Targeted Prioritized Processing in Overloaded Data Stream Systems

by

Karen Works

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

November 4, 2013

APPROVED:

Prof. Elke A. Rundensteiner
Worcester Polytechnic Institute
Advisor

Prof. Emmanuel Agu
Worcester Polytechnic Institute
Committee Member

Prof. Matthew O. Ward
Worcester Polytechnic Institute
Committee Member

Dr. Kajal Claypool
MIT Lincoln Laboratory
External Committee Member

Prof. Craig E. Wills
Worcester Polytechnic Institute
Head of Department

Abstract

We are in an era of big data, sensors, and monitoring technology. One consequence of this technology is the continuous generation of massive volumes of streaming data. To support this, stream processing systems have emerged. These systems must produce results while meeting near-real time response obligations. However, computation intensive processing on high velocity streams is challenging. Stream arrival rates are often unpredictable and can fluctuate. This can cause systems to not always be able to process all incoming data within their required response time. Yet inherently some results may be much more significant than others. The delay or complete neglect of producing certain highly significant results could result in catastrophic consequences. Unfortunately, this critical problem of targeted prioritized processing in overloaded environments remains largely unaddressed to date. This dissertation now addresses four issues. First, the problem of optimally processing the most significant tuples identified by the user at compile-time is addressed. Proactive Promotion (PP), a new targeted prioritized processing data stream (TP) methodology for preferential CPU resource allocation, selectively pulls the most significant tuples ahead of less significant tuples. Second, a new aggregate operator, TP aggregation (TP-Ag), that effectively increases the accuracy of aggregate results produced for TP systems is proposed. TP-Ag generates reliable results from incomplete aggregation group populations by selectively controlling which subsets in the aggregate group population are used to generate each result. Third, the problem of identifying significant tuples at run-time via dynamic determinants is addressed. Preferential Result (PR), a novel TP methodology for preferential CPU resource allocation, uses carefully designed statistics to identify effective dynamic determinants online and efficiently determines the placement of where in the query pipeline each dynamic determinant is evaluated. Fourth, a new join operator, PR Join operator (PR-Join), that efficiently produces the join results in significance order is proposed. To achieve this, PR-Join empowers the atomic join scan process to be interruptible at a finer level of granularity than previously achieved. The experimental studies in this dissertation explore a rich diversity of workloads, queries, and data sets, including real data streams. The results substantiate that the proposed approaches are a significant improvement over the state-of-the-art approaches.

Executive Summary

Main Problem: The main problem addressed in this dissertation is to develop data stream processing models and operators that prioritize the processing of critical tuples in data stream management systems. In such systems the load may adapt over time. In addition, such systems may at times get overloaded. When this occurs, not all incoming tuples will be processed within the response time required by the end-user. These critical tuples are tuples that are essential to produce results for. They should be processed before less critical tuples. They can be identified by either a multi-tiered preference model specified by the user at compile-time or by using dynamic preferences identified at run-time by the data stream management system.

Importance: Users now require data stream management systems that produce results while meeting near-real time response obligations. Our TP models and operators allow users to gain a fine level of control over which tuples are processed when a data stream management system is overloaded. This control is essential in data stream management systems where certain results are more critical than others and each result has a degree of criticalness compared to the other results. If the most critical results are not produced then the end-user may be missing some

critical information. Any decision made by the user based upon the incomplete knowledge is likely to be sub par.

There are no proposed systems that support allowing users to gain a fine level of control over which tuples are processed when a data stream management system is overloaded. We are the first to look at such an approach.

Main Accomplishments:

- Our data stream query optimization approach efficiently determines how best to allocate resources to ensure the processing of certain subsets of tuples within the query pipeline. For each subset of critical tuples, we consider the cost of precedence determination to identify these tuples. Our model determines the best position in the query plan to perform the precedence determination for each subset of critical tuples.
- Our execution infrastructures support the online adaption of how resources are allocated without requiring any infrastructure changes. This allows the system to quickly adjust how resources are allocated.
- Our operators produce accurate results while ensuring that resources are allocated to producing the most significant results first.
- Our cost models factor in the overhead of precedence determination.
- Our systems support complex and costly precedence determination. They are designed to support any precedence criteria that can be specified by the user or identified at run-time.

General Take-Away Message: In data stream management systems, where some results are more critical than others, our TP models and operators can produce a

higher quantity of the most critical results. Ultimately TP allows the end-user to make better decisions by providing the user with the most critical information.

Acknowledgments

I express my sincere thanks to my advisor Professor Elke Rundensteiner. In addition, I thank the members of my Ph.D. committee, Professor Matthew O. Ward, Professor Emmanuel Agu, and Dr. Kajal Claypool. Especially, I thank Worcester Polytechnic Institute, the Graduate Assistance in Areas of National Need Program (a.k.a. GAANN), and the National Science Foundation (NSF grants: 0917017, 0414567, and 0551584) for their financial support. I thank my husband, Jim Works, for his unconditional love, long standing encouragement, and support. Last but not least, I thank my DSRG peers for their feedback.

Dedication

I dedicate my dissertation to God.

”I can do all things through Him who gives me strength.” Philippians 4:13

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The recent advances in hardware and software have enabled the capture of different measurements of data in a wide range of fields. Technology that generates rapid, continuous, and large volumes of data streams include readings from sensors used in applications, such as weather sensors [Uni02, HFAE03, FJK$^+$05], health sensors [SB03], network sensors [Uni02], online auctions, credit card operations [TMSF03, TTPM03], financial tickers, and web server logs. The world is primed for a variety, scale, and importance of real-time applications - from dynamic traffic management, environmental monitoring, to health care. Query processing technology that supports real-time query processing on such continuously changing, often unpredictable, and possibly unbounded data streams, is crucial for these mission critical applications. A flurry of research activity addressing the problem of data stream processing has arisen, ranging from the design of continuous query operators [CLYY92, UF99, DGGR02, DRH03, GÖ03, VNB03, SW04, DR04, MLA04, DMRH04, WRGB06, SB13, CFMKP13] to new operator scheduling algorithms [UF01, BBD$^+$02, HFAE03, CcR$^+$03, SPZ$^+$05, DLX07, WXL$^+$09, QL10, FY11, ZWL13], and beyond. Database companies have developed data stream processing systems. These efforts include Microsoft SQL Server StreamInsight [Stra], Oracle Event Processing (OEP) [Ora], IBM System S [IBM], Coral8 [Cor], StreamBased Systems [Strb], and Esper [Esp]. However, the critical problem of processing tuples in significance order where tuple precedence is multi-tiered remains unaddressed, as demonstrated below.

Processing queries over large volumes of data arriving from high-speed data streams with fluctuating arrival rates for days, months, or even indefinitely can be extremely resource intensive. At times these systems may not be able to process all incoming data *within their response time* [CcC$^+$02], especially when supporting complex queries with large operator states. Yet inherently some results may be *more valuable* to the user than others. Ideally, when CPU resources are limited, the precious resources should be dedicated to assure the production of the most significant results, followed by the next most important ones, and so on, until all resources are exhausted. The delay or even complete neglect of processing such highly significant query results can have catastrophic consequences.

1.2 Electronic Monitoring Applications

Electronic monitoring applications (or *EMA*s) [GP00, KPP$^+$02, LCH$^+$06] track the most up-to-date data on objects to answer continuous queries. Objects periodically transmit their current status. They process continuous queries online over large volumes of data arriving from high-speed data streams. EMAs can be resource intensive. At times EMAs cannot process all incoming data [CcC+02], especially when queries are complex and have huge operator states. The problem is further complicated by the significance of objects which is often multi-tiered as motivated by the examples that follow.

EMA Example: Outpatient Health Care

EMAs track people with dementia [LCH+06]. It is critical to monitor people away from their proper location for an extended time (i.e., lost). Monitoring people who live on their own (i.e., need help) may be reduced based on whether or not resources remain after processing people likely to be lost. Until enough resources are available, monitoring other people could be temporarily skipped. These EMAs can experience technical glitches such as overloads [KMH+10].

1.3 Running Examples & Their Challenges

1.3.1 Home Arrest Example

The *Home Arrest Example* is based upon EMAs that monitor prisoners assigned to home arrest [GP00]. Such EMAs can also be overloaded. In October 2010, an application monitoring released sex offenders across 49 states shut down for 12 hours [Pre10]. Consider a *Home Arrest* EMA that reports any prisoner at an improper location who is within 3 miles of a police officer (see CQL query [ABW06] below). Patrol cars (via GPS) and prisoners (via ankle bracelets) continuously submit their locale. First, each prisoner's current locale is compared to a table of permitted locations. Then the results are joined with the officer's current locales.

Home Arrest Query:

SELECT PL.PrisonerID, PL.Locale, OL.OfficerID

FROM prisonerLocation PL, prisonerInfo PI, officerLocation OL

WHERE PL.Locale != PI.ProperLoc **AND** PL.PrisonerID = PI.PrisonerID

AND Distance(PL.CurrentLoc, OL.CurrentLoc) $\leq$ 3 miles

WINDOW 30 *seconds*

Consider the following criteria to establish the monitoring order of prisoners (Table 1.1). First, escaped violent prisoners (i.e., may cause harm) must be monitored with the highest level of urgency. Next, prisoners at an improper location (i.e., likely to be in violation) shall be monitored. Finally, if resources permit, prisoners known to be flight risks ought to be monitored.

Table 1.1: Home Arrest Example: Criteria for Desired Resource Allocation

System Load	**Desired CPU Resource Allocation Order**
System not overloaded	all tuples processed in FIFO order
System mildly overloaded	tuples from prisoners 1) escaped and likely to cause harm 2) likely to be in violation 3) considered a flight risk
System moderately overloaded	tuples from prisoners 1) escaped and likely to cause harm 2) likely to be in violation
System extremely overloaded	tuples from prisoners 1) escaped and likely to cause harm

As illustrated by real events [Pre10, Net12], EMAs may not be able to accurately estimate their system load. Without an accurate estimation, the system may have inadequate resources to support the load and will shut down. It is critical to ensure that the EMAs do not shut down and that the monitoring of certain objects continues until additional resources can be provided to the EMA. This requires an infrastructure capable of selectively allocating CPU resources to significant tuples in progressive order - assuring that at any time the available resources are allocated to processing the most significant tuples before less significant ones throughout the pipeline. The overhead of processing may adapt over time. Thus such a data stream management system (or DSMS) should be agile to rapidly adapt to system load changes (i.e., act *proactive*ly) and pull certain more significant tuples ahead of less significant ones (i.e., *promote* specific tuples). For example, highly significant tuples are generated by escaped violent prisoners (Table 1.1).

1.3.2 Stock Market Example

EMAs also monitor stocks online [KPP$^+$02]. Online financial applications can also get overloaded. In 2012, the London Stock Exchange shut down after a rash of computer-generated orders overwhelmed the system [Net12].

Consider an application that monitors the average price of stocks by their business sector that appear in recent news and "street research" (See *Stock Market Join Query* below).

Stock Market Join Query: /*Operators*/

SELECT S.company_name, S.symbol, S.price

FROM Stock as S, News as N, Blogs as B

WHERE contains(S.BusinessSector, N.BusinessSector) /*op_1*/

AND contains(S.BusinessSector, B.BusinessSector) /*op_2*/

WINDOW 30 *seconds*

Consider an application that monitors the average price of stocks by their business sector that appear in recent news and “street research” (See *Stock Market Aggregate Query* below).

Stock Market Aggregate Query: /*Operators*/

SELECT avg(S.price)

FROM Stock as S, News as N, StreetResearch as SR

WHERE contains(S.sector, News[10 min]) /*op_1*/

AND contains(S.sector, StreetResearch[15 min]) /*op_2*/

Group by S.sector /*op_3*/

WINDOW 30 *seconds*

Mutual fund companies often invest in diverse stock portfolios. It is critical to ensure that every object of a certain class (e.g., their aggressive investments) is monitored. While the monitoring of other objects (e.g., their conservative investments) may be reduced based on what resources remain after processing the more significant objects. Until all important objects can be processed within their response time, monitoring of certain less important objects can be temporarily skipped altogether (e.g., stocks under evaluation).

Consider an overloaded EMA executing the Stock Market Join Query in the Stock Market Example. CPU resources will be dedicated to process the most significant tuples (i.e., aggressive stock investments) ahead of any other tuples.

Resources should be allocated to particular tuples based upon the application’s desired result precedence order (Table 1.2) and the amount of available resources.

Table 1.2: Stock Market Example: Desired Result Precedence Order

System Load	**Desired Result Precedence Order**
System not overloaded	all results produced
System mildly overloaded	1) aggressive investments 2) conservative investments 3) stocks under evaluation
System moderately overloaded	1) aggressive investments 2) conservative investments
System extremely overloaded	1) aggressive investments

When the *Stock Market Application* is extremely overloaded, only the tuples most critical for the application should be dedicated CPU resources. These tuples create the most critical results, namely, results about aggressive investments. These most critical results are formed when *news tuples* join with aggressive *stock tuples* based upon their business sector, i.e., op_1. Next, these join results from operator op_1 are joined with *blog tuples* based upon their business sector, i.e., op_2.

We distinguish between two classes of tuples that create these most critical results. The first class corresponds to so called native *significant tuples*. That is, significant tuples satisfy *static precedence criteria* defined explicitly by the user at *compile-time*. For example, a significant tuple in the stock stream can be identified as an aggressive or as a conservative investment simply by checking if its attributes match the given significance criteria (Table 1.2).

The second more interesting class of tuples are *promising tuples*. Promising tuples are tuples estimated to be *highly likely to produce significant results by association*, i.e., by joining with significant tuples. For example, tuples in the news stream may join with significant stock tuples and thus produce significant query results due to their association with their join partners. In this case, the criteria to identify promising tuples are indeed *dynamic*. This requires knowledge of which join attributes of the current significant stock tuples are also prevalent in tuples in both the news and blog streams. Put differently, this requires identifying which business sectors of significant stock tuples appear in both recent news and recent blogs. The identification of such dynamic criteria must be accomplished at *run-time*. This is not only complex but also prohibitively costly.

Promising tuples are deemed likely to join with significant tuples at specific join operators. However, these promising tuples may also join with insignificant tuples. Hence, unfortunately, such promising tuples may create both significant and insignificant join results. Producing all join results generated by these promising tuples places less desirable results into the pipeline.

The issues listed above may cause highly significant results to not be produced. This can result in dire consequences such as a large monetary loss.

Now consider an overloaded EMA executing the Stock Market Aggregate Query in the Stock Market Example. Recall that the CPU resources will be dedicated to process the most significant tuples (i.e., aggressive stock investments) ahead of any other tuples.

This may cause the average price per business sector produced by aggregate operator op_3 in the Stock Market Aggregate Query to be skewed, incomplete, or even erroneous. Namely, under limited resources tuples from all other less significant tuples may have been ignored and thus not be reflected in the aggregate result. In this case, the aggregate result produced will be generated only from those most significant tuples in the aggregate group that reach operator op_3. Clearly, this generated aggregate result may not match the final aggregate result that would have been produced if all tuples in the aggregate group had reached operator op_3. The average price computed for business sectors that appear in recent news and blogs would correspond to the average price of the aggressive stocks in that business sector. It may however not correspond to the average price of all stocks in that business sector.

Given such an aggregate result, a broker would also not necessarily know which subset of tuples was considered when computing the aggregate value. Without such knowledge, a broker is bound to make sub-optimal trades, which may lead to investors losing money.

1.4 Requirements for the Multi-Tiered Priority-Driven Processing Problems

The above scenarios motivate the need for distinct levels of response according to the risks of the objects being monitored. That is, resource allocation must be aligned to the monitoring priorities of objects and the system load. We refer to the system that achieves this as targeted prioritized data streams (or *TP*). TP utilize user-defined preference criteria to determine the order in which tuples should be allocated resources.

We derive the following requirements for TP system models.

1) Prioritization is a multi-level preference scheme. 2) Tuples typically arrive at the system without any assigned significance. 3) Each resource allocation preference level has its own distinct criteria to identify tuples at this significance level. 4) Determination and assignment of significance to tuples at run-time can be costly. 5) Certain criteria can only be found at runtime and depend upon the current tuples

in other streams. 6) Priority determination must be lightweight to not delay the production of significant results.

We derive the following requirements for aggregate operators in TP systems. 1) The user must be informed about which tuples were used to create the result. Consider the example above. The aggregate result should at the very least annotate that it is formed from mostly aggressive investments only (i.e., a sub group population) instead of from all investments (i.e., the full group population). 2) The aggregate operator must be designed to generate certain aggregate results only from sub group population(s) where each population best represents their actual sub group population. There may not even be enough resources to process all tuples from a sub group population, e.g., all tuples from aggressive investments. In such cases, a result may be produced that does not match the actual aggregate result for aggressive investments. This again could cause a broker to make improper trades. Aggregate results produced from these incomplete sub group populations may be inaccurate. That is, the incomplete sub group population that the result is generated from (e.g., some stocks from aggressive investments) may not accurately represent the complete sub group population (e.g., all stocks from aggressive investments). An aggregate operator must be designed to generate certain aggregate results only from sub group population(s) where each population best represents their actual sub group population.

We derive the following requirements for join operators in TP systems. 1) The join operator must control which join results are emitted from the join process. That is, only a selective subset of the possible join results that can be produced by tuple t_i during the state scan should actually be produced at any given time. This defies the common design assumption in stream joins that encapsulate the state scan task as one atomic process [WA91]. 2) To achieve this, the join scan process must be *interruptible*, i.e., the join operator must be able to halt the production of join results from tuple t_i at any time. 3) Correspondingly, the join operator needs the capability to reinstate the processing of tuple t_i to *resume the production* of still missing join results. 4) Such a join operator must be carefully designed to not either erroneously produce duplicate join results or fail to create all significant join results. 5) The join process must be light weight. Otherwise it defeats the purpose of TP, namely, of optimizing the production of significant results under severe resource shortages.

1.5 TP Problem Challenges

Determining the significance of a tuple (a.k.a. *precedence determination*) may not be cheap. Reconsider the Home Arrest Query. To locate the most significant tuples (i.e., escaped violent) requires the current locale and time attributes to be compared to stored known permitted locations by time. The number of permitted locations by time may be too large for main memory. This may require a costly database look-up. To identify the next most significant tuples (i.e., prisoners likely to be in violation) may also be expensive. Being likely to be in violation is not a constant value; it pertains to a prisoner's recent movements. It requires a spatial-temporal query to compare the sensor locations of prisoners with the spatial restrictions imposed upon them. This again requires access to permitted locations by time. This leads to the observation that *precedence determination itself is expensive and must be taken into consideration for design of effective EMA processing*.

The cost of precedence determination affects the latency of tuples. Latency is also affected by the time spent on performing precedence determination of incoming tuples against other significant levels. EMAs must consider *the cost of precedence determination (i.e., which and where precedence criteria are evaluated) as it affects the latency of tuples in each tier*.

It is challenging to locate promising tuples (e.g., join partners for significant stock tuples) at each join operator. For this, we propose to exploit the join criteria of such stock tuples. These selected join criteria effectively act as selection preference criteria. They thus could then be pushed backwards through the query pipeline to identify and pull forward promising tuples from news or blog streams before their respective join operators.

However, promising tuples may join not only with significant but also with insignificant tuples. That is, they may also produce undesired insignificant query results. The key challenge is thus how to select dynamic criteria online that are effective at identifying promising tuples that create the largest quantity of significant query results while not clogging the pipeline with critical tuples.

To complicate the problem, queries may contain multiple join operators. Each join operator may have its own distinct join criteria. It thus can be rather costly to determine the most effective combination of criteria that should be exploited to identify promising tuples. If the optimization overhead is costly then the production of significant query results may be delayed or even worse yet prevented. In the Stock Market application this could result in the company losing money or in the worse case, going bankrupt.

Informing the user about which tuples were used to create each aggregate result is challenging. We must design an aggregate operator that can quantify for each aggregate result its group population, e.g., only stocks from aggressive investments.

In addition, designing an aggregate operator to generate certain aggregate results only from sub group population(s) where each population best represents their actual sub group population is also challenging.

It is challenging to always produce the most significant join results first. A sophisticated solution is required that examines the join process in a new light. In particular, we must control which join results are emitted from the join process. That is, only a selective subset of the possible join results that can be produced by tuple t_i during the state scan should actually be produced at any given time.

1.6 Key TP Tasks

The multi-tiered priority-driven processing problem provides the following key tasks that need to be addressed.

One task is to develop a flexible framework that can adaptively and continuously change how resources are allocated such that the more significant tuples are pulled ahead of less significant ones. To complicate matters, determining the significance of tuples introduces additional costs at a time when resources are already at a premium. Hence at runtime, the query executor must efficiently adapt which and where significant tuples (i.e., tuples that meet precedence criteria) are preferentially dedicated resources in the query pipeline. A query optimizer must be developed to effectively determine online when and how to adapt how resources are allocated to promotable tuples.

Another task is to ensure that correct results are produced. Namely, correct results may not be produced by particular query operators if only certain tuples are processed. In particular, under limited resources only the most significant tuples may be processed and thus be available to create the aggregate results with. Ideally, the aggregate operator should generate correct aggregate results from only the tuples pulled forward. That is, the optimization decisions made by TP should not be adjusted. In addition, the aggregate operator should limit the aggregate results produced to those whose generation population is representative of their actual population. TPs needs such an aggregate operator.

A third task is to ensure that the greatest number of possible significant results are produced. To achieve this, TP must dedicate resources to both: 1) significant tuples (i.e., tuples that meet precedence criteria) and 2) promising tuples (i.e., tuples that facilitate the production of important results by combining with a significant tuple from another stream). However this problem cannot be corrected by simply using a semantic load shedder as identifying promising tuples is based upon the run-time data content of significant tuples. The state-of-art approaches do not address this issue. A solution is required that can support queries with more complex functionality; namely, allowing the promising tuple in one stream to be identified based upon the data content of significant tuples in another.

An additional task is to design operators that utilize CPU resources to produce results in significance order. In particular, join operators produce results by combining tuples from multiple streams. In addition, tuples can have different types of importance (i.e., significant or promising). The importance of the join results that can be produced by a single tuple could potentially vary based on the importance of the tuple's join partners. TP requires a join approach that allocates CPU resources based upon the importance of potential join results and the importance of the tuples waiting to produce results.

Another task is to efficiently support the resource allocation preferences of multiple queries. Each query may have its own defined resource allocation preferences. To complicate matters, portions of a query pipeline may be shared by more than one query. In such instances, multiple resource allocation preferences may be defined for tuples that reside in the same portion of the query pipeline. The resource allocation preferences of the queries may have conflicting requirements for the same stream. The requirements across the queries may also overlap. The questions now arise. How to best honor the resource allocation preferences of all the defined queries that share segments of the query pipeline? How to ensure that the resource allocation implemented does not cause significant tuples in the query pipeline to be completely starved of resources?

A final task is to develop a flexible architecture that allows the user to adaptively and continuously change their resources allocation preferences of any query online. This is a complicated issue. Once the user selects new resource allocation preferences, the current inprocess query plan must be adapted. In such instances, inprocess tuples that have been pulled forward may no longer be considered important. Conversely, inprocess tuples that have been ignored may need to be rapidly pulled forward. In other words, inprocess tuples that were allocated resources according to the old user preferences may need special considerations to ensure that the new user preferences can rapidly be employed.

1.7 State-Of-the-Art and Their Shortcomings

1.7.1 Current Resource Allocation Methods

Allocating Resources to the Most Significant Tuples

State-of-the-art resource allocation methods (i.e, shedding [ACc$^+$03a, ZSC$^+$03, BDM04, NR07, WQL$^+$10, MZS10, SH12, BKZS12, LQJQ12, TNP13] and spilling [LZR06, WRM10]) allocate resources based on tuple significance by dropping less significant tuples from the workload. They respectively drop or temporarily put to disk insignificant tuples. A single binary decision of whether or not to process each tuple is made, typically at the start of the pipeline [ANWS06]. Thereafter the remaining tuples are processed in FIFO order. However, as illustrated below, these prior methods do not tackle several key challenges of EMAs.

Consider the Home Arrest Query executed in a moderately overloaded system (Fig. 1.1). Here, both shedding and spilling would only process tuples from escaped violent prisoners (e.g., black circles) or prisoners likely to be in violation (e.g., dark gray circles). Consider what happens when the workload suddenly fluctuates and many violent prisoners escape (e.g., many black circles arrive). In this case, all resources should swiftly be dedicated to tuples from escaped violent prisoners (e.g., black circles). In addition, any tuples from prisoners likely to be in violation (e.g., dark gray circles) should be ignored. In comparison, the state-of-the-art methods instead process all inflight tuples in FIFO order to completion *before* processing more significant incoming tuples. In short, these methods allocate resources to all inflight tuples regardless of workload fluctuations. This may cause very significant tuples to not be processed to completion. It may result in

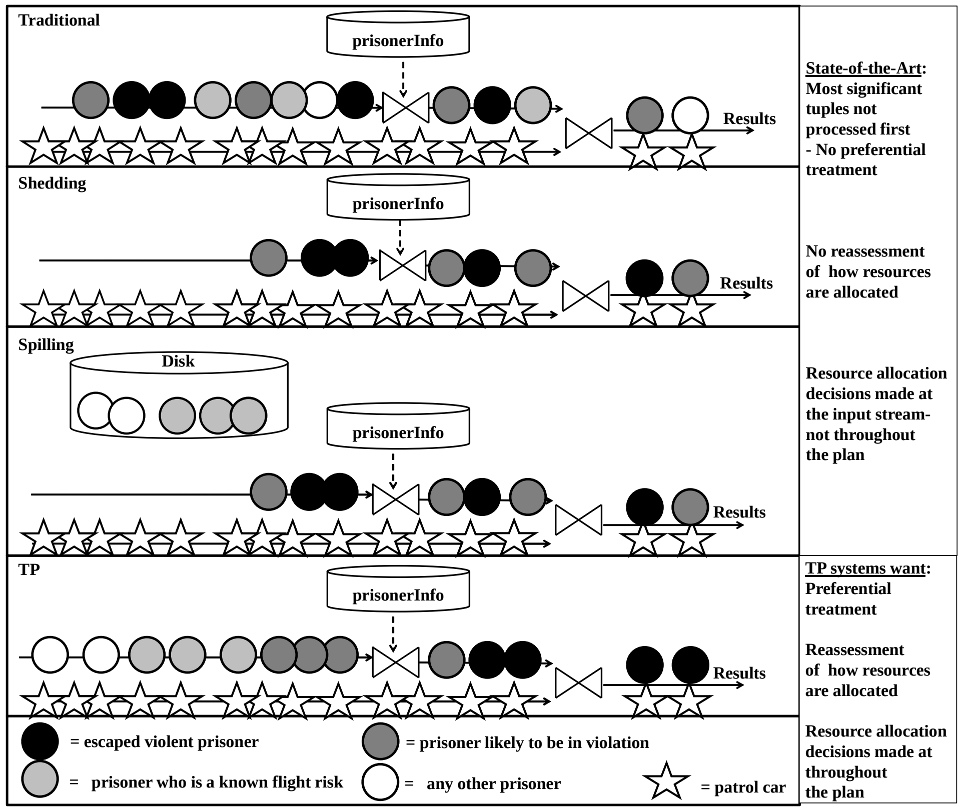

Traditional
prisonerInfo
Results
State-of-the-Art: Most significant tuples not processed first - No preferential treatment
Shedding
prisonerInfo
Results
No reassessment of how resources are allocated
Spilling
Disk
prisonerInfo
Results
Resource allocation decisions made at the input stream- not throughout the plan
TP
prisonerInfo
Results
TP systems want: Preferential treatment
Reassessment of how resources are allocated
Resource allocation decisions made at throughout the plan
= escaped violent prisoner
= prisoner likely to be in violation
= prisoner who is a known flight risk
= any other prisoner
= patrol car

Figure 1.1: Resource Allocation in State-Of-the-Art Systems vs Desired Resource Allocation in TP

dire consequences, e.g., an escaped violent prisoner may harm someone.

In the ideal case, EMAs would prefer to allocate resources to these most significant incoming tuples from escaped violent prisoners (e.g., black circles) first. Then the remaining resources (if any exist) would be allocated to the inflight tuples from prisoners likely to be in violation (e.g., dark gray circles). Such an approach may cause all very significant tuples to be processed to completion and may prevent an escaped violent prisoner from harming someone. This would require EMAs to have *agile resource allocation control* to promptly adjust how resources are allocated throughout the pipeline.

Allocating Resources to the Critical Tuples

We now explore the more state-of-the-art related to the general and complex problem of optimizing the processing of significant tuples also identified by dynamic criteria.

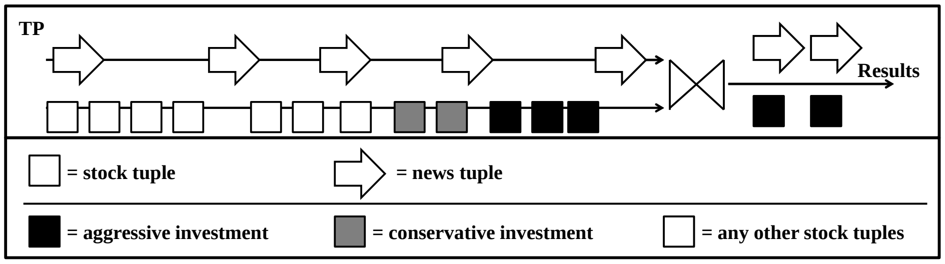


Figure 1.2: Issues with Allocating Resources to the Critical Tuples in TP

Consider a TP that executes the Stock Market query in a moderately overloaded system (Fig. 1.2). In this scenario, the TP (Ch. 4) would still be able to identify significant tuples in the stock stream using the static criteria. However, no static criteria exist to identify promising tuples from the news stream. Thus, to reduce the overload, news tuples would need to be randomly dropped [ACÇ$^+$03b]. This can cause severe problems. Most notably, news tuples that would have joined with significant stock tuples may be dropped. Consequently, some significant join results (e.g. aggressive investments) may not be produced.

The other related work is the inter-operator feedback punctuation framework [FMTL09]. In this approach operators use punctuations to communicate interest in particular subsets of the data stream. Each operator sends a ”request” to its adjacent operator in the pipeline asking for specific tuples to be pulled forward. In other words, each individual operator makes local resource allocation decisions that serve its particular interest. In complex queries that contain multiple join operators, each join operator may end up requesting different types of tuples. This could waste resources by pulling tuples forward that are only important to one operator but may be irrelevant to other operators. Ultimately, such an uncoordinated approach may produce fewer rather than more significant query results. TP instead requires a *global coordinated approach* that considers the impact of pulling promising tuples forward on the production of significant query results.

1.7.2 Current Aggregate Operators

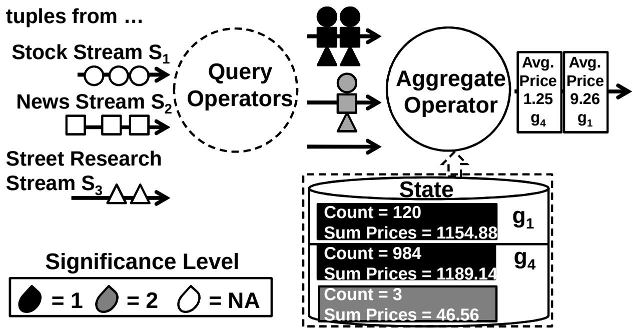


Figure 1.3: Aggregation Example

The state-of-the-art aggregation methodologies [BDM04, TZ06] designed to produce reliable results in TP systems adjust which tuples are processed *regardless of their significance*. [TZ06] limits the tuples dropped from a window w_p. Aggregate results for window w_p are generated from all tuples from window w_p and are non-skewed. [BDM04] adapts where and how many tuples are randomly dropped to ensure that all aggregate results produced are bound by a given error rate. These approaches require overhead to consider how to adjust the allocation of resources at a time when resources are scarce. In addition, they ignore the desired resource allocation order specified by the user. That is, they do not consider how to build reliable aggregate results from the significant tuples already pulled forward.

Consider the stream aggregate operator op_3 [DGGR02] in a TP system under duress that implements the Stock Market Aggregate Query defined above. The aggregate operator op_3 computes the average price of stocks in each business sector for sectors mentioned in recent news and in street research (Fig. 1.3). The average price for business sector group g_1 can only be produced from significant tuples at level 1. While the average price for business sector g_4 could be produced from only tuples at significant tuples at level 1, only tuples at level 2, or a combination of significance tuples at levels 1 and 2. However, the three significant tuples at level 2 may not be representative of the actual partial population of tuples at level 2. That is, TP requires an aggregate operator than can selectively control which partial populations are used to generate each result. The challenge is how to determine which set of tuples that arrived at the aggregate operator best represent the actual sub group population of tuples that would have arrived at the aggregate operator if resources were available.

