary cost balancing.

Weave-Group To Lowest (WG_{TL}) This approach also builds on G_{TL}, but as opposed to W_{TL}, it allows both *selective weaving* and grouping *ACQs* together.

Algorithm: Similar to G_{TL} and W_{TL}, WG_{TL} first sorts the *ACQ* trees, then iteratively assigns each *ACQ* to the node with the current smallest cost. At a node, an *ACQ* is either *woven* with the smallest costing tree in the node or added as a separate tree, whichever leads to the minimum cost increase.

Discussion: The WG_{TL} has similar runtime cost as W_{TL} as both optimizers use the *Weavability* calculations only once per *ACQ*. Even though WG_{TL} attempts to take advantage of grouping, it does not produce much better execution plans than W_{TL}. By focusing only on the lowest cost tree on a node, it weaves together some poorly compatible *ACQs*, leading to comparatively low quality execution plans.

6.5.2 Category "To Nodes"

Optimizers in this category follow the "To Nodes" algorithm depicted in Algorithm 5.

Weave to Nodes (W_{TN}) This optimizer is directly based on the single node *WeaveShare* algorithms, thus it is targeted at minimizing the *Total Cost*.

Algorithm: W_{TN} starts its execution the same way as the single node *WeaveShare*. If it

Algorithm 5 The *"To Nodes"* Algorithm

Input: A set of Q Aggregate Continuous Queries, N computation nodes, and *Category*
Output: Execution plan P
Create an execution tree $(t_1, t_2, \ldots, t_Q)$ for each query
$T \leftarrow Q$ ▷ T is the number of remaining trees
loop
  $MaxReduction \leftarrow -\infty$ ▷ maximum cost reduction is set to minimum
  for $i = 0$ *to* $T - 1$ **do** ▷ iterate over all trees
    for $j = 1$ *to* T **do** ▷ iterate over all trees again (to cover all pairs)
      $CostRed \leftarrow weave(t_i, t_j)$ ▷ cost reduction if weaving trees t_i and t_j
      if $CostRed > MaxReduction$ **then** ▷ find largest $CostRed$
        $MaxReduction \leftarrow CostRed$ ▷ and save it to $MaxReduction$
        $ToWeave \leftarrow (t_i, t_j)$ ▷ trees t_i and t_j are saved to be weaved later
      end if
    end for
  end for
  if $MaxReduction > 0$ **then** ▷ there is a benefit in weaving
    $weave(ToWeave)$ ▷ weave saved trees
  else
    switch *Category* **do**
      case $WeaveOnly$
        if $T \leq N$ **then**
          end (Return P)
        else
          $weave(ToWeave)$ ▷ weave saved trees
        end if
      case $WeaveAndGroup$
        $P \leftarrow G_{TL}(T)$ ▷ run *GroupToLowest* optimizer on remaining T trees
        end (Return P)
    end switch
  end if
  $T \leftarrow T - 1$
end loop

reaches the point where the current number of trees is less than or equal to the number of available nodes, W_{TN} stops and assigns each tree to a different node. If, however, *WeaveShare* finishes execution, and the current number of trees is still greater than the number of available nodes, the W_{TN} optimizer continues the *WeaveShare* algorithm (merging trees pairwise), even though it is no longer beneficial for total cost. The execution stops when the number of trees becomes equal to the number of available nodes.

Discussion: Since W_{TN} is a direct descendant of *WeaveShare*, it is optimized to produce the

minimum *Total Cost*. However, since W_{TN} allows only one execution tree per node, in order to match the number of nodes to number of trees, W_{TN} forces *WeaveShare* to keep merging trees with less compatible *ACQs*. Hence, W_{TN} generates, in general, more expensive plans than the basic *WeaveShare*. Additionally, W_{TN} does not perform any load balancing, hence it can generate query plans with execution trees whose computational requirements exceed the capacity of the node with the most powerful CPU.

Weave-Group to Nodes (WG_{TN}) Like W_{TN}, this optimizer is also directly based on the single node *WeaveShare* algorithm and is targeted at minimizing the *Total Cost*.

Algorithm: The WG_{TN} optimizer starts by executing single core *WeaveShare* and, similarly to W_{TN}, stops execution if it reaches the point where the current number of trees is equal to or less than the number of available nodes. However, if *WeaveShare* finishes execution and the current number of trees produced is greater than the number of available nodes, WG_{TN} assigns them to the available nodes, without *weaving* them, in a balanced fashion by applying the G_{TL} optimizer. First, all trees are sorted by their costs, and, starting from the most expensive ones, the trees are assigned to the nodes with the smallest current total cost.

Discussion: Unlike W_{TN}, the WG_{TN} optimizer is designed to produce the minimum *Total Cost* and the minimum *Max Cost*. The latter is not always possible, since the execution trees produced by *WeaveShare* are sometimes of significantly different costs, and the used load balancing technique cannot produce the desired output. WG_{TN} can achieve a better *Total Cost* than W_{TN} by not forcing trees that do not *weave* well together to merge, which would have increased the total cost of the plan. However, the penalty of grouping execution trees on nodes without merging them is that each tuple has to be processed as many times as the number of trees on a node. This effectively increases the *Total Cost* by a factor equal to the input rate multiplied by the number of the trees on each node. Clearly, the higher the input rate of a stream, the more costly it will be for the system to group trees without *weaving* them.

6.5.3 Category "Inserted"

Optimizers in this category follow the "Inserted" algorithm depicted in Algorithm 6.

Algorithm 6 The *"Inserted"* Algorithm

Input: A set of Q Aggregate Continuous Queries, N computation nodes, and *Category*
Output: Execution plan P
Assigning first N queries to N nodes $(n_1, n_2, \ldots, n_N)$ as separate trees
Calculate node costs for all N nodes
$Q \leftarrow Q - N$ ▷ Q is the number of remaining queries to be assigned
$WeaveCost \leftarrow \infty$ ▷ weave cost is set to maximum
for $i = 0$ *to* Q **do** ▷ iterate over the queries until all are grouped/weaved
  $MinNode \leftarrow findMinNode()$ ▷ determine the node with the current smallest cost
  for $j = 0$ *to* N **do** ▷ iterate over all nodes
    for $k = 0$ *to* NumTrees in n_j **do** ▷ iterate over all trees within a node
      $TempCost \leftarrow weave(q_i, t_k)$ ▷ determine plan cost if weaving query q_i into tree t_k
      if $TempCost < WeaveCost$ **then** ▷ find smallest $TempCost$
        $WeaveCost \leftarrow TempCost$ ▷ and save it to $WeaveCost$
        $ToWeave \leftarrow (q_i, t_k)$ ▷ query q_i is saved to be weaved to tree t_j later
      end if
      switch *Category* **do**
        case $WeaveOnly$
          $weave(ToWeave)$ ▷ weave saved trees
        case $WeaveAndGroup$
          $GroupCost \leftarrow group(q_i, MinNode)$ ▷ cost of $MinNode$ if q_i is grouped
          if $GroupCost < WeaveCost$ **then**
            $group(q_i, MinNode)$ ▷ group q_i to $MinNode$ as a separate tree
          else
            $weave(ToWeave)$ ▷ weave saved trees
          end if
      end switch
    end for
  end for
end for
end (Return P)

Weave Inserted (W_I) This approach is based on the *Insert-then-Weave* optimizer introduced in [20], in which every ACQ is either weaved in an existing tree or assigned to a new tree, whichever results in the smallest increase in the *Total Cost*. The difference of the W_I optimizer from the original *Insert-then-Weave* approach is that W_I keeps a fixed number of trees equal to the number of nodes in the distributed system.

Algorithm: W_I starts by randomly assigning an ACQ to each available node, then iterating through the remaining $ACQs$. For each node it computes the new cost if the ACQ under

consideration is woven into the execution tree on the node and assigns the ACQ to the node that has the smallest new cost.

Discussion: W_I is attempting to optimize for the *Max Cost*, as well as the *Total Cost*, by taking into account both the *Weavability* of the inserted ACQ with every available node and performing cost-balancing of the computation nodes. The downside of W_I is that, since load balancing is the first priority of W_I, it sometimes assigns $ACQs$ to nodes with underlying trees with which they do not *weave* well. This happens in cases where the tree that *weaves* poorly with the incoming ACQ currently has the smallest cost. Additionally, since W_I is limited to one execution tree per node, the $ACQs$ that do not *weave* well with any of the available trees are still merged into one of these trees. This increases the *Total Costs* of the generated plans.

Weave-Group Inserted (WG_I) This optimizer is also a version of the *Insert-then-Weave* approach and similar to W_I. However, since the WG_I optimizer does not have to be limited to only one execution tree per node, it utilizes grouping to keep the *Total Cost* low while maintaining load balance between nodes.

Algorithm: WG_I starts by randomly assigning an ACQ to each available node, then iterating through the remaining $ACQs$ similarly to W_I. By trying to weave each ACQ under consideration into every execution tree in every node, WG_I determines each node's minimum new cost and the most compatible underlying tree. Finally, the ACQ is either woven to the selected tree on the node with the minimum new cost or added as a separate tree to the tree with the minimum old cost, based on which option leads to the minimum *Total Cost* increase.