1.7.3 Current Join Solution

Consider a symmetric binary hash join [WA91] op_1 employed in a PR system that implements the above Stock Market Join Query (Fig. 1.4). Join operator op_1 joins stock tuples (stream s_1) with news tuples (stream s_2) based upon their business sectors. Tuples in the query pipeline are stored in queues indexed by their significance. The top queue contains the most significant tuples (i.e., level 1). The middle queue contains the second most significant tuples (i.e., level 2). The bottom queue contains the insignificant tuples (i.e., level NA).

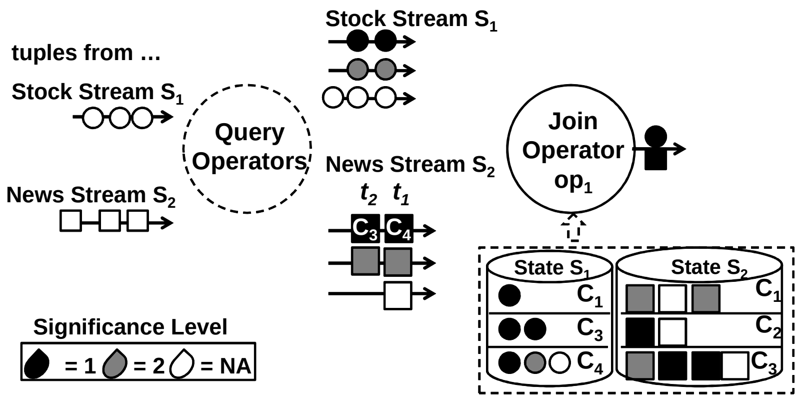


Figure 1.4: Example of Join Operator employed in PR

In Figure 1.4, the leading tuple in the most significant news stream queue (i.e., the top news stream queue) is processed first. This news tuple t_1 satisfies join criteria c_4, namely, business sector equal to $Advertising$. It will produce two significant results (at significance levels 1 and 2) and one insignificant result (at level NA) by joining with the three stock tuples in state s_1 from the advertising business sector. Next, the second news tuple t_2 in the most significant news stream queue is processed. It satisfies join criteria c_3, namely, business sector equal to $Retail$. The news tuple t_2 produces two significant results at level 1 by joining with the two stock tuples at significance level 1 in state s_1 from the retail business sector.

We notice that the results created by tuple t_2 are more significant than some of the results produced by tuple t_1. In other words, this join operator *repeatedly produces less significant join results (with t_1) before other more significant join* *results (with t_2)*. When resources are scarce, this could create a ripple-effect along the pipeline as resources are wasted on insignificant tuples may multiply and clog the pipeline.

We observe that this problem is caused by the join operator being *result significance-agnostic*. When it processes tuple t_i, it produces *all* join results for tuple t_i.

Reconsider the join operator op_1 in the PR in Figure 1.4. Ideally, to produce the join results in significance order, news tuple t_1 should not be forced to produce all its join results at once. This leads to the following insight. Namely, the news tuple t_1 should only produce the most significant results at significance level 1, while thereafter, the processing of news tuple t_1 should be *interrupted*. Then, the resources should be dedicated to producing significant join results at significance level 1 generated from other probe tuples. Only when no more join results at significance level 1 can be produced, should the processing of news tuple t_1 be *reinstated*.

1.8 Dissertation Objectives

Of the key TP tasks outlined above, this dissertation addresses the following four tasks:
Task 1 (Proactive Promotion): Design a TP framework, including developing an efficient special-purpose operator to establish tuple significance and enhancements to standard operators that support the ability to adaptively and continuously change how resources are allocated such that the more significant promotable tuples are pulled ahead of less significant ones. In addition, an optimizer must be developed that can efficiently locate the optimal TP plan to support promotable tuples in a

single query at run-time. Further the architecture must be able to efficiently adapt at run-time which, where, and when promotable tuples are preferentially allocated resources throughout the query plan.

We refer to the TP framework described in task 1 as *proactive promotion* or *PP*. PP *proactive*ly pulls certain more significant tuples ahead of less significant ones (i.e., *promotes* specific tuples).

Proactive Promotion Contributions:

1). Our promotion continuous query language (P-CQL) supports the specification of multi-tiered monitoring criteria. We propose a continuous query algebra to determine and propagate tuple significance.

2). We analyze the PP optimization problem of choosing which priority determinants to use for precedence determination and where in the pipeline to evaluate each determinant. The PP optimization complexity is shown to be exponential in the number of determinants.

3). We design the rank order pruning PP optimizer which exploits the rank of the tuples (i.e., how some tuples are more significant than others) to prune the search space. We prove that it locates the optimal query plan. Its complexity is polynomial in the number of determinants.

4). Our PP execution infrastructure agilely supports online resource allocation adjustments without requiring any expensive query plan infrastructure changes.

5). Our experimental study shows that PP consistently produces more significant results, up to 18 fold on a wide variety of data sets, compared to the state-of-the-art approaches with minimal run-time overhead.

Task 2 (TP Aggregate Operator): TP requires an aggregate operator that produces non-skewed aggregate results. It must support producing only valid results generated from portions of tuples that belong to an aggregate group. In addition, it must quantify the quality of each possible aggregate result. It should only produce aggregates results that are derived from sets of tuples that most closely represent their actual aggregate group populations. This requires new logic and data structure design to support the generation of aggregate results from subset aggregate group populations bound by significance levels. It also requires the results to be denoted by how representative their subset population is of the actual population.

We refer to the aggregate operator outlined in task 2 as *TP-Ag*. TP-Ag is an aggregate operator designed for any TP framework.

TP Aggregate Operator Contributions:

1). We formulate the TP-Ag operator problem.

2). We enhance our TP-Ag operator to selectively control which subsets in the aggregate group population are used to generate each result.

3). We propose a carefully designed estimation model and application of Cochran's sample size methodology to measure if any subset of the actual population is large enough to generate a reliable aggregate result.

4). We propose a method to track the number of tuples that do not reach the aggregate operator due to limited resources.

5). We outline and propose the physical and logical design of our novel TP-Ag operator.

6). Our experiments demonstrate that TP-Ag produces up to 90% more accurate aggregate results than the state-of-the-art methodologies on a wide variety of data sets and workloads.

Task 3 (Preferential Results): Enhance the TP framework to support queries with more complex functionality such that the most significant and promising tuples are pulled ahead of less significant and promising ones. This requires the formulation of a cost model to determine which significant and promising tuples to pull forward and how to locate such tuples in the PP plan. Different strategies to locate the tuples as well as underlying cost models may be required to support promising tuples. The TP Executor should be extended to efficiently support both user defined and complex significance promotion with minimal overhead at run-time. In addition, the *optimizer* must be enhanced to efficiently locate the optimal PP plans which support the pulling of promising tuples at run-time.

We refer to the TP framework described in task 3 as *Preferential Result* (or *PR*). Task 1 (Ch. 4) presents a TP infrastructure to solve a simple subproblem of priority-based processing, namely, it introduces an infrastructure for processing tuples identified by static criteria only. PR explores the more general and complex problem of optimizing the processing of significant tuples also identified by dynamic criteria.

Preferential Results Contributions:

1). Our PR Executor supports custom algebraic operators that propagate and assign preference related meta data to tuples. Our algebra supports at runtime the online adjustment of which tuples are significant or promising throughout the query pipeline.

2). We formally define the PR optimization problem of generating a PR plan which involves discovering criteria to identify promising tuples, selecting which criteria to use to pull particular significant, and/or promising tuples forward, and deciding where in the plan to evaluate selected each criteria. The PR optimization search time complexity is shown to be exponential in the number of both dynamic criteria and the join operators that promising tuples are pulled forward for.

3). Our PR Optimizer design for criteria placement provides an efficient optimization algorithm that prunes the query plan search space. First, it utilizes a statistics reduction methodology to eliminate inferior statistics used to find criteria to locate promising tuples online. Second, it reduces the number join operators that pull promising tuples by combining the needs of multiple consecutive join operators. The complexity of PR-Prune is shown to be significantly less than exhaustive PR optimization.

4). Our PR Adaptor changes the PR query plan to support online resource allocation adjustments. It is aided by our PR execution infrastructure that allows the PR plan to adapt without requiring any expensive query plan infrastructure changes.

5). Our experimental study, using real data, synthetic data sets, and a wide variety of queries, shows that PR consistently produces between 1.3 to 23 fold more more significant results than the state-of-the-art systems.

Task 4 (Preferential Result Join Operator): PR requires a join operator that utilizes the available CPU resources to maximize the production of significant results in precedence order. The join operator must have special policies and data structure design to achieve effective selection of which tuples to process based upon their significance. It must also support the production of some (not all) join results from a tuple. That is, it must allow the production of some results to be purposely delayed while others are pushed forward. It must only create proper join results and not create any duplicate join results.

We refer to the join operator addressed in task 4 as *PR-Join*. PR-Join is a join operator designed for PR.

Preferential Result Join Operator Contributions:

1). We analyze the limitation of current join operators that do not produce join results in significance order and formulate the result significance-aware preferential result join operator problem.

2). We develop the foundation of how to support atomic and non-atomic join result production, and propose policies that optimally select at run-time when to use them.

3). We propose an innovative PR-Join operator design that is shown to efficiently support non-atomic join result production. Our PR-Join operator design is comprehensive. It includes both infrastructure as well as logic design.

4). We prove that Our PR-Join only produces correct results and never produces duplicate results.

5). Our experiments demonstrate that PR-Join produces up to 190 fold more significant query results than the state-of-the-art methodologies on a wide variety of data sets, workloads, and queries.

1.9 Dissertation Organization

The rest of this dissertation is organized as follows. Related work is outlined in Chapter 2. Basic background of this dissertation is provided in Chapter 3. Chapter 4 introduces the Proactive Promotion model (Task 1). The background and design for a TP Aggregate Operator is discussed in Chapter 5 (Task 2). The Preferential Result methodology for the online identification of dynamic criteria for priority determination (Task 3) is proposed in Chapter 6. The Preferential Result Join Operator (Task 4) is presented in Chapter 7. The dissertation work conclusion is covered in Chapter 8. Finally, Chapter 9 contains ideas for future work.

Chapter 2

Related Work

2.1 Data Streams

2.1.1 Tuple Level Resource Reduction

There are many resource allocation approaches that *reduce the workload*. One approach is load shedding. Load shedding drops less significant tuples. It only allocates resources to the tuples not dropped. Once a tuple is chosen to be processed, it will not be shed at any point along the query pipeline.

Aurora [ACc$^+$03a, ZSC$^+$03] is a system to manage data streams for monitoring applications. It supports real-time requirements. To achieve this, they proposed using load shedding to reduce the system of less critical tuples. Their key idea was to propose load shedding as a means to control the workload.

Tatbul et al. [TÇZ$^+$03] explored a technique for dynamically inserting and removing drop operators into query plans as required by the current load. They considered both semantic and random shedding. Their cost model does not consider the cost of the drop operators to evaluate tuples. It assumes that this cost is low.

Reiss et al. [RH05] proposed the Data Triage architecture. It supports systems with bursty arrival rates that can fluctuate. During such bursts, Data Triage captures an estimate of the query results that the system did not have time to compute. They combined these results with the query results to generate more accurate statistics. These statistics are used to evaluate which tuples should be shed.

Tatbul et al. [TÇZ07] proposed load shedding techniques for distributed stream processing environments. They modeled the distributed load shedding problem as a linear optimization problem. They proposed a distributed approach. It was built for dynamic environments in large-scale deployments.

Nehme et al. [NR07] proposed a load shedding technique for spatio-temporal stream data. Their load shedding model considered spatio-temporal properties by grouping similarly moving objects into clusters. Then they shed selective objects within each cluster. The locations of the objects shed are approximated based upon their associated clusters.

Wang et al. [WQL$^+$10] proposed a load shedding technique for real-time data stream applications. The goal of their approach is to reduce the workload while at the same time preserving the system timing constraints. They proposed different modes. These modes define how the load on the stream is adjusted.

Ma et al. [MZS10] proposed a semantic load shedding technique for real-time data stream applications that utilizes a priority table. It considers both the execution costs and tuple attribute values when deciding which tuples are shed.

Basaran et al. [BKZS12] proposed a load shedding method that applies distributed fuzzy logic. It considers the per-stream backlog and selectivity of each query operator. Their approach is event-driven. This allows it to react to bursty workloads.

Lin et al. [LQJQ12] proposed a linear programming based load shedding method for distributed data stream processing systems. It models the system load as a simple query network with network constraints. It considers two factors. These factors are the amount of available CPU and network resources.

Labrinidis et al. [TNP13] proposed a load shedding strategy that manages the load shedding without requiring any input from users, namely, any manually tuned parameters. Their approach works with complex query networks containing joins, aggregations or shared operators.

In contrast to these approaches, TP seeks to adaptively adjust how resource allocation throughout the query pipeline. These approaches simply decide to process a tuple or not and never revisit this decision. In TP, a tuple may be allocated resources for a portion of the query pipeline. Later on, if more significant tuples are present then this same tuple may be denied resources. This allows the more significant tuples to be processed.

2.1.2 Tuple Level Resource Reorder

Another resource allocation approach *reorders the workload*. Spilling temporarily moves less significant tuples to memory to be processed later and allocates resources to the tuples not spilled.

Spilling XJoin [UF00] is a non-blocking join operator that adapts to data arrival rates. It has a small memory footprint. It's goal is to limit the number of tuples in the workload. To achieve this, it temporarily pushes tuples from portions of the query to disk when the memory is exhausted.

Hash-Merge Join [MLA04] is also a new non-blocking join algorithm that adapts to data arrival rates. It's goal is also to produce results. To achieve this, it utilizes a policy that adaptively flushes certain tuples stored in-memory for a given operator to disk storage when the memory is exhausted.

Liu et al. [LZR06] considered how spilling tuples in one operator affects the other operators in the same plan. They introduced several state spill strategies. The bottom-up state spill strategy treats all data in one operator state equally. The partition-level data spill strategies consider different characteristics of the input data such as the local and global output.

In contrast to these approaches, TP seeks to adaptively reorder tuples throughout the query pipeline. These approaches simply reorder tuples at a given operator in the query plan. In TP, the order in which tuples are processed may change throughout the query pipeline. This ensures that the more significant tuples to be pulled forward so that the most significant results can be produced.

Locally reorder the workload There are also resource allocation approaches that *locally reorder the workload.*

Gedik et al. [GWYL05] introduced a load shedding approach based upon a localized decisions of which tuples are best to perform join operations on. These decisions are made locally at each individual join operator. Their approach performs load shedding based upon three different changes in the system that affect which and how many join results are produced. These areas are the input stream rates, the time correlation between the streams, and the join direction (or which stream is pushing out the join results).

Fernandez-Moctezuma et al. [FMTL09] introduced a load shedding approach that makes localized decisions about how to allocate resources. It identifies subsets of interest in the data stream between neighboring operators in the query pipeline and allocates resources to these subsets of interest. It utilizes punctuations to communicate information about the subsets of interest between operators.

In contrast to these approaches, TP requires centralized decisions to be made about how it is best to allocate resources at each operator in the query pipeline. TP is concerned about the throughput of significant results across the entire query. These approaches are focused on the throughput of significant results at individual operators.

2.1.3 Join Operators that support Tuple Level Resource Reduction and Reorder Systems

The current join operator designs in the literature all process tuples using the *atomic result production* approach introduced by the symmetric binary hash join [WA91]. In the atomic result production approach, all join results that can be generated from a given tuple are produced when that tuple is processed.

A symmetric binary hash join [WA91] maintains a state for each input stream. When it processes a tuple, first it stores the tuple in the state associated with the tuples input stream. Next, results are created by joining the tuple with all its join partner tuples in the state of the other input stream.

The MJoin operator [VNB03] joins tuples from input stream s_x with other streams by determining via statistics the best order of streams which the tuples from input stream s_x should probe against. Then a symmetric binary join algorithm is used to produce results. This is orthogonal to our effort. We could also apply a MJoin approach in a TP system.

The STeM operator [RDH03] is a join operators used in adaptive query processing techniques. It is not concerned with significance nor load shedding. It allow the system to support adaptive query processing where tuples are processed by different orders of operators. The goal of adaptive query processing is to maximize the overall throughput. SteM is a semi-join operator. It joins incoming tuples to a single stream state.

Kang et al. [KNV03] proposed a cost model that considers the cost of performing the join from each of the incoming streams to the join operator individually. Based upon this cost model, they propose strategies for maximizing the efficiency of processing joins in three scenarios. These scenarios are where one stream faster than the other, when the system can not process all tuples from the input streams, and when memory is limited.

The STAIRS operator [DH04] was also built to support adaptive query processing techniques. It is also not concerned with significance nor load shedding. STAIRS, an extension to the ripple join operator, allows the system to dynamically adjust the intermediate tuples stored in the cache to optimally process tuples based upon changes in the statistics and/or performance of operators in the query plan. It allows certain tuples to undo some join operations that were performed on them. The goal is to unblock these tuples so that they can perform other join operations.

Gedik et al. [GWYL07] extended join operators to makes localized decisions about how to allocate resources to process the most profitable segments of the join windows. It uses knowledge of the time correlation of join partners in a MJoin operator to optimize the stream join order amongst multiple streams.

Dong-Hong et al. [HXZ$^+$06, HWXZ07] explored several load shedding techniques to benefit queries containing a single sliding window join. They developed efficient methods to collect statistics on each stream and a load shedding strategy whose goal is to maximize the join results produced. They proposed strategies to address when streams have high and/or variable arrival rates.

Law et al. [LZ07] explored load shedding strategies for multi-way joins. Their approach utilized a sketching-based technique to estimate the contribution that each tuple has on the number of results produced. The estimated contribution is used when determining which tuples will be processed and not shed.

Lin et al. [LL07] presented an adaptive load shedding approach for windowed stream joins. When resources are scarce, they do not drop tuples from the input streams. They instead shed tuples from the join state. They propose different methods to adjust to three possible run-time adaptions. These run-time adaptions are the input stream rates, time correlation between the streams, and join directions.

Ren et al. [RJH07] presented a load shedding technique for sliding window joins based upon clustering-based indexing. They proposed to collect statistics on the indices for each sliding window. The statistics are stored using a clustering technique. These clustered statistics are used when determining which tuples will be processed (or shed) to maximize the number of join results produced.

Ma et al. [MLZ$^+$09] proposed a load shedding strategy to support shared window joins in multi-queries stream processing systems. Their load shedding approach reduces overhead by adjusting the size of the sliding windows. Their approach tries to find the common subset of tuples required by the majority (if not all) the queries.

Kwon et al. [KLK11] also proposed a load shedding strategy to support shared window joins in multi-queries stream processing systems. Their approach determines which tuple will be processed (or shed) based upon the join attribute values. For a given join attribute value, the decision made is based on the streams where there exists tuples that contain the join attribute value.

TP requires a join operator that can support the production of some (not all) join results from a tuple. That is, it must allow the production of some results to be purposely delayed while others are pushed forward.

2.1.4 Aggregate Operators that support Tuple Level Resource Reduction and Reorder Systems

Some *aggregation operators* proposed to support data stream systems that utilize tuple level resource allocation and reduction aim to only produce non-skewed aggregate results (Sec. 1.7.3) by requiring that certain tuples from selective windows are never shed. This is limiting in what tuples will and will not be processed. It does not address the TP systems where the user selects which tuples will and will not be processed. These approaches simplify aggregation because they force a complete set of tuples from these windows to arrive at the aggregate operator.

Hellerstein et al. [HHW97] proposed an online interface that allows users to both observe the progress and halt the execution of their aggregation queries. In their approach, load shedding is initiated by the end user. To help ensure the most accurate aggregate results are produced, their approach returns the output in random order, adjusts the rate at which different aggregates are computed, and computes running confidence intervals. The running confidence intervals are displayed to the user.

Olston et al. [OW00] proposed a system that creates an aggregate result from cached results and the actual data set with the goal of creating an aggregate result within a high confidence interval range as quickly as possible. They propose a methodology to determine which results to cache that considers the trade off between precision and performance. Their algorithm delivers an answer that is bound by a specified precision constraint.

Babcock et al. [BDM04] proposed a system that supports load shedding. The goal of the system is drop tuples such that accuracy of the aggregate results produced are within certain limits. They consider the probability that dropping certain tuples has on the accuracy of query answers produced by the multiple queries.

Longbo et al. [LZZM07] propose a load shedding system for continuous sliding window join-aggregation queries over data streams. Their load shedding strategy partitions the domain of the join attribute into certain sub-domains. Then they filter out selected input tuples based on their join values.

Guo et al. [GH09] proposed a load shedding approach for aggregation queries with sliding windows. They analyzed the characteristics of subset model and deficiencies of current load shedding methods. Their load shedding algorithm is based on the strategy of dropping tuples from certain window.

Senthamilarasu et al. [SH12] proposed load shedding techniques for queries consisting of one or more aggregate operators with sliding windows. Their load shedding method utilizes a window function that divides the input into portions of the windows of the aggregate operators. It then utilizes this function to probabilistically determine which tuple to shed.

Akin to these approaches, any aggregate operator in a TP system must also contend with the time versus accuracy trade off. TP requires an aggregate operator that can creates a reliable aggregate result using only the available tuples within the group population. The approach should not adjust how the TP system is allocating resources. It should not change which tuples are pulled forward.

2.1.5 Prioritization of Results

The current prioritization techniques can be classified by the purpose of the prioritzation decisions, specifically either for the purpose of 1) ordering or 2) restricting the results.

Ordering the Results There are many efforts that introduce methods whose focus is to *order the results produced*.

Raman et al. [RRH99] proposed a non-blocking mechanism for reordering tuples over data streams. Their reordering policies are based on different performance goals.

The *Juggle operator* [RH02], built for exploratory queries, allows the users to adapt the order in which results are generated. Users can specify which specific rows or columns should be produced first. They proposed a query processing architecture that generates partial results quickly and adapts to changing user interests.

Jacobi et al. [JBG$^+$10] proposed an out-of-order execution of tuples in the data stream. They approach reorders tuples in the data stream system per user preferences. Their system produces all results. The prioritized results are simply produced earlier than the insignificant results.

These approaches produce all results regardless of how long it takes to produce them. They do not restrict the results produced by any response time limit. As a result, results may take as long as they need to be produced and resources are allocated to producing all results. TP systems require that results be produced within a response time limit.

Restricting the Results There are many efforts that introduce methods whose focus is to *restrict which results are produced*. They can be categorized into three different areas. These areas are *Top k*, *web search*, and *preference queries*.

These approaches limit the number of results produced by ranking tuples based upon the user preferences. They return the most preferred results, often a fixed small cardinality. For instance, they may only return the top most results that will fit on a screen. They do not concern themselves with the overhead of ranking the tuples, nor how this affects the latency of the results produced.

Top K Mouratidis et al. [MBP06] proposed top-k query methodology that supports multi data stream queries. Their approach explores methods to store the most recent data in main memory. The goal is to produce rapid online results. To achieve this, they adjust when the system returns computed or partially precomputed results.

Ilyas et al. [IBS08] provided a survey where they described top-k processing techniques in relational databases. They outlined and analyzed the current techniques by their query model, data access method, implementation, support of certainty, and scoring functions provided.

Shen et al. [SCL$^+$12] proposed a top-k framework which supports multiple top-k queries. They allow each query to define their own scoring function and sliding window. They support complex scoring functions and out-of-order data streams.

Web Search Teevan et al. [TMB09] proposed methods that utilize information about the end-user to identify the most relevant results for that person. They group people using different criteria. Then they explored the similarity of query selection and how relevant they find the results produced across people in each group.

Durao et al. [DLDC12] explored different methods to determine a users result generation preferences. They utilized the tagging activity of the user. Their user preference model considers multiple factors.

Kramar et al. [KBB13] proposed a method that adds additional keywords to end-user search queries. It utilized social network knowledge generated from the user's activity on the Web.

Preference Queries Chomicki [Cho07] proposed two properties of the semantic optimization of preference queries. The first property concerns validating preference relations against integrity constraints. The second property concerns the order in which axioms are satisfied in integrity constraints. There properties are applied to the evaluation and optimization of preference queries.

Roocks et al. [REH$^+$12] proposed a context-aware preference query framework. It supports end-user input, and domain-specific and contextual knowledge input. They introduced methods to optimize and generate preference queries. using these inputs

TP has a novel prioritization goal, namely, producing the *most significant results given the resources available*. When resources are sufficient, TP aims to produce all results. When resources are scare, TP aims to produce as many of the most significant results as possible. TP considers the overhead of ranking the tuples and how it affects the latency of the results produced.

2.1.6 Operator Scheduling

Broadly, operator scheduling methods determine which operator to run and for how long to improve different metrics.

Single Query There are many efforts that introduce methods whose focus is to improve the metrics of a single query.

Urhan et al. [UF01] proposed policies for pipelined query plans to produce fast results. Their rate based scheduling policy is based upon a response time metric for a query. They also proposed a scheduling policy to support Top-K queries where certain results may be more important than others. This policy dynamically regulates which tuples are processed.

Aurora [ACc+03a, ZSC+03], the first system to propose load shedding, used a Round Robin method to schedule the order in which each query is executed. They also proposed a scheduling policy based upon the average tuple latency metric to determine the order in which operators within a query are executed.

Sharaf et al. [SLCP05] proposed a scheduling policy for continuous queries. The objective is to maximize the number of most current results produced. Their scheduling policy decides the execution order of continuous queries. It considers query properties and the variability of new data in the input streams.

Tick scheduling [OYY+05] proposed a deadline-scheduling strategy for continuous queries. The goal is to minimize the latency of each individual tuple. Their strategy utilizes a batch scheduling plan to reduce the overhead.

Multiple Queries Similarly, there are many efforts that introduce methods whose focus is to improve the metrics across multiple queries.

Hammad et al. [HFAE03] proposed a multi-query scheduling method that uses a metric based upon reducing the average response time per query. Their approach prioritized shared join operators (i.e., join operators that process tuples for more than one query) by adapting the window constraints.

Babcock et al. [BBMD03] proposed an adaptive, load-aware, multi-query scheduling method for data stream systems. Their approach optimizes the order of selections, projection operators in single-stream queries. It is driven by a memory usage metric.

Babcock et al. [BBD$^+$04] extended their multi-query scheduling method for data stream systems. Their enhanced approach utilized a combined memory usage and response time metric. They prove that this approach is near-optimal in minimizing runtime memory usage.

Sharaf et al. [SCLP06] proposed a multi-query scheduling policy for data stream systems that balances the needs for performance with the fairness needs of multiple queries. They demonstrated that optimizing for average response time across queries is a distinct new metric.

Deng et al. [DLX07] proposed a multi-query scheduling policy for data stream systems that seeks to minimize the memory overhead and output latency. It achieves this by estimating the future workload. Their approach utilizes a scoring function to determine the execution order of the operators.

Wu et al. [WTZ09] introduced a multi-query scheduling strategy that recasts the problem into a job scheduling problem. It utilizes input from the end-users to measure the QOS metric.

Wang et al. [WGM10] extended the Chain scheduling to support complex directed acyclic graph query plans. Their approach enhances stream tuples with scheduling meta-data.

Falt et al. [FY11] introduced a multi-query scheduling strategy for parallel processing systems. They explored the scheduling of complete tasks. Their approach utilizes metrics based upon hardware statistics of each node in the system.

Adapting the Scheduling Method Sutherland et al. [SPZ$^+$05] proposed an adaptive scheduling policy for data stream systems. Their approach adapts the scheduling algorithm used online to the current statistics. Their approach can support meeting multiple metric objectives.

Ghalambor et al. [GSA09] proposed an adaptive scheduling policy for data stream systems. Their approach located the best scheduling method to use for different scenarios. They periodically select a scheduler based upon the current system statistics to support multiple metric objectives.

Mohammadi et al. [MSAH11] proposed an adaptive scheduling policy for data stream systems that utilized machine learning. Their system adapts parameters to improve the output latency from knowledge gained about the system.

Real Time Deadlines Metrics Other efforts use *real time deadlines metrics* to schedule operators.

Schmidt et al. [SLSL05] proposed a real-time scheduling strategy that seeks to improve the quality of service of multiple queries. In particular, they consider that sections of the query pipeline may be shared by queries and how to meet the constraints of multiple queries.

Wei et al. [WSS06] proposed a real-time data stream system. It utilized an earliest deadline first strategy to schedule the periodical queries. They proposed periodic query semantics for real-time data stream systems.

Ma et al. [MLWW09] proposed scheduling strategies for real-time data stream systems. They modeled the progress of one tuple as a real-time task instance. Their approach schedules the operator path with the earliest deadline of the tuples in the input queue.

In contrast to these approaches, TP seeks to select the order in which tuples are processed and control the amount of CPU resources dedicated to processing tuples based upon how critical they are throughout the query pipeline. Traditional operator scheduling approaches make coarse grained decisions on how to utilize the CPU resources available. TP requires fine grained decisions on how to utilize the CPU resources available.

2.1.7 Query Optimization

Query optimization considers how to adjust how the query is processed to improve a given metric. Query optimization techniques can be classified by what the optimization decisions are based upon; optimizing for 1) individual tuples, 2) groups of tuples, or 3) all tuples.

Individual Tuples Query optimization techniques for individual tuples consider which operator each tuple should be processed by next.

Eddies [AH00] optimizes query processing by adapting the query execution plans for each individual tuple processed. They use a router to adaptively send tuples to individual operators in a query based on localized heuristics. It was the first to propose to use a router to adapt the order of operators in which individual tuples are processed.

Madden et al. [MSHR02] proposed a system that utilized the Eddies approach over multi-queries. They propose sharing of work. To support this, they implemented special query operators.

Tian et al. [TD03] proposed routing policies for the Eddies system executing a distributed stream query plan. They monitor performance by evaluating the response time and system throughput. They also evaluated how the routing policy effects the system throughput.

Groups of Tuples Query optimization techniques for groups of tuples consider the order of operators that each tuple in a selected group should be processed by.

CBR [BBDW05] adapt query execution plans to optimize query processing for groups of tuples. It is a specialized version of Eddies. It adaptively routes tuples through individual operators in a query based on the data content of the tuples.

QMesh [NWRB09, NWL$^+$12] groups incoming tuples by characteristics and routes each group along a predefined path determined to be optimal for tuples with such characteristics. The main idea of QMesh is to compute multiple routes (i.e., query plans). Each path is designed for a particular subset of the data with distinct statistical properties. They define a classifier model. It assigns the best route to process incoming tuples based upon their data characteristics.

Nehme et al. [NRB09] introduced an adaptive version of QMesh which collects current run-time statistics. Their approach is to identify concept drifts. These drifts indicate when a change in the environment has occurred. Their execution infrastructure is easily adaptable. It only requires changes in the classifier operator.

All Tuples Query optimization techniques for all tuples consider the order of operators that all tuples should be processed by.

Kabra et al. [KD98] proposed a method that detects the sub-optimality of a query plan at run-time. It then attempts to correct the problem by adapting how the resources are allocated to operators in that query plan or by changing the query plan itself.

Stillger et al. [SLMK01] proposed comparing the optimizer's estimates with the actual cost values. From this comparison, it computes adjustments to cost estimates and statistics. These adjustments are used during future query optimizations. This allows the optimizer to learn whether or not their estimates are correct from the system.

Golab et al. [GÖ03] classified continuous queries into four types of update characteristics. They propose the best implementations of query operators based upon these four types of update characteristics. Their implementations include the best data structures to store tuples in the state.

Babu et al. [BMM$^+$04] proposed an optimized approach to determine the optimal order of filters in a continuous query. The goal is to minimize the processing cost. They propose a greedy approach that is proven to converge to an ordering within a small constant factor of optimal.

Babcock et. al. [BC05] proposed using the probability distribution of the approach selected. Their optimizer selects the appropriate query plan after considering the relative importance of predictability vs. performance preference of the user. Prior to optimization, the user selects the trade-off between the two goals of predictability and performance (which could be at odds sometimes) to find the appropriate query plan.

Safaei et al. [SH10] proposed a parallel processing system for continuous queries over data streams. Their optimizer models the query as a weighted graph. It utilizes the finding the shortest path in a weighted graph problem to determine how to best allocate resources to the operators.

Guirado et al. [GRR13] proposed an optimizer for parallel processing database systems. Their optimizer exploits tasks shared by multiple query tasks. It also supports the replication of multiple instances of data. This allows tasks that share the same data to be run in parallel.

Balkesen et al. [BTO13] also proposed an optimizer for parallel processing database systems. They introduced a framework that decides how to adjust the distribution of which nodes process the incoming streams. The goal is to minimize the size of the cluster and balance the load across the nodes.

TP seeks to adjust the order in which tuples are processed not the order in which operators are executed within the query. The query optimization approach is orthogonal to our effort. TP will work with any query optimization approach.