Discussion: WG_I is optimized for both *Max Cost* and *Total Cost*. However, even though WG_I allows grouping of execution trees, it does not always achieve a good *Total Cost*. This happens (similarly to W_I) in cases when the tree that *weaves* poorly with the ACQ under consideration has the smallest cost and is located in the node with the smallest current node cost, which forces WG_I to *weave* the non-compatible $ACQs$.

Note A preprocessing step can be carried out for all optimizers by merging all $ACQs$ with identical slides into the same trees, since such $ACQs$ *weave* together perfectly. This reduces the workload down to a number of execution trees with multiple $ACQs$ with the same slides.

Note that this preprocessing is always beneficial in terms of the *Total Cost*, however, it is only beneficial in terms of the *Max Cost* if the distributed system has low number of nodes compared to the number of input *ACQs*. Otherwise, since the number of entities in the workload is decreased, it is more challenging to achieve balance among the high number of computating nodes.

6.6 Experimental Evaluation

In this section, we summarize the results of our experimental evaluation of all the optimizers for distributed processing environments listed in Table 7.

6.6.1 Experimental Setup

In order to evaluate the quality of our proposed optimizers, we used our testbed described in Section 5.4.1. We implemented all of the optimizers discussed above as part of this system. Our *workload* and *experimental parameters* utilized in our evaluation are the same as in Section 5.4.1, however we add a new parameter N_{num}, which represents the number of nodes in the target system.

We **measured the quality of plans** in terms of the cost of the plans 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. Thus, our comparison does not include the actual execution of the plans on a distributed environment, which we address later in Chapter 7. All results are taken as averages of running each test three times.

6.6.2 Experimental Results

6.6.2.1 Exp 1: Evaluation of Distributed Environment Optimizers

Configuration (Table 8) To compare the quality of produced plans by the distributed optimizers, we tried to cover as broad a range of different parameters as possible. Thus, we ran a set of 256 experiments, which correspond to all possible combinations of the parameters

Table 8: Experimental Parameter Values (Total number of combinations = 256)

Parameter	Q_{num}	N_{num}	λ	S_{max}	Z_{skew}	O_{max}	Gen
Values	250, 500	4, 8, 16, 32	10, 100	25, 50	0, 1	10, 100	Nrm, Div
# options	2	4	2	2	2	2	2

from Table 8 (i.e., our entire search space). For each one of these experiments, we generated a new workload according to the current parameters and executed all of the above mentioned optimizers on this workload.

Results (Figure 40 and Tables 9 and 10). Out of a very large number of results, we observed that the *Weave to Nodes* (W_{TN}) and *Weave-Group to Nodes* (WG_{TN}) produced good plans in terms of *Total Cost*, while *Weave Inserted* (W_I) and *Weave-Group Inserted* (WG_I) performed the best in terms of *Max Cost* (Figure 40). However, we noticed that in the majority of the cases where the W_{TN} and W_I optimizers produced the best plans (in terms of *Total Cost* and *Max Cost*, respectively), their matching optimizers from the *Weave and Group* category (WG_{TN} and WG_I) produced output of either equal or very similar

Table 9: WG_I vs WG_{TN} breakdown (for 256 experiments)

Max Cost	**Weave-Group Inserted** (WG_I)	**Weave-Group to Nodes** (WG_{TN})
Wins	Best in **80%** of cases	Best in **17%** of cases
Loses	Not best in 20% of cases, and within **3%** from the best on average	Not best in 83% of cases, and within **48%** from the best on average

Total Cost	**Weave-Group Inserted** (WG_I)	**Weave-Group to Nodes** (WG_{TN})
Wins	Best in **5%** of cases	Best in **90%** of cases
Loses	Not best in 95% of cases, and within **9%** from the best on average	Not best in 10% of cases, and within **0.2%** from the best on average

(a) **Max Cost** Comparison

(b) **Total Cost** Comparison

Figure 40: **Average Plan Quality** (from 256 experiments) where **0%** and **100%** are the average plan costs of all **best** and **worst** plans, respectively, across all optimizers. The error bars show the standard deviations Consistent with the definition of a standard deviation, about 68% of all plans produced by these optimizers lie in this margin.

Table 10: Average Plan Generation Runtime (for 256 experiments)

Optimizer	G_{Rand}	W_{Rand}	WG_{Rand}	G_{RR}	W_{RR}	WG_{RR}	G_{TL}	W_{TL}	WG_{TL}	W_{TN}	WG_{TN}	W_I	WG_I
Time (sec)	0.01	2.31	0.02	0.01	2.34	0.01	0.01	12.6	9.11	2.95	2.83	5.68	3.94

quality. In some other cases where W_{TN} and W_I performed poorly, the optimizer *Group to Lowest* (G_{TL}) performed better. In such cases, our optimizers WG_{TN} and WG_I were still able to match the best plans produced by G_{TL} with equal or better quality plans in most of the cases. Thus, we concluded that the WG_{TN} and WG_I optimizers were able to successfully adapt to different environments and produce the best plans in terms of *Total Cost* and *Max Cost*, respectively. It is intuitive that both winning plans are from the *Weave and Group* category, which allows them to benefit from both weaving and grouping capabilities. WG_{TN} is best in terms of *Total Cost* because its strategy is to continuously find pairs of trees that decrease the overall plan cost the most when combined. In contrast, WG_I is best in terms of *Max Cost* because its strategy is to continuously perform tree insertions that lead to smallest node cost increases.

To compare and contrast the two winning optimizers, we provide the breakdown of their performances in Table 9. From this table, we see that in terms of *Max Cost*, WG_{TN} significantly falls behind WG_I, since balancing is not the first priority of WG_{TN}. In terms of *Total Cost*, WG_{TN} always either wins or is within 0.2%, and WG_I falls behind, but not as significantly, since it is on average within 9% of the winning optimizer.

Additionally, we have recorded the runtimes of our optimizers (Table 10), and we see that plan generation time on average does not exceed 13 sec per plan for all optimizers, which is fast considering that after an execution plan is generated and deployed to the *DSMS*, it is expected to run for a significantly longer time.

Conclusions WG_{TN} and WG_I produce the best execution plans in terms of *Total Cost* and *Max Cost*, respectively. WG_{TN} falls behind WG_I in terms of *Max Cost* more significantly than WG_I falls behind WG_{TN} in terms of *Total Cost*. All optimizers generate plans fast ($<$ 13 sec).

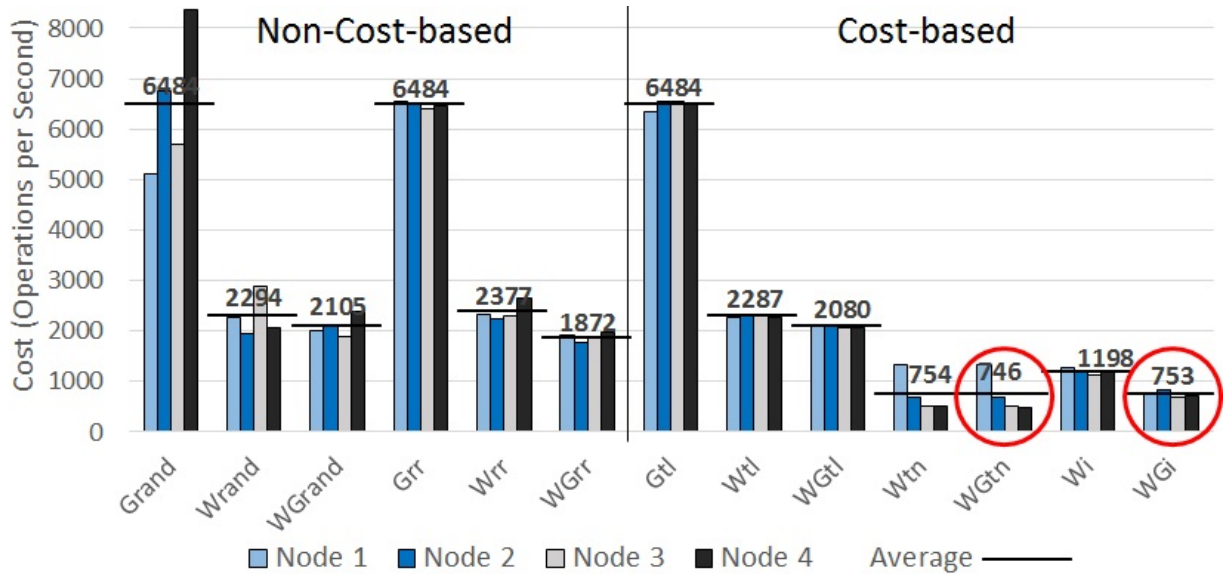

Figure 41: Costs per node in a 4-node system

6.6.2.2 Exp 2: Load Balancing

Configuration To show how all proposed algorithms compare in terms of balancing load and minimizing the total plan cost, we fix a few parameters ($Q_{num} = 250$, $N_{num} = 4$, $\lambda = 100$, $S_{max} = 25$, $Z_{skew} = 1$, $_{max} = 100$, $Gen = Nrm$) and run this experiment while recording the individual node costs of produced plans for all optimizers.