2.1.8 Monitoring Load Changes

Monitoring load changes has been studied in the literature.

Schlimmer et al. [SG86] proposed methods that are able to tolerate noise (less than perfect feedback) and drift (concepts that change over time) in data over time. They use these to predict the expected outcome by considering the noise and drift.

Haas et al. [HNS94] proposed to use sampling techniques to estimate the selectivity and cost of a join operator. They showed that the bounds on the cost of sampling depends on the size of the input relations, the number of input relations, and the precision required. They covered how skewed data effects the cost of sampling.

Poosala et al. [PI97] proposed two approaches that use histograms to estimate the selectivity and cost of a join operator. The first approach uses a multi-dimensional histogram. The second approach uses the Singular Value Decomposition (SVD) technique from linear algebra.

Wei et al. [WPSS06] proposed a Quality-of-Service management scheme for periodic queries in data streams systems. Their method utilizes query workload estimators. These estimators use profiling and sampling to predict the future query workload.

TP requires that the workload of critical tuples (i.e., the estimated number of incoming tuples that belong will produce a significant result) in each operator to be measured. TP utilizes the sampling-based approach to estimate the data stream query workloads proposed by [WPSS06]. Other sampling methods including decaying the importance of older statistics [SG86] could also be applied.

2.1.9 Commercial Stream Systems

Database companies have developed data stream processing systems.

Microsoft SQL Server StreamInsight [Stra] supports building data processing over real-time event streams. It can handle high volumes of event stream data.

Oracle Event Processing (OEP) [Ora] supports the processing of tuples in real-time. It is designed for service oriented and/or event-driven applications that need real-time intelligence.

IBM System S [IBM] is a streaming architecture. It can deliver nearly instantaneous results. It supports the creation of incomplete results from partial knowledge. More refined results can be created as more data is processed.

Coral8 [Cor] develops event processing software. They have many products. One of their products, Engine, is a platform for developing, deploying, and scaling complex event processing applications. Studio, another product, is a CEP application development environment. The company has a strategic partnership with IBM.

StreamBased Systems [Strb] develops high performance Complex Event Processing (CEP) applications and software that analyzes real-time streaming data.

Esper [Esp] is a complex event processing component for Java. It can support real-time stream processing. It enables more efficient development of stream processing applications.

2.2 Publish/Subscribe Systems

In publish/subscribe systems there are *publishers* and *subscribers*. The publishers post messages. They do not know which subscribers (if any) can read their messages. Each message is classified into different classes. Subscribers request to receive messages from certain classes.

Banavar et al. [BCM$^+$99] proposed a content-based publish/subscribe systems. It allows subscribers to request messages based on the content of published messages. They proposed a link matching algorithm that determines which messages should be forwarded.

Fabret et al. [FJL$^+$01] proposed a novel data structures and implementation to support high performance publish/subscribe systems. Their main memory algorithm filters the content of messages. It utilizes a clustering scheme to minimize the number of searches over the possible subscribers.

Campailla et al. [CCC$^+$01] proposed an efficient algorithm to filter incoming messages in high performance publish/subscribe systems. Their approach scales the published messages with the subscriptions. Their model uses binary decision diagrams.

Tam et al. [TAJ04] proposed a distributed content-based publish/subscribe system. Their approach is based upon distributed hash tables systems. These systems have been effectively used in large peer-to-peer networks.

Carvalho et al. [CAR05] proposed a scalable QoS-aware publish/subscribe system. Their model decentralizes the message-broker based upon underlying network-level QoS reservation mechanisms. To improve efficiency, they replicate the information for each message classification.

Moro et al. [MBT07] proposed a filtering approach for XML aware publish/subscribe systems. Their model is based upon a tree-like indexing structure that organizes the queries based on their similarity. They provided lower and upper bound estimations on when pruning is required.

Patel et al. [PRGK09] proposed a a peer-to-peer feed-based publish-subscribe service. To create a ”‘fair system”’, they utilize methods to relayed the messages to true subscribers of that class of messages. They also take into consideration the characteristics of subsets in the population of subscribers.

Cheung et al. [CJ10] proposed a load balancing scheme for distributed publish/subscribe system. Their load balancing framework contains detection and mediation mechanisms at the local and global load balancing levels. They propose three offload algorithms. Each is designed to load balance on a particular performance metric.

Zhang et al. [ZMJ12] proposed a total ordering publish/subscribe system.

Their approach incorporates total ordering into the publish-subscribe logic instead of treating it as a separate function. Their solution is fail-safe. It gracefully handles conditions where it is not possible to support total order.

Vavassori et al. [VSL$^+$13] presented a context-aware publish-subscribe model. Their approach decouples each subscription from the changing context in which it is produced. It then filters the contextual subscription bounds to reduces the subscription cost per class.

In contrast to publish/subscribe systems, TP queries define the attributes and criteria of the most critical results that should be produced. In addition, TP queries also define which subset of tuples in the incoming streams should be processed when resources are scarce.

2.3 Networking Packet Priority Scheduling and Queuing

Prior data stream system problems have been mapped to networking problems (e.g., *Eddies* [AH00]). Similar to the TP problem can be mapped to network packet priority scheduling and queuing research.

Lin et al. [LM97] proposed a network routing protocol for nonadaptive, fragile, and robust network traffic. It tracks the number of packets per flow. This is used to impose a loss rate on each flow that is based upon the flow's buffer use. They demonstrate that their approach is able to isolate non-adaptive greedy traffic more effectively.

Lakshman et al. [LS98] proposed a packet classification schemes. It classifies packets by using a range match on more than 4 packet header fields. Their algorithm supports two classification dimensions. Their approach is applicable to many areas, including security policy enforcement, resource management decisions, and routing.

Paris et al. [PJS99] proposed an active queue management policy to protect TCP from all UDP flows. It also ensures reasonable throughput and latency for well-behaved UDP flows and provides congestion avoidance benefits and protection for TCP.

Wang et al. [WSS01] proposed a packet-scheduling protocol to support a premium service in the differentiated service architecture. It adds a weight to each packet. Packets are scheduled based upon their weight. To adjust which packets are scheduled, weights on packets are adaptively adjusted.

Kumar et al. [KMPS04] proposed a packet-scheduling protocol for wireless networks. The focus of their work is to develop fully-distributed (decentralized) protocols. They modeled the network as a graph and proposed approximation and near-optimal approximation algorithms.

Ashour et al. [ALN08] proposed a multi-queue infrastructure for network traffic. They provided a cost model that estimates the queue length and delay survivor functions for a priority queuing system with varying service rate. They outlined a method to support variable service of packets using the multi-queue infrastructure.

Karim et al. [KNTA12] proposed a priority scheduling protocol for packets. It is a three-class priority packet scheduling tiered- scheme. The highest tier contains emergency real-time packets. They can preempt all other packets. The other two tiers are prioritized based upon the sensor nodes locale. They address starvation by allowing the lowest priority packets to preempt other packets after a certain period of time has elapsed.

Yang et al. [YU12] proposed a packet scheduling protocol that adaptively changes the transmission rate to adjust to the traffic load and available energy. At the same time, they seek to minimize the transition time of all packets. They developed optimal off-line scheduling policies that minimize the the transition time of all packets.

Jiang [JJ12] proposed a differentiated queuing service method to support quality of service requirements for different applications. They utilize a controller that queues packets according to their qos requirements. The controller can also downgrade packets who are perceived to not be able to meet their qod requirements to reduce network resource waste.

Similar to TP, these approaches also pull significant objects forward. However, query processing adds additional complexities not seen in networking. First, incoming tuples in data stream systems do not know the last operator (akin to the packet destination) that they will be processed by. Rather, they could be filtered out at any operator in the pipeline. Second, the amount of work required to complete processing a single tuple in transit increases or decreases depending on if the tuple produces multiple join results or if the tuple gets filtered out and produces few results.

2.4 Operating Systems Scheduling and Queuing

Operating system scheduling and queuing consider how to adjust the order in which the tasks are processed to improve a given metric.

Davis et al. [DTB93] addressed the problem of jointly scheduling tasks with both hard and soft time constraints. They proposed the foundations for synchronization, release jitter and stochastic execution times. These are the basis for slack stealing algorithms.

Zhang et al. [ZHC02] proposed a two-phase task scheduling method that considers both the task as well as the voltage required to execute the task. The goal is to minimize the energy consumption of real-time dependent tasks executing on a given number of variable voltage processors. Their approach orders the task assignment to maximize the opportunities to reduce the voltage overhead.

Goossens et al. [GFB03] proposed a task scheduling method for periodic tasks upon multiprocessor platforms. Their priority-driven algorithm schedules periodic task systems upon multiple platforms. It is based upon the total utilization of the system and the maximum utilization of any specific task.

Abdelzaher et al. [ASL04] proposed a task scheduling method for aperiodic tasks. They derived a bound for schedulability of aperiodic tasks. It is based upon a metric that measures the amount of available schedulability overtime.

Davis et al. [DB05] explored a scheduling systems on a single processor where there is a hierarchical priority of tasks. Their approach addressed fixed priority pre-emptive scheduling at both global and local levels.

Kim et al. [KLJ$^+$09] proposed a task scheduling method for virtual machines. It utilizes inference techniques. Their approach estimates the I/O-bounds of certain tasks. Their scheduler selectively chooses tasks to executed with the goal of ensuring that the tasks incoming events are promptly processed.

De et al. [DNLR09] proposed a task scheduling method for embedded systems. Their approach utilizes a protection scheme. It prevents lower priority tasks from interfering with higher priority tasks. Their algorithm works offline and reduces the time low priority tasks are preempted by high priority ones.

Lakshmanan et al. [LKR10] proposed scheduling methods of periodic real-time tasks on multiprocessors under the fork join structure used in OpenMP. They outlined the best and worse case scenarios. They proposed an efficient multiprocessor scheduling algorithm.

Dubey et al. [SD13] proposed a task scheduling method for Cloud Computing service operation. It incorporates a task QoS metric. Their approach supports the tasks requested by many users at the same time where each task may have different QoS requirements.

Similar to TP, these approaches also pull significant objects forward. However, in query processing a tuple (i.e., task) may expire and never complete processing. In addition, the amount of work required to complete processing a single tuple in transit increases or decreases depending on if the tuple produces multiple join results or if the tuple gets filtered out and produces few results.

Chapter 3

Background

This chapter defines general background knowledge about data stream management systems, Continuous Query Language (a.k.a CQL) queries, query plans, aggregate and join operators needed for the remainder of this dissertation. It also provides information on CAPE [RDS$^+$04]. CAPE is the DSMS where our proposed TP systems and operators were implemented.

3.1 Data Stream Management System (DSMS)

In the DSMS model, data providers send an unbounded sequence of tuples to be processed by a set of continuous queries $\{q_1, \ldots q_n\}$. Each tuple t_i is a triplet $[SId, A, at]$ where $t_i.SId$, $t_i.A$, and $t_i.at$ are respectively the stream identifier, set of attribute values, and arrival time of t_i.

Consider the stream model example in Figure 3.1. In this example there are three sets of data providers, namely, prisoners, police officers, and patrol cars. A stream is dedicated to each data provider. Streams 1, 2, and 3 are respectfully dedicated to prisoners, police officers, and patrol cars. In Figure 3.1 each stream uses a distinct shape to represent their tuples. Streams 1, 2, and 3 are respectfully represented by rectangles, circles, and triangles.

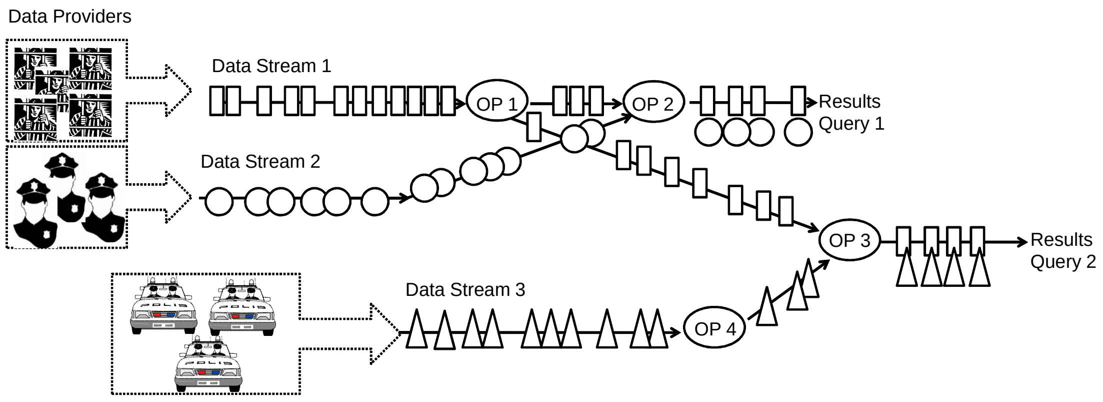


Figure 3.1: Example of a DSMS

3.2 CQL Query

The Continuous Query Language (CQL) [ABW06] supports the specification of continuous queries. The main difference between CQL and Structured Query Language (SQL) (i.e., the standard query language for traditional databases) is the window specification. DSMS can receive an endless amount of data. However, most DSMS are only interested in producing results from the most recent information. A window specifies the finite set of recent tuples from the unbounded stream that are used to create results. To define which tuples belong to the set of recent tuples, users specify window bounds. For instance, a user may specify the window bounds via a start time st and end time et. The user can define window bounds as time-based (e.g., tuples that arrived within the last 30 seconds) or count-based (e.g., the last 30 tuples that arrived).

There are two types of window semantics, sliding window and hopping window. In sliding windows, there is only one window, i.e., the current window. All tuples in the current window compose the set of recent tuples. As tuples arrive from the data providers, they are included in the set of recent tuples. Older tuples are removed from the set of recent tuples when the arrival date of the tuple t_i is beyond the windows start time and end time, i.e., $t_i.at \leq st \leq et$. The set of recent tuples evolves and adapts over time as incoming tuples arrive. A tuple t_i belongs to the set of recent tuples if and only if its arrival time stamp is $t_i.at$ is within the windows starts time and ends time, i.e., $st \leq t_i.at \leq et$. Hence, the current window adapts overtime by changing its start and end time to match the specified window bounds. The window bounds slide to include the most recent incoming tuples.

In hopping windows, each window has a set distinct start and end time. The bounds of each window do not overlap, the next window starts at the end time of the preceding window. Hence, multiple windows may exist. As tuples arrive from the data providers, they are added to the set of tuples that compose the window whose bounds contain the tuple's arrival time. A tuple t_i belongs to the set of tuples that compose a window if and only if its arrival time stamp $t_i.at$ is within the windows starts time and ends time, i.e., $st \leq t_i.at \leq et$. When the current time is beyond a windows bounds, it produces any and all results generated by its tuples, i.e., $st \leq et \leq \mathit{Current\ Time}$. Then all the tuples in the set of tuples that compose the window are purged.

3.3 Query Plans

A CQL query [ABW06] q_i can be represented by one of many *query plan*s p_j. Each plan p_j corresponds to a directed acyclic graph composed of operators as nodes and data exchange interfaces as edges. Operators take in set(s) of tuples from data exchange interfaces, produce results by running an algebraic function on the set(s) (e.g., filter or union), and output the resulting tuples to data exchange interfaces. Data exchange interfaces (e.g., queues or stacks) transfer tuples between operators.

Reconsider the example of a stream model in Figure 3.1. In this example there are two queries. Namely, Figure 3.1 contains query 1 and query 2, i.e., q_1 and q_2 respectfully. Query q_1 is represented by query plan p_1. Query plan p_1 is the acyclic graph composed of algebra operator nodes OP_1 and OP_2 and their connecting edges. In Figure 3.1 query q_2 is represented by query plan p_2. Query plan p_2 is the acyclic graph composed of algebra operator nodes OP_1, OP_3, and OP_4 and their connecting edges.

The PP (Ch. 4) and PR (Ch. 6) frameworks support both stateless, blocking stateful, and nonblocking stateful operators. Stateless operators (e.g., select and project) do not need to maintain a state (i.e., a table) of data to produce results. Nonblocking stateful operators (e.g., join, union, and intersection) maintain a state (i.e., a table) of data used to produce results. They do not need to wait until the state contains the entire set of recent tuples in the current window to be able to produce results. Blocking stateful operators (e.g., group-by, aggregate (e.g., max, min, and count), and difference) also maintains a state of data used to produce results. They do need to wait until the state contains the entire set of recent tuples in the current window to be able to produce results. A description of typically used

CQL operators in PP and PR models can be respectively found in Chapters 4 and 6. The two most commonly used stateful operators work are now explained, namely, aggregation (i.e., a blocking stateful operator) and join (i.e., a nonblocking stateful operator).

3.4 Aggregate Operators

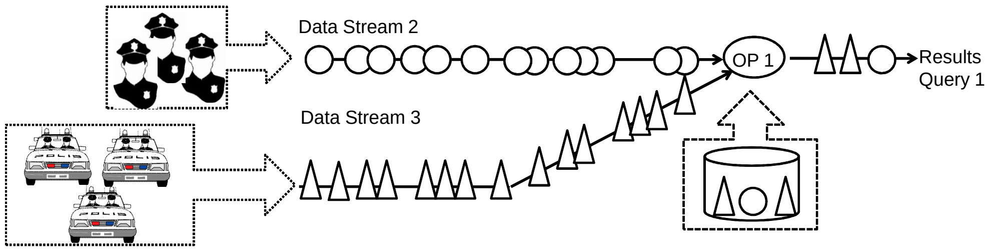


Figure 3.2: Aggregate Operator Example

Aggregate operators maintain a state and incrementally return a value for the given aggregate function. Figure 3.2 is the query plan for a query looking for the officer or police car closest to a location (i.e., minimum distance). Consider the aggregate operator OP_1 in Figure 3.2. OP_1 maintains a state for all incoming tuples. For each incoming tuple t_i, the aggregate operator stores t_i in the state and periodically returns an updated result.

Consider tuple t_i that arrives from a policeman on stream 2 at aggregate operator OP_1 in Figure 3.2. First, the aggregate operator stores tuple t_i in the state. Then when OP_1 is set to return a result, the tuple stored in the state that is closest to the given location will be returned as a result.

Periodically, aggregate operators will purge their states of any stored tuple that are no longer required, e.g., any tuples stored in the state whose arrival time is older than the query window of any future tuples that will arrive at the aggregate operator.

3.5 Join Operators

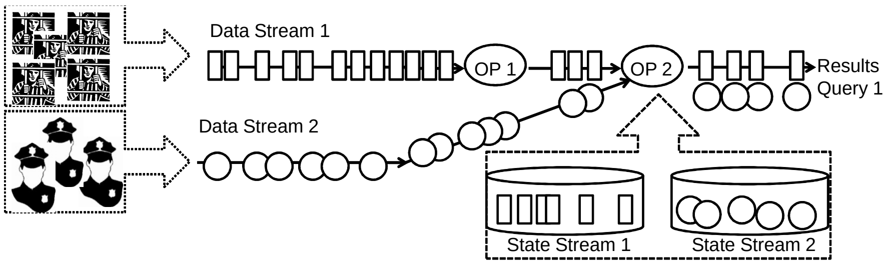


Figure 3.3: Join Operator Example

A symmetric binary hash join [WA91] maintains a state for each input stream that it combines. For example, in Figure 3.3, join operator OP_2 maintains a state for stream 1 as well as a state for stream 2. Each incoming tuple t_i to a join operator is stored in the state according to tuple t_i's stream identifier, i.e., $t_i.SId$. Results are created by joining t_i with tuples in the states of the other streams according to the join and window predicates specified in the CQL query.

A *join predicate* is expressed in the WHERE clause of a CQL query composed of 1) a join expression $(=, <, >, \geq, \leq)$ (e.g., $\leq$ 3 mi), 2) an attribute stream reference (e.g., OL.CurrentLocation), and 3) another attribute stream reference (e.g., OL.CurrentLocation), i.e., Distance(PL.CurrentLocation, OL.CurrentLocation) $\leq$ 3 mi (Home Arrest Query in Section 1.2).

A *window predicate* expressed in the CQL query is composed of 1) a window bound expressed in the WITHIN clause (e.g., 30 seconds), and 2) a window semantic expressed in the Range clause (e.g., slide every 1 second) (Home Arrest Query in Section 1.2).

Figure 3.3 is the query plan for the Home Arrest Query in Section 1.2. Consider tuple t_i that arrives from data provider for a policeman on stream 2 at join operator OP_2 in Figure 3.3. First, tuple t_i will be stored in the state for stream 2. Then results will be created by joining tuple t_i with prisoner tuples in the state for stream 1 such that the distance between officer tuple t_i and any prisoner tuple in the state for stream 1 is ≤ 3 miles and the tuples arrival times are within 30 seconds of each other.

Periodically, join operators will purge their states of any stored tuple that are no longer required. Any tuples stored in the state whose arrival time is older than the query window of any future tuples that will arrive at the join operator.

3.6 The CAPE DSMS

Our proposed TP system and operators were implemented in the same DSMS, namely, CAPE [RDS$^+$04] (Fig. 3.4). To ensure fairness, and all systems we compare against were also implemented in CAPE.

CAPE is composed of many modules. The *Stream Receiver* gathers incoming tuples from external devices and sends them to the *Execution Engine*. The *Execution Engine* executes the query plan by processing the incoming tuples through a set of query operators. The *Operator Scheduler* determines when and how long each operator in the query plan is executed based on information gathered by the *QoS Inspector*. The *Operator Configurator* adjusts how each operator processes tu-

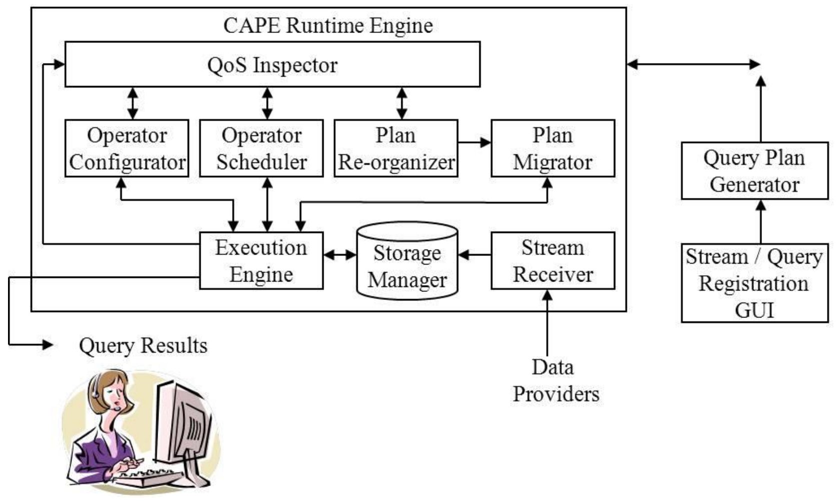


Figure 3.4: CAPE Data Stream Management System

ples based on information gathered by the *QoS Inspector*. The *Plan Re-organizer* selects a new optimal query plan based on information gathered by the *QoS Inspector*. The *Plan Migrator* adapts the current plan to the new optimal query plan selected by the *Plan Re-organizer*. The *Query Plan Generator* creates a query plan from a CQL [ABW06] query.

Our proposed TP systems and operators enhance the *Execution Engine* module of CAPE and add additional optimization component modules to CAPE. CAPE was developed in JAVA. Our enhancements to the *Execution Engine* module of CAPE and add additional optimization component modules were also developed in JAVA.

Bottlenecks can occur in CAPE when more tuples arrive at an operator than the operator can process within the time allocated to it by the *Operator Scheduler*. It can occur at any operator in the query plan. The *QoS Inspector* collects statistics to identify bottlenecks. When a bottleneck is identified, the *Operator Scheduler* will adjust when and how long each operator in the query plan is executed. The *Plan Re-optimizer* may also selects a new optimal query plan.

When bottlenecks occurs and the system is overloaded, state-of-the-art resource allocation methods (i.e, shedding and spilling) respectively drop or temporarily put to disk insignificant tuples. They utilize specialized operators in the *Execution Engine* whose sole purpose is to shed or spill specified tuples. They also have additional optimization component modules that identify when shedding or spilling needs to occur and which tuples must be shed or spilled. However, these approaches do not halt the most critical tuples from expiring (Sec. 1.7.1). Our proposed TP systems collects statistics to identify such situations and adapt how resources are allocated in the query plan to ensure that the most significant tuples do not expire. Our enhancements to the *Execution Engine* module of CAPE and add additional optimization component modules are outlined in Chapters 4 and 6.

Chapter 4

Proactive Promotion: Preferential Resource Allocation in Stream Processing Systems

4.1 The Proactive Promotion Query Model and Problem Definition

This chapter provides details on our work towards Task 1 (i.e., Proactive Promotion). The main goal of Task 1 is to design a TP framework that supports the ability to adaptively and continuously change how resources are allocated such that the more significant promotable tuples are pulled ahead of less significant ones.

4.1.1 Proactive Promotion Queries

We extend CQL [ABW06] to support a processing time limit and multi-tiered monitoring criteria. Our P-CQL query q_h is a CQL query [ABW06] augmented by a *lifespan* clause and a set of *monitoring levels*. Below is the P-CQL extension to the Home Arrest Query (Sec. 1.3.1) and Stock Market Queries (Sec. 1.3.2) using the monitoring order in Table 1.1.

(P-CQL Extension to Home Arrest Query)

LIFESPAN 1000 *milliseconds*

RANK 1

CRITERIA PI.CommittedViolentOffense = TRUE

AND Distance(PL.Locale, PI.ProperLoc) $\geq$ 1 mi

RANK 2

CRITERIA PI.RecentlyFoundInWrongLocation = TRUE

RANK 3

CRITERIA PI.KnownFlightRisk = TRUE

(P-CQL Extension to Stock Market Queries)

```
LIFESPAN 1000 milliseconds
RANK 1 /* aggressive investments */
CRITERIA (S.ownedByCompany=TRUE) AND (S.aggressive=TRUE)
RANK 2 /* conservative investments */
CRITERIA (S.ownedByCompany=TRUE) AND (S.conservative=TRUE)
```

```
RANK 3 /* stocks under evaluation */

CRITERIA (S.underEvaluation= TRUE)
```

Table 4.1: Notations for PP Query Plans

Notation	*Meaning*
q_h	a query
$q_h.lf$	lifespan of query q_h
$q_h.M$	set of monitoring levels of query q_h
t_i	a tuple
m_k	a monitoring level in the set of monitoring levels $q_h.M$
$m_k.rnk$	rank of monitoring level m_k
$m_k.mem$	membership criteria of monitoring level m_k
p_j^{pp}	a PP query plan
$p_j^{pp}.AM$	set of activated monitoring levels in PP query plan p_j^{pp}
PS_{m_k}	promotable subset for monitoring level m_k of query q_h
op_l	an operator
s_l	a stream
w_p	a window
$T_{p_j^{pp},PS_{m_k}}$	est. latency for tuples in promotable subset PS_{m_k} through PP query plan p_j^{pp}

The special clauses added to CQL to form P-CQL include:

• The *lifespan* clause $q_h.lf$ is the processing time limit, e.g., 1000 milliseconds above. That is, the query results generated by tuple t_i are only valuable to the receiving application if they are received within the query lifespan. Thus, if the time spent processing tuple t_i exceeds the lifespan then tuple t_i is no longer processed.

• The *rank* and *criteria* clauses together specify the user's preferences about which results would be preferred over other results when resources are scarce (Table 1.1). These clauses are referred to as the *monitoring levels* $q_h.M$. Each *monitoring level* m_k defines its rank $m_k.rnk$ and membership criteria $m_k.mem$. *Rank* specifies the significance of m_k in relation to other levels. Each rank is an integer value ranging from $[1,\infty]$ where 1 is the most significant rank. Monitoring level m_k is more significant than level m_l if $m_k.rnk < m_l.rnk$. *Membership criteria* $m_k.mem$ are the predicates that a tuple must meet to have the rank of m_k. They can contain multiple predicates combined using conjunction and disjunction, and reference attributes from different streams. Tuple t_i meets the criteria of m_k if $m_k.mem(t_i) = true$.

4.1.2 Tuple Significance

At times, there may be insufficient resources to pull forward all significant tuples, i.e., preferentially allocated resource to all tuples that match the membership criteria of a monitoring level. Given the available resources, the PP optimizer only pulls forward some significant tuples. We refer to the monitoring levels used to identify the tuples pulled forward as *activated monitoring level*s. The set of activated monitoring levels AM is a subset of the set of monitoring levels $AM \subseteq q_h.M$.

Definition 1 *Promoted tuple t_i is designated with a single rank, namely, the most significant activated monitoring level of which tuple t_i meets the criteria.*

Consider that tuple t_i is from an escaped violent prisoner (i.e., $m_1.mem(t_i) = true$) and flight risk (i.e., $m_3.mem(t_i) = true$). Monitoring levels 1, 2, and 3 are activated, i.e., $AM = \{m_1, m_2, m_3\}$. Then tuple t_i will be assigned rank m_1.

Definition 2 *A back burner tuple has not yet been designated a rank.*

4.1.3 Proactive Promotion Query Plans

A P-CQL query q_h can be represented by many alternative *PP query plan*s. Each PP plan p_j^{pp} is represented as a directed acyclic graph composed of *PP algebra operator*s (Sec. 4.2.1) as nodes and *data exchange interface*s that transfer tuples between operators as edges. A rank classifier RC is a specialized operator that computes and assigns ranks to tuples. All other operators are variants of CQL operators extended to propagate promoted and back burner tuples (Sec. 4.2.1).

The data exchange interface is designed to efficiently pull more significant tuples ahead of less significant tuples. If a traditional buffer were deployed between operators, then all tuples, both promoted and back burner, would be intermingled. Prior to processing, each operator would have to sort or scan the complete input buffer to locate the most significant tuples. This may introduce unacceptable delays. Inspired by priority queuing in differentiated services architectures [ALN08], PP instead uses a multi-queue approach. Each operator supports a queue for each monitoring level and one for back burner tuples. All operators place their results into the appropriate queue of the next down stream operator.

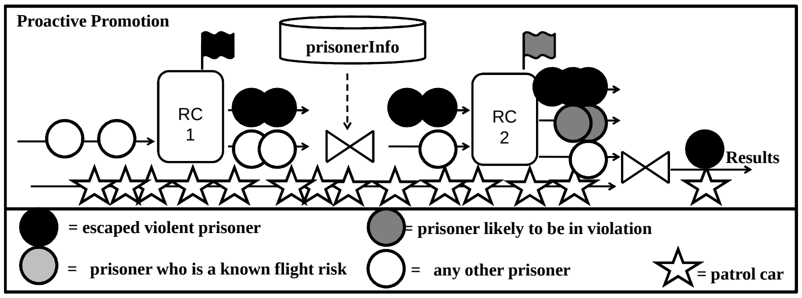

Proactive Promotion
prisonerInfo
RC 1
RC 2
Results
= escaped violent prisoner
= prisoner likely to be in violation
= prisoner who is a known flight risk
= any other prisoner
= patrol car

Figure 4.1: PP Plan p_1^{pp} for Home Arrest Query (Sec. 1.3.1)

Consider PP plan p_1^{pp} for the Home Arrest Query (Sec. 1.3.1) and P-CQL extension (Sec. 4.1.1) in Figure 4.1 in a moderately overloaded system. In Figure 4.1, the flag above the RC operator designates which ranks the RC evaluates and assigns to tuples. First, rank classifier RC1 locates and assigns rank m_1 to tuples from escaped violent prisoners (e.g., black circles). Next, the join operator compares each prisoner's current locale to a table of permitted locations. Then rank classifier RC2 locates and assigns rank m_2 to tuples from prisoners likely to be in violation (e.g., dark gray circles). The last join operator joins the results with the officers in close proximity.

Each PP plan specifies *which monitoring levels are activated* and *in which rank classifier the membership criteria of each activated monitoring level is evaluated. PP plan* p^{pp}_j for query q_h defines a set of activated monitoring levels $p^{pp}_j.AM$ ($p^{pp}_j.AM \subseteq q_h.M$). For each activated monitoring level $m_k \in p^{pp}_j.AM$, PP plan p^{pp}_j specifies which rank classifier(s) performs monitoring level m_ks membership evaluation. For instance, in Figure 4.1, p^{pp}_1's activated monitoring levels are monitoring levels m_1 and m_2, i.e., $p^{pp}_1.AM = \{m_1, m_2\}$. Monitoring level 1 is evaluated in rank classifier RC1, while level 2 is evaluated in rank classifier RC2.

4.1.4 Proactive Promotion Problem

Broadly, the Proactive Promotion Problem is to locate the PP plan that minimizes the latency for tuples for the largest sequence of monitoring levels in consecutive order within the given CPU resources. To ensure that the most significant tuples are monitored before less significant tuples, their latency should be less than the query lifespan. The goal is to not allow any highly significant tuples to expire, and thus, to assure less significant tuples are only processed if resources exist. To achieve this, we must identify for each monitoring level the subset of the tuples in the workload that satisfy the monitoring level's membership criteria.

Definition 3 *<u>Promotable subset</u> PS_{m_k} for monitoring level is the set of tuples that would have the rank of $m_k.rnk$ if monitoring level m_k was activated per Def. 1.*

Table 4.2: Example: Ranking PP Plans

PP Plan (p_x^{pp})	CPU $Overhead$	$T_{p_j^{pp},PS_{m_1}}$	$T_{p_j^{pp},PS_{m_2}}$	$T_{p_j^{pp},PS_{m_3}}$
p_1^{pp}	1022	52	65	69
p_2^{pp}	1256	69	68	50
p_3^{pp}	1956	60	77	95
p_4^{pp}	1006	83	50	87

Notion of PP Plan Optimality. We use $T_{p_j^{pp},PS_{m_k}}$ to denote the estimated latency for tuples in promotable subset PS_{m_k} processed by PP plan p_j^{pp}. Intuitively, the optimal PP plan ensures that the latency is less than the query lifespan for tuples from the largest consecutive sequence of monitoring levels starting from the most significant level down to as many levels as possible.

Consider the example in Table 4.2 where the query lifespan lf is equal to$70ms$. Assume the resources required to execute each PP plan in Table 4.2 are within the available system capacity. PP plan p_1^{pp} is the best among them as explained below. For the most significant monitoring level m_1, PP plan p_4^{pp} has a latency greater than lf, i.e., $lf \leq T_{p_4^{pp},PS_{m_1}}$. For the next most significant monitoring level m_2, PP plan p_3^{pp} has latency greater than lf. For all monitoring levels, PP plan p_1^{pp} and PP plan p_2^{pp} have a latency less than or equal to lf. However, compared to PP plan p_2^{pp}, PP plan p_1^{pp} has the lowest CPU overhead cost and thus is the preferred solution.

Definition 4 PP plan p_i^{pp} is the **optimal** PP plan for P-CQL query q_h iff compared to any other PP plan p_j^{pp} for P-CQL query q_h:

A) For the most significant monitoring level, i.e., $(m_k.rnk = 1)$, p_i^{pp} has a latency for monitoring level m_k less than or equal to the query life span (i.e., $T_{p_i^{pp},PS_{m_k}} \leq q_h.lf$) and p_j^{pp} does not (i.e., $T_{p_j^{pp},PS_{m_k}} > q_h.lf$). If both p_i^{pp} and p_j^{pp} have a latency

less than or equal to the query lifespan for monitoring level m_k (i.e., $T_{p_i^{pp},PS_{m_k}} \leq q_h.lf$ and $T_{p_j^{pp},PS_{m_k}} \leq q_h.lf$), then for the monitoring level with the next highest significance m_l, p_j^{pp} has a latency greater than the query life span (and so on). If for all monitoring levels both PP plans p_i^{pp} and p_j^{pp} have latencies less than the query life span then p_i^{pp} utilizes less CPU resources than p_j^{pp}.
B) The resources required to execute PP plans p_i^{pp} and p_j^{pp} are within the available system capacity respectively.

4.2 PP Query Processing

4.2.1 PP Query Algebra

PP algebra extends traditional operators from the continuous query algebra [GO05] to propagate an incoming tuple's rank to its result tuples. In addition, special purpose rank classifier operators (RCs) compute and assign the rank to tuples. Generally, each operator op_i starts processing tuples from the highest priority queue. When this queue is empty and allocated CPU resources remain, operator op_i processes tuples from the next highest priority queue. Operator op_i places its results into the appropriate incoming queues of the next operator.
Rank Classifier (RC) is an unary operator with an assessment set parameter AS. RC assigns rank to tuples in its input stream by invoking the priority determinants specified by its assessment set parameter AS parameter. The assessment set parameter AS indicates which of the activated monitoring levels the RC evaluates. Each tuple t_i processed is designated the most significant of either t_i's current rank or the rank of the priority determination functions that are being evaluated and that t_i satisfies. Tuples are sent with their assigned rank, if any, to the next operator.

Projection discards unwanted attributes from tuples in its input stream. ***Selection*** drops tuples in its input stream that don't satisfy the selection condition. Both send their results to the next operator without changing their rank.

Join, a symmetric binary hash join [WA91], maintains a state for each input stream (s_p and s_q). To process tuple t_i, first t_i is stored with its rank in the state of t_i's stream s_p. Next, results are created by joining t_i with tuples t_j in the other stream s_q's state. Each result (t_i,t_j) is assigned the highest rank of its composing join partners, i.e., t_i and t_j. Join result (t_i,t_j) with its rank, if any, is sent to the next operator. To prevent the blockage of significant results, each join operator allocates the same number of CPU resources to process tuples from each of its incoming streams.

As confirmed by our experiments (Sec. 4.5), our simple join strategy is effective for the following reasons. First, if tuple t_i is significant (i.e., $m_k.mem(t_i) = true$), then any join result that t_i contributes to will be significant by definition (i.e., have at least rank $m_k.rnk$). When the system is overloaded, all resources are devoted to processing the most significant tuples first. This automatically produces only significant results. Second, for the same reason, join states for streams containing promoted tuples will mostly contain significant tuples and few, if any, back burner tuples. Thus even when a back burner tuple joins with such a state, it is likely to combine with promoted join partners and produce significant results.

Reconsider the Home Arrest Query (Sec. 1.3.1) using the P-CQL extension in Section 4.1.1 executed in a moderately overloaded system (e.g., PP plan in Fig. 4.1). In this case, the prisoner stream pulls significant tuples forward, i.e., tuples from escaped violent or likely to be in violation prisoners. As such, the state of the join operator that combines the current locales of prisoners and officers will contain significant prisoner tuples. Any incoming officer tuple processed will create significant results as the prisoner tuples it is matched against are likely all significant. While other join methods are perceivable, such as actively attempting to pull potential join partners up the pipeline, they require additional resources without any guarantee of success. Therefore, we do not consider such strategies here, as they add extra demands on the already scarce resources.

4.2.2 Stateful Operators & Out-of-Order Handling

PP actively causes tuples to become out-of-order by pulling some tuples ahead of others. Out-of-order methods from the literature [LLG$^+$09] were designed to tackle out-of-order issues caused by external causes such as network transmissions. Broadly, these methods synchronize the progression of time within the query pipeline to address correctness and completeness of query processing [LLG$^+$09]. These existing methods can also be applied to PP.

Broadly, this requires stateful operators to organize and manage their states. First, states must allow stored tuples to be located by query window to support the correct production of results. Second, to ensure that states do not grow unbounded, they must purge stored tuples. However, states should only purge stored tuple t_i when it can be guaranteed that t_i will not create any results in the future.

To safely purge tuples from states, similar to [LMT$^+$05], each leaf operator op_i periodically sends progress indicator punctuations when op_i will no longer process any tuples from a set period of time. When operator op_i receives such notification, first op_i assures that no more tuples are waiting to be processed related to this period of time. Then progressively each operator sends similar punctuations to its next operator down stream (and so on).

4.2.3 PP Algebraic Rewriting

Rank classifiers (RCs) evaluate and assign rank to tuples. They don't modify tuple attributes, remove nor produce tuples. Clearly, creating a new PP plan by changing where monitoring levels are evaluated by splitting, merging, removing, or relocating RCs (or rather their assessment sets) maintains the logical correctness of the original plan semantics. Hence, rewrite rules can be designed similar to those for pushing selections through a plan [Che99]. Given adequate resources, a rewritten PP plan would produce the same results as the original plan. Given insufficient resources, fewer results may be produced, namely, only the most significant ones. However, each result output is still guaranteed to be valid, i.e., satisfies all query predicates.

4.3 PP Optimization

4.3.1 PP Architecture

The PP architecture (Fig. 4.2) is composed of the PP Executor, PP Monitor, PP Optimizer, and PP Adaptor. The PP Executor runs the physical PP plan (Sec. 3.4). The PP Monitor gathers statistics (Sec. 3.5.1). Periodically, the PP Monitor triggers the PP Optimizer to find a more optimized PP plan (Sec. 3.5.3-3.5.5). The PP Adaptor adapts the current PP plan into the new optimal PP plan (Sec. 3.6). To reduce overhead, the PP Monitor, PP Optimizer, and PP Adaptor run on a separate thread from the PP Executor.

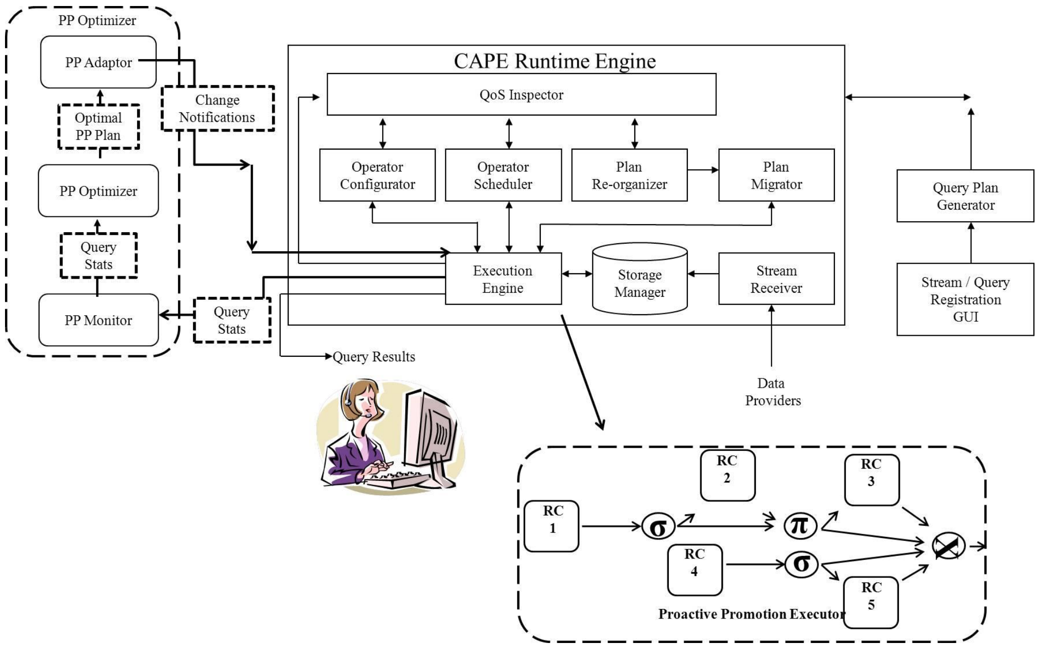


Figure 4.2: PP Architecture

4.3.2 PP Monitor

The PP Monitor quickly and cheaply identifies *when* to trigger the PP Optimizer. Similar to [WPSS06], the PP Monitor monitors the work load. The PP Monitor gathers statistics to estimate the estimated latency for each promotable subset (Sec. 4.3.3). These statistics include the number of expired tuples for each promotable subset, the queue lengths, and average tuple computation time of all operators. The PP Monitor will trigger the PP Optimizer when, either tuples from a promotable subset expire or there may be adequate resources to process all incoming tuples. This consists of three cases.

The first case is when tuples from promotable subset PS_{m_k} expire and monitoring level m_k is deactivated. Adapting to a PP plan where monitoring level m_k is activated will pull tuples in promotable subset PS_{m_k} forward.

The second case is when tuples from promotable subset PS_{m_k} expire, monitoring level m_k is activated, and less significant monitoring levels are also activated. In this case, the latency of promotable subset PS_{m_k} may be reduced by adapting to a PP plan that changes the evaluation location of monitoring level m_k and/or de-activates some less significant activated monitoring levels. This would reduce the precedence determination overhead and dedicate more resources to tuples in promotable subset PS_{m_k}.

The third case is when no tuples expire and some monitoring levels are activated. In this case, PP Optimization will explore PP plans that deactivate some monitoring levels. This may reduce the precedence determination overhead and dedicate more resources to processing tuples.

4.3.3 Latency Metric of Promotable Subsets

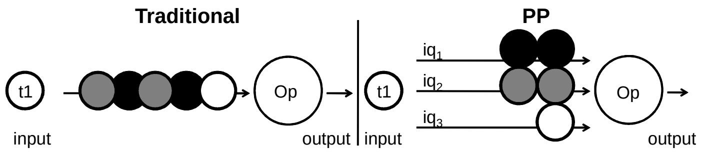


Figure 4.3: Traditional vs PP Operator Latency Example

Estimated Operator Latency of a Promotable Subset: The estimated latency for tuples in PS_{m_k} at operator op_l, $T_{op_l,PS_{m_k}}$, is the sum of the average tuple wait time at rank $m_k.rnk$ and the average tuple computation time $T_t(op_l)$ (or the time for operator op_l to process a tuple) (Equation 4.1). The average tuple wait time is the time it takes to process all tuples with greater or equal significance that arrived before the new tuple. It is affected by the workload of tuples at each rank, i.e., the queue lengths $QL(op_l, m_k)$. The average tuple wait time at rank $m_k.rnk$ is the product of the sum of the lengths of each queue with greater or equal significance and the computation time $T_t(op_l)$. The estimated latency for back burner tuples through op_l, $T_{op_l,bb}$, is simply a special case of $T_{op_l,PS_{m_k}}$, namely, waiting for all queues to empty prior to processing. Latency notations are listed in Table 4.3.

$$T_{op_l,PS_{m_k}} = ((\sum_{x=1}^{m_k} QL(op_l, x)) * T_t(op_l)) + T_t(op_l) \tag{4.1}$$

Consider tuple $t1$'s latency through operator op_l in Fig. 4.3. If tuple $t1$'s $rank = 1$ (in queue iq_1) then tuple $t1$'s latency is $3 * T_t(op_l)$. If tuple $t1$'s $rank = 3$ (in queue iq_3) then tuple $t1$'s latency is $6 * T_t(op_l)$.

Estimated Query Latency of a Promotable Subset: The estimated latency for tuples in promotable subset PS_{m_k} through PP plan p_j^{pp}, $T_{p_j^{pp},PS_{m_k}}$ (Equ. 4.2), where rank classifier op_m evaluates monitoring level m_k consists of two parts. 1) The estimated latency before tuples in promotable subset PS_{m_k} are preferentially allocated resources $\sum_{x=op_1}^{op_m} T_{x,bb}$, i.e., processed as back burner tuples at each operator prior to and through rank classifier operator op_m. 2) The estimated latency after they are preferentially allocated resources $\sum_{x=op_{(m+1)}}^{op_n} T_{x,PS_{m_k}}$, i.e., processed as promoted tuples at rank m_k at each operator after rank classifier operator op_m.

$$T_{p_j^{pp},PS_{m_k}} = (\sum_{x=op_1}^{op_m} T_{x,bb} + \sum_{x=op_{(m+1)}}^{op_n} T_{x,PS_{m_k}}) \tag{4.2}$$

Table 4.3: Latency Notations for PP Query Plans

Notation	*Meaning*
$T_{p_j^{pp}, PS_{m_k}}$	est. latency for tuples in promotable subset PS_{m_k} in PP plan p_j^{pp}
$T_{op_l, PS_{m_k}}$	est. latency for tuples in promotable subset PS_{m_k} in operator op_l
$T_t(op_l)$	avg. tuple computation time in operator op_l
$QL(op_l, m_k)$	tuple queue len. for monitoring level m_k in operator op_l

4.3.4 The PP Plan Search Space

Observation: We now observe that the estimated latency for a promotable subset depends upon where the evaluation of each activated monitoring level m_k occurs.

The goal of PP is to ensure that given the available memory tuples from the most significant promotable subsets do not expire. Namely, when resources are scarce, they are first dedicated to ensuring that the most significant tuples do not expire. If resources remain, then they are dedicated to ensuring that the next most significant tuples do not expire (and so on). This means that we first seek to reduce the latency of tuples in the most significant promotable subset. In addition, the latency of tuples in the most significant promotable subset is affected by which monitoring levels are activated and where each priority determinant is evaluated. Hence, as shown in the example PP plan in Section 3.3, the optimal PP plan may evaluate the priority determinants of the most significant monitoring level in an RC early in the query pipeline and the priority determinants of another less significant monitoring level in an RC later on.

There is no benefit to simply placing the evaluation of each priority determinant in the earliest RC in the plan. The placement of where each priority determinant is evaluated depends upon the selectivity and cost of each operator. That is, if the cost of evaluating a priority determinant is high then it may take less CPU processing time to evaluate it after a highly selective operator rather than before. Consider the following scenario. There are 300 back burner tuples in the pipeline before operator op_1. It costs 2 milliseconds to evaluate if a tuple satisfies priority determinant d_1. It costs 1 millisecond for operator op_1 to process a tuple. The selectivity of operator op_1 is $.5$, i.e., many tuples are dropped.

Now, consider the latency of tuples in promotable subset PS_{m_k}. The cost to evaluate the 300 back burner tuples in an RC after operator op_1 against priority determinant d_1 is 300 milliseconds (e.g., op_1 process 300 tuples) + 300 milliseconds (e.g., RC process 150 tuples as 1/2 of the tuples are dropped by operator op_1) = 600 milliseconds total. However the cost to evaluate the 300 back burner tuples in an RC before operator op_1 against priority determinant d_1 and then be processed by operator op_1 is 600 milliseconds (e.g., RC process 300 tuples) + X milliseconds (e.g., where X is the number of tuples that have been promoted to rank $m_k.rnk$) = 600+X milliseconds total. Even if one tuple is promoted to rank $m_k.rnk$ then it will cost 601 milliseconds. In this case, it does not make sense to move the evaluation to the earliest RC in the plan.

Locating the Optimal PP Plan: The PP Optimizer optimizes resource allocation by exploring which monitoring levels to activate and where in the pipeline to evaluate each activated priority determinant, i.e., in which RC. Beyond adapting resource allocation, the traditional question of the optimal order of operators within a PP plan must also be explored.

Two possible alternatives to locate the optimal PP plan are 1) Similar to [AH00, NRB09], we could support a separate optimal ordering of operators for each promotable subset using a *multiroute architecture*. However, this would not allow the precedence determination overhead to be reduced or ideally even eliminated online. Rather, potentially expensive precedence determination would need to be performed on each and every single incoming tuple to locate their best route prior to processing. 2) Alternatively, a *combined optimization* approach would consider both the regular and PP costs at the same time [RG00]. However, finding an optimal query plan considering only the regular costs is known to be NP-hard [BMM$^+$04]. To consider both costs would only increase the complexity. If the system is overloaded, such expensive reordering of operators may delay the processing of significant tuples even further.

To eliminate these issues, we propose a two-phased approach. Our approach supports agile changes to resource allocation online by separating the resource allocation from the standard plan operator ordering decisions. First, the query optimizer using traditional query optimization [RG00] selects an optimal order of operators as query plan p_i. Then, our *PP Optimizer* transforms this query plan p_i into an optimized PP plan p_i^{pp} by locating the optimal allocation of resources (Sec. 4.1.3) for this query pipeline. Whenever a new query plan ordering is chosen, the resource allocation (i.e., a new PP plan) is determined for the new query plan.

Creating a PP Plan: Transforming a traditional query plan p_i into a PP plan p_i^{pp} requires the addition of rank classifiers, i.e., RCs. Any number of RCs could be placed in query plan p_i. However, there is no benefit in placing two RCs directly adjacent to each other. Their functionality can always be merged into one RC (Sec. 4.2.3). Thus, one RC is placed before each standard operator. Any RC that evaluates no monitoring levels (i.e., empty assessment set) is skipped. Recall the RCs in PP plan p_1^{pp} for the Home Arrest Query in Figure 4.1.

Where can a Priority Determinant be Evaluated? A monitoring level may contain multiple criteria combined via conjunction and disjunction (Sec. 4.1.1). No benefit exists in evaluating some but not all criteria of a given monitoring level in an RC. In such a case, whether or not a tuple satisfies the partial criteria will not be enough to establish the tuple's significance. Thus the membership criteria of each monitoring level is transformed into a disjunctive normal form. We refer to each disjunct as a *priority determinant*. Each RC uses priority determinants to determine tuple significance. An RC can only evaluate a priority determinant d_i if the attributes used by the predicates of d_i are available in the incoming tuples to the RC. Each d_i is associated with an *evaluation path* of RCs that begins at the first and ends at the last RC in the pipeline where all predicates of d_i are defined.

Reconsider PP plan p_1^{pp} (Fig. 4.1). If priority determinant d_i contains attributes from the prisoner stream then the evaluation path of d_i would contain RC1 and RC2. However, if d_i contains attributes from the prisoner information table then the evaluation path of d_i would only contain RC2.

How many RCs Evaluate a Priority Determinant?

Observation: *Each priority determinant d_i of monitoring level m_k needs to only be evaluated by one RC in the plan.*

Reconsider PP plan p_1^{pp} in Figure 4.1 where priority determinant d_i's evaluation path contains RC1 and RC2. If RC1 evaluates d_i then all tuples that satisfy d_i will have been promoted before they reach RC2. That is, RC2 will not pull any new tuples forward by evaluating d_i.

Observation: *The estimated latency for promotable subset PS_{m_k} is affected by which RC evaluates the priority determinants of m_k.*

Reconsider PP plan p_1^{pp} where priority determinant $d_i \in m_k$ and d_i's evaluation path contains RC1 and RC2 (Fig. 4.1). If RC1 evaluates priority determinant d_i then tuples in promotable subset PS_{m_k} will be preferentially allocated resources in both join operators. If RC2 evaluates priority determinant d_i then tuples in PS_{m_k} will only be preferentially allocated resources in the last join operator.

Two Critical Factors of the PP Search Space Size are 1) the number of RCs that can evaluate each priority determinant and 2) the number of possible sets of activated monitoring levels.

Theorem 1 *The PP search space size for query plan p_i is $\sum_{am_x=1}^{2^{|q_h.M|}} \prod_{ep_y=1}^{|EP|} |ep_y.RC|^{|AM_{ep_k}|}$. AM_{ep_k} is the set of priority determinants from the activated monitoring levels whose evalation path is ep_k.*

Proof *The number of PP plans for query plan p_i is equal to the sum of PP plans for each possible activated monitoring level set, i.e., $\sum_{am_x=1}^{2^{|q_h.M|}}$. There are $2^{|q_h.M|}$ combinations of creating an activated monitoring level set AM from the monitoring levels in $q_h.M$. am_x is one possible activated monitoring level set AM.*

For activated monitoring level set am_x all PP plans must be generated. This is the sum of the combinations of placing each priority determinant from the activated monitoring levels whose evalation path is ep_k (i.e., AM_{ep_k}) into an RC in evaluation path ep_k. ep_y is one evaluation path in the set of all evaluation paths EP. $ep_y.RC$ is the set of RCs in evaluation path ep_y. This problem can be mapped to the well known problem of placing each ball in a set of n distinguishable balls into a bin from a set of m distinguishable bins [Ree07]. Each ball represents a priority determinant d_i whose monitoring level is in the set of activated monitoring levels AM_{ep_k}. Each bin represents a RC $ep_y.RC$ in evaluation path ep_y. Each PP plan is modeled by placing a priority determinant d_i (i.e., a ball) into one of its $|ep_y.RC|$ bins. The number of PP plans for am_x is thus $\prod_{ep_y=1}^{|EP|} |ep_y.RC|^{|AM_{ep_k}|}$.

The complexity of the PP problem is exponential in the number of priority determinants (Thm. 1). Hence, it is impractical to exhaustively search for the optimal PP plan for complex PP queries with many priority determinants.

4.3.5 Pruning the PP Search Space

As foundation for pruning the search space, we now establish properties of ranking that reduce the latency of promotable subsets. First, tuples that skip being evaluated by RC_n reduce their latency by the processing costs of RC_n. Second, the evaluation order of monitoring levels within an RC and across the evaluation paths of the plan can allow some tuples to skip being evaluated by some priority determinants. This can further reduce latency.

Property 1*: If the rank already assigned to promoted tuple t_i is more significant than the monitoring levels in RC_j's assessment set, i.e., $\forall m_l \in RC_j.AS$ $(m_l.rnk > t_i.rnk)$, then t_i is guaranteed to never be assigned a rank by RC_j. Thus t_i can skip being evaluated by RC_j.*

For PP plan p_1^{pp} (Fig. 4.1), promoted tuple t_i is assigned rank 1. Rank Classifier RC2 evaluates the determinants for monitoring level m_2 and assigns rank 2 to tuples. Tuples are assigned the highest rank that they satisfy the criteria for (Sec. 4.1.2). Tuple t_i is already assigned a higher rank than 2. Thus RC2 will never assign a rank to tuple t_i.

Property 2*: Rank classifier RC_j that evaluates the priority determinants of the monitoring levels in its assessment set in rank order reduces the latency of significant tuples compared to evaluating the same criteria not in rank order.*

For PP plan p_1^{pp} in Figure 4.1, consider that rank classifier RC1 evaluates the determinants for monitoring levels m_1 and m_2. Assume that first RC1 evaluates the determinants for monitoring levels m_2 and then for m_1. If tuple t_i belongs to the most significant promotable subset PS_{m_1} then t_i will need to be evaluated by determinants for both monitoring levels. Now assume that first RC1 evaluates the determinants for monitoring level m_1 and then for m_2. In this case, tuple t_i will only need to be evaluated by determinants for m_1.

From these properties, we derive that the fewer priority determinants evaluated by a tuple before it is promoted the lower its latency. The order of the ranks of where each priority determinant is evaluated across RCs from the first RC to the last RC in the plan also lowers the latency of significant tuples. We now show that PP plans that evaluate monitoring levels out of order are never optimal. Hence they can safely be removed from the search space.

Definition 5 *PP Plan p_i^{pp} is called an* ordered PP plan *if given any RC_j in PP plan p_i^{pp}, no RC past RC_j in the pipeline, say RC_k, evaluates determinants for any activated monitoring level more significant than the activated monitoring levels of the determinants evaluated by RC_j, i.e., $(\forall m_m \in RC_j.AS)$ $(\forall m_l \in RC_k.AS)$ $(m_m.rnk \leq m_l.rnk)$. All other plans are called* unordered PP plans.

Theorem 2 *For any unordered PP plan p_i^{pp} of query plan p_i, there exists an ordered PP plan p_j^{pp} (Def. 5) of query plan p_i such that the ordered PP plan p_j^{pp} is superior to the unordered PP plan p_i^{pp} (Sec. 4.1.3).*

Proof *First, consider a PP plan with one RC and several activated monitoring levels. Per property 2, all activated monitoring levels should be evaluated in rank order in the sole RC. Hence the* ordered PP plan *is superior.*

Now, assume an unordered PP plan p_i^{pp} for query q_h is optimal. Let p_i^{pp} have multiple RCs and activated monitoring levels. m_j is the most significant monitoring level evaluated out-of-order. The priority determinant $m_j.d_k$ of m_j is evaluated by RC_l. Thus, by assumption, some monitoring levels less significant than m_j are evaluated prior to RC_l. The latency for promotable subset PS_{m_j} through p_i^{pp} includes *three costs. First, prior to RC_l, tuples in PS_{m_j} are processed as back burner tuples. This includes the cost to evaluate and promote tuples less significant than m_j, i.e.* tuples evaluated out-of-order. *Second, in RC_l, tuples are evaluated by determinant $m_j.d_k$. Third, after RC_l, tuples in PS_{m_j} are processed as promoted tuples.*

We now create an ordered PP plan p_j^{pp} from the unordered PP plan p_i^{pp} by relocating the priority determinants less significant than m_j which are evaluated prior to RC_l. The newly constructed ordered PP plan p_j^{pp} now evaluates monitoring levels in significance order up to RC_l. The latency for promotable subset PS_{m_j} through p_j^{pp} also includes the same three costs outlined above. However, in contrast to the unordered PP plan p_i^{pp}, the cost prior to RC_l does not include the cost to evaluate and promote tuples less significant than m_j.

The latency of promotable subset PS_{m_i} for PP plans p_i^{pp} and p_j^{pp} are similar with two exceptions. First, the unordered PP plan p_i^{pp} has additional overhead prior to RC_l to evaluate less significant monitoring levels. Second, the unordered PP plan p_i^{pp} processes tuples less significant than m_j before tuples in PS_{m_j} prior to RC_l. This may increase the latency of PS_{m_j}. Namely, compared to the ordered PP plan, the unordered PP plan will almost always process more tuples before tuples in PS_{m_j} in operators prior to RC_l. The only case when the unordered PP plan will process an equal number of tuples as the ordered PP plan is when all tuples less significant than m_j arrive last. Compared to the ordered PP plan, the unordered PP plan will never process less tuples before tuples in PS_{m_j} in operators prior to RC_l.

Thus, ordered PP plan p_j^{pp} has a lower latency for promotable subset PS_{m_i} and is superior to p_i^{pp}. Contradiction.

Theorem 3 *The optimal PP plan p_i^{pp} for a plan p_i will always be an ordered PP* *plan (per Definition 5).*

Proof *An unordered plan p_j^{pp} is never optimal because an ordered plan superior to p_j^{pp} always exists (Thm. 2).*

4.3.6 Rank Order Pruning PP Optimizer

Per Thm. 3, we are now ready to introduce the Rank Order Pruning Optimizer or *ROP*. ROP significantly reduces the search space, yet is guaranteed to find the optimal PP plan. ROP performs an ordered evaluation of the activation and evaluation of each monitoring level in the plan from the most to least significant (Fig. 4.4). First, ROP initializes a default PP plan by placing an RC before each standard operator in the plan (*line 1*). For each priority determinant $m_i.d_j$ for the most significant monitoring level m_i, ROP locates the "best" RC in $m_i.d_j$'s evaluation path to evaluate $m_i.d_j$ within the available system capacity (*lines 6-16*). The best RC results in an estimated latency lower than the query lifespan and has the lowest overhead. If no best RC is found then either all possible RCs result in an estimated latency higher than the lifespan or the overhead to evaluate $m_i.d_j$ is above the available system capacity. In this case, m_i is not activated, the search stops. The previously found optimal PP plan is returned (*line 17*). Otherwise, m_i is activated, $m_i.d_j$ is added to the assessment set of the best RC (*line 18*). The search continues with the next most significant monitoring level. Each search begins at the RC that is either the best RC for the last monitoring level evaluated (i.e., maintaining rank order) or the earliest possible RC for the priority determinant. This continues until either a monitoring level is found to not be worthwhile for activation (*line 19*), no more resources are available (*line 17*), or all monitoring levels are activated (*line 21*).

```
ROP(S=stats.,$p_i$=plan,$availCPU$=avail.res.,$q_h.l$=lifespan)
1: Initialize $p_i^{pp}$ from $p_i$; lastRC $\leftarrow$ last RC in $p_i^{pp}$;
2: for $ep_i = ep_1$ to $|EP|$ do
3:     $startRC[ep_i] \leftarrow$ first RC in $ep_i$; {Init. each eval. path}
4: end for
5: for $m_i = m_1$ to $m_n$ where $n = |q_h.M|$ do
6:     for $d_j = d_1$ to $d_p$ where $p = |m_i.d|$ do
7:         $bestRC[d_j] \leftarrow$ null; bestCost $\leftarrow \infty$; {each deter. $d_j$ in level $m_i$}
8:         for $RC_k$ = $startRC[d_j.ep]$ to lastRC do
9:             assume $m_i.d_j$ is evaluated in rank classifier $RC_k$;
10:            estLat $\leftarrow T_{p_i^{pp},PS_{m_i}}$
11:            estCost $\leftarrow$ est. CPU overhead;
12:            If(estLat $< q_h.l$) and (estCost $\leq$ bestCost) and (estCost $\leq$ availCPU)
13:            then $bestRC[d_j] \leftarrow RC_k$; bestCost $\leftarrow$ estCost; endif
14:        end for
15:        if ($bestRC[d_j] \neq null$) then
           $startRC[d_j.ep] \leftarrow bestRC[d_j]$; endif
16:     end for
17:     if (estCost $\geq$ availCPU) then return $p_i^{pp}$; endif
18:     if ($\forall bestRC[] \neq null$) then add each $m_i.d_j$ to $bestRC[d_j].AS$ in $p_i^{pp}$
19:     else return $p_i^{pp}$; endif
20: end for
21: return $p_i^{pp}$;
```

Figure 4.4: Rank Order Pruning PP Optimizer *ROP*

Theorem 4 *ROP finds an optimal PP plan.*

Proof *ROP searches all ordered PP plans and returns the best one. An ordered plan will be optimal (Thm. 3). ROP is guaranteed to find an optimal PP plan.*

Theorem 5 *ROPs complexity is $\sum_{k=1}^{|EP|}(|ep_k.rc| * |AM_{ep_k}|)$ where am_x is the sequence of monitoring levels activated starting from the most significant monitoring level.*

Proof *In the worst case scenario, the first RC of each evaluation path is optimal for all priority determinants of every monitoring level. Then, each determinant needs to consider all possible RCs in its evaluation path.*

4.4 Resource Allocation Adaption

Our agile execution framework makes online resource allocation adjustments without requiring infrastructure changes. That is, operators are not added, removed, or reordered. Instead, upon receiving a notification from the PP Adaptor, each operator adapts which tuples are preferentially allocated resources. To not delay adaption, a control exchange interface is dedicated to transfer such notifications between each operator and the PP Adaptor.