Results (Figure 41). The results depict the typical behavior of the proposed algorithms in a 4-node environment. Since algorithms W_{TN} and WG_{TN} are optimized mostly for *Total Cost*, they produce plans with very imbalanced node loads. However, their *Total Costs* (as well as their Average Costs) are low. On the other hand, W_I and WG_I produce plans that are well balanced, and, at the same time, WG_I produces plans that also have a low *Total Cost* (practically as low as WG_{TN}).

Conclusions Algorithms that are producing execution plans with the lowest *Total Cost* typically perform poorly in terms of balancing load among the different nodes.

6.7 Summary

In this chapter, we explored how the sharing of partial aggregations can be done in the environment of distributed *DSMSs*. We formulated the problem as a distributed multi-query optimization which combines the sharing of partial aggregations and assignment to servers to produce high quality plans that keep the total cost of the execution low and balance the load among the computing nodes. We presented a classification of optimizers based on whether or not they are cost-based and how they utilize the concept of *Weavability*. We implemented and experimentally compared all of our proposed optimizers.

Our evaluation showed that the *Weave-Group Inserted* (WG_I) optimizer delivers the best quality in terms of load balancing among the nodes in the system, which makes it the most beneficial for Cloud service providers, since balancing helps conserve energy and prevents the need to over-provision systems hardware. At the same time, our evaluation showed that the *Weave-Group to Nodes* (WG_{TN}) optimizer best minimizes the total plan cost, which makes WG_{TN} the most beneficial for clients, since the monetary cost of ACQ computation in multi-tenant environments becomes lower.

A closer look at the performance profiles of the two winning optimizers suggests that it might be more advantageous to choose the WG_I optimizer in the case where both service providers and clients should be satisfied "equally" – WG_I falls behind in terms of *Total Cost* less significantly (only 9% on average) than WG_{TN} does in terms of *Max Cost* (load balancing).

7.0 Multi-Query Optimization of Incrementally Evaluated Sliding-Window Aggregations

In this chapter, we re-examine how the principle of sharing is applied in *Incremental Evaluation* (*IE*) techniques as well as in *Multi-Query* (*MQ*) optimizers. We provide a theoretical analysis of all of the available *IE* techniques that accurately determines their average operational cost per slide (Ω) given any set of input *ACQs*. We also propose a new *MQ* optimization solution that achieves significant improvement in execution costs by combining the new *IE* techniques with the state-of-the-art *MQ* optimizers using the analysis above.

In the next section, we discuss how the new *IE* techniques can be used in *MQ* optimizers. We present our theoretical analysis and our solution at combining *IE* techniques and *MQ* optimization in Section 7.2. We discuss our experimental findings in Section 7.3, and present our conclusions in Section 7.4.

7.1 Introduction

With the introduction of the *SlickDeque* technique, the final aggregation for a single query can now be performed in constant time with no more than 2 operations per slide. Thus, we believe that sharing at the level of partial and final aggregation has reached its limit. In this chapter we focus on *MQ* optimization because it is the next logical step for further improving *SWAG* and still has multiple unaddressed challenges.

Currently, the state-of-the-art *MQ* optimizers only work with the outdated *Panes* [33] and *Pairs* [31] techniques for *IE*. Thus, the opportunity arises to explore the suitability of new and more efficient *IE* techniques for use in combination with the *MQ* optimizers. To this end, we propose a novel solution of using the new *IE* techniques as part of state-of-the-art *Multi-Query* (*MQ*) optimizers in a way that significantly reduces the execution plan costs.

It is intuitive that by combining new *IE* techniques and the *MQ* optimizers, significant benefits in *ACQ* processing can be achieved. To accomplish this, the plan cost calculation

process needs to be adjusted. The need for such adjustment can be extrapolated from the *Weavability* cost calculation formula (also discussed in Section 2.4.2):

$$C = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{7.1}$$

where m is the number of the trees in the plan, λ is input rate in tuples per second, E_i is *Edge rate* of tree i (the number of partial aggregations performed per second), and Ω_i is the total number of final-aggregation operations performed per edge of tree i.

Currently optimizers *WeaveShare* and *TriWeave* always calculate Ω_i as follows:

$$\Omega_i = \sum_{j=1}^{n} \frac{r_j}{s_j} \tag{7.2}$$

As demonstrated in Section 2.3.2, such a number of final-aggregation operations is only applicable for the outdated *Panes* technique, and all other compared *IE* techniques perform fewer operations.

For the new *IE* techniques the Ω estimation is more complex due to the variability of operation numbers between different slides, and dependability on the input data, given that our *MQ* optimizers need to be able to estimate Ω for any number of *ACQs* with any periodical properties. Thus, a theoretical analysis (presented below) of all the *IE* techniques is necessary.

7.2 Estimating Ω

In order to evaluate how different *Incremental Evaluation* (*IE*) techniques perform when used in *Multi-Query* (*MQ*) optimizers, we need to calculate the number of final aggregation operations (Ω) that they perform on average per slide (i.e. after receiving each new partial) given nQ unique *ACQs*. After that Ω is used in their corresponding cost formulas in *MQ* optimizers. The range and slide of each query q_i we denote as r_i and slide s_i respectively (i is a sequential number of a query). *IE* techniques must support *MQ* processing in order to be used in such optimizers, which rules out the *TwoStacks* and *DABA* techniques presented

in Section 2.3.2. Our analysis of Ω for the rest of the techniques (*Panes*, *FlatFAT*, *FlatFIT*, and *SlickDeque*) follows below.

Panes. It is intuitive that Ω for this naive technique in single query environments can be calculated as range divided by slide (r/s) since the query range is assembled from r/s slides. Similarly, in MQ environments, since each added query increases Ω by its range divided by slide, we calculate it as follows:

$$\Omega = \sum_{i=1}^{nQ} \frac{r_i}{s_i} \tag{7.3}$$

FlatFAT. Given that this technique utilizes a binary tree for its calculations, in single query scenarios $\Omega = nQ \cdot log_2(nP_{max})$, where nP_{max} is the total number of partials (or leaf nodes) in the tree. nP_{max} is also the longest query range that can be processed by this structure. The Ω formula follows from the fact that the number of levels in a binary tree are $log_2(nP_{max})+1$, and on each update *FlatFAT* updates the tree in a bottom-up fashion from the leaf to the root. The answer to the query with the longest range in this case could be simply taken from the root of the tree without additional operations. For each additional query with a unique range that consists of $nP_i < nP_{max}$ partials, the aggregate is composed from a minimum set of internal tree nodes that covers the number of partials (tree leaves) nP_i. Thus, given that nP_i can be calculated as r_i divided by the average partial length, each new query q_i increases Ω by $\lfloor log_2(nP_i - 1) \rfloor + \frac{nP_i - 1}{2^{\lceil log_2(nP_i) \rceil}}$, resulting in the formula:

$$\Omega = \sum_{i=1}^{nQ} (\lfloor log_2(nP_i - 1) \rfloor + \frac{nP_i - 1}{2^{\lceil log_2(nP_i) \rceil}}) \tag{7.4}$$

Even though the actual number of partials processed is likely to be different for each slide, in the long run the cost per partial averages out, making this estimation valid. For quick approximate calculations Ω can be also estimated as $log_2(nP_i - 1) + 1$ for each unique query.

FlatFIT. For this technique, we estimate Ω as $3 \cdot nQ$, i.e., each unique ACQ requires about 3 operations per slide. We show in Section 3.3.1 that Ω is 3 operations per slide for a single query environment and $3 \cdot nQ$ per slide in a *max-multi-query* environment (a MQ environment with the maximum number of queries covering all possible ranges from 1 to r_{max}). Thus, intuitively each added query should be adding 3 operations per slide to Ω. We confirmed

this formula experimentally by testing a large number of various query sets. Even though there was slight variability in the results (due to the effect of periodic properties of $ACQs$), Ω always stayed close to $3 \cdot nQ$ and never crossed $2 \cdot nQ$ or $4 \cdot nQ$. Given our intuition above, our closest estimation for *FlatFIT* appears to be:

$$\Omega = 3 \cdot nQ \tag{7.5}$$

which we use in our experiments with cost-based MQ optimizers.

SlickDeque (Inv). In single query environments this technique has $\Omega = 2$ because there are only two operations performed for each new partial: (1) the aggregation of the arriving partial with the running aggregate, and (2) the inversion operation of the expiring value (e.g., subtraction in the case of Sum). Similarly, in the case with multiple queries we get Ω by multiplying the number of running aggregates by 2. Unfortunately, the number of running aggregates does not always equal nQ due to the different periodic properties of $ACQs$, an ACQ might assemble its final aggregate from different numbers of partials on different stages of execution, which means we need to keep running aggregates for all of these possibilities. However, if several queries need the same running aggregate (aggregating same number of partials) it is shared. Thus, in order to calculate the exact number of running aggregates required per query set we need to create a composite slide and iterate over it while counting all possible numbers of partials needed at every edge, and to get Ω we finally multiply the number of unique running aggregates by 2. Currently we do not know if there is a faster approach to determine this. Due to the complexity of this calculation in our experiments we use an approximation: first we divide each query range r_i by the average partial aggregate length to get nP_i, and then take the count of all the unique nP_i values and multiply by 2, resulting in the following formula:

$$\Omega = \#\text{ of unique } nP_i \tag{7.6}$$

Given that in our experiments we generally have nQ that is larger than the number of unique slides (which are generated by factoring a large number), the variance of nP_i values

is low, which makes our estimation valid (we also verified this experimentally), however this estimation might slightly vary in the other case.