Each operator op_i preferentially allocates resources to incoming promoted tuples. Incoming promoted tuples to operator op_i are designated a rank by an RC that proceeds operator op_i. The activated monitoring levels evaluated by any RC that proceeds operator op_i may change. Hence, operators must adapt which incoming tuples they preferentially allocate resources to.

4.4.1 Adaption and In-process Tuples

Run-time adaption is triggered when the PP Optimizer selects a new PP plan (Sec. 4.3). There are two types of possible changes between the current and new PP plan. First, the number of activated monitoring levels may change. Second, the RC where each priority determinant is evaluated may change. The number of activated monitoring levels decreases or increases when respectively some activated levels are deactivated or some deactivated levels are activated. The evaluation location of a priority determinant may move to a new RC that resides in the pipeline after or before the current RC where the evaluation occurs.

Operators adjust how resources are allocated to in-process tuples by changing which queue they reside in. Only tuples in priority queues will be preferentially allocated resources. Operators *promote* or *demote* tuples by respectively placing them into the proper priority or back burner queue of the next operator.

Over time, a monitoring level may be activated then deactivated then reactivated. Similarly, an inprocess tuple may be promoted then demoted then repromoted again. To avoid repeated precedence determination, once a tuple is assigned a rank it retains its rank for the duration of processing. However, the queue in which the tuple resides may change.

In-process promoted tuples. When the number of levels decreases, operators demote any promoted tuple that is no longer associated with an activated monitoring level. When the number of levels increases, operators promote any *demoted tuple* residing in the back burner queue whose associated monitoring level is activated.

When an evaluation location is moved to a new RC that resides after the current RC, a promoted tuple will be demoted for a portion of the pipeline, namely, until they reach the new RC. Nothing happens to promoted tuples when an evaluation location is pushed before the current RC. Such tuples should be promoted.

In-process back burner tuples. Nothing happens to back burner tuples when either the number of levels decreases or an evaluation location is moved after the current RC. These tuples should remain back burner tuples.

This is not the case when an evaluation location is moved before the current RC or the number of levels increases. Current in-process back burner tuples that belong to the promotable subset of an activated monitoring level and reside in the pipeline between the new and the current RC could be identified and promoted. However, there is very limited number of such tuples that will currently reside in this small portion of the query pipeline. Thus, we choose not to undertake any special checks as the overhead outweighs the gains achievable.

4.4.2 Skipping Extraneous RCs

The physical query plan remains constant while parameters of each operator are adjusted to adapt resource allocation. At times a RC may no longer evaluate any monitoring levels (i.e., its assessment set $= \emptyset$) and will be skipped. To skip extraneous RCs, each standard operator controls where it sends its results. That is, operators send results to either: 1) its adjacent down stream RC and thereafter to the next down stream standard operator or 2) directly to the down stream standard operator (skipping the RC).

Monitoring levels are evaluated in rank order (Thm. 2). Thus, a promoted result never needs to be evaluated by any subsequent RCs (Sec. 4.3.5), i.e., they skip all future RCs. Back burner tuples are only sent to the next RC if its assessment set is not empty. Hence, the special routing decision is executed once and is cheap. In short, each operator either sends **all** back burner results to the next RC or directly to the next standard operator.

4.4.3 PP Plan Adaption

The optimal PP plan selected by the PP Optimizer defines changes to the parameters of each operator. For each RC, it defines an assessment set. For each standard operator op_i it defines where operator op_i's back burner results are sent and which promotable subsets operator op_i preferentially allocates resources to. The optimal PP plan is forwarded to the PP Adaptor which reconfigures the current PP plan into the new one as follows. The PP Adaptor informs each standard operator op_i and RC op_j of required changes to their parameters via a notification. Upon receipt of a notification, operators and RCs adjust their parameters. In summary, PP supports agile online adjustments to resource allocation without requiring any infrastructure changes.

4.5 Experimental Evaluation of PP

4.5.1 Experimental Set Up

Experimental Methodology: All experiments are conducted on nodes in a cluster that consists of 20 processing nodes. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory. Each system uses three machines. One runs the query plan. One monitors statistics, determines, and implements any changes to resource allocation. One tracks for each monitoring level the running online count of the number of results and expired tuples.

We explore: 1) if PP decreases the latency and increases the throughput of significant tuples, 2) how PP is affected by the number of monitoring levels, % of incoming tuples in each promotable subset, and lifespan parameters, 3) PP's run time CPU overhead and memory overhead, and 4) ROP's performance in locating the optimal PP plan compared to the alternative.

These variables most directly affect PP. The number of tuples in a promotable subset in the incoming streams affects the number of significant tuples in the workload. The number of monitoring levels affects the number of different membership criteria, i.e., increases the complexity of precedence determination. The lifespan affects the total number of tuples, significant or not, in the workload, i.e., all tuples may remain in the system for a longer duration. This clearly puts more pressure upon the system by imposing more resource consumption.

Alternative Solutions: We compare PP to traditional DSMS with no shedding, semantic shedding, and random shedding. These alternatives are respectively referred to as trad, sem, and rand. *Rand* randomly selects tuples to process at the incoming stream based on the estimated number of tuples that can be processed within their lifespan given the current statistics. *Sem* uses both the monitoring level criteria and the estimated number of tuples that can be processed to select which incoming tuples from which promotable subsets to process. *Sem* sheds all other tuples. Both *sem* and *rand* process tuples in FIFO order and determine significance (i.e., whether or not to process tuples) as soon as possible. *PP* instead selects which monitoring levels to activate and where in the query plan each activated level criteria is evaluated. *PP* processes all tuples in significance order and assigns significance at the best location in the pipeline.

In the *PP*, *rand*, and *sem* approaches, the PP Monitor collects statistics on the latency of all tuples. In addition, for *PP* and *sem* the PP Monitor also collects statistics on the latency of each promotable subset. Then *rand* and *sem* perform a cost analysis of which tuples to shed. *PP* uses the PP Optimizer to determine which monitoring levels to activate and where it is best to evaluate each priority determinant.

Data Streams and Queries: To vary the workload, three key features are varied for each data-set, namely: 1) number of monitoring levels, 2) percent of incoming tuples in each promotable subset, and 3) query lifespan.

Each data-set is denoted by a triplet. Data-set '5-10-15' has 3 monitoring levels, i.e., $m_1 - m_2 - m_3$. 5% of the tuples in the prisoner stream belong to promotable subset PS_{m_1}, 10% to promotable subset PS_{m_2}, and 15% to promotable subset PS_{m_3}.

Each data-set varies the number of significant tuples in the prisoner stream. For each data-set, the prisoners in each promotable subset are randomly chosen. Each data-set has a different workload and number of possible significant results.

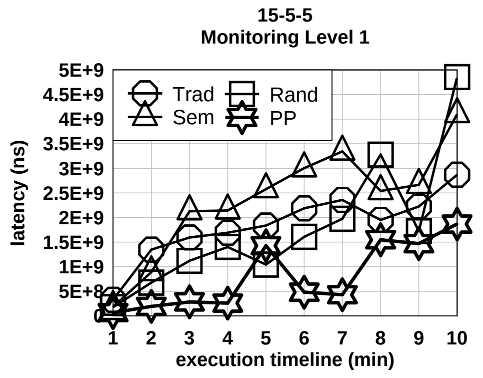
Our experiments use the Home Arrest query (Sec. 1.3.1) and P-CQL extension (Sec. 4.1.1). The prisoner data streams is a stream from [DL] that contain the daily movement of people in Portland, Oregon. The officer data stream was created by dividing Portland, Oregon into regions. Each officer randomly moves modeling a police man patrolling an assigned region.

Metrics and Measurements: All results are the average of 3 experimental runs executed for 10 minutes. Our experiments measure for each promotable subset PS_{m_k} 1) the latency of PS_{m_k} (Sec. 4.3.3) and 2) throughput of PS_{m_k} or the number of results produced that belong to PS_{m_k}.

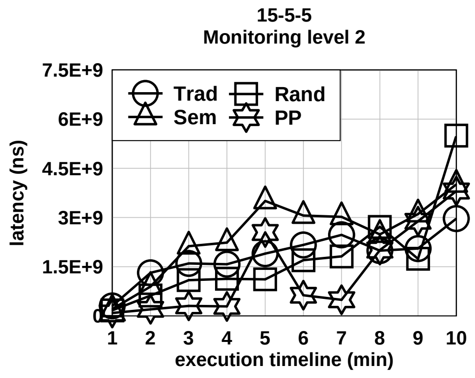
Each experiment can produce roughly 750,000 results (both significant and insignificant) over 10 minutes (See the execution-run-time CPU overhead experiment below). The number of possible significant results compared to insignificant results is small by design. Namely, PP is designed to allocate resources to ensure the production of the most significant results. Hence, we measure how many significant results are produced for each promotable subset PS_{m_k}.

4.5.2 Experimental Results

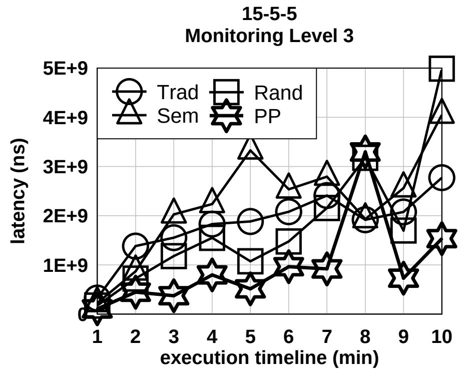
Effectiveness in decreasing latency and increasing throughput First, we compare *PP*'s effectiveness in decreasing latency to the alternative solutions. The dataset used is '15-5-5'. Respectively, 15%, 5%, and 5% of the prisoner stream contains tuples in PS_{m_1}, PS_{m_2}, and PS_{m_3}. The lifespan l = 7500000 ns. Figures 4.5 a, b, and c respectively show the average latency over 10 minutes for promotable subsets PS_{m_1}, PS_{m_2}, and PS_{m_3}. Overall compared to the alternatives, *PP* consistently achieves the lowest latency for each monitoring level. At specific points in time, *PP* may not have the lowest latency for a given level. Consider minute 5. At this moment, PP adjusts the activated level set to only contain m_1 and m_2. That is, *PP* deactivates m_3 to allow more resources to be dedicated to tuples in PS_{m_1} and PS_{m_2}.



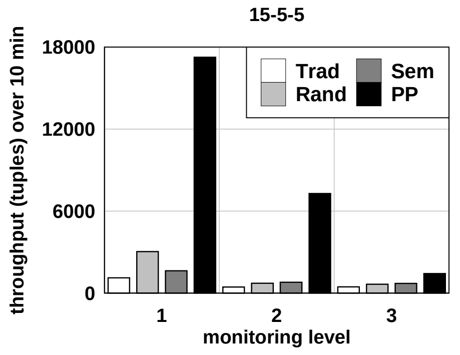
a) Avg. Latency for Monitoring Level 1



b) Avg. Latency for Monitoring Level 2



c) Avg. Latency for Monitoring Level 3



d) Cumulative Throughput

Figure 4.5: Effectiveness in Decreasing Latency and Increasing Throughput

This does lower the latency of the most significant monitoring levels, namely, at minute 6, *PP* again achieves the lowest latency for m_1 and m_2. *PP* is effective at continuously adapting which monitoring levels are activated to minimize the latency of the most significant tuples.

Now, we compare PP's effectiveness in increasing throughput for the data-set defined above (Fig. 4.5 d). We observe that, besides reducing the latency, *PP* also increases the throughput of most significant results. Consider how many more significant results were produced by PP at each level to the alternatives. Compared to the alternative solutions, PP produced from 467% - 1444% more of the most significant results (m_1). While for the least significant results (i.e., m_3) the range is from 101% - 212%. In summary, compared to the alternative solutions, *PP* produced a greater quantity of the most significant results.

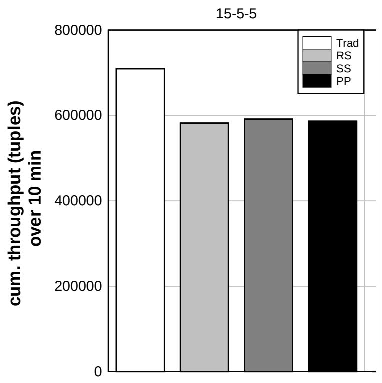
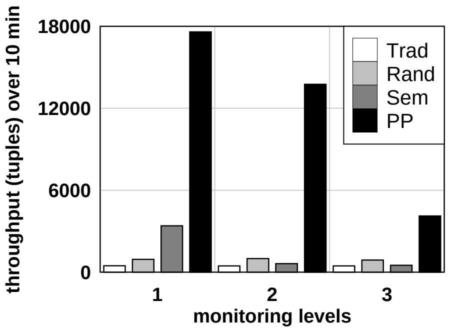


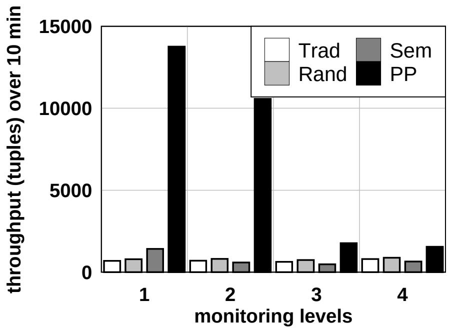
Figure 4.6: Overall Throughput of Results

Figure 4.5.2 shows the overall throughout of all query results (regardless of significance) using the experimental set up outlined above. Overall compared to the alternatives, *PP* achieved the lowest overall throughput. The number of results produced by PP, Sem and Rand are within 1 % of each other. Compared to Trad, PP produced 17 % fewer results. This is as expected. PP dedicates resources to ensuring that when resources are scarce that the most significant results are produced. The overhead to support PP takes away resources from processing insignificant tuples. This is by design. PP is designed for EMAs where they is a need to ensure that at all costs certain tuples are processed.

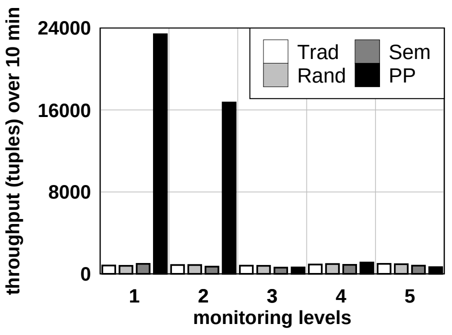
Varying the Number of Monitoring Levels We now vary the number of monitoring levels from 3 to 6, i.e., data-sets 5-5-5, 5-5-5-5, 5-5-5-5-5, and 5-5-5-5-5-5 (Fig. 4.7



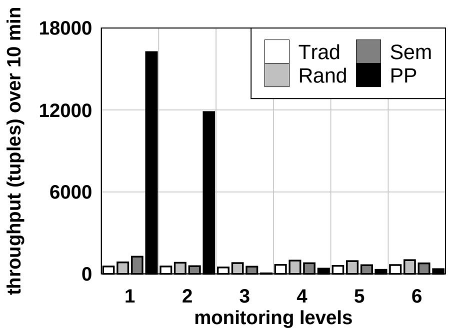
a) data-set 5-5-5 (l=7500000 ns)



b) data-set 5-5-5-5 (l=7500000 ns



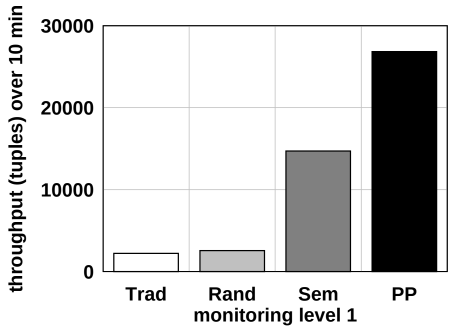
c) dataset 5-5-5-5-5 (l=7500000 ns)



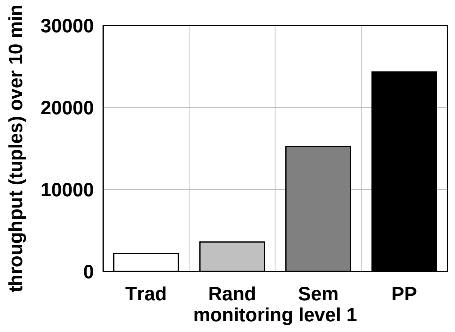
d) dataset 5-5-5-5-5-5 (l=7500000 ns)

Figure 4.7: Varying the Number of Monitoring Levels

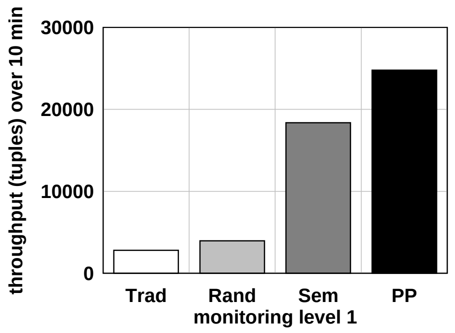
a-d). The percentage of incoming tuples in each promotable subset $= 5\%$ and lifespan $l = 7500000$ ns are constant. No matter how many monitoring levels exist, *PP* produces a markedly larger quantity of significant results. Clearly, the overhead of supporting many monitoring levels has little effect on *PP* producing a larger quantity of the most significant results than the other approaches. This is as expected as both *rand* and *sem* make a single coarse binary decision on resource allocation (Sec. 1.7.1). In contrast, *PP* efficiently pulls the most significant tuples (e.g., m_1) forward throughout the pipeline no matter how many other tuples may coexist in the same pipeline.



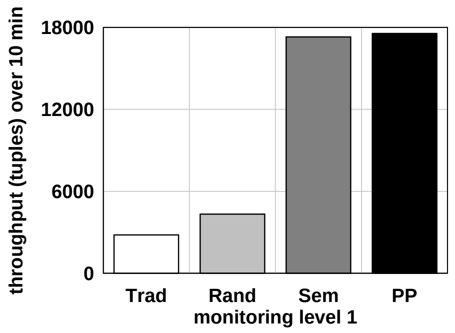
a) dataset 20-0-0 (l=7500000 ns)



b) dataset 30-0-0 (l=7500000 ns)



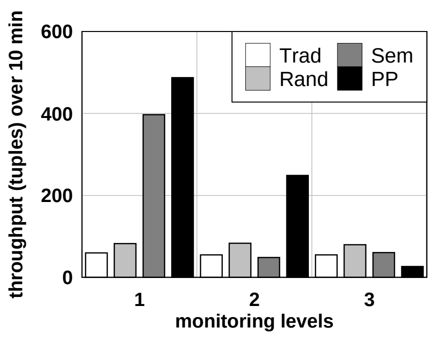
c) dataset 40-0-0 (l=7500000 ns)



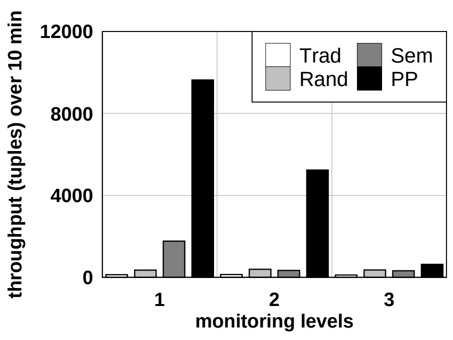
d) dataset 50-0-0 (l=7500000 ns)

Figure 4.8: Varying the Number of Incoming Tuples in each Promotable Subset

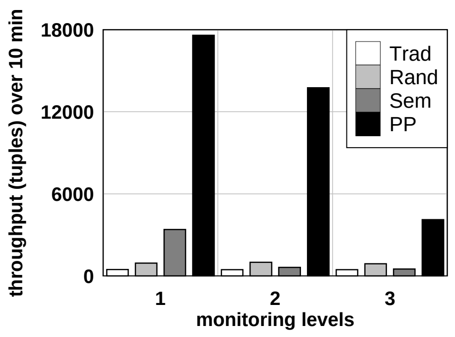
Varying the Number of incoming Tuples in each Promotable Subset Now, we vary the percentage of incoming tuples in each promotable subset from 20% to 50%, i.e., data-sets 20-0-0, 30-0-0, 40-0-0, and 50-0-0 (Fig. 4.8 a-d). The number of monitoring levels $= 1$ and lifespan $l = 7500000$ ns are constant. In all experiments, *PP* again increases the throughput of the most significant results compared to the other approaches. As the percentage of incoming tuples in each promotable subset increases, the throughput gains achieved by *PP* compared with the alternative solutions narrows. As shown, increasing the % of incoming tuples a promotable subset to a large portion of the stream data (e.g., 50%) also increases the likelihood of approaches like *rand* processing more significant tuples by chance. Thus, scenarios where the majority or the entire stream is highly significant would not benefit from deploying *PP*. The stream is so saturated with significant tuples that there are few insignificant tuples to pull them ahead of.



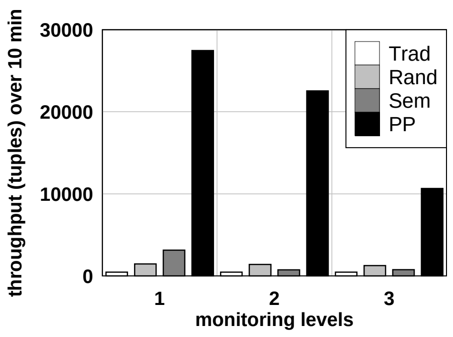
a) dataset 5-5-5 (l= 2500000 ns)



b) dataset 5-5-5 (l= 5000000 ns)



c) dataset 5-5-5 (l= 7500000 ns)



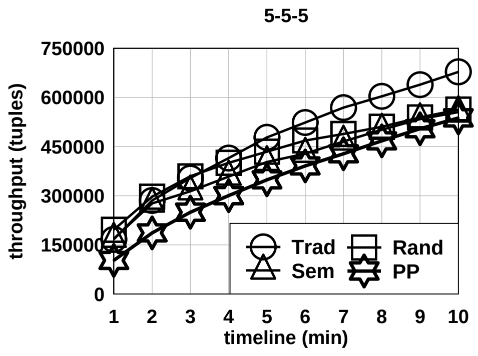
d) dataset 5-5-5 (l=10000000 ns)

Figure 4.9: Varying the Size of the Lifespan

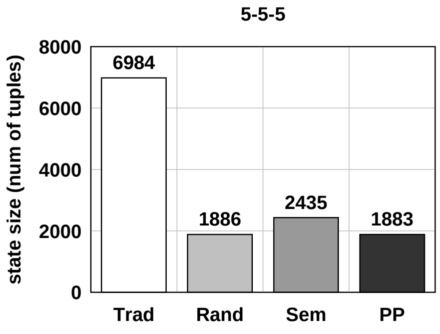
Varying the Size of the Lifespan We now vary the size of the lifespan, i.e., $l = 2500000$ ns, $l = 5000000$ ns, $l = 7500000$ ns, and $l = 10000000$ ns (Fig. 4.9 a-d). The number of monitoring levels ($= 3$) and % of incoming tuples in each promotable subset ($= 5\%$) are constant, i.e., data-set $= 5\text{-}5\text{-}5$. Compared to the alternative approaches, *PP* consistently has a higher throughput of most significant results. The throughput of most significant results produced by *PP* increases as the lifespan increases. These results were expected. As the lifespan increases, fewer significant tuples expire. In shedding, this increases the number of in-process less significant tuples. When a highly significant tuple t_i arrives, there will be more in-process less significant tuples that t_i will need to wait behind. In contrast, *PP* pulls the most significant tuples ahead of any less significant ones. In *PP* the latency of the most significant tuple is thus not affected by any in-process less significant tuples.

Overhead We now measure the execution-run-time CPU overhead by evaluating the cumulative throughput in the worst case for *PP* using data-set $5\text{-}5\text{-}5$ (Fig. 4.10 a). The worst case is when no tuples expire, i.e., $l = \infty$ and thus the PP Optimizer (Sec. 4.3.2) is never triggered. No monitoring levels will ever be activated. No tuples will be promoted. The overhead of *PP* and shedding is the cost to gather and evaluate run-time statistics. Namely, even if the system is never overloaded, these systems would continue to analyze run-time statistics. In this worst case, *PP* produced only slightly fewer results after 10 minutes than *sem* and *rand*, namely 3.1% and 4.5% respectively. Thus in summary, as our experiments above demonstrate the benefits outweighs *PP*'s minor overhead.

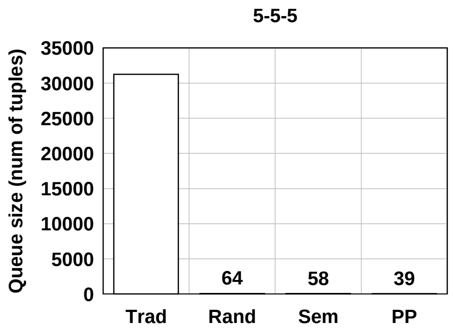
We now measure the memory overhead by assessing the average number of tuples in the join states and input queues when PP promotes tuples (Fig. 4.10 b & c). This experiment uses data-set $5\text{-}5\text{-}5$ and lifespan $l = 7500000$ ns. *Trad* has the most



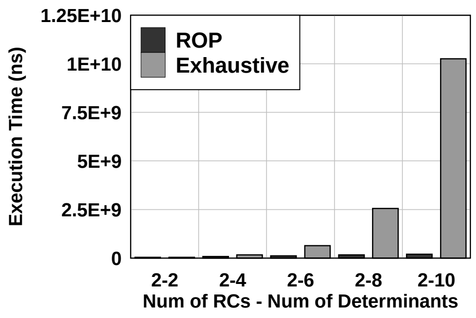
a) CPU Overhead: 5-5-5 $l = \infty$



b) Avg. State Size: 5-5-5 l=750000 ns



c) Avg. Queue Size: 5-5-5 l=750000 ns



d) Optimizer Search Time

Figure 4.10: Overhead & Effectiveness of ROP

tuples in its state and queue. It is unable to keep up with the volume of incoming tuples. Thus, they accumulate. If this were not true then promotion or shedding would not be required. All other methods compared to *trad* have negligible queue and state sizes because they reduce load to the point that they can keep up with the stream. *Rand* has roughly the same state size as *PP*. Although *PP*'s state contains more significant tuples (Sec. 4.2.1). This is confirmed by our experiments as *PP* produces more significant results. *Sem* has a larger state size than *PP*, roughly 29.3%

more. This is also expected. *Sem* never sheds any in-process tuples. Thus, although *sem*'s states contain many tuples, many of them will not be highly significant. This causes *sem* to devote more resources to producing less significant results which reduces the resources dedicated to creating the most significant results.

Effectiveness of ROP Now we evaluate the effectiveness of ROP. We compare the average search time of our Rank Order Pruning Algorithm (ROP) (Sec. 4.3.5) to the *exhaustive PP plan search* when varying the key factor as identified by our cost model (Thm. 5), namely, the number of priority determinants (Fig. 4.10 d). Both methods return the same optimal PP plan (Sec. 1.3.1) for the Home Arrest query (Sec. 1.3.1). Recall the PP plan (e.g., Fig. 4.1) for the Home Arrest query. These experiments use one monitoring level m_1. However, the number of priority determinants defined for m_1 varies from 2 to 10. ROP and *exhaustive* have roughly the same execution time when the number of determinants is equal to 2. Yet ROP takes significantly less execution time than the exhaustive approach as the number of determinants increases, roughly 1-20 fold. As expected, when the number of determinants increases, ROP exponentially performs better than the exhaustive method (Thm. 5).

Effects of the Selectivity of the Significant Tuples on the Throughput of Significant Results If the significant tuples in the stream are highly selective then some of these tuples will be dropped at an operator in the query pipeline before they produce query results. Compared to when the selectivity is low, in this case fewer resources will be required to process the significant tuples. This will reduce the overhead on the system. Given this reduction, it is likely that the majority of significant tuples will be processed within their lifespan. Thus, in this case, it is not likely that any tuples will be preferentially allocated resources. In addition, there will be adequate resources to produce all significant results.

There are many systems where it is critical to ensure the processing of certain tuples. In some of these systems, the selectivity of the significant tuples may not be known at compile-time. Reconsider the Home Arrest example. The selectivity of prisoner tuples who try to escape is not likely to be stable, periodic, or predictable. Prisoners are human beings with unpredictable behavior. However, as seen by real events [Pre10], these systems can get overloaded. Given how critical it is to process these significant tuples, such applications require a TP system.

Effects of the Data Set of the Significant Tuples chosen on the Throughput of Significant Results As stated above, the selectivity of the significant tuples effects whether or not any tuples will be preferentially allocated resources. In some cases, there will be adequate resources to produce all significant results. In other cases, preferential allocation of resources will ensure that the most significant tuples are processed. In such cases, resources will allocated first to ensuring that as many of the most significant results that can be produced are produced. The significant tuples in our data sets were randomly chosen. As shown by the throughput of the trad system utilizing these data sets a TP system is required. If this were not the case then Trad would have outperformed the other systems (similar to the overhead experiment).

In any cases where a TP system is required, PP ensures that resources are dedicated to producing the most significant results first. The throughput of most significant results produced by PP will depend upon the availability of resources. In our workload examples above the amount of available resources is variable. However in all cases, PP produces a higher quantity of the most significant query results compared to the competitors.

4.5.3 Summary of Experimental Findings

Key findings:

1) Whenever preferential resource allocation is required, *PP* consistently lowers latency and increases the throughput of significant results by many orders of magnitude compared to the state-of-the-art approaches.

2) *PP* is increasingly more effective than the state-of-the-art approaches when the number of monitoring levels or life-span expands.

3) *PP* is worse than the state-of-the-art approaches in scenarios where the majority of the stream population is significant.

4) Compared to the state-of-the-art shedding approaches, PP's CPU run-time overhead is negligible (roughly 3-4.5%).

5) ROP locates an optimal PP plan faster than exhaustive search. The time saved by ROP is significant if there are many priority determinants.

6) All PP experiments include the cost of promotion. Clearly the benefits of *PP* outweigh the minor overhead observed.

Chapter 5

Aggregation in Targeted Prioritized Data Streams

5.1 State-of-the-Art Aggregation

This chapter provides details on our work towards Task 2 (i.e., TP Aggregate Operator). The main goal of Task 2 is to design an aggregate operator that produces non-skewed aggregate results.

We focus on the aggregation operator in the TP context with the goal of producing reliable aggregate results from the significant tuples pulled forward by the TP.

Basic Aggregate Operator: Typically an *aggregate operator* [CKT08] computes a function over the set of tuples that belong to the same aggregate group within the current query window w_p of the data stream. The aggregate groups are defined by the user in the CQL query [ABW06]. For example, in our Stock Market Aggregate Query, the user chooses to group tuples by business sector. Incoming tuples are stored in the state and associated with their respective aggregate group that they contribute to. When it is determined that no future incoming tuples for the current window w_p will arrive then the aggregate result(s) are generated for each aggregate group in w_p.

Out-of-Order Aggregate Operator: In the TP context, aggregation causes new challenges. Some TP systems [WR11] (Ch. 4) actively cause tuples to become out-of-order by pulling some tuples ahead of others. Thus, TP aggregate operators require special support to know when all tuples from a window have been processed so that results could then be produced. Such special support for out-of-order result progression is not new, rather prior work [LLG$^+$09] addresses out-of-order issues caused by external causes such as network transmissions. Broadly, these methods synchronize the progression of time within the query pipeline to assure correctness and completeness of query processing [LLG$^+$09]. These existing methods can also be applied to support aggregate operators in TP.

We now explain this process in more detail. An aggregate operator should only produce results for window w_p when no tuples from window w_p will be processed in the future. To support this, [LMT$^+$05] proposed to use punctuations to trigger the creation of aggregate results whenever a window is complete. To quickly identify tuples within a given window, windows are divided into groups of tuples (a.k.a. *panes*) whose arrival times are a constant length portion of the query window [LMT$^+$05].

Each pane p_q is assigned a pane number. When a tuple arrives, it is associated with a pane number. This pane number determines which windows the tuple will produce results for. Each query window is composed of a set number of panes. Each aggregate result is only generated from tuples whose panes compose the specified query window w_p.

To discern when no tuples from pane p_q will be used to produce aggregate results in the future, the progress of tuples is tracked. Each leaf operator op_o periodically sends a punctuation when operator op_o has no more tuples from pane p_q to process. Upon receiving such a notification, the aggregate operator produces all results for the window whose set of panes ends with pane p_q. Thereafter, any tuples stored in the state within the panes that are guaranteed to not produce any aggregate results in the future are purged.

State-Of-the-Art TP Aggregate Operator: We now outline an aggregation operator using the state-of-the-art aggregation methodology outlined above. As seen in Section 1.7.3, the results produced by this operator may be skewed and unreliable.

```
Algorithm Aggregation Operator( $Q_p()$ pane complete punct. queue, $C_{avail}$ avail. res., $Q_{inc}(s_1)$ queues for stream $s_1$)
 1: while (($C_{avail}() > 0$) and ($Q_p()$ is not empty)) do
 2:     Punc ← first pane complete punctuation in $Q_p()$
 3:     create aggregate results that for window that ends with pane in Punc
 4:     place aggregate results into input queue of next query plan operator
 5:     purge state of tuples within panes that will not produce any future results
 6: end while

 7: LevelProcessed ← 1
 8: while (($C_{avail}() > 0$) and ($Q_{inc}(s_1)$ is not empty)) do
 9:     if ($Q_{inc}(s_1, LevelProcessed)$ contains a tuple) then
10:         Tup ← first tuple in $Q_{inc}(s_1, LevelProcessed)$
11:         store Tup in state and associate with the aggregate group and pane
12:     else
13:         LevelProcessed ← LevelProcessed + 1
14:         if LevelProcessed > max(Monitoring Level) then
15:             LevelProcessed ← Insignificant Tuples;
16:         end if
17:     end if
18: end while
```

Figure 5.1: State-Of-the-Art TP Aggregate Operator

The **State-Of-the-Art TP aggregation algorithm** works as follows (Fig. 5.1).

Each incoming notification punctuation states that no more tuples from pane p_x will arrive. Once a punctuation has been received, then aggregate results are created for the window whose last pane is pane p_x (line 3). These results are sent to the next operator (line 4). Finally, tuples stored in the state within panes that will not produce any future results are purged (line 5). This continues until either no resources or incoming punctuations remain (line 1).

If after processing all incoming punctuations, resources remain then starting from the most significant incoming queue tuple t_i is selected to be processed (line 10). Next tuple t_i is stored and associated with the aggregate group and pane of tuple t_i (line 11). After all tuples from one significance level have been processed, tuples in the next level are processed (lines 13-16). This continues until either no resources remain or all queues are empty (line 8).

5.2 TP-Ag Problem Definition

Our TP-Ag operator must meet the following requirements.

1). It must produce the most reliable aggregate result from the largest set of tuples that arrived at the aggregate operator and belong to selective sub group populations. That is, the set used to create each result must be estimated to represent the actual selected sub group populations of tuples that would have arrived at the aggregate operator if resources were available.

2). It must annotate each aggregate result with the sub group population(s) that the result is generated from.

3). It can not adjust which tuples the TP optimizer chose to process (Sec. 4.1).

That is, it can not ignore the desired resource allocation order specified by the user. It must build reliable aggregate results from the significant tuples already pulled forward.

5.3 TP Aggregation Foundation

5.3.1 Population Each Aggregate Result is Generated From

Per the requirements (Sec. 1.5), each aggregate result must be annotated with which tuples it is generated from. In TP, tuples chosen to be processed satisfy the membership criteria of an activated monitoring level (Sec. 4.1). Thus we propose to logically divide the population of tuples within an aggregate group and pane into subsets based upon their significance levels. Each population is divided into subsets, namely, one subset for each significance level and one for insignificant tuples.

We denote aggregate result a_i as $(val;\ g_j;\ w_k;\ rnk;\ ss_l)$. Here, val denotes the aggregate result value (e.g., average stock price 9.26 in Fig. 1.3). g_j denotes the aggregate group (e.g., g_1 in Fig. 1.3), w_k the query window, and rnk the significance level of the aggregate result a_i. Subsets flag ss_l annotates which tuples the aggregate result a_i is generated from. ss_l is represented as a bit vector with one bit for each monitoring level $lvl_1, lvl_2, ..., lvl_n$ and one bit for insignificant tuples. For instance, subsets flag $ss_1 = 100$ signifies that the aggregate result a_i is generated from only tuples at significance level 1. Subsets flag $ss_2 = 110$ signifies that the aggregate result a_i is generated from only tuples at significance levels 1 and 2. Finally, subsets flag $ss_3 = 111$ signifies that the aggregate result a_i is generated from all tuples (both significant and insignificant).

Traditionally, aggregate operators produce a single aggregate result for each group and window. It is possible for each subset in an aggregate group and window to create an aggregate result. Given the limited resources, we propose to follow the traditional method. We will combine multiple subsets in an aggregate group and window to create a single aggregate result. The goal of TR-Ag is to produce the single most reliable aggregate result from the largest number of subsets for an aggregate group. To achieve this, TR-Ag selectively choses which of the available subset(s) are used to create each aggregate result.

5.3.2 Sample Population Evaluation Strategies

Result Accuracy

Aggregate result a_i is generated from a sample population $spop_m$. The sample population $spop_m$ corresponds to the tuples that belong to the subset(s) in an aggregate group g_l and window w_k that were used to generate aggregate result a_i. Some tuples that belong to group g_l and window w_k may expire before reaching aggregate operator op_o and not be in the sample population $spop_m$. The actual population pop_m is the set of tuples in the aggregate group g_l and window w_k that given adequate resources should have reached aggregate operator op_o and been in the sample population $spop_m$. The sample population is often a subset of the actual population (i.e., $spop_m \subset pop_m$). The aggregate result a_i may not be correct, i.e., match the aggregate answer a_i^* ($a_i^* \neq a_i$). The aggregate answer a_i^* is the aggregate result generated from the actual population pop_m.

Reconsider the aggregate operator op_3 in Figure 1.3. The sample population $spop_1$ for aggregate group g_4 includes 987 tuples where 984 tuples have significance level 1 and 3 tuples have significance level 2. Some tuples may expire prior to reaching aggregate operator op_3 and belong to the actual population pop_1 for group g_4. The aggregate results produced by sample population $spop_1$ will be inaccurate if the sample population $spop_1$ does not accurately portray the actual population pop_1.

Sample Population Accuracy Determination Strategies

Comparing the Mean of the Sample and Actual Population: One method to determine if a sample population accurately portrays the actual population is to compare the mean of the sample and the mean of the actual population. The Hoeffding [Hoe63] inequality equation computes the probability that the mean result $\bar{x}$ for a sample population $spop_m$ deviates from the the expected mean answer μ by an error threshold ϵ or $Pr\{|\bar{x} - \mu| \geq \epsilon|\mu|\}$. Many aggregation operators [HHW97, OW00, LMT$^+$05] that process aggregate results from most (if not all) tuples in a given query window use the Hoeffding equation. However, these approaches seek to create accurate aggregate results from all tuples within specific windows. To achieve this, they limit the number of tuples dropped from such windows. That is, they adjust how resources are allocated. This adds an additional overhead at a time when resources are scarce. In addition, their resource allocation decisions will not reflect the desired resource allocation order specified by the user (Sec. 6.1.1).

We instead seek to build reliable aggregate results solely from the tuples pulled forward by controlling which of the available subset(s) are used to create an aggregate result. TP-Ag cannot use the Hoeffding equation to determine the accuracy of the sample population. The Hoeffding equation requires that the actual mean μ be precisely measured. TP cannot ensure that tuples do not expire before reaching TP-

Ag op_o. Thus to calculate the actual mean μ requires knowledge of which expired tuples will satisfy the query constraints of all operators prior to TP-Ag op_o. Such an evaluation would amount to running the full query. Clearly, this is prohibitively costly. We now explore a less costly approach.

Estimating the Required Sample Size: Another method to determine if a sample population accurately portrays the actual population is to estimate the sample size required to determine the actual mean within a given error threshold [Coc77]. If the size of sample population, denoted by $|spop_m|$, is less than the estimated required sample size $|spop^{est}|$, then the sample population $spop_m$ may not accurately represent the actual population pop_m. No aggregate results should be generated from such sample populations.

We propose to use the Cochran's sample size formula [Coc77]. It determines the sample size by considering the limits of the errors in the mean values of items in the sample population. $|pop_m|$ is the size of the actual population. ϵ is the user selected error rate. σ is the standard deviation of the actual population. z is the user selected confidence level or the estimated percentage of the values in the sample population within two standard deviations of the mean of the actual population. The Cochran's sample size formula [Coc77] is $|spop^{est}| = (z^2 * \sigma^2 * (|pop_m|/(|pop_m| - 1)))/(\epsilon^2 + ((z^2 * (\sigma^2)/(|pop_m| - 1))))$. Roughly, $z^2 * \sigma^2 * (|pop_m|/(|pop_m| - 1)))$ represents the percentage of tuples from the sample population are within the confidence interval of the estimated mean. $(\epsilon^2 + ((z^2 * (\sigma^2)/(|pop_m| - 1))))$ represents the percentage of tuples from the sample population that per the error rate must be within the confidence interval of the estimated mean.

The Cochran's sample size formula requires that the aggregate values in the population follow a normal distribution. While not all streaming data has this distribution, many practical streams are known to experience such fluctuations. One example are stock market prices. They can be mapped to the normal distribution [Fam65]. That is, the market can be viewed as a large number of independent or weakly dependent random events.

The standard deviation of the actual population σ is commonly calculated using the standard deviation of the sample population and Bessel's correction [Hoy88] as $\sigma = \sqrt{(1/(|spop_m| - 1)) * \sum_{x=1}^{|spop_m|} (x_i - \bar{x})^2}$. To estimate the actual population size $|pop_m^{est}|$, we thus must measure how many expired tuples would have reached TP-Ag operator op_i if given adequate resources, as further explained below.

5.3.3 Actual Population Size Measurement

The estimated actual population size $|pop_m^{est}|$ is the sum of the sample population size $|spop_m|$ and the estimated number of tuples that expire too early $|expTE(op_x, g_l, lvl_p)|$ over a window.

Reconsider the aggregate operator op_3 (Fig. 1.3). If tuple t_1 expires while waiting in the incoming queue to operator op_2 then tuple t_1 has satisfied the constraints of operator op_1 and all operators before operator op_1. However, tuple t_1 may still not satisfy the constraints of any subsequent operators. In other words, the probability that the expired tuple t_i at significance level lvl_p and aggregate group g_l would have reached aggregate operator op_o is the product of the probability that any tuple t_i at significance level lvl_p and aggregate group g_l that expires at operator op_x satisfies the constraints of all operators between operator op_x and aggregate operator op_o.

The probability of tuples at significance level lvl_p and aggregate group g_l that expire at operator op_x but if given adequate resources would have reached aggregate operator op_o is the product of the selectivity of tuples at significance level

lvl_p and aggregate group g_l for each operator between op_x and aggregate operator op_o, i.e., $P(op_x, op_o, lvl_p, g_l) = \prod_{op=op_x}^{operator\ precedes\ op_o} sel(op, lvl_p, g_l)$. The selectivity of tuples at significance level lvl_p and aggregate group g_l at operator op_o is denoted as $sel(op_x, lvl_p, g_l)$.

To compute the estimated number of tuples that expire too early $|expTE(op_x, g_l, w_k, lvl_p)|$, we track how many tuples expire at each operator op in the query path before aggregate operator op_o by group g_l, significance level lvl_p, and aggregate operator op_o or $exp(op, g_l, lvl_p)$. In addition, we also track the probability that a tuple that is processed by operator op will reach the aggregate operator op_o or $P(op, op_o, lvl_p)$. To determine how any tuples that expire at operator op but could have reached aggregate operator op_o, we simply compute the product of the probability of expired tuples reaching aggregate operator op_o and the number of tuples that expire at operator op or $P(op, op_o, lvl_p) * |exp(op, g_l, lvl_p)|$. Thus, the estimated number of tuples that expire too early $|expTE(op_x, g_l, w_k, lvl_p)|$ for group g_l, significance level lvl_p, and aggregate operator op_o is for each operator in the query path before aggregate operator op_o (i.e., $\sum\limits_{op=op_s}^{op_o}$) the sum of the product of the probability of expired tuples reaching aggregate operator op_o and the number of expired tuples (i.e., $P(op, op_o, lvl_p) * |exp(op, g_l, lvl_p)|$), i.e., $|expTE(op_x, g_l, lvl_p)| = \sum\limits_{op=op_s}^{op_o} (P(op, op_o, lvl_p) * |exp(op, g_l, lvl_p)|)$. The estimated number of tuples that expire over a window at operator op_x that belong to group g_l and significance level lvl_p is $|exp(op_x, g_l, lvl_p)|$.

5.3.4 Sample Population Selection Policy

The key idea of TP-Ag is that only sample populations representative of their actual populations are used to create aggregate results. The aggregate result a_i can be generated from a sample population that includes one or more subset(s) in the aggregate group g_l and window w_k. There are many ways of selecting which subset(s) are included. TP is based upon certain tuples being more significant than others. Thus, we propose to include in the sample population the largest number of reliable subsets in consecutive significance order. For example, aggregate result a_i could be created from only tuples at significance level 1, only tuples in significance levels 1 and 2, or all tuples.

Reconsider the aggregate operator op_3 (Fig. 1.3). An aggregate result could be created in the two subsets for group g_4. These subsets include 984 tuples at significance level 1 and 3 tuples at significance level 2 respectively. Assume that the estimated actual population size $|pop_m^{est}|$ for this sample population is 1984 tuples. The estimated standard deviation σ is 7.9. The error rate ϵ is .1. The critical standard score z is 1.96 (95% confidence level). Then the estimated required sample size $|spop^{est}|$ (i.e., $= (1.96^2 * 7.9^2 * (1984/(1984-1)))/(.1^2 + ((1.96^2 * (7.9^2)/(1984-1))))$) is 1832 tuples. The sample population contains 984 tuples. It is not large enough to create a reliable aggregate result , i.e, $984 < 1832$.

However, an aggregate result could be created from the sample population for group g_4 that only includes the 984 tuples at significance level 1. Assume that the estimated actual population size $|pop_m^{est}|$ for this sample population is 1000 tuples. The estimated standard deviation σ is 5.9. The error rate ϵ is .1. The critical standard score z is 1.96 (95% confidence level). Then the estimated required sample size $|spop^{est}|$ (i.e., $= (1.96^2 * 5.9^2 * (1000/(1000-1)))/(.1^2 + ((1.96^2 * (5.9^2)/(1000-1))))$) is 930 tuples. The sample population contains 984 tuples. It is large enough, i.e, $984 > 930$. Hence, an aggregate result will be produced for this sample population.

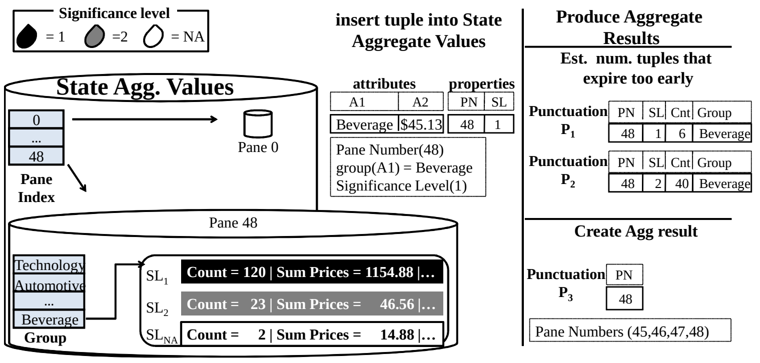


Figure 5.2: TP-Ag State Example

5.4 TP Aggregation Operator

We now propose our TP aggregate operator (or TP-Ag).

5.4.1 Tracking Expired Tuples

Periodically each operator op_x sends statistics on the selectivity of tuples $sel(op_x, lvl_p, g_l)$ at significance level lvl_p and aggregate group g_l at operator op_x to the TP Optimizer. From these statistics, the TP Optimizer informs each operator op_x in the query (Sec. 5.4.1) of the probability that tuples at significance level lvl_p from aggregate group g_l expire at operator op_x but if given adequate resources would have reached TP-Ag op_o.

Operators must also track the number of tuples that expire too early over a window by aggregate group and significance level (Sec. 5.3.3) and send this count to the TR-Ag operator. TR-Ag will use this count to estimate the actual population size when deciding whether or not a sample population should produce a result. To achieve this, TR-Ag uses an *expiration count punctuation*. Each operator tracks the estimated number of tuples that expire too early by pane, aggregate group, and significance level. When no tuples in a pane remain to be processed by an operator, the operator sends an expiration punctuation for each aggregate group and significance level where tuples expired. Upon receiving an expiration punctuation, each operator adds on their estimated number of tuples that expire too early. The expiration punctuation moves along the pipeline until it reaches the TP-Ag (Sec. 5.4.3). When, the expiration punctuations reach the TP-Ag they contain the number of tuples that expire too early over a window for each aggregate group and significance level.

Consider expiration punctuations $p_1 < 48, 1, 6, Beverage >$ and $p_2 < 48, 2, 40, Beverage >$ in Figure 5.2. Expiration punctuation p_1 states that 6 tuples from pane 48, significance level 1, and group Beverage are estimated to have expired too early. While expiration punctuation p_2 states that 40 tuples from pane 48, significance level 2, and group Beverage are estimated to have expired too early.

5.4.2 TP-Ag Physical Design

To support the production of aggregate results from certain subset(s) of the sample populations, TP-Ag must support the efficient look-up and purging (Sec. 5.4.4) of stored aggregate variables by pane (Sec. 5.1), group, and significance level.

State Design: TP does not always process tuples in arrival time order (Sec. 5.1). Thus, the state can contain tuples from more than one query window. That is, a multitude of tuples with the same aggregate group and significance level are likely to exist across multiple panes. However, the number of tuples within each pane is limited. Thus stored tuples are first grouped by the panes they belong to. Next, tuples are indexed by their aggregate group. Finally, tuples are stored by their significance level. Consider the insertion of stock tuple $t_1 < Beverages, \$45.13 >$ with pane 48 and significance level 1 into state aggregate values (Fig. 5.2). Stock tuple t_1 is stored in the state for pane 48, business sector $Beverages$, and significance level 1.

5.4.3 TP-Ag Algorithm

TP-Ag supports the production of aggregate results from representative sample populations that contain tuples from the largest number of consecutive significance levels in significance order.

Consider the production of aggregate results triggered by punctuation $p_2 < 48 >$ (Fig. 5.2). p_2 signals that all tuples from pane 48 have either expired or been processed by the operators that reside in the query pipeline prior to the aggregate operator. First, TP-Ag locates the group attributes for pane 48 which are Technology, Automotive, ..., and Beverage. Then for each group attribute (e.g., Beverage), it creates a sample population from all tuples from the window (composed of 4 panes) that ends with pane 48 (i.e., pane $45 - 48$). If the size of the sample population is greater than or equal to the estimated required sample size then TP-Ag creates an aggregate result for tuples from this sample population. Otherwise, TP-Ag creates a new sample population by removing the least significant subset from the current sample population. The process of testing the current sample population and creating a new sample population continues until either a result is generated or the sample population is empty. In the latter case, no aggregate result will be produced. Then TR-Ag moves to the next aggregate group for pane 48 (and so on...). For any result created, a subsets flag (Sec.5.3.2) that signifies which significance levels were in the sample population is added to the aggregate result tuple.

```
Algorithm TP-Ag Operator( $Q_{ep}()$ exp. punct. queue, $Q_{np}()$ not. punct. queue, $C_{avail}$ avail. res., $Q_{inc}(s_1)$
queues for stream $s_1$)
 1: while (($C_{avail}() > 0$) and ($Q_{np}()$ is not empty)) do
 2:     Punc ← first punctuation in $Q_{np}()$
 3:     while (($Q_{ep}()$ is not empty) and (first punctuation in $Q_{ep}$ = pane in Punc)) do
 4:         ExpPunc ← first punctuation in $Q_{ep}()$
 5:         store the values in ExpPunc
 6:     end while
 7:     popFlag ← to a bit of length num promotion levels +1 and each bit = 1
 8:     for each group $g_l$ in defined by the pane in Punc do
 9:         while (no result has been produced) and (significant subsets can be removed) do
10:             sample population ← population defined by the pane in Punc and group $g_l$ and subsets in popFlag
11:             if sample population represents actual population then
12:                 create aggregate results for sample population
13:                 add the popFlag to the result
14:                 place aggregate results into input queue of next query plan operator
15:             else
16:                 set the last bit in popFlag that is a 1 to a 0
17:             end if
18:         end while
19:     end for
20:     purge state of tuples within panes that will not produce any future results
21: end while
22: LevelProcessed ← 1
23: while (($C_{avail}() > 0$) and ($Q_{inc}(s_1)$ is not empty)) do
24:     if ($Q_{inc}(s_1, LevelProcessed)$ contains a tuple) then
25:         Tup ← first tuple in $Q_{inc}(s_1, LevelProcessed)$
26:         store Tup in state and associate with the pane, aggregate group, and significance level
27:     else
28:         LevelProcessed ← LevelProcessed + 1
29:         if LevelProcessed > max(Monitoring Level) then
30:             LevelProcessed ← Insignificant tuples;
31:         end if
32:     end if
33: end while
```

Figure 5.3: TP Aggregation Operator *TP-Ag*

PP-Ag algorithm works as follows (Fig. 5.3). For each incoming notification punctuation to the aggregate operator, first any incoming expiration punctuation for the same pane is stored (lines 3-6). Then, the groups in the window pane are identified (line 8). Next, for each group, we test to see if the sample population for all tuples represents the actual populations (line 11). If so, results are created, flagged with the significance levels of tuples in the sample population, and sent to the next operator (lines 12-14). Otherwise, the sample population is reduced by tuples that belong to the least significant subset (line 16) and the test starts over (line 11). This continues until either a result is produced or all sample populations have been explored (line 9). Then, results for the next group are produced. This continues until either no resources remain or there are no more incoming punctuations (line 1). Finally, tuples stored in the state within panes that will not produce any future results are purged (line 20).

If after processing all punctuations resources remain then starting from the most significant incoming queue, tuple t_i is selected to be processed (line 25). Next, tuple t_i is stored and associated with the pane, aggregate group, and significance level of tuple t_i (line 26) (Sec. 5.4.2). Once all tuples from one level are processed, then tuples in the next level are processed (lines 28-30). This continues until either no resources remain or all queues are empty (line 23).

5.4.4 Memory Resource Management

Beyond CPU resources, memory resources may also be limited.

State Management: To ensure complete results, tuples stored in states are not purged if they may create aggregate results in the future (Sec. 5.1). However, this purging method assumes that sufficient memory is available to store all tuples that will create future aggregate results. This may not always be the case. In the case of insufficient memory, we propose to purges tuples from the oldest panes in the state first. Our approach is based upon the fact that the majority of aggregate results generated from the oldest tuples would have already been produced. Memory resources are allocated to storing the freshest tuples.

Queue Management: In the case of insufficient memory, the incoming queues (Sec. 4.1.3) must also be purged. We also utilize the oldest pane method defined above to purge the queues.

5.5 Experimental Evaluation of TP-Ag

5.5.1 Experimental Set Up

Alternative Solutions. We compare TP-Ag (or TP w/ TP-Ag) to the state-of-the-art aggregate operators in TP systems. That is, we compare to the out-of-order aggregate operator [LMT^{+}05] implemented in PP (or PP) [WR11] (Ch. 4) and the data stream aggregation operator [CKT08] implemented in semantic (or sem) and random (or rand). PP requires the out-of-order aggregate operator as PP processes tuples out of arrival time order (Sec. 5.1). Sem and rand do not require any punctuations to trigger the creation of aggregate results because they process tuples in FIFO order. We also compare to the traditional aggregate operator [CKT08] implemented in a non-targeted prioritized data stream systems (i.e., a data stream system that processes tuples in FIFO order) (or trad). We compare to trad to demonstrate that our experimental scenarios require a TP system. TP-Ag uses the critical standard score $z = 1.96$ (95% confidence level).

TP w/ TP-Ag, PP, and sem use the same monitoring level criteria to select the tuples processed. Rand randomly selects tuples to process in FIFO order based upon the estimated number of tuples that can be processed within their lifespan. Trad simply processes all tuples in FIFO order. It neither drops nor allocates resources to tuples based upon their significance.

Data Streams and Query. The experiments use the Stock Market Aggregate Query in Section 1.3.2. There are three monitoring levels (Sec. 6.1.1) used to pull significant tuples forward.

The stock market stream was created from stock ticker information on the S&P 500 stocks gathered over July 18, 2012 via Yahoo Finance [Yah]. A selection committee from Standard & Poor's determines which of the 500 leading companies publicly traded in the U.S. stock market are in the S&P 500. It is considered to be a good model of how well the U.S. economy is doing.

News and blog data streams were created by randomly selecting sectors from the global industry classification standard (GICS). GICS, developed by Morgan Stanley Capital International (MSCI) and Standard & Poorś, contains 10 sectors that categorize the S&P 500 stocks. Most experiments use Data Set 1 which mimics the monitoring levels of a financial company monitoring diversified mutual funds. Given the stock market's ability to change rapidly, the stocks in many mutual funds are diversified. That is, the stocks chosen to belong to a fund are distributed across different business sectors and investment types (i.e., aggressive versus conservative investments). It is highly unlikely that all stocks in a mutual fund from a GICS sector will be from aggressive (level 1) or conservative investments (level 2). It is equally improbable that a financial company would be evaluating all stocks from one sector (level 3). Thus, Data Set 1 was created by randomly selecting 5% of the 500 stocks (or 25 stocks) to have genuine significance at each of the three levels.

Hardware. All experiments are conducted on nodes in a cluster that consists of 20 processing nodes. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory.

Metrics and Measurements. PP, sem, rand, and trad generate aggregate results from a sample population. Each sample population includes all tuples that arrive at the aggregate operator. In PP and sem, only the most significant tuples reach the aggregate operator, while in rand and trad, both insignificant and significant tuples are equally likely to reach the aggregate operator. TP w/ TP-Ag produces an aggregate result from the sample population whose size is sufficient based upon the required sample size. To validate whether or not the results are correct, each aggregate result produced by any of these methods is annotated with the significant levels of the tuples in the sample population (i.e., a subsets flag (Sec. 5.3.2)).

There are *eight possible sample populations* for aggregate results produced by these methods. That is, the sample population could contain tuples at significance level 1, tuples at significance level 2, tuples at significance level 3, tuples at significance levels 1 and 2, tuples at significance levels 1 and 3, tuples at significance levels 2 and 3, tuples at significance levels 1 2 and 3, or all tuples.

For each experiment, the actual aggregate answers (Sec. 5.3.2) for each of the eight possible sample populations listed above were found. That is, for each possible sample populations the query was executed through the trad system using a dataset that only contains tuples from the given sample populations. Then, for each query execution the tuples in the input stream is adjusted so that they belong to a sample population $spop_m$ in the eight possible sample populations. To generate the accurate results, the query lifespan is set to ∞. Thus, no tuples expire and the actual aggregate answers are generated from all tuples in the stream. These aggregate results are the actual aggregate answers. We stored each accurate result with their the group, window, and subsets flag.

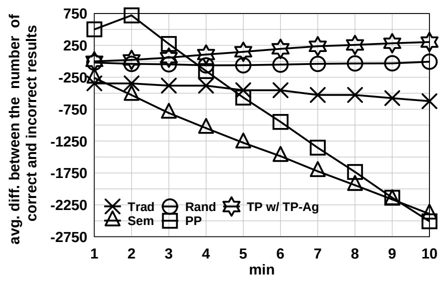
The experiments were run 3 times for 10 minutes. The results are the average of these runs. Our experiments measure for all aggregate results produced the percentage of correct aggregate results. Each aggregate answer produced is compared to the actual aggregate answer for the same group, window, and subsets flag. Any result that is within 5% of the actual answer is considered to be correct.

The goal of TP-Ag is to improve the percentage of aggregate results produced that are correct. Thus, this is what we measure.

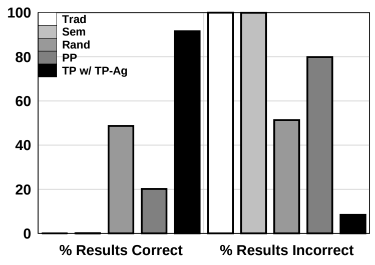
Methodology. We explore the following: 1) Is TP-Ag more effective at producing a larger percentage of correct significant aggregate results than the state-of-the-art solutions? 2) What effect does the number of significant tuples that belong to each aggregate group have on the effectiveness of the TP-Ag strategy compared to the state-of-the-art solutions? 3) How do changes in the error rate (Sec. 5.3.2) affect the percentage of correct results produced by TP-Ag? 4) What is TP-Ag's runtime CPU and memory overhead in the worst case scenario compared to the state-of-the-art solutions?

We vary the *number of significant tuples that belong to each aggregate group* and the *error rate* as they directly affect TP-Ag. When the *number of significant tuples that belong to each aggregate group* decreases, this reduces the number of tuples in each sample population. The *smaller the sample population* is the more likely that the result produced may be skewed. Consider a significant tuple t_i that expires before reaching the aggregate operator. Sample population $spop_m$ is the sample population that tuple t_i would have belonged to if tuple t_i had not expired. The aggregate result produced by sample population $spop_m$ will be more affected if the sample population $spop_m$ contains few tuples (smaller population) rather than many tuples (larger population). Decreasing the *error rate* increases the accuracy in the estimated required sample size. This should in turn increase the percentage of correct aggregate results produced by TP-Ag. These variables affect TP-Ag's ability to produce accurate results. Thus, we experiment by varying them.

5.5.2 Experimental Results



a) Average difference between the number of correct and incorrect aggregate results produced



b) Average % Correct Significant Results Over 10 Min

Figure 5.4: Effectiveness at Increasing the % of Correct Aggregate Results

Effectiveness at Increasing the Percentage of Correct Aggregate Results Produced. First, we compare the percentage of correct aggregate results produced by each approach. This experiment uses Data Set 1 and Stock Market Aggregate Query where the query lifespan = 1,000,000 ns and the window size = 500 tuples. Figure 5.4 a shows the average difference between the number of the correct and incorrect aggregate results produced at each minute. This measures whether more correct (if number is positive) or incorrect results were produced (if number is positive). Overall TP w/ TP-Ag compared to sem, rand, and trad consistently produces more correct aggregate results.

PP produced more correct aggregate results than TP w/ TP-Ag at startup (minutes 1 through 3). However, after the system start-up (minutes 4 through 10) PP produced a larger number of incorrect aggregate results than correct aggregate results. This is as expected. Namely, PP has less overhead than TP w/ TP-Ag. In addition, the aggregate results produced by PP will only be incorrect when significant tuples expire. This only occurs after the queues are full.

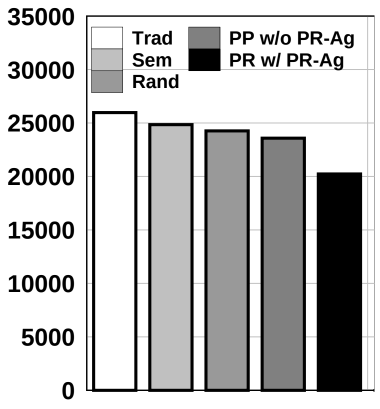
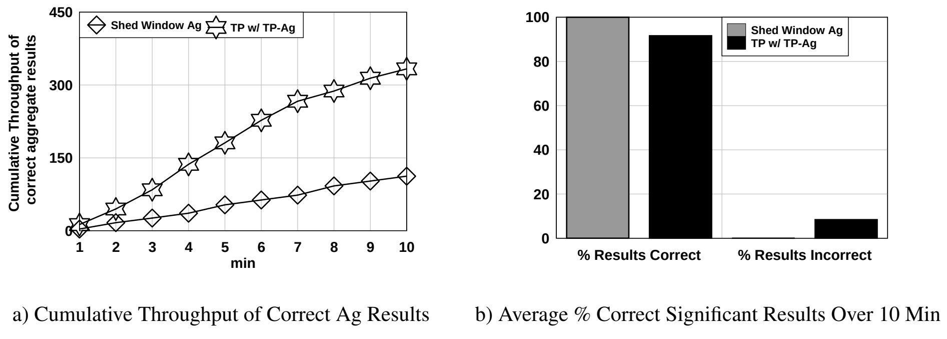


Figure 5.5: Overall Throughput of Correct Results

As the overall percentage of correct and incorrect significant results in Figure 5.4 b shows, compared to all alternative solutions, TP w/ TP-Ag produced a much higher percentage of correct aggregate results. Of all the aggregate results produced by TP w/ TP-Ag, 91.5% were correct. The percentage of correct aggregate results produced by trad, sem, rand, and PP respectively was 0.0%, 0.09%, 48.6%, and 20.1%. Our results support that TP w/ TP-Ag is effective at increasing the percentage of correct aggregate results produced compared to competitor solutions in scenarios that require a TP system.