SlickDeque (Non-Inv). Previously [44] we proved that Ω is bounded by 2 operations per slide (in single query environments), however in this work we worked out a more accurate estimation and accounted for MQ overhead in order to use this technique in MQ optimizers. *SlickDeque (Non-Inv)* performs exactly 2 operations per slide if we do not account for the following two cases: (1) expiration of partials at the head of the deque, and (2) deletion of the head node of the deque. When either of the two cases occur, 1 operation is performed for that slide instead of 2.

Case (1) happens when the partial stayed on the deque for the entire max query range worth of partials (nP_{max}), which means that there was no input partials that could displace the expiring partial from the deque (e.g., if Max is calculated, there was not any input partial greater or equal to our expiring partial). The probability of that happening (given uniform input) is 1 to nP_{max}, where nP_{max} is the number of partials in the query with the longest range. Thus we subtract $1/nP_{max}$ from Ω to account for the average number of times this happens in a long running process.

Case (2) happens when any input partial displaces the head node of the deque (e.g., if Max is calculated, a partial higher than all the nodes including the head node arrives). The probability of that happening is again $1/nP_{max}$ per slide since that is the probability of the new partial displacing the most valuable partial from the latest nP_{max} (e.g., the highest value if Max is calculated), thus we subtract another $1/nP$ from Ω.

Now, in order to account for MQ cases we have to account for operations required by the algorithm to return query answers. In a single query environment this could be simply done by returning the value of the head node on the deque, however if we need to return answers to several queries with different ranges we traverse the deque. During the planning stage all queries are ordered descendingly by their ranges, which makes it possible during execution to get answers to all queries in just one full traversal of the deque. Each query requires at least one operation to compare its required nP_i to the current iterator position within the deque. To account for that we add nQ to our cost estimate Ω. Also, to account for the operations that need to be performed to traverse the deque (in the worst case) we add

Table 11: Estimated Final Aggregation Costs

IE technique		Operations Per Edge (Ω)
Panes		$\sum_{i=1}^{nQ} \frac{r_i}{s_i}$
FlatFAT		$\sum_{i=1}^{nQ}(\lfloor log_2(nP_i - 1)\rfloor + \frac{nP_i - 1}{2^{\lceil log_2(nP_i)\rceil}})$
FlatFIT		$3 \cdot nQ$
Slick Deque	Inv.	$\#$ *of unique* nP_i
	Non-I.	$2 - 2/nP_{max} + nQ + \sum_{i=1}^{nP_{max}} \frac{1}{i!}$

the number of operations equal to the average length of the deque during execution. Given the uniform input, as shown in [44], the length of the deque on average equals the sum of the inversed factorials of sequential natural numbers from 1 to nP_{max}, where nP_{max} is again the maximum number of partials needed by any query to assemble its answer, and can be expressed as $\sum_{i=1}^{nP} \frac{1}{i!}$. This follows from the fact that the probability of randomly picking x numbers ordered in a particular way (e.g., ascending) is 1 to $x!$. Thus, we estimate Ω for *SlickDeque (Non-Inv)* as:

$$\Omega = 2 - 2/nP_{max} + nQ + \sum_{i=1}^{nP_{max}} \frac{1}{i!} \tag{7.7}$$

We summarize our theoretical findings in Table 11.

7.3 Experimental Evaluation

In this section, we present the results of our experimental evaluation of using the new *IE* techniques in *MQ* optimizers by (1) generating execution plans for the *IE* techniques and comparing their estimated costs, and (2) actually executing several generated plans and comparing the practical performance.

7.3.1 Plan Generation Setup

In this part of our evaluation we show the significance of *IE* technique selection on generated plan costs using our *plan generation* testbed described in Section 5.4.1. Towards this we utilized our Java platform where we implemented the *WeaveShare* and *TriWeave MQ* optimizers as described in [20] and [19], and augmented them with the support of different Ω calculations (estimation of final aggregations) for our compared *IE* techniques.

Our plan generation **experimental parameters** are:

[***IE technique***] It specifies which algorithm is used for Ω calculations. The available techniques are: *Panes*, *FlatFAT*, *FlatFIT*, *SlickDeque* (Inv and Non-Inv).

[$\boldsymbol{Q_{num}}$] Number of *ACQs*. We assume that all *ACQs* are installed on the same data stream and their aggregate functions allow them to share partial aggregations among them. The actual function does not have any effect on performance other than the ability to share partial aggregations.

[$\boldsymbol{S_{max}}$] Maximum slide length provides an upper bound on how large slides of our *ACQs* can be. The minimum slide allowed by the system is one. The slides are drawn from the set of factors of S_{max}.

[$\boldsymbol{\lambda}$] The input rate describes how fast tuples arrive through the input stream in our system.

[$\boldsymbol{Z_{skew}}$] Zipf distribution skew depicts the popularity of each slide length in the final set of *ACQs*. A Zipf skew of zero produces uniform distribution, and a greater Zipf skew is skewed towards large slides.

[$\boldsymbol{O_{max}}$] Maximum overlap factor defines the upper bound for the overlap factor. The range of each *ACQ* is determined by drawing an ovelap factor from a uniform distribution between one and O_{max} and multiplying it by *ACQ*'s slide.

7.3.2 Plan Generation Results

To compare the sensitivity of the estimated plan costs produced by our new *IE* techniques to the parameters Q_{num}, S_{max}, O_{max}, λ, and Z_{skew}, we ran five experiments where we varied one of these parameters at a time while keeping the rest of them fixed. The parameters were selected separately for each experiment in a way that would highlight the differences in the

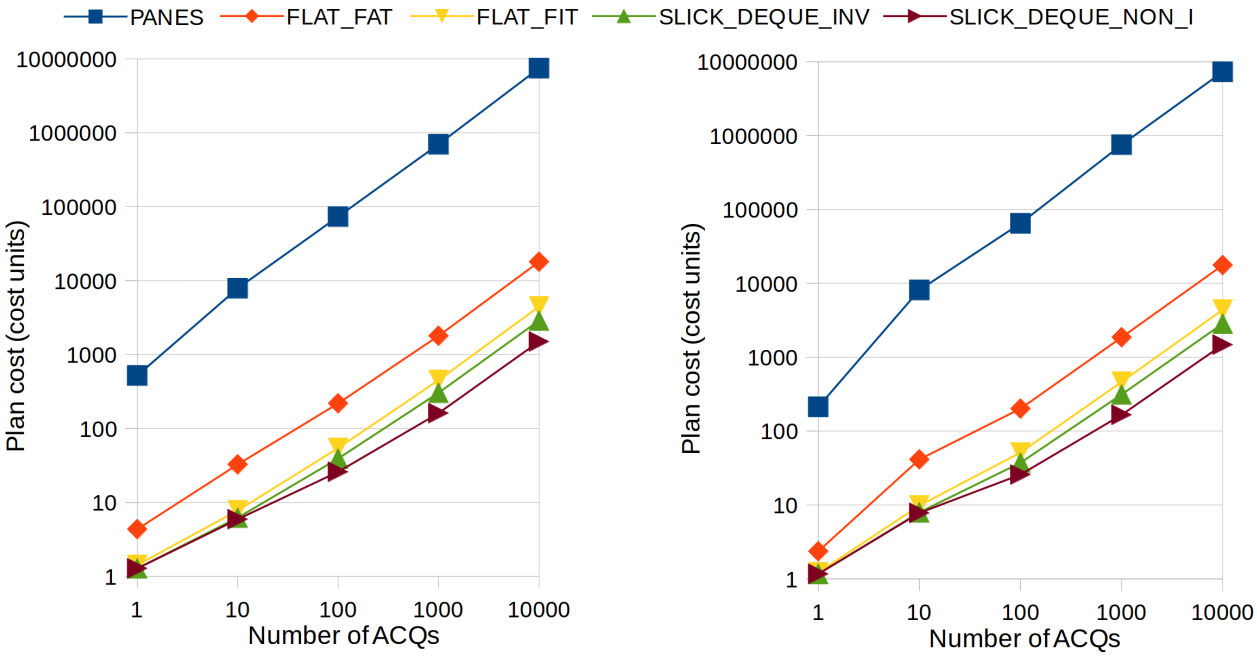

Figure 42: Plan cost with increasing number of queries using WeaveShare (left) and Tri-Weave(right)

Table 12: Experiment Parameters

#	Q_{num}	S_{max}	O_{max}	λ	Z_{skew}
1	*1-10K*	1K	10K	1	0
2	100	*10-100K*	10K	1	0
3	100	1K	*100-1M*	1	0
4	100	1K	10K	*0.01-100*	0
5	100	1K	10K	1	*(-1)-1*

scalabilities of the five compared *IE* techniques. The experimental parameters are specified in the Table 12. All results are taken as averages of running each experiment ten times.