Figure 5.5.2 shows the overall throughout of all correct aggregate results (regardless of significance) using the experimental set up outlined above. Overall compared to the alternatives, *PP* achieved the lowest overall throughput of correct aggregate results. Compared to PP, TP w/ PR-Ag produced 14% fewer correct results. TP w/ PR-Ag produced 16% fewer correct results than rand. It produced 18% fewer correct results than sem. Finally, compared to trad, TP w/ PR-Ag produced 22 % fewer correct results. This is as expected. TP-Ag requires additional overhead. This takes away resources from producing insignificant aggregate results. This is by design. TP-Ag is designed for EMAs where they is a need to ensure that at all costs certain tuples are processed.



a) Cumulative Throughput of Correct Ag Results b) Average % Correct Significant Results Over 10 Min

Figure 5.6: TP w/ TP-Ag versus State-of-the-art

TP w/ TP-Ag versus State-of-the-art. We now compare TP w/ TP-Ag to state-of-the-art aggregate operators for TP systems [BDM04, TZ06]. These systems limit which tuples are dropped from specific windows. We refer to these systems as *Shed Window Ag*. We implemented Shed Window Ag in CAPE [RDS$^+$04]. First, we compare the percentage of correct aggregate results produced by each approach. This experiment also uses Data Set 1 and Stock Market Aggregate Query where the query lifespan = 1,000,000 ns and the window size = 500 tuples.

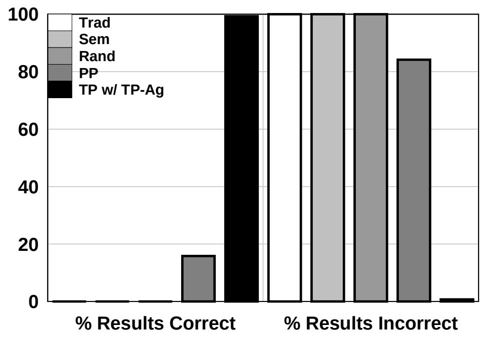
As the overall percentage of correct and incorrect significant results in Figure 5.6 b shows, all aggregate results produced by Shed Window Ag were correct. Of the aggregate results produced by TP w/ TP-Ag produced 91.5% were correct. Clearly, Shed Window Ag will always produce correct aggregate results. Recall that Shed Window Ag will ensure that no tuples from specific windows are dropped or expire. As a result, Shed Window Ag will only produce correct aggregate results. In contrast, TP w/ TP-Ag seeks to produce results that are estimated to be correct from incomplete windows of tuples.

However, Shed Window Ag may not produce as many aggregate results as TP w/ TP-Ag. As Figure 5.6 a shows, TP w/ TP-Ag produced roughly 2.9 fold more correct aggregate results than Shed Window Ag. Shed Window Ag will process all tuples (both significant and insignificant) from selected windows. This requires a significant amount of CPU overhead. Hence, Shed Window Ag will not produce as many correct aggregate results as TP w/ TP-Ag.

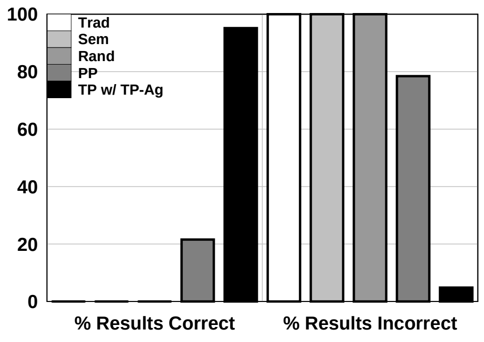
Clearly, Shed Window Ag and TP w/ TP-Ag have different goals. The goal of Shed Window Ag is to produce correct aggregate results by adjusting how resources are allocated. The goal of TP w/ TP-Ag is to build reliable aggregate results from the significant tuples pulled forward by the TP. Thus, henceforth we no longer compare TP w/ TP-Ag to Shed Window Ag.

Varying the Sample Population Size. We now explore how the number of the significant tuples in each aggregate group affects TP-Ag. This experiment uses the Stock Market Aggregate Query where the query lifespan = 1,000,000 ns and the window size = 500 tuples. All significant tuples belong to two GICS sector groups. This experiment uses four Data Sets (i.e., DS25, DS50, DS75, and DS100). Each Data Set adapts the percentage of significant tuples that belong to the two GICS sector groups. In DS25, 25% of the stocks in the two sectors are significant. That is, 75% of the tuples in the two sectors are insignificant. Similarly, in DS50, DS75, and DS100, respectively 50%, 75% and 100% of the tuples in the two sectors are significant. This causes the sample population size to vary.

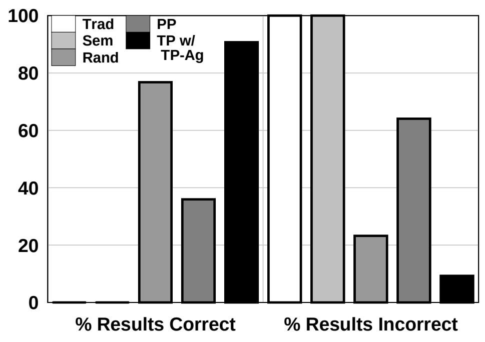
Figures 5.7 a-d show the overall percentage of correct and incorrect aggregate



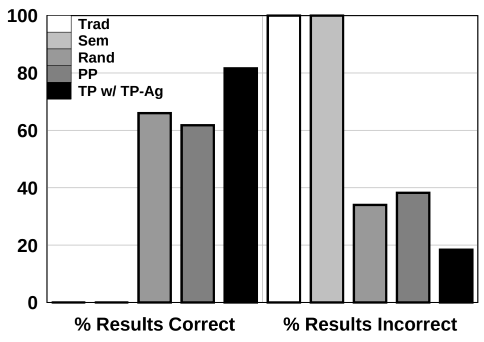
a) Data Set DS25



b) Data Set DS50



c) Data Set DS75



d) Data Set DS100

Figure 5.7: Varying the Sample Population Size

results respectively for DS25, DS50, DS75, and DS100. As can be seen, compared to the alternative solutions, TP w/ TP-Ag produced the highest percentage of correct aggregate results. The closest competitors were rand and PP. In DS25 (where the sample populations are the smallest for the two groups), TP w/ TP-Ag produced 100% and 84.0% more correct aggregate results than rand and PP. In DS100 (where the sample populations are the largest for the two groups), TP w/ TP-Ag produced 19.1% and 24.2% more correct aggregate results than rand and PP. This is as expected. Namely, TP-Ag achieves the highest gains when few tuples in the sample population expire. Fewer tuples will expire when there are fewer significant tuples in the stream (visa versa). TP-Ag is best suited for environments where the stream is not saturated with significant tuples. When it is saturated, most tuples are significant and thus more significant tuples are likely to expire.

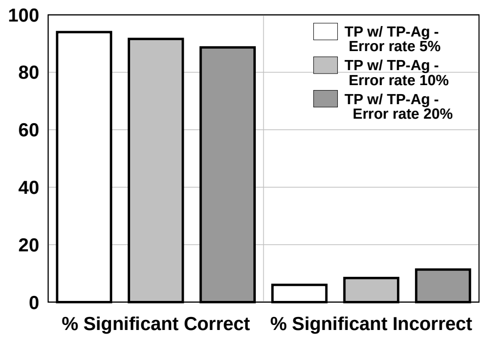


Figure 5.8: Varying the Error Rate

Varying Error Rate. Now, we compare the percentage of correct aggregate results produced by TP w/ TP-Ag when the error rate (i.e., the desired level of precision ϵ of Cochran's sample size formula (Sec. 5.3.2)) varies. This experiment uses Data Set 1 and Stock Market Aggregate Query where the query lifespan = 1,000,000 ns and the window size = 500 tuples. We vary the error rate ϵ from 5%, 10%, to 20%. Figure 8 shows the percentage of correct and incorrect aggregate results produced. Overall the highest percentage of correct aggregate results was produced when the error rate ϵ is 5%. While the lowest percentage was produced when the error rate ϵ was 20%. The percentage of correct aggregate results produced by TP w/ TP-Ag for the error rate ϵ from 5%, 10%, to 20% was respectively 93.9%, 91.5%, and 88.6%. As expected, decreasing the error rate (i.e., higher level of precision of Cochran's sample size formula) increases the percentage of correct aggregate results achieved by TP w/ TP-Ag (vice versa).

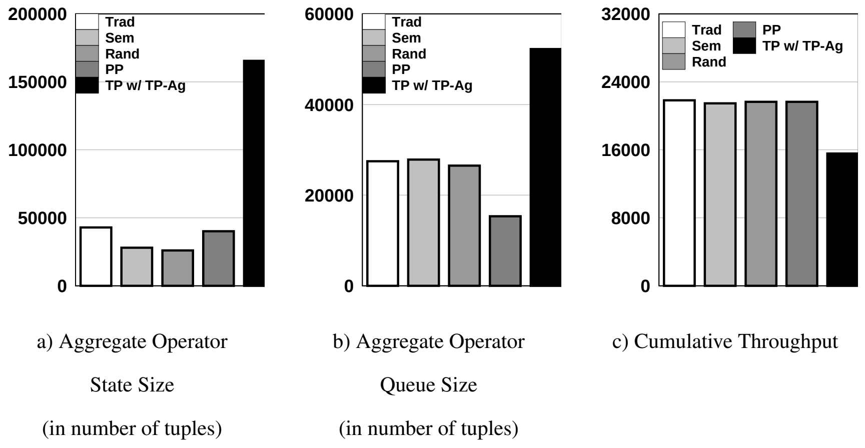


Figure 5.9: Memory & Execution-Runtime CPU Overhead

Execution-Runtime CPU Overhead. To measure the runtime overhead we evaluate the cumulative throughput using the worst case scenario for TP w/ TP-Ag. In the worst case scenario, no tuples expire (i.e., query lifespan $=\infty$) (Fig. 4.8 c). As a consequence, for each aggregate result we always end up working with the same sample population. Specifically, each aggregate result is produced from a sample population that contains all tuples in a window and aggregate group. The overhead of TP systems is the cost to gather and evaluate runtime statistics. Even if these systems are never overloaded, they continue to evaluate how to best allocate resources. In addition, TP-Ag has the additional overhead of tracking statistics (Sec. 5.4.1) to estimate the actual population (Sec. 5.3.3), evaluating the required sample size (Sec. 5.3.2), and determining if there is a sample population for each group and window whose size is comparable to the required sample size (Sec. 5.3.4). This experiment uses the Stock Market Aggregate Query where the query lifespan is ∞ and the window size is 250 tuples.

As can be seen in our results, the difference between the throughput of TP w/ TP-Ag and trad, sem, rand, and PP is respectively 40.2%, 37.9%, 39.1%, and 39.0%. For systems with extremely limited resources, TP w/ TP-Ag may not be a good approach. However, TP w/ TP-Ag is a great fit for systems that require a TP system and desire reliable accuracy in the aggregate results produced.

Memory Overhead. To measure the memory overhead we evaluated the average number of tuples in the state and input queue of the aggregate operator using the worst case scenario for TP w/ TP-Ag (outline above) (Fig. 4.8 a & b). As our results demonstrate, the memory overhead of TP w/ TP-Ag is higher than the current state-of-the-art approaches. The state of the aggregate operators in trad, sem, rand, and PP respectively have 74.0%, 83.0%, 84.2%, and 75.7% less tuples in their states than TP w/ TP-Ag. The queues of the aggregate operators in trad, sem, rand, and PP respectively have 47.4%, 46.6%, 49.2%, and 70.6% less tuples in their queues than TP w/ TP-Ag. This is as expected. Namely, the TP-Ag design relies upon a memory-intensive physical design to support the production of results from subsets of the actual sample population. Again, TP w/TP-Ag is a great fit for systems that require a TP system and desire reliable accuracy in the aggregate results produced. Ensuring the production of reliable aggregate results however carries an overhead.

5.5.3 Summary of Experimental Findings

We now summarize our key findings.

1) TP-Ag is effective at increasing the percentage of correct aggregate results produced in TPs (TP-Ag produces up to 91% more correct aggregate results).

2) Decreasing the error rate increases the percentage of correct aggregate results

achieved by TP w/ TP-Ag and vice versa.

3) TP-Ag is best suited for environments where the stream is not saturated with significant tuples. When the stream is saturated with significant tuples, more significant tuples are likely to expire.

4) TP w/TP-Ag is a great fit for systems that require a targeted prioritized data stream system and desire reliable accuracy in the aggregate results produced.

Chapter 6

Utilizing Dynamic Precedence Criteria For Producing Significant Results

6.1 PR Optimization Problem

This chapter provides details on our work towards Task 3 (i.e., Preferential Results). The main goal of Task 3 is to design a TP framework that pulls the most significant and promising tuples are pulled ahead of less significant and promising ones.

6.1.1 PR Queries

In the PR model, a set of P-CQL queries $\{q_1,\ldots q_j\}$ process continuous streams $\{s_1,\ldots s_n\}$ of tuples (Symbols in Table 6.1). Each P-CQL query is a CQL query

[ABW06] extended to support a processing time limit and multi-tiered monitoring criteria. Recall the P-CQL extension to the *Stock Market Queries* (Sec. 6.1.1).

Notation	*Meaning*
t_i	a tuple
$t_i.srnk$	significant rank of t_i
$t_i.prnk$	promising rank of t_i
$t_i.op_c$	designated operator of t_i
$t_i.rnk$	rank of t_i (max rank of $t_i.srnk$ or $t_i.prnk$)
q_j	a query
$q_j.lf$	lifespan of query q_j
$q_j.SML$	set of static monitoring levels of query q_j
sml_k	a static monitoring level in $q_j.SML$
$sml_k.srnk$	significant rank of sml_k
$sml_k.mem$	membership criteria of sml_k
DML	set of dynamic monitoring levels
dml_l	a dynamic monitoring level in DML
$dml_l.s_x$	stream that tuples must reside in for dml_l
$dml_l.op_c$	designated operator for dml_l
$dml_l.prnk$	promising rank of dml_l
$dml_l.mem$	membership criteria of dml_l
p_m^{pr}	a PR query plan
$p_m^{pr}.A_{SML}$	set of activated static monitoring levels in p_m^{pr}
$p_m^{pr}.A_{DML}$	set of activated dynamic monitoring levels in p_m^{pr}
s_n	a stream
op_o	an operator
$ER_s(p_m^{pr}, srnk)$	expir. rate of poten. sig. tuples at sig. rank $srnk$ in p_m^{pr}
$ER_p(p_m^{pr}, prnk)$	expir. rate of poten. prom. tuples at prom. rank $prnk$ in p_m^{pr}

Table 6.1: Notations for PR Query Plans.

The special clauses added to CQL to form P-CQL include:

• The *lifespan* clause $q_j.lf$ indicates the time limit for processing a tuple. That is, the query results generated by tuple t_i are only valuable to the receiving application if they are received within the query lifespan. For this reason, if the time spent processing tuple t_i exceeds the lifespan then t_i *expires*, i.e., is no longer processed.

• The *rank* and *criteria* clauses together specify the user's preferences about which results would be preferred over other results when resources are scarce. These clauses are specified as a part of the query, i.e., at compile-time. Hence, they are referred to as *static monitoring levels* $q_j.SML$. Each static monitoring level sml_k consists of a *significant rank* $sml_k.srnk$ and *membership criteria* $sml_k.mem$. The significant rank $sml_k.srnk$ denotes the degree of sml_k's significance. The static monitoring level sml_k is more significant than level sml_l if $sml_k.srnk < sml_l.srnk$.

Consider the Stock Market Join query (Sec. 1.3.2) with the P-CQL extension (Sec. 4.1.1). The query lifespan is 1,000 milliseconds. Static monitoring levels sml_1, sml_2, and sml_3 respectively identify aggressive investments, conservative investments, and stocks under evaluation. Static monitoring level sml_1 is more significant than static monitoring level sml_3, i.e., $(sml_1.srnk = 1) \wedge (sml_3.srnk = 3)$ thus $(sml_1.srnk < sml_3.srnk)$.

The optimizer periodically selects which static monitoring levels are used to identify the tuples to pull forward. These currently selected monitoring levels are referred to as being *activated* denoted by A_{sml}. If no resource shortage exists then no monitoring levels would be activated. Tuples will then be processed in FIFO order. However, if a resource shortage arises, then some monitoring levels would be activated and tuples will be processed in significance order based on guidance derived from these activated priorities.

6.1.2 Significant Tuples

Tuples that satisfy the membership criteria of an activated static monitoring level are said to be *significant* tuples (Ch. 4).

Definition 6 *A significant tuple t_i is designated with a significant rank $t_i.srnk$ which corresponds to the most significant of all the activated static monitoring levels that*

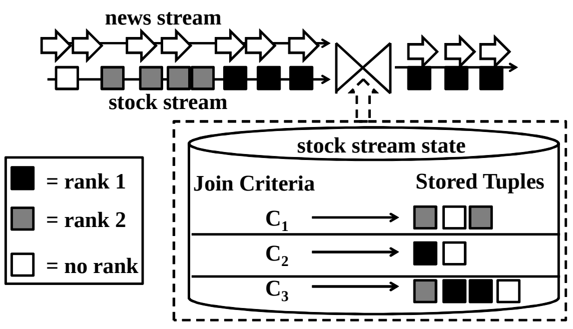


Figure 6.1: Estimated Significant Tuples Example

t_i satisfies the criteria of.

Consider stock tuple t_i that is both an aggressive investment (i.e., $sml_1.mem(t_i) = true$) and is under evaluation (i.e., $sml_3.mem(t_i) = true$). A_{SML} contains static monitoring levels 1 and 3, i.e., $A_{SML} = \{sml_1, sml_3\}$. Tuple t_i's significant rank is thus 1, i.e., $t_i.srnk = 1$.

6.1.3 Promising Tuples

Tuples likely to create significant query results by joining with significant tuples at a join operator are called *promising* tuples. To explain this, let us consider a symmetric binary hash join operator op_i [WA91] that combines tuples from streams s_1 (e.g., news stream) and s_2 (e.g., stock stream). This join operator op_i uses a state data structure to store incoming tuples for the news s_1 and stock s_2 streams in the news and stock stream state respectively. Results are created by combining an incoming tuple t_i from one stream (e.g., news stream) with matching tuples t_j in the state for the other stream (e.g., stock stream state) based upon the join criteria.

Consider news tuple t_i and the two stock tuples stored in the stock state that satisfy join criteria c_2 (with business sector = $Advertising$) (Fig. 6.1). One of the stock tuples from the advertising business sector is a significant tuple, while the other is not. If news tuple t_i is from the advertising business sector (i.e., satisfies the join criteria c_2) then t_i will be a promising tuple. That is, t_i has a high chance to produce a significant join result when it joins with the significant stock tuple from the advertising business sector in the stock stream state. However, this news tuple t_i may also join with the insignificant stock tuples from the advertising business sector and thus produce insignificant join results.

Observation: If a tuple is promising or not may *change during processing*. Significant tuples can produce significant query results on their own and thus are *significant for the entire query pipeline*. Thus if stock tuple t_i is an aggressive investment then it retains its significance throughout the pipeline (Fig. 6.1). In contrast, promising tuples are only promising because of their potential to join with significant tuples at a future join operator op_o. After proceeding past this operator op_o they may no longer have any known potential of producing significant query results. Hence after they have been processed by operator op_o, these promising tuples should no longer be preferentially allocated resources. They are *only promising for a portion of the query pipeline*, namely, until they reach the operator op_o.

Reconsider Figure 6.1. Consider news tuple t_j from the advertising business sector (i.e., join criteria c_2). In this case, tuple t_j has the potential to join with a significant stock tuple from the advertising business sector due to the existence of such a tuple in the stock stream state. Thus tuple t_j is a promising tuple only until it reaches the join operator in Figure 6.1.

The set of *dynamic monitoring level*s (DML) are constructed at run-time by the optimizer to indicate the criteria and rank of promising tuples at join operators (Sec. 6.5.1). Each dynamic monitoring level dml_l denoted as $(s_n;\ op_c;\ prnk;$ $< attrib_1, attrib_2, ..., attrib_n >)$ designates 1) the stream $dml_l.s_n$ of the promising tuples, 2) the operator at which the promising tuples are predicted to join with significant tuples called the designated operator $dml_l.op_c$, 3) a promising rank $dml_l.prnk$, and 4) the membership criteria $dml_l.mem$ defined by attribute pattern $< attrib_1, attrib_2, ...,$ $attrib_n >$. Under limited resources, the optimizer also selectively activates some dynamic monitoring levels (Sec. 6.1.1).

The attribute pattern of a membership criteria $dml_l.mem$ specifies the values for each attribute $attrib_i\ (1 \leq i \leq x)$ in the x attributes of tuples in stream s_n. For example, attribute pattern $< Advertising, *, * >$ states the condition that the first attribute of the tuple must equal business sector $Advertising$.

Assume for example that dynamic monitoring level dml_l identifies a news tuple t_j that satisfies the join criteria c_2, namely, that business sector $= Advertising$. Dynamic monitoring level dml_l would thus be $(s_1; op_2; 1; < Advertising\ , *, * >)$. It states that tuples from stream s_1 whose business sector $= Advertising$ are designated to be promising tuples at rank 2 until they reach join operator op_2.

Definition 7 *A promising tuple t_i has one promising rank $t_i.prnk = dml_l.prnk$ and designated operator $t_i.op_c = dml_l.op_c$. The promising rank $t_i.prnk$ is the most significant of all the activated dynamic monitoring levels that t_i satisfies. In order for tuple t_i to retain its promising rank (and preferential resource allocation) for the longest duration of the pipeline, the designated operator $t_i.op_c$ is set to be the designated operator furthest down the pipeline of all the activated dynamic monitoring criteria at promising rank $t_i.prnk$ that tuple t_i satisfies.*

6.1.4 Tuple Rank

Tuple t_i can have both significant and promising rank. Each designation refers to distinct significant results that tuple t_i may create. Significant rank, being global, applies to all results that tuple t_i creates at any operator. Promising rank, being localized, applies to some results that tuple t_i creates at a specific operator only.

Tuple t_i can be allocated resources based upon both its significant and promising ranks. Multiple design choices are possible from selecting the maximum of its significant and promising ranks to computing a combined weighted rank. The later option requires periodic recalculation of the weighted ranks as the ranks adapt. Given limited resources and this extra recalculation cost, we adopt the former simpler method for the remainder of this manuscript. Although, in principle, any method could be chosen.

Tuple t_i is assigned a *rank* attribute $t_i.rnk$ that is the maximum of t_i's significant and promising ranks. This lets us quickly determine if t_i should be preferentially allocated resources.

6.1.5 PR Optimization Problem

A *PR query plan* is used to represent a P-CQL query q_j. We model each PR plan p_m^{pr} as a one directional flow network composed of PR algebra operators as nodes and data exchange interfaces that transfer tuples between operators as edges (Sec. 6.3.1). The PR query algebra is composed of significance classifier operators and PR augmented standard operators as introduced in Sec. 6.3.2.

The optimal PR plan allocates resources to tuples to maximize the throughput of the most significant query results in precedence order. Such a plan ensures that any tuples with the highest rank rnk are processed first when resources are limited. To achieve this, tuples with significant and/or promising rank of rnk must be processed before those with lower or no rank.

Thus any tuples with significant rank $srnk$ must be processed throughout the query plan within the query lifespan. Hence if all tuples that can have the significant rank of $srnk$ are being devoted adequate processing cycles then the number of such tuples that expire throughout the query pipeline (or the expiration rate of potential significant tuples $ER_s(p_m^{pr}, srnk)$ at significant rank $srnk$) should be low and ideally zero.

Definition 8 *Expiration Rate of Potential Significant Tuples $ER_s(p_m^{pr}, srnk)$ is the number of tuples that satisfy static criteria for rank $srnk$ that expire throughout the query pipeline.*

Such a PR plan must also improve the flow of tuples that can have the promising rank $prnk$ until they reach their designated operator. The number of promising tuples that expire before reaching their designated operator (or the expiration rate of potential promising tuples $ER_p(p_m^{pr}, prnk)$ at promising rank $prnk$) should be low (ideally zero).

Definition 9 *Expiration Rate of Potential Promising Tuples $ER_p(p_m^{pr}, prnk)$ is the number of tuples that satisfy dynamic criteria at promising rank $prnk$ that expire before they reach their designated operator in p_m^{pr}.*

Definition 10 *The goal of our PR optimizer is to identify the* optimal PR plan *that, compared to all possible PR plans, minimizes the expiration rate of both significant (Def. 8) and promising tuples (Def. 9) for each rank starting from the most significant rank within the available system capacity.*

Table 6.2: Example: Ranking PR Plans

PR Plan (p_m^{pr})	CPU $Over$ $head$	$ER_s(p_m^{pr},1)$ / $ER_p(p_m^{pr},1)$	$ER_s(p_m^{pr},2)$ / $ER_p(p_m^{pr},2)$	$ER_s(p_m^{pr},3)$ / $ER_p(p_m^{pr},3)$
p_1^{pr}	1022	0 / 0	2 / 35	15 / 75
p_2^{pr}	1256	0 / 0	2 / 35	15 / 75
p_3^{pr}	1956	0 / 0	2 / 80	95 / 89
p_4^{pr}	1006	9 / 89	123 / 90	90 / 87

Consider the example in Table 6.2. Assume the resources required to execute each PR plan are within the available system capacity. PR plan p_1^{pr} is the best for the following reasons. For the most significant rank $rnk = 1$, PR plan p_4^{pr} has an expiration rate of potential significant tuples greater than 0, i.e., $ER_s(p_4^{pr},1) > 0$. For the next most significant rank $rnk = 2$, PR plan p_3^{pr} has an expiration rate of potential promising tuples greater than PR plans p_1^{pr} and p_2^{pr}. For all ranks, PR plans p_1^{pr} and p_2^{pr} have equal expiration rates. However, compared to PR plan p_2^{pr}, p_1^{pr} has the lowest CPU overhead cost. Thus, PR plan p_1^{pr} is the preferred solution.

6.2 PR Architecture

The online adaptive PR architecture (Fig. 6.2) contains the PR Executor, PR Monitor, PR Optimizer, and PR Adaptor. The PR Executor (Sec. 6.3) runs the current PR plan. The PR Monitor (Sec. 6.4), PR Optimizer (Sec. 6.5.5), and PR Adaptor

(Sec. 6.6) support the online PR plan adaption by respectively collecting statistics, selecting a new PR plan, and adapting the current PR plan to the new one.

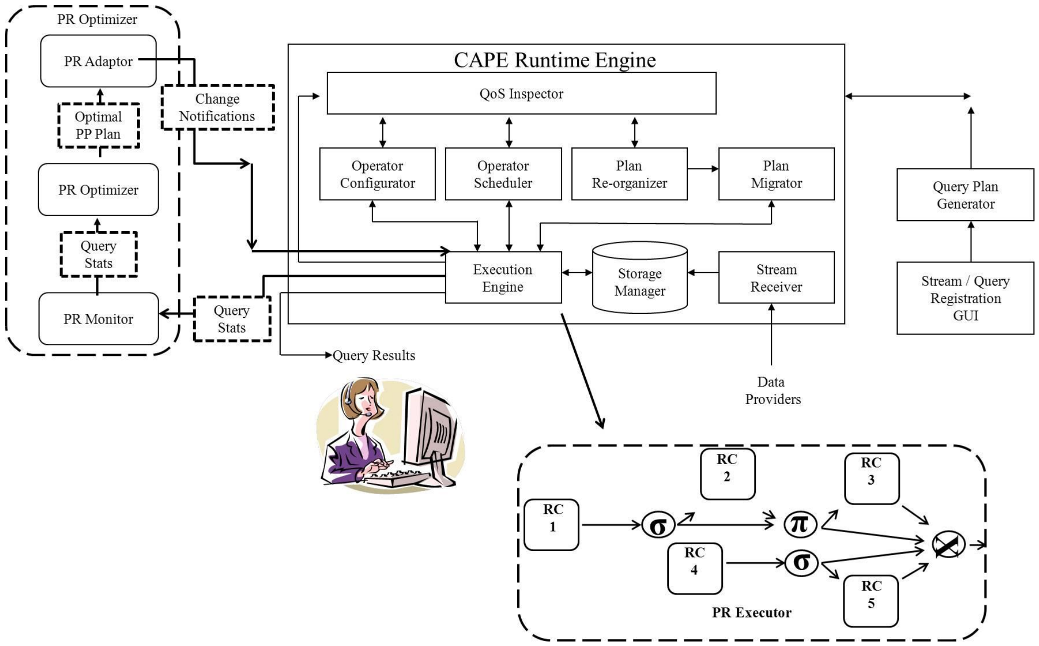


Figure 6.2: PR Architecture

6.3 PR Executor Infrastructure

We now outline our design for the back bone of PR (i.e., the PR Executor Infrastructure) and how it executes a PR plan efficiently.

6.3.1 Pulling Tuples Ahead of Others

The *data exchange interface* transfers tuples between operators. To efficiently pull certain tuples forward, we employ a multi-queue approach. Operators support one queue for tuples with each possible rank and one for insignificant tuples. Significant and promising tuples with the same rank reside in the same queue.

If no monitoring levels are activated then all tuples reside in the insignificant queue. In this case, the query operators would process the tuples in FIFO order. Otherwise, each tuples reside in the queue that corresponds to their rank. In this situation, operators process tuples in rank order. Operator op_o starts processing tuples from the most significant queue. When this is empty and resources remain, operator op_o moves to the second most significant queue. Each result t_i is placed into the incoming queue for t_i's rank of the next down stream operator.

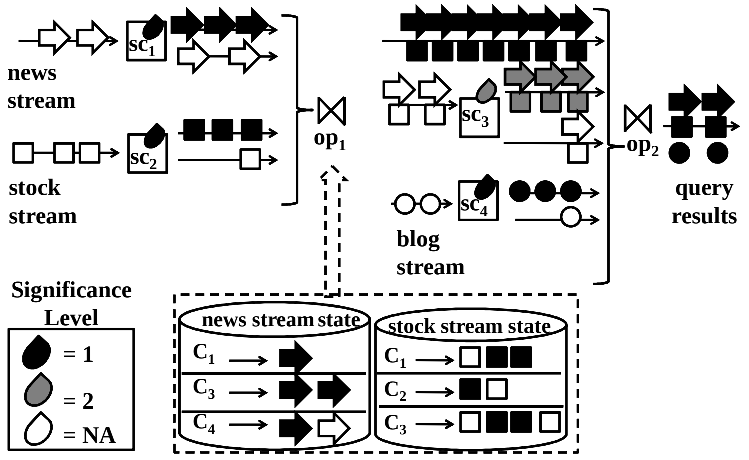


Figure 6.3: Stock Market Join Query PR Plan

Consider the queues for the news stream in Figure 6.3. Operator op_1 has an incoming queue for news tuples with each possible rank (e.g., rank 1) and one for insignificant incoming news tuples.

6.3.2 PR Query Algebra

Our PR algebraic operators support both significant and/or promising tuples where the rank may adapt at run-time (Sec. 6.3.3). In PR algebra, traditional operators [GO05] process tuples as usual and propagate the appropriate preference related meta data to the results. *Significance classifier* operators assign preference related meta data to tuples.

Projection removes specified attributes from tuples in its input queues. ***Selection*** removes tuples in its input queues that do not satisfy the specified selection condition. Both send their results with no changes of their preference related meta data to the next operator.

Join [WA91] creates results (t_i,t_j) by matching tuples from streams s_1 and s_2. First, tuple t_i is stored with its preference related metadata in the state of tuple t_i's stream s_1. Then, results (t_i,t_j) are created by joining tuple t_i with tuples t_j in stream s_2's state. Next, result (t_i,t_j)'s preference related metadata are set. That is, result (t_i,t_j) is assigned the most significant among the preference related metadata of tuples t_i and t_j. If tuples t_i and t_j have the same promising rank then result (t_i,t_j) is assigned the designated operator of tuples t_i and t_j that is furthest along the pipeline. Lastly, the result (t_i,t_j) is sent to the next operator.

If tuple t_i is a promising tuple and its designated operator is this current join operator, then prior to processing tuple t_i's promising rank and designated operator attributes are set to null. Then tuple t_i's rank is set to its significant rank, i.e., $t_i.rnk = t_i.srnk$. In this case, tuple t_i is not known to be a promising tuple beyond this join operator but it may turn into a significant tuple at a lower rank.

Significance Classifier (or *SC*) is a special-purpose operator with static (S_{AS}) and dynamic assessment set (D_{AS}) parameters. It creates results by assigning preference related metadata to tuples in its input stream and then sends these results to the next operator.

The static S_{AS} and dynamic D_{AS} assessment sets contain the respective criteria of the activated static or dynamic monitoring levels assessed by the SC operator op_o.