7.3.2.1 Exp 1: Number of ACQs Sensitivity (Figure 42)

In this test we varied the Q_{num} from 1 to 10,000. Clearly increasing the Q_{num} also in-

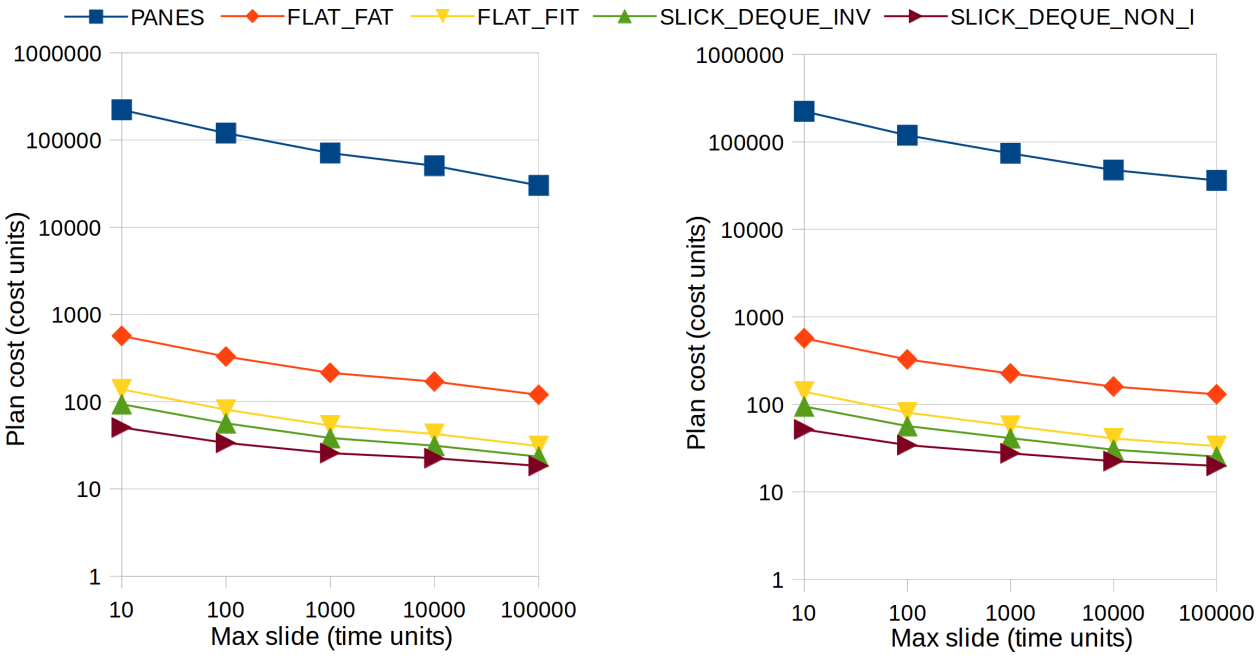

Figure 43: Plan cost with increasing max slide using WeaveShare (left) and TriWeave(right)

creases the amount of required calculations, causing higher costs for all of the generated plans (for both *WeaveShare* and *TriWeave* optimizers). The growth rates (depicted in Figure 42) of all underlying *IE* techniques are similar to what we expected from the theoretical analysis of the time complexities of their underlying algorithms. Thus we see that using *SlickDeque* (Non-Inv) and *SlickDeque* (Inv) show the best results by outperforming the closest competing *IE* technique (FlatFIT) by up to 3x and the current state-of-the-art *Panes* technique by up to 5,000x. Additionally notice that *TriWeave* outperformed *WeaveShare* algorithm by 8% on average.

7.3.2.2 Exp 2: Max Slide Sensitivity (Figure 43)

In this test we varied the S_{max} from 10 to 100,000. As opposed to to Exp 1, increasing the S_{max} decreases the amount of required calculations. This happens because with a higher max slide parameter, the generated *ACQs* have longer slides, which results in longer distances between the edges (where the final aggregations are performed). This way the workload for

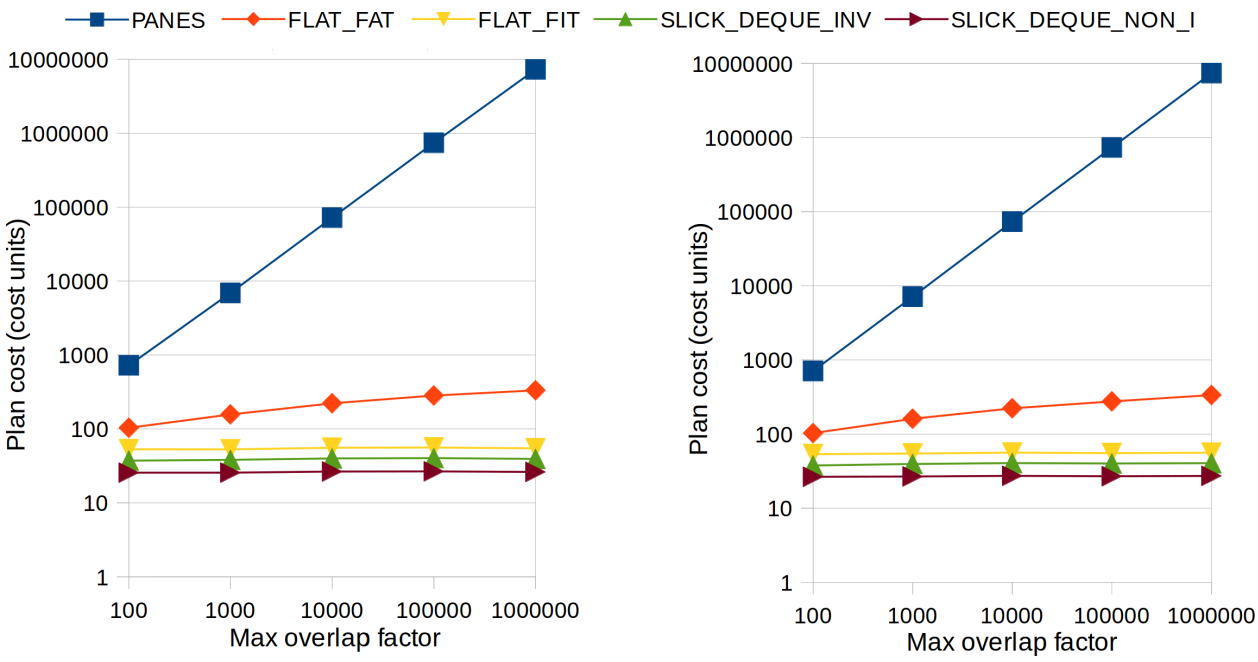

Figure 44: Plan cost with increasing max overlap using WeaveShare (left) and Tri-Weave(right)

the final aggregator is reduced while keeping the same workload for the partial aggregator, resulting in a decreased total cost. Notice again that *SlickDeque* shows vastly superior performance by surpassing all other algorithms by up to 4,300x. *WeaveShare* and *TriWeave* optimizers performed similarly in this experiment (within 2%).

7.3.2.3 Exp 3: Overlap Factor Sensitivity (Figure 44)

In this test we varied the O_{max} from 100 to 1,000,000. Similarly to Exp 1, increasing the O_{max} also increases the amount of required calculations (in most cases). This follows from the fact that increasing O_{max} increases query ranges, and increased ranges require more partials to be assembled during each final aggregation. However, algorithms *FlatFIT* and *SlickDeque* (both Inv and Non-Inv) have constant complexity in terms of increasing window, thus their performance remains largely unaffected by the increasing ranges (which can be observed in Figure 44). As a result, we can see that the difference between the

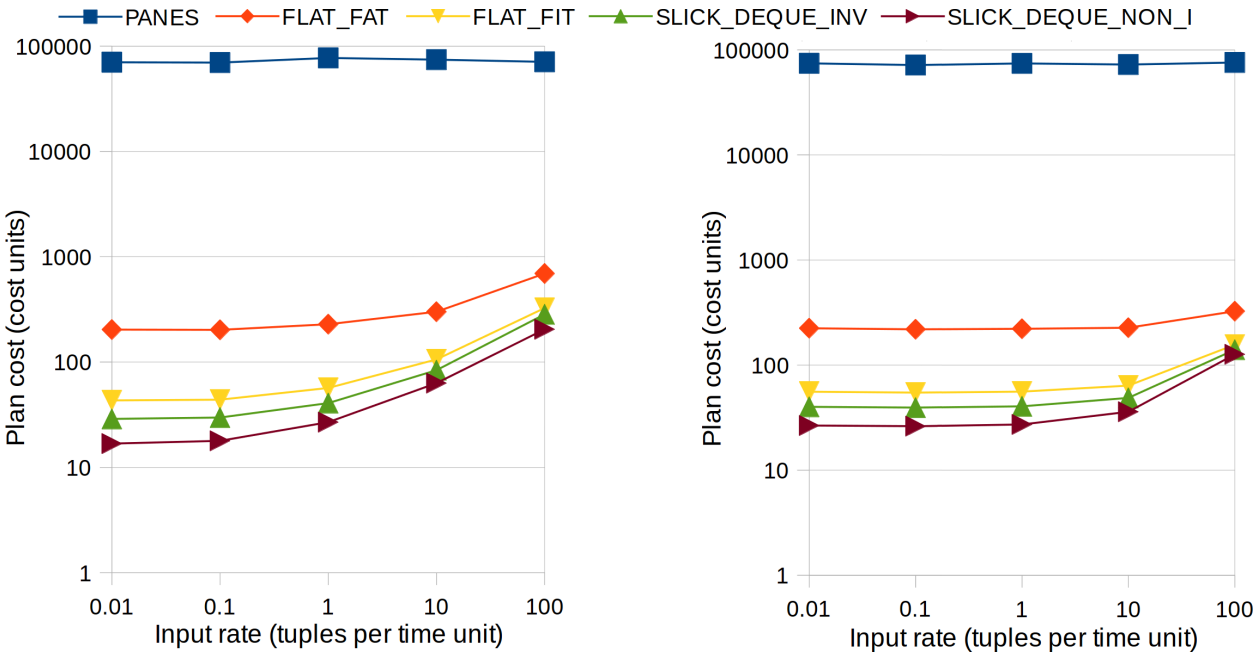

Figure 45: Plan cost with increasing input rate using WeaveShare (left) and TriWeave(right)

best performing *SlickDeque* technique and the currently used Panes technique grows much faster than in the first two experiments, and reaches 270,000x improvement. *WeaveShare* and *TriWeave* optimizers again performed similarly in this experiment (within 2%).

7.3.2.4 Exp 4: Input Rate Sensitivity (Figure 45)

In this test we varied λ from 0.01 to 100. Increasing λ increases the amount of required calculations because with higher input rates partial aggregators have to do more work aggregating the input tuples (which can be seen in Equation 7.1). Notice that the performance of the *Panes* algorithm is not significantly affected by the increasing input rate. This happens because the cost of the *Panes* algorithm is largely dominated by the final aggregator cost, and the increase in partial aggregation cost is proportionally small. *SlickDeque* again outperforms other algorithms by up to 3,000x, and *TriWeave* outperforms the *WeaveShare* optimizer by 18% on average.

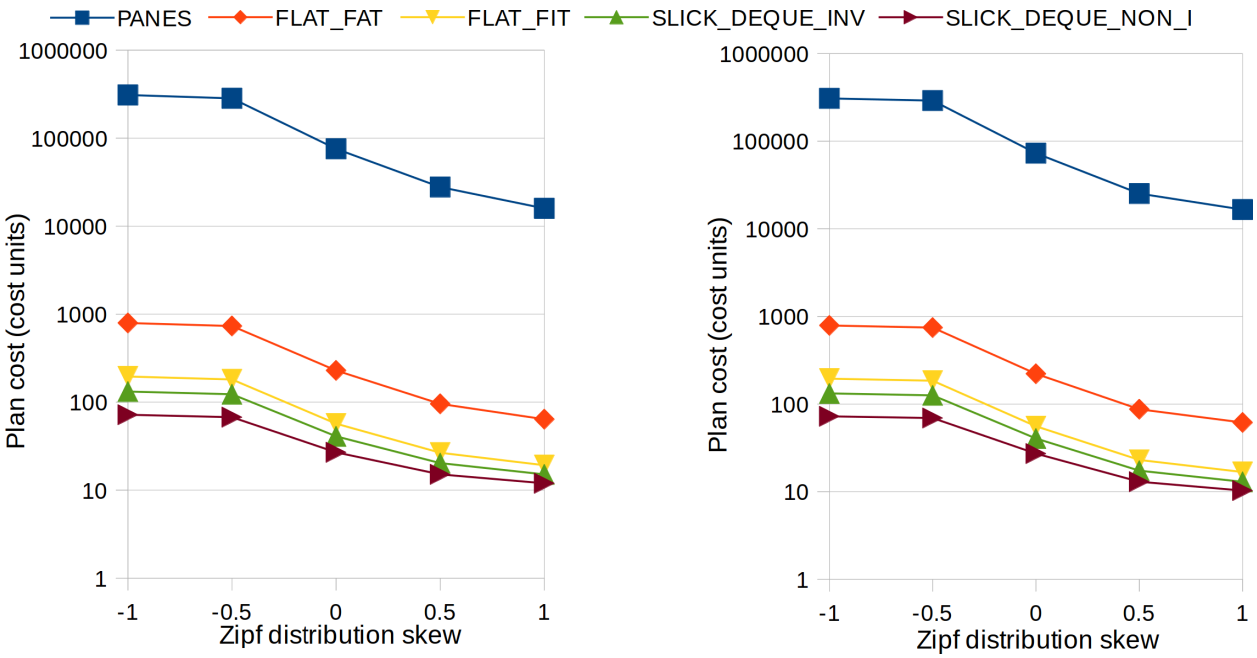

Figure 46: Plan cost with increasing zipf using WeaveShare (left) and TriWeave(right)

7.3.2.5 Exp 5: Slide Skew Sensitivity (Figure 46)

In this test we varied the Z_{skew} from -1 to 1. This experiment is similar to the max slide scalability experiment, because in both experiments we are gradually increasing the amount of *ACQs* with large slides and therefore decreasing the amount of required calculations. The difference here is that when significantly skewing all slides drawn from the same set to one side (when Z_{skew} is close to -1 and 1), they start repeating, which lessens the affect on the costs (we can see flatter lines on the figures in these places). *SlickDeque* outperforms all the other *IE* techniques by up to 4,200x, and *TriWeave* outperforms *WeaveShare* by 5% on average.

7.3.2.6 Plan Generation Summary.

The above experimental results showed that our *SlickDeque* technique delivers the best quality execution plans when used as part of *MQ* optimizers. This was to be expected given that *SlickDeque* is the most advantageous *IE* technique in terms of throughput as shown in

Table 13: Practical Evaluation Parameters

Q_{num}	S_{max}	O_{max}	λ	Z_{skew}
100	1K	10K	1	0

Chapter 4. We also observed that the *TriWeave* optimizer produces slightly better execution plans compared to *WeaveShare*.

7.3.3 Practical Evaluation Setup

In order to verify the correctness (and practical significance) of the plan cost estimations produced by our updated *MQ* optimizers in the first part of our evaluation, we executed a few selected plans on a real dataset using our own *execution* platform written in C++ and examined their performance. The detailed testbed description can be found in Section 3.4.1.

Setup. We used our *plan generation* platform to generate one plan using *WeaveShare* and one plan using the *TriWeave* optimizers for each of the compared *IE* techniques using the query load parameters (specified in Table 13) that correspond to the middle point of each figure from our plan generation experiments (Exp. 1-5).

Evaluation Metrics. Generally, a cost of a plan is estimated as the required computation power to process this plan. It is clear that there is a reverse relationship between the plan cost (measured in operations per second) and the actual throughput (measured as the number of actual query answers received per second). Thus, to perform a fair comparison we converted the plan cost to estimated throughput by inverting it (i.e. $1/Cost$).

7.3.4 Practical Evaluation Results

To measure the actual throughput, we ran each execution tree of a plan for 30 minutes at full speed while counting query results returned by the system. We added them together to get the total number for the plan, and divided them by 1,800 to get the number per

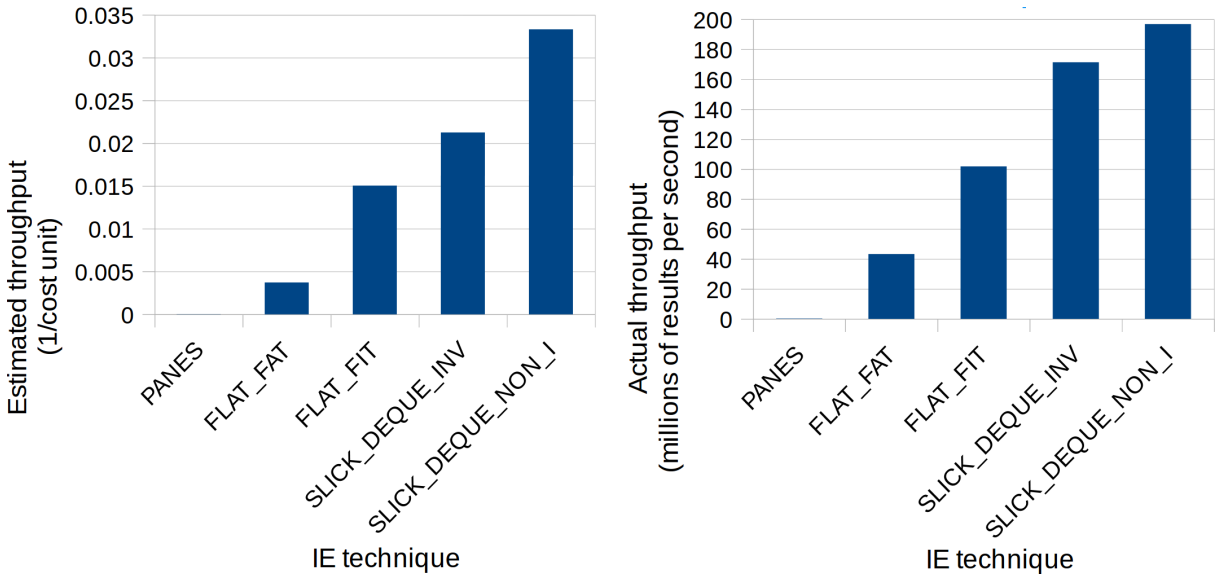

Figure 47: WeaveShare: estimated throughput in 1/cost_unit per second (left), actual throughput in results per second (right)

second. Even though the estimated and actual throughputs are measured in different units, they should correlate and be proportionally similar if our calculations are correct.

7.3.4.1 Exp 6: WeaveShare: estimated vs actual throughput (Figure 47)

In this test we can see that visually our actual throughput correlates with the expected one, which gives us confidence that our updated *WeaveShare* optimizer performs cost estimation of all the *IE* techniques rather accurately. The statistics say that if we normalize the scales of both estimated and actual throughputs by equating their largest readings (in Figure 47), the average deviation between estimated and practical readings averages 31%, which is a good result given the dependency of the actual performance on a variety of system/environmental factors. Thus, we conclude that our estimations of the *IE* techniques are accurate enough to be used in the *MQ* optimizer *WeaveShare*.

7.3.4.2 Exp 7: TriWeave: estimated vs actual throughput (Figure 48)

In this test we can again see that our actual throughputs are visually similar to the

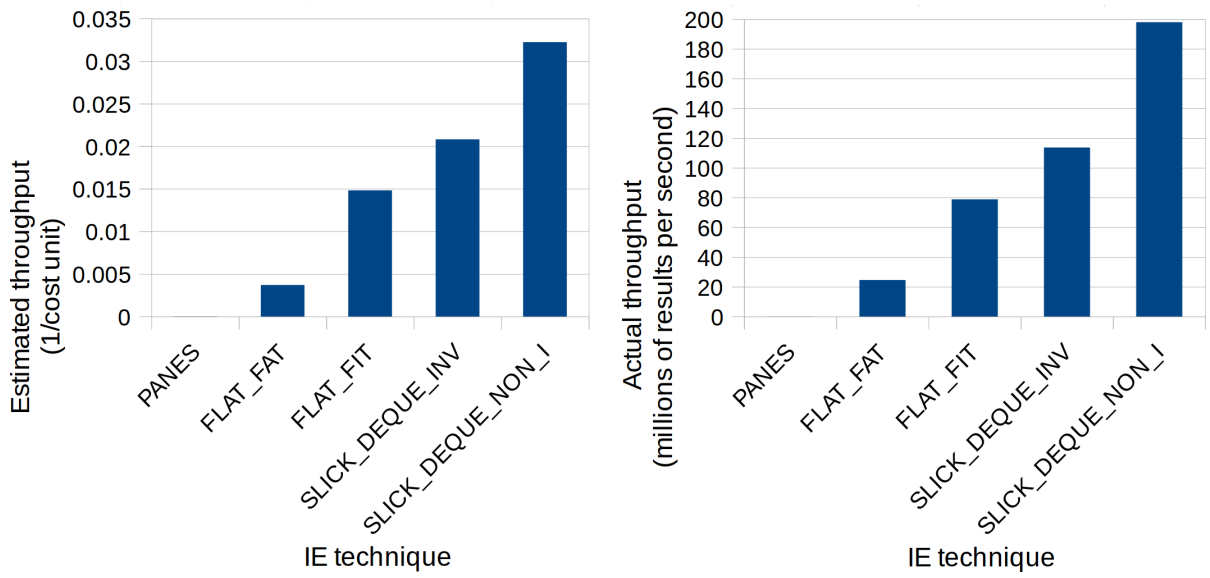

Figure 48: TriWeave: estimated throughput in 1/cost_unit per second (left), actual throughput in results per second (right)

expected ones. If we again normalize the scales of both estimated and actual throughputs in Figure 48, the deviation is on average only 14%, which is even better than in Exp. 6. Again we have confidence that our calculations have meaning and can be used in the *MQ* optimizer *TriWeave*.

7.3.4.3 Practical Evaluation Summary.

The correlation that we see between the estimated and the actual performance numbers (with an average deviation of only 22%) gives us confidence that our new final aggregation cost calculations (Ω) are valid and can be utilized in cost-based *MQ* optimizers.

7.4 Summary

The key contribution of this chapter is combining the recently developed *IE* techniques with the cost based *MQ* optimizers. We provided a theoretical analysis of all the available

IE techniques that determines their average operational cost (Ω) per slide given any set of input *ACQs*.

We used this analysis in our cost estimation formulas and experimentally compared all of the *IE* techniques as part of the *MQ* optimizers. We identified that *SlickDeque* allowed the optimizers to produce the most efficient execution plans by outperforming the rest of the techniques by up to 270,000x in terms of the estimated plan cost.

We also used the above analysis to experimentally show that our estimated performance is on average within 22% of the practical performance using a real dataset, which proves the validity of using our cost calculations (Ω) as part of any cost-based *MQ* optimizer.

8.0 Conclusions & Future Work

8.1 Summary of Contributions

This dissertation aimed to improve the state-of-the-art algorithms and optimizers used for processing *SWAG*, which are at the core of modern data analytics. After we identified the shortcomings in current *IE* techniques and *MQ* optimizers we examined how the *IE* techniques and *MQ* optimizers apply the principle of sharing, which lead us to develop a *A Taxonomy of all IE techniques* (Chapter 2) available today. This taxonomy organized these techniques in terms of applicability, complexity, and usability in *MQ* environments, and provided the foundation that led to our **hypothesis**:

Sliding-window aggregation processing can benefit from (1) improving the performance of Incremental Evaluation by exploiting the algebraic properties of ACQ's underlying aggregate operations and (2) developing new Multi-Query optimizers that can target multi-node distributed environments and efficiently generate high quality execution plans by exploiting the new Incremental Evaluation techniques.

We supported the first part of our hypothesis with the development of two novel *IE* techniques, which we evaluated both theoretically and experimentally to demonstrate their practical impact:

- *FlatFIT* (Chapter 3), a new efficient final aggregation technique that allows high *ACQ* processing throughput by dynamically storing the intermediate results and their corresponding pointers in a novel indexing structure, which indicates how far ahead *FlatFIT* can skip in each step of its calculation, reducing the number of partials used in performing a final aggregation. We experimentally show that *FlatFIT* achieves up to a 17x throughput improvement over *FlatFAT* (the state-of-the-art approach at the time) for the same input workload while using less memory.
- *SlickDeque* (Chapter 4), another new final aggregation technique that maintains both high throughput and low latency in *ACQs* processing by treating *ACQs* differently based on their invertibility property. The invertible operations are processed using *SlickDeque*

(Inv), our new modified *Panes* (Inv) approach, while non-invertible *ACQs* are processed with *SlickDeque* (Non-Inv), our novel deque-based algorithm that intelligently maintains and utilizes intermediate partial aggregates allowing a greater level of reuse of previously calculated results. We show that *SlickDeque* maintains 283% lower latency spikes on average while achieving up to 345% throughput improvement over the state-of-the-art approaches along with requiring up to 5 times less memory.

Towards the second part of our hypothesis for developing new practical *MQ* optimizers we developed (1) a closed formula for efficient calculations of overlapping windows, and (2) multiple new approaches of *MQ* optimization for multi-tenant cloud environments that utilize concepts of *weaving* and *grouping* and take advantage of our new formula:

- *F1* (Chapter 5), a novel closed formula that accelerates all of the *Weavability*-based *Multi-Query* optimizers by replacing the iterative and calculation-heavy *Bit Set* method with a closed formula for *Weavability* calculations. We showed that $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple *ACQs* by up to 60,000x, and that it is superior to the current approach in both time and space complexities.
- *Distributed ACQ Optimizers* (Chapter 6), a set of novel *Weavability*-based *Multi-Query* optimizers for processing *ACQs* in a distributed environment, including *Weave-Group to Nodes* (WG_{TN}) and *Weave-Group Inserted* (WG_I) optimizers, that produce plans of a significantly higher quality than the rest of the optimizers by minimizing the total cost (where WG_{TN} is best in 90% cases) and achieving better load balancing (where WG_I is best in 80% cases).

Finally, to connect all the dots in our hypothesis we show how our newly proposed *IE* techniques can be integrated into the *MQ* optimizers to achieve maximum performance efficiency in *SWAG* processing:

- *New Cost Estimation* (Chapter 7), a new approach based on a theoretical analysis of all of the available *IE* techniques that accurately determines their average operational cost per slide (Ω) given any set of input *ACQs*, which allows estimating their performance on average within 22% of the actual performance.

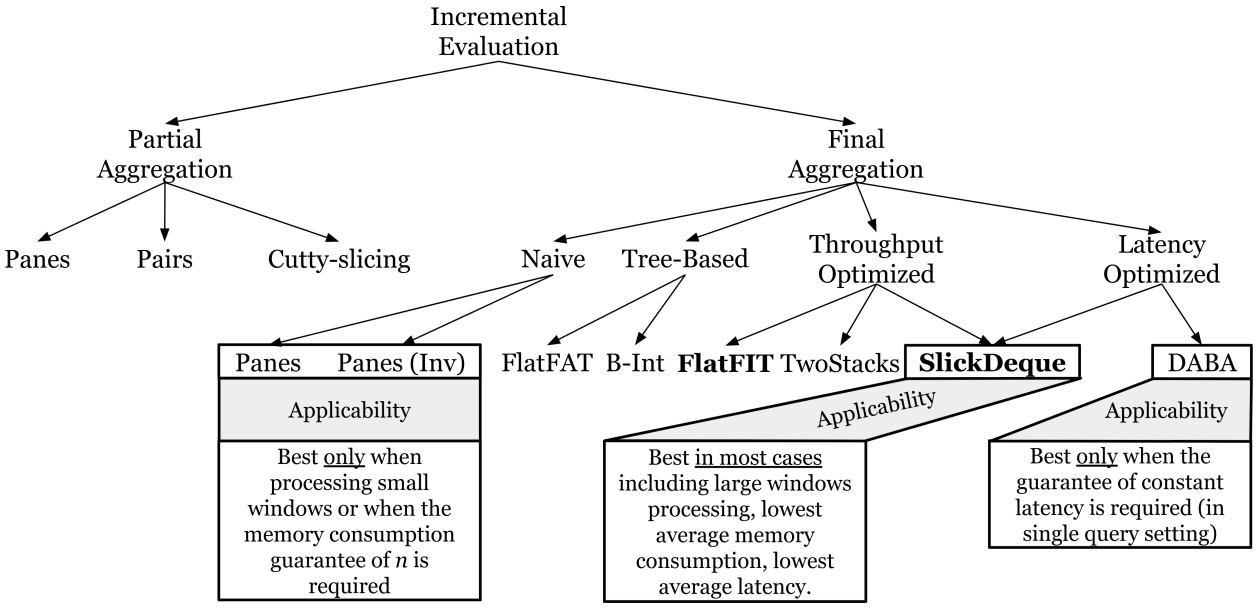

Figure 49: Incremental Evaluation Taxonomy and Applicability. Our contributions are bolded.

- *MQ Optimization of IE ACQs* (Chapter 7), a novel solution that achieves up to 270,000x improvement in execution cost by combining the new *IE* techniques with the state-of-the-art *MQ* optimizers using the *New Cost Estimation* above.

In addition to algorithmic contributions, this dissertation produced two experimental testbeds that allow extensive experimental *SWAG* evaluations to be carried out using both synthetic and real data sets: (1) a C++ based execution platform for measuring the performance of different *IE* techniques, and (2) a Java based *MQ* optimization platform for generating execution plans by selectively combining large numbers of *ACQs* into execution trees. These platforms can be potentially used for other experimentations as well.

All our experimental and theoretical findings in this dissertation lead us to a better understanding of how the processing of aggregate continuous queries could be optimized to achieve higher processing performance. As part of our investigation we determined the space of applicability of *IE* techniques, which we summarize in Figure 49. We also learned that the new *IE* techniques are not only applicable but also favorable when used as part of single-

and multi- node *MQ* optimizers, allowing them to achieve significantly higher computational efficiency.

8.2 Future Work

Clearly, there are many potential directions for future work derived from this dissertation. Our *IE* techniques and *MQ* optimizers can be expanded and improved to support:

- *Heterogeneous computation environments*, where each node has a different computational capacity, which needs to be taken into account by the optimizer.
- *Dynamic computation environments*, where nodes can be added/removed on-the-fly, and the execution plan needs to be adjusted accordingly.
- *Evolving workloads*, where the execution plans are adjusted based on the current demand on the system, which includes dropping low priority *ACQs* in a system with insufficient resources.
- *Approximate query answers*, where CPU time and memory can be saved by sacrificing query result accuracy.
- *Out-of-order processing*, where the *ACQs* need to take into account late arrivals.
- *ML-driven* MQ *optimization*, where machine learning techniques can be applied to speed up and/or improve our *MQ* plan generation process.

The obvious next step of this work is to implement the algorithms and optimizations proposed in this dissertation on a real, general-purpose production system. One such system in which our work can be incorporated is Apache Flink, which provides a general interface to efficiently process window aggregations using general stream slicing [49].

8.3 Broad Impact

In the current business environment, a growing number of applications are becoming available to wider audiences, resulting in an increasing amount of data being produced. A

large volume of this generated data often takes the form of high velocity streams. At the same time, online data analytics have gained momentum in many applications that need to ingest data fast and apply some form of computation, such as predicting outcomes and trends for timely decision making. This dissertation addresses the challenges of efficiently processing large amounts of data arriving with high velocities in the form of streams, and our contributions open the possibilities to meet the near-real-time requirements of analytical applications, whether they are business, health care, science, security, social media, etc.

Financially, this dissertation enables the enterprises to increase their profits by allowing them to process analytics more efficiently and thus satisfying more client requests using the same resources. In multi-node settings our work allows for the balancing of workloads among computation nodes, preventing the need to over-provision in order to cope with unbalanced workloads. This ultimately reduces infrastructure costs and saves energy while still meeting the requirements of the installed analytical queries. Not only does this save on the monetary cost of hardware, but it also saves on the manpower required to install, configure, and support infrastructure throughout its lifecycle.

The above advancements result in cost decreases for online analytics, making the technology more easily available to a number of industries, including health care, science, and social media, empowering user to make better business decisions with a high degree of confidence in the supporting data. Since almost every industry is growing their use of big data, the opportunities to apply this work will only continue to grow.

Bibliography

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 2003.

[2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[3] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *SIGMOD*, 2004.

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[6] I. Brigadir, D. Greene, P. Cunningham, and G. Sheridan. Real time event monitoring with trident. In *RealStream*, 2013.

[7] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *DataEngConf*, 2003.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. 2015.

[9] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, 2016.

[10] S. Chandrasekaran et al. Telegraphcq: continuous dataflow processing. In *SIGMOD*, 2003.

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

[12] C. Chung, S. Guirguis, and A. Kurdia. Competitive cost-savings in data stream management systems. In *COCOON*. 2014.

[13] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 2002.

[14] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD*, 2008.

[15] 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. *TKDE*, 2007.

[16] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, 2002.

[17] 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. *Data mining and knowledge discovery*, 1997.

[18] T. S. Group. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 2003.

[19] S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, 2012.

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

[21] A. Gupte and R. Agrawal. Solving big data challenges with data science at uber. Uber Engineering.

[22] M. A. Hammad et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[23] D. Harris. Facebook shares some secrets on making mysql scale. Gigaom.

[24] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *DEBS*, 2012.

[25] Y. Ji, T. Heinze, and Z. Jerzak. Hugo: real-time analysis of component interactions in high-tech manufacturing equipment (industry article). In *DEBS*, 2013.

[26] J. M. Joseph. Predicate decomposition for signature generation. In *MS Project, University of Pittsburgh*, 2016.

[27] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: a visualization-oriented time series data aggregation. *VLDBJ*, 2014.

[28] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. Vdda: automatic visualization-driven data aggregation in relational databases. *VLDBJ*, 2016.

[29] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *VLDBJ*, 2017.

[30] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. Concept-driven load shedding: Reducing size and error of voluminous and variable data streams. In *Big Data*, 2018.

[31] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[32] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[33] 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*, 2005.

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

[35] B. Moon, I. F. V. López, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *DataEngConf*, 2000.

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

[37] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *VLDB*, 2017.

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

[39] T. K. Sellis. Multiple-query optimization. *TODS*, 1988.

[40] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 2008.

[41] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. F1: Accelerating the optimization of aggregate continuous queries. In *CIKM*, 2015.

[42] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Processing of aggregate continuous queries in a distributed environment. In *BIRTE*, 2015.

[43] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*, 2017.

[44] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Slickdeque: High throughput and low latency incremental sliding-window aggregation. In *EDBT*, 2018.

[45] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *DEBS*, 2017.

[46] K. Tangwongsan, M. Hirzel, and S. Schneider. Sliding-window aggregation algorithms. *PVLDB*, 2019.

[47] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *VLDB*, 2015.

[48] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.

[49] J. Traub, P. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *EDBT*, 2019.

[50] E. Weisstein. Binomial series. Wolfram MathWorld.

[51] E. Weisstein. Euclidean algorithm. Wolfram MathWorld.

[52] E. Weisstein. Harmonic number. Wolfram MathWorld.

[53] E. Weisstein. Prime number theorem. Wolfram MathWorld.

[54] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *DataEngConf*, 2001.

[55] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS*, 1995.

[56] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[57] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.