SCs process each tuple t_i by comparing the criteria in S_{AS} and D_{AS} to tuple t_i in consecutive order starting from the most significant criteria in S_{AS} and D_{AS}. Once tuple t_i satisfies a static criteria then tuple t_i is not compared to any dynamic criteria of the same or lower rank. This is because if tuple t_i is a significant tuple at rank rnk then tuple t_i is guaranteed to be a significant tuple at rank rnk for the duration of tuple t_i's processing. However, if tuple t_i is a promising tuple at rank rnk then this tuple t_i is only guaranteed to be a promising tuple at rank rnk for a portion of the query pipeline. Even if tuple t_i satisfies a dynamic criteria no static criteria comparisons are eliminated.

Before each standard operator in the PR plan p_m^{pr} we place a SC operator. The optimizer determines which monitoring levels are activated and the static S_{AS} and dynamic D_{AS} assessment set of each SC (Sec. 6.5). Then the optimizer notifies each SC of changes to their assessment sets.

Each SC in the PR plan assigns preference related metadata to particular tuples. The assessment sets of an SC may be empty. In this case, the SC will not evaluate the preference related metadata of any monitoring levels and all tuples will skip being sent to the SC (Sec. 6.6).

Consider the PR Plan in Figure 6.3. Incoming news and blog tuples are respectively evaluated by significance classifier operators sc_1 and sc_4 against dynamic criteria to identify promising tuples at rank 1. We denote the rank of the criteria in the assessment sets evaluated by each SC in Figure 6.3 by the color of the tear drop in the top of the SC. First, incoming stock tuples are evaluated by significance classifier operator sc_2 against static criteria to locate significant tuples at rank 1. Then join operator op_1 joins news tuples with stock tuples. Depending upon their rank, results from op_1 are routed to either s_3 and then to join operator op_2 or directly to join operator op_2 (Sec. 6.6). SC sc_3 evaluates incoming tuples against static criteria to identify significant tuples at rank 2. Finally, the join operator op_2 produces query results by joining blog tuples with combined stock/news tuples.

6.3.3 Adapting Rank of Tuples

Cases when tuple t_i's rank may adapt:

1) Tuple t_i's rank may be *elevated* when t_i is assigned a significant and/or promising rank by an SC.

2) Tuple t_i's rank may be *degraded* when t_i is a promising tuple and t_i reaches its designated operator.

3) Tuple t_i's rank may be either *elevated* or *degraded* when the optimizer selects a new PR plan (Sec. 6.5). Tuple t_is rank is respectively more or less significant than the lowest rank of the monitoring levels activated in the new PR plan.

Each result t_i created from an *elevated* or *degraded* tuple t_j is placed into a different queue than the incoming queue that held tuple t_j. An operator places result t_i into the appropriate queue based upon t_i's rank and the current activated monitoring levels. If tuple t_i is placed into a priority queue then henceforth operators will preferentially allocate resources to t_i. If tuple t_i is placed into the insignificant queue then henceforth operators will not preferentially allocate resources to t_i. Tuple t_i retains the values of its preference related metadata in case t_i is elevated or degraded in the future.

6.4 PR Monitor

The PR Monitor gathers statistics to track the progress of tuples that have the potential to be significant and/or promising tuples. Static membership criteria are defined in the P-CQL extension at compile-time (Sec. 6.1.1). The PR Monitor uses these static criteria to collect statistics for each operator on the how many incoming tuples expire that have the potential to be significant tuples at rank rnk.

However, the dynamic membership criteria that identifies promising tuples in the current system is unknown at this point. To identify this criteria requires knowledge of which current join attributes of tuples that have the potential to be significant tuples are also prevalent in tuples in the join partner stream. Thus, the PR Monitor gathers statistics that the PR Optimizer will use to identify such criteria. We refer to these criteria as *potential dynamic membership criteria*.

Tuples can be both significant and promising. There is no need to pull forward promising tuples at rank rnk if such tuples have the potential to be significant tuples at the same or higher rank than rnk. Such tuples will already be preferentially allocated resources at a more significant rank. The goal instead is to identify the join criteria of tuples that have the potential to be promising tuples, but do not have the potential to be significant tuples at a rank more significant than rank rnk.

To figure out what could serve as dynamic membership criteria, the PR Optimizer needs to identify at each join operator the join criteria of incoming tuples that have the potential to be significant tuples. Such tuples may expire before reaching the next join operator in the query pipeline. Thus, the PR Monitor tracks the attributes of tuples that arrive at join operators as well as the attributes of tuples that expire before they reach the next join operator in the pipeline. Each join operator op_i generates a histogram of the count of tuples that have the potential to be significant tuples at rank rnk and arrive at operator op_i by their join criteria and input stream. Each non-join operator op_j generates a histogram of the count of tuples that have the potential to be significant tuples at rank rnk and arrive at operator op_j by the join criteria of the next join operator in the query pipeline and input stream. The PR Monitor combines these counts to represent the *frequency of potential significant tuples*.

Definition 11 *The frequency of potential significant tuples $F_{Sig(op_o,s_n,rnk,c_p)}$ is the count of all potential incoming tuples to join operator op_o from stream s_n that could be a significant tuple at rank rnk and satisfy join criteria c_p.*

Consider the PR Plan (Fig. 6.3) for the Stock Market Join Query Example (Sec. 1.3.2). The join criteria of potential significant tuples at rank 1 into the join operator op_1 from the stock stream are join criteria c_1, c_2, and c_3. Assume that join criteria c_1, c_2, and c_3 respectively are business sector equal to Energy, Advertising, and Drug Retail.

The PR Optimizer could only use the frequency of potential significant tuples to locate potential dynamic criteria. However, tuples that satisfy a particular join criteria may not exist in all streams. Reconsider the PR Plan (Fig. 6.3). If there are no news tuples about the Advertising business sector (i.e., join criteria c_2) then evaluating join criteria c_2 as a dynamic criteria will not identify any promising tuples. Clearly evaluating such criteria simply adds overhead without any benefit.

Thus, the PR Monitor also collects statistics to identify at each join operator the join criteria of incoming tuples that have the potential to be promising tuples. Such tuples may also expire before reaching the next join operator in the query pipeline. Hence, the PR Monitor tracks the attributes of tuples that arrive at join operators as well as the attributes of tuples that expire before they reach the next join operator in the pipeline. Each operator op_i (join or otherwise) generates a histogram of the count of tuples by their join criteria and input stream (a.k.a. the *frequency of potential promising tuples*) that satisfy the following criteria. First, they must have the potential to be promising tuples at rank rnk. Second, they must not have the potential to be significant tuples at a rank more significant than rank rnk. Third, they must be potential incoming tuples to join operator op_i.

In the PR Plan (Fig. 6.3), the join criteria of potential promising tuples from the news stream at rank 1 into the join operator op_1 are criteria c_1, c_3, and c_4.

Definition 12 *The frequency of potential promising tuples* $F_{Prom(op_o,s_n,rnk,c_p)}$ *is the count of all tuples that could be incoming tuples to join operator* op_o *from stream* s_n *at rank* rnk *that satisfy* join criteria c_p *where the following conditions hold. First, significant join partner tuples exist that also satisfy criteria* c_p *at join operator* op_o *(i.e.,* $F_{Sig(op_o,s_m,rnk,c_p)} > 0$*). Second, these tuples do not have the potential to be significant tuples at a rank more significant than rank* rnk*.*

The number of possible join criteria is exponential given the possible domains and range of join criteria values. Clearly, this can cause potential memory limitations in collecting statistics. We propose to reduce the number of frequencies collected by using a heavy hitter algorithm [MM02]. Informally, while collecting the statistics each operator periodically removes any statistic whose frequency falls below a preset error rate. When all statistics have been collected, each operator returns only the statistics whose frequencies are above a preset threshold. In addition, if the join criteria is from a continuous domain then statistics could be gathered on a range of values. To simplify the problem, we assume that the join criteria is from a discrete domain. Thus, statistics are gathered on each distinct join criteria value.

Periodically, each operator transmits their statistics to the PR Monitor. Once the PR monitor has collected statistics from all operators, it then sends them to the PR Optimizer.

6.5 PR Optimizer

Upon receiving the statistics, the PR Optimizer selects the best order of operators within the query plan and then generates a new PR plan. First the *initial PR plan* is created by placing an SC with empty assessment sets before each standard operator in p_m. Only one SC is required as the static and dynamic assessment sets of multiple adjacent SCs can be merged. From this initial PR plan, all possible PR plans can be created by adjusting which static and dynamic criteria are evaluated and where (i.e, in which significance classifier(s)) each static or dynamic criteria is evaluated.

Generating one possible PR plan roughly involves the following two steps:

Step 1: Create all possible dynamic monitoring levels.

Step 2: Iteratively for each rank rnk in significance order:

a: Select which static and dynamic monitoring levels to activate.

b: Determine where in the plan to evaluate each static and dynamic criteria from the monitoring levels activated in Step 2a.

We now explore the details behind each of these steps.

6.5.1 Creating Dynamic Monitoring Levels

The PR Optimizer creates a dynamic monitoring level for each join operator op_o, stream s_n, rank rnk, and *join criteria* c_p where the frequencies of both potential significant and promising tuples are greater than 0, i.e., $F_{Sig(op_o,s_n,rnk,c_p)} > 0$ and $F_{Prom(op_o,s_n,rnk,c_p)} > 0$. When either frequency equals zero then either there are no tuples in one of the steams that satisfy join criteria c_p or tuples that satisfy join criteria c_p already have the potential to be assigned to a rank more significant than rnk.

Consider PR Plan (Fig. 6.3). Dynamic criteria at rank 1 for join operator op_1 are join criteria c_1 and c_3. Join criteria c_2 is not classified as a dynamic criteria. Although there are significant stock tuples from the Advertising business sector (i.e., join criteria c_2), there are no news tuples in this Advertising business sector. Hence, the frequency of potential promising tuples that satisfy join criteria c_2 is 0, i.e., $F_{Prom(op_1,newsStream,c_2,1)} = 0$. Similarly, join criteria c_4 is also not classified as dynamic criteria.

Each dynamic criteria c_p may identify significant join partners at join operator op_o from more than one rank. In this case, promising tuples that satisfy c_p will create join results at more than one rank. To keep this practical, each promising tuple that satisfies c_p is assigned the highest rank of all significant join partner tuples that also satisfy c_p.

Consider PR Plan (Fig. 6.1). Stock tuples from the Drug Retail business sector (i.e., join criteria c_3) are significant tuples with ranks of 1 and 2. Thus, the rank of the dynamic monitoring level created to locate news and blog tuples with this Drug Retail business sector (i.e., c_p = join criteria c_3) is rank 1.

6.5.2 Selecting Which Static and Dynamic Monitoring Levels to Activate

Some of the potential dynamic monitoring levels generated may reference join operators that will not have any incoming significant tuples. A join operator op_o is said to be designated if and only if some of its incoming tuples will be significant tuples. Whether or not significant tuples arrive at join operator op_o depends upon which static monitoring levels are activated and where they are evaluated. As a consequence to determine which dynamic monitoring levels at rank rnk to active the PR Optimizer must first find which join operators are designated. Hence, we must first select which static monitoring levels to activate and determine where in the plan to evaluate each static criteria before dynamic monitoring levels should be considered.

From these insights, we now revise the steps executed by the PR Optimizer. The revised steps are underlined.

Step 1: Create dynamic monitoring levels.

Step 2: Iteratively for each rank rnk in significance order:

a: Select which static monitoring levels to activate.

b: Determine where in the plan to evaluate each static criteria for the monitoring levels activated in Step 2a.

c: Identify designated operators, i.e., join operators where significant tuples at rank rnk may arrive.

d: Select which dynamic monitoring levels to activate.

e: Determine where in the plan to evaluate each dynamic criteria from the monitoring levels activated in Step 2d.

6.5.3 Determining Where to Evaluate each Static and Dynamic Criteria

The PR Optimizer must determine which SC(s) should evaluate which criteria of the activated monitoring levels. We now discuss how many SCs must evaluate each static and dynamic criteria to ensure that all possible significant or promising tuples are pulled forward. Each static or dynamic criteria has an *evaluation path*, i.e., an ordered set of operators that begins at the first and ends at the last operator in the plan that can evaluate the criteria.

Static Evaluation Path: Only one significance classifier (SC) in the evaluation path needs to evaluate a given static criteria because significant tuples retain their rank for the duration of processing. Thus the PR optimizer only needs to locate the best SC to assess each static criteria.

Dynamic Evaluation Path: The rank of promising tuple t_i has a short lifespan because tuple t_i will drop its promising rank when it reaches its designated operator. Further along the pipeline, the tuple t_i may be assigned another promising rank.

During its processing, a tuple may be pulled forward to different designated operators along the query pipeline. Sometimes the evaluation paths of different criteria may overlap. To ensure that for dynamic criteria c_p all promising tuples are pulled forward, dynamic criteria c_p may need to be evaluated at multiple SCs, namely, after any of its designated operator. This is because tuples may lose their current promising rank at each designated operator. This thus would be the first place to check if it should be assigned the promising rank of a designated operator further along the query pipeline.

Static vs Dynamic Evaluation Paths: A static evaluation path ends at the last

SC operator in the query plan as significant tuples remain significant for the entire query pipeline. In contrast, a dynamic evaluation path ends at the SC operator that proceeds their associated designated operator in the query pipeline. Hence to determine the possible dynamic evaluation paths we must locate the designated operators in the query pipeline.

6.5.4 PR Plan Search Space

We now explore the size of the search space to exhaustively search over all possible options of which levels are activated and where each criteria is evaluated to find the optimal PR plan.

Static Priority Determination: Recall that each static criteria is evaluated in one SC in its static evaluation path. In the PR Plan (Fig. 6.3), consider identifying significant tuples from the stock stream at rank 1, i.e., tuples from aggressive investments. Such tuples can either be identified by SC_2 (i.e., before join operator op_1) or by SC_3 (i.e., after join operator op_1 and before join operator op_2).

Complexity of Static Priority Determination: Assume that in PR plan p_m^{pr}, there are $|SEP|$ static evaluation paths. Each static evaluation path sep_k contains $|sep_k.sc|$ SC operators. There are $|SCrit(sep_k, rnk)|$ static criteria whose static evaluation path is sep_k and rank is rnk. For rank rnk and static evaluation path sep_k, there are $|sep_k.sc|^{|SCrit(sep_k,rnk)|}$ possible combinations of which SC evaluates each static criteria in $SCrit(sep_k, rnk)$. Hence, for rank rnk, there are $\prod_{k=1}^{|SEP|} |sep_k.sc|^{|SCrit(sep_k,rnk)|}$ possible PR plans where the static criteria at rank rnk can be evaluated.

Reconsider the PR Plan in Figure 6.3. Assume that only one static evaluation path sep_1 exists that contains SC operators SC_1 and SC_2. In addition, assume that there are 4 static criteria whose static evaluation path is sep_1 and rank is rnk. In this case, for rank rnk, there are 2^4 or 16 possible PR plans where the static criteria at rank rnk can be evaluated.

Dynamic Priority Determination: In contrast, each dynamic criteria could be evaluated in many (and even all) SCs in its dynamic evaluation path. In the PR Plan (Fig. 6.3), consider locating promising tuples for join operator op_1 from the news stream from the Energy business sector, i.e., join criteria c_1. Such tuples will be promising at rank 1 for designated operator op_1. Operator SC_1 would be the only operator required to identify these tuples using the promising criteria (i.e., before join operator op_1). This is because there are no designated operators between SC_1 and the designated operator op_1.

Now consider locating promising tuples for join operator op_2 from the news stream from the Energy business sector, i.e., join criteria c_1. These tuples will be promising tuples at rank 1 with designated join operator op_2. Such tuples can be located by any SCs before join operator op_2, namely, SC_1 and SC_3.

If only SC_3 evaluates join criteria c_1 then such tuples only need to be located by SC_3. This is because there are no designated operators between SC_3 and the designated operator for join criteria c_1, i.e., designated operator op_2. However, this is not true if SC_1 evaluates join criteria c_1 and operator op_1 is a designated operator between SC_1 and join operator op_2, i.e., the designated operator for join criteria c_1. In this case, to ensure that promising tuples for both designated operators op_1 and op_2 are pulled forward, SC_3 will also need to evaluate join criteria c_1.

Segments within a dynamic evaluation path where a tuple's rank can change exist between every pair of consecutive designated operator in the dynamic evaluation path. We refer to these segments as *dynamic re-evaluation paths*.

For the last example above, a dynamic re-evaluation path exists between operators op_1 and ends at SC_3. This is because SC_3 is the first SC after designated operator op_1 and the last SC before the next designated operator op_2. The optimizer must select one SC in each dynamic re-evaluation path to evaluate the dynamic criteria.

Complexity of Dynamic Priority Determination: Consider dynamic criteria c_p whose dynamic evaluation path is dep_l. Dynamic criteria c_p can be evaluated by any of the $|dep_l.sc|$ SCs in dynamic evaluation path dep_l. Assume that dynamic criteria c_p is evaluated by SC sc_x. In this case, dynamic criteria c_p must be re-evaluated in each of the dynamic re-evaluation paths that proceed sc_x in dynamic evaluation path dep_l. This ensures that any tuple that satisfies dynamic criteria c_p and another dynamic criteria within dynamic evaluation path dep_l will be assigned as a promising tuple for the duration of dynamic evaluation path dep_l. $|DREP(sc_x, dep_l)|$ denotes the number of dynamic re-evaluation paths that proceed sc_x. Each dynamic criteria c_p must be evaluated by one of the $|drep_m.sc|$ SCs in each $drep_m$ dynamic re-evaluation paths that proceed sc_x in dynamic evaluation path dep_l. Thus, assuming that dynamic criteria c_p is evaluated by SC sc_x, there are $\prod_{m=1}^{|DREP(dep_l,sc_x)|} |drep_m.sc|$ possible combinations of which other SCs evaluate dynamic criteria c_p.

Thus, for each dynamic criteria c_p whose dynamic evaluation path is dep_l, there are $\sum_{x=1}^{|dep_l.sc|} \prod_{m=1}^{|DREP(dep_l,sc_x)|} |drep_m.sc|$ possible combinations of which SCs evaluate dynamic criteria c_p. sc_x is the SC in dynamic evaluation path dep_l selected by the optimizer. $|DREP(sc_x, dep_l)|$ denotes the number of dynamic re-evaluation paths that proceed sc_x in dynamic evaluation path dep_l. $|drep_m.sc|$ denotes the number of SCs in dynamic re-evaluation path $drep_m$ where dynamic criteria $d\text{-}crit_i$ can be re-evaluated.

There are $|DCrit(dep_l, rnk)|$ dynamic criteria whose dynamic evaluation path is

dep_l and rank is rnk. Hence, for each rank rnk, there are

$\prod_{l=1}^{|DEP|}(\sum_{x=1}^{|dep_l.sc|}\prod_{m=1}^{|DREP(sc_x)|}|drep_m.sc|)^{|DCrit(dep_l,rnk)|}$ possible PR plans where the dynamic criteria at rank rnk can be evaluated.

The PR search space thus is:

$\sum_{rnk=1}^{|q_j.SML|}\prod_{k=1}^{|SEP|}|sep_k.sc|^{|SCrit(sep_k,rnk)|}*$

$\prod_{l=1}^{|DEP|}(\sum_{x=1}^{|dep_l.sc|}\prod_{m=1}^{|DREP(dep_l,sc_x)|}|drep_m.sc|)^{|DCrit(dep_l,rnk)|}$.

We notice that the complexity of dynamic priority determination in the PR problem is exponential in the number of dynamic criteria and the number of designated operators. It is impractical to exhaustively search for the optimal PR plan with many dynamic criteria and designated operators.

6.5.5 PR Prune Optimization Strategy

We now introduce our PR Prune Optimization strategy that reduces the complexity of dynamic priority determination. PR Prune eliminates inferior dynamic criteria before creating dynamic monitoring levels (i.e., Step 2 below). This reduces $|DCrit(dep_l,rnk)|$). As we will see below, PR Prune discriminately chooses which dynamic monitoring levels to activate. Finally, PR Prune also reduces the number of designated operators it creates (i.e., Step 2d above). This reduces $|DREP(dep_l,sc_x)|$).

Roughly, PR-Prune consists of the following steps: The differences between the logic of the PR Prune and the original PR Optimizer steps are underlined.

Step 1: <u>Eliminate inferior dynamic criteria.</u>

Step 2: Create dynamic monitoring levels.

Step 3: Iteratively for each rank rnk in significance order:

a: Select which static monitoring levels to activate.

b: Determine **where** in the plan to evaluate each static criteria from the monitoring levels activated in Step 2a.

c: Identify the designated operators, i.e., join operators where significant tuples at rank rnk may arrive.

d: Reduce the number of designated operators found in Step 2c.

e: Select which dynamic monitoring levels to activate.

f: Determine where in the plan to evaluate each dynamic criteria from the monitoring levels activated in Step 2e.

Pruning of Inferior Dynamic Criteria

Dynamic priority determination is more complex than static priority determination. The number of static criteria at rank rnk is inherently small as they are defined by users at compile-time. In contrast, the number of dynamic criteria can be prohibitively large. That is, there may be a huge number of join criteria at each join operator that identify promising tuples at rank rnk.

Observation: *Some dynamic criteria c_p may identify promising tuples that produce more significant query results than others.*

In the PR Plan (Fig. 6.3), dynamic criteria at rank 1 in join operator op_1 are from tuples related to the Energy and Drug Retail business sectors. More precisely, two join results from the Energy business sector will be produced when the promising tuple at rank 1 joins with the two significant tuple at rank 1 using join criteria c_1. Only four join results from the Drug Retail business sector will be produced for join criteria c_3.

Evaluating the optimal determination location of inferior criteria adds overhead. Each dynamic criteria we do not evaluate reduces the complexity of dynamic priority determination by : $(\sum_{x=1}^{|dep_l.sc|}\prod_{m=1}^{|DREP(dep_l,sc_x)|}|drep_m.sc|)$ where dep_l is the dynamic evaluation path of $d\text{-}crit_i$.

A dynamic criteria is inferior to the others typically when the product of the frequencies of potential significant and promising tuples is extremely low. To eliminate these inferior dynamic criteria, we propose a reduction method that eliminates inferior dynamic criteria. We remove any dynamic criteria if the product of the frequencies of potential significant and promising tuples is below a preset threshold.

Activation Order of Dynamic Monitoring Levels

Rather than exhaustively searching through all possible dynamic monitoring levels to decide which ones to activate, PR-Prune starts with the dynamic monitoring levels that are estimated to produce the largest cardinality of significant join results. This corresponds to the criteria with the largest product of the frequencies of potential significant and promising tuples. This helps ensure that resources are allocated to the most promising tuples first.

From these insights, we thus again refine the logic of the PR Prune. The revised steps are underlined.

Step 1: Eliminate inferior possible dynamic criteria.

Step 2: Create dynamic monitoring levels.

Step 3: Iteratively for each rank rnk in significance order:

a: Select which static monitoring levels to activate.

b: Determine where in the plan to evaluate each static criteria from the monitoring levels activated in Step 2a.

c: Identify the designated operators, i.e., join operators where significant tuples at rank rnk may arrive.

d: Reduce the number of designated operators found in Step 2c.

e: Iteratively for each dynamic monitoring levels at rank rnk in order of highest to lowest product of the frequencies of potential significant and promising tuples.

f: Active the dynamic monitoring levels selected in Step 2e.

g: Determine where in the plan to evaluate each dynamic criteria from the monitoring level activated in Step 2f.

Reducing the Dynamic Evaluation Paths

Eliminating a designated operator reduces the number of dynamic evaluation paths. This reduces the number of dynamic evaluation paths or $|DEP|$ by one. Thus, per the PR search space, this reduces the number of possible PR plans by $(\sum_{x=1}^{|dep_l.sc|} \prod_{m=1}^{|DREP(dep_l,sc_x)|} |drep_m.sc|)^{|DCrit(dep_l,rnk)|}$ plans.

Reducing the number of dynamic evaluation paths may also reduce the number of dynamic re-evaluation paths. This reduces $|DREP(sc_x, dep_l)|$ or the number of dynamic re-evaluation paths that proceed SC sc_x. Thus, per the PR search space, each dynamic re-evaluation path eliminated reduces the number of possible PR plans by $(\sum_{x=1}^{|dep_l.sc|} |drep_m.sc|)^{|DCrit(dep_l,rnk)|}$ plans.

In the PR Plan (Fig. 6.3), assume that tuple t_i satisfies the dynamic criteria c_1 and c_2 at rank rnk for respective designated operators op_1 and op_2. Assume that tuple t_i does not satisfy any other criteria. There are two possible ways in which t_i could be processed. First, tuple t_i could be evaluated by an SC against dynamic criteria c_1. In this case, tuple t_i would be a promising tuple at rank rnk with designated operator op_1. After operator op_1, tuple t_i would then be evaluated by an SC against dynamic criteria c_2. At this point, tuple t_i would become a promising tuple at rank rnk with designated operator op_2. The second alternative is that prior to operator

op_1, tuple t_i could be evaluated by an SC against dynamic criteria c_2. In this case, tuple t_i would be a promising tuple at rank rnk with designated operator op_2.

In both cases, tuple t_i would be promoted as a promising tuple at rank rnk across both operators. In the first case, tuple t_i would be evaluated twice and promoted as a promising tuple at rank rnk across each operator individually. In the second case, tuple t_i would be evaluated once but promoted as a promising tuple at rank rnk across both operators.

Clearly, there is less overhead if we allocate resources to promising tuples for the longest duration of query processing. Hence, it is best to assign tuple t_i to the designated operator that is furthest along the query pipeline, i.e., operator op_2.
Observation: *If multiple consecutive join operators in the query pipeline have the same join criteria then only the last join operator in the sequence should be used to identify potential dynamic criteria.*

Reconsider PR Plan (Fig. 6.3). Join operators op_1 and op_2 both use the same join criteria attribute, i.e., business sector. If we use the dynamic criteria located at operator op_1 then there is no guarantee that the same business sectors exist in the news and blog streams over the same query window. That is, the significant join result tuples generated by op_1 may not contain a business sector in the blog stream. This would cause no result to be created by join operator op_2. Preferentially allocating resources to pull forward news tuples from such a business sector may not generate any additional significant query results. However, since op_1 and op_2 both use the same join criteria attribute the frequency statistics can be shared across both operators. Then dynamic criteria learned by operator op_2 can ensure that such tuples will match some tuples in both the news as well as blog stream.

In summary, PR-Prune eliminates designated operators by only assigning the last join operator of all adjacent join operators that use the same join criteria to be the designated operator. Then PR-Prune identifies the dynamic criteria of the join operator furthest along the query pipeline.

PR-Prune Plan Search Space

Compared to PR, PR-Prune reduces the following sets; the set of dynamic evaluation paths, the set of dynamic re-evaluation paths, and the set of dynamic criteria. However, the possible combinations of where each static ad dynamic determinant can be re-evaluated in the PR plan is the same for both PR and PR-Prune.

The **PR-Prune search space** for plan p_i is thus:

$\sum_{rnk=1}^{|q_j.SML|} \prod_{k=1}^{|SEP|} |sep_k.sc|^{|SCrit(sep_k,rnk)|} *$

$\prod_{l=1}^{|RDEP|} (\sum_{x=1}^{|dep_l.sc|} \prod_{m=1}^{|RDREP(dep_l,sc_x)|} |drep_m.sc|)^{|RDCrit(dep_l,rnk)|}$.

$RDEP$ is the reduced set of dynamic evaluation paths. $RDREP(dep_l, sc_x)$ is the reduced set of dynamic re-evaluation paths that follow sc_x in dynamic evaluation path dep_l. $RDCrit(dep_l, rnk)$ is the reduced set of dynamic criteria whose dynamic evaluation path is dep_l and rank is rnk.

6.5.6 Optimal Order of Operators in PR Plan

Selecting the optimal ordering of operators within a plan is NP-hard [BMM$^+$04]. The complexity only increases if we simultaneously consider both the optimal ordering of operators and allocation of resources. Given a query plan p_m selected using traditional query optimization techniques [RG00], the PR optimizer locates the optimized PR plan p_m^{pr} for a given traditional query plan p_m.

6.6 PR Adaptor

After the optimizer selects a new optimized PR plan, the PR Adaptor adapts the current PR plan to the new one. A key to efficiently supporting the adaption of PR plans is the retention of an SC before each standard operator. This allows the PR plan to adapt online without requiring any operators or data exchange interfaces to be added, removed, or reordered.

To not delay adaption, a control exchange interface is dedicated to sending notifications between the PR Adaptor and each operator. To adapt which and where static and dynamic criteria are evaluated, the PR Adaptor simply sends each significance classifier a notification about their new static and dynamic assessment sets.

Operators do not send tuples to any SC whose assessment sets are empty. Such SCs are skipped. To skip extraneous SCs, standard operators control where they send results. They send results to either: 1) the down stream significance classifier sc_p and then to the down stream operator op_o or 2) directly to op_o, i.e., skipping sc_p. To adapt where results are sent, the PR Adaptor notifies each operator of the whether or not to send their results to the down stream SC.

In short, PR can quickly adapt the PR plan online without requiring any infrastructure changes. Instead, each operator locally adapts online how they allocate resources to any future inprocess tuples (Sec. 6.3.3).

6.7 Experimental Evaluation of PR

6.7.1 Experimental Setup

Alternative Solutions. We compare PR with the PR-Prune Optimizer (or PR-Prune) to our PR with basic PR Optimizer solution (or PR). Both approaches identify and pull promising tuples forward. However, PR-Prune reduces the optimization time for criteria placement which allows PR-Prune to pull promising tuples forward sooner than PR. We also study traditional data stream management systems which does not employ any resource allocation methodology (or Trad). Trad demonstrates that our experimental scenarios require a resource allocation methodology to ensure the throughput of the most significant results. We also analyze the state-of-the-art resource allocation methodologies, namely, semantic (or Sem), random (or Rand), and Proactive Promotion (or PP) (Ch. 4).

PP is the closest competitor to PR. However, it only pulls significant tuples forward. It does not identify nor pull forward promising tuples. Sem selects which incoming significant tuples will be processed (or dropped) upon their arrival. Rand randomly selects tuples to process upon their arrival based upon the estimated number of tuples that can be processed within their lifespan given the current statistics. Both Sem and Rand process all tuples in FIFO order. In contrast, PP locates significant tuples at the optimal SC operator in the query pipeline using a cost-based optimization strategy. PP processes all tuples in rank order. Studying these approaches highlights the benefits of pulling promising tuples forward (i.e., PR and PR-Prune). Trad simply processes all tuples in FIFO order. It neither sheds nor allocates resources to specific tuples based upon their rank.

PR-Prune, PR, PP, and Sem use the same set of static criteria to identify significant tuples. Unlike the other approaches, PR-Prune and PR generate dynamic criteria to identify promising tuples. In our experiments, operators in PR-Prune and PR send join criteria statistics whose frequency is above 5% to the PR monitor.

Experimental Methodology. Compared to alternative solutions, we explore the following research questions: 1) Is PR-Prune more effective at increasing the throughput of the most significant results? 2) What affect does the number of promising tuples in the streams have on the effectiveness of PR-Prune? 3) How does varying the number of join operators or dynamic evaluation paths affect PR-Prune? 4) How does the optimization search time of PR-Prune compare to that of the PR? 5) What is the runtime CPU and memory overhead of PR-Prune?

Our experiments adapt the variables that most directly affect PR. The quantity of promising tuples affects the number of tuples in the workload that may benefit from being pulled forward. PR's effectiveness is based upon promising tuples arriving at their designated join operator and creating significant join results.

The number of join operators in a query affects the query complexity and number of tuples in the workload that may benefit from being pulled forward.

Varying the number of dynamic evaluation paths affects the optimization search time and the number of operators over which each promising tuple can be pulled forward.

Queries. Most experiments use the Stock Market Join Query (Sec. 1.3.2) with the P-CQL extension. There are three static monitoring levels that define the significant tuples in the stock stream. We henceforth refer to this query as the *Stock Market Query*.

Data Streams. The stock market stream used is the same stock market stream generated in Section 5.5.

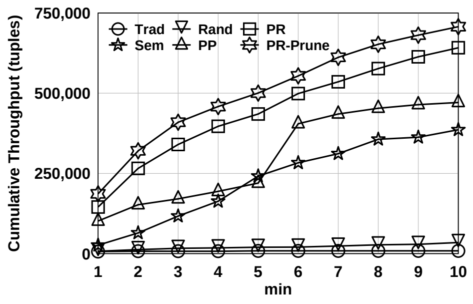
In *Stock Data Set 1* , 4.6% of the stocks in the stock stream were randomly chosen to respectively have significant ranks 1, 2, and 3.

Similar to generated in Section 5.5, our news and blog data streams were created by randomly selecting from either the sectors in the Global Industry Classification Standard (GICS). In these experiments we also generated news and blog data streams from the subsectors in the Industrial Classification Benchmark (ICB). These streams represent the industries or business sectors mentioned in current postings.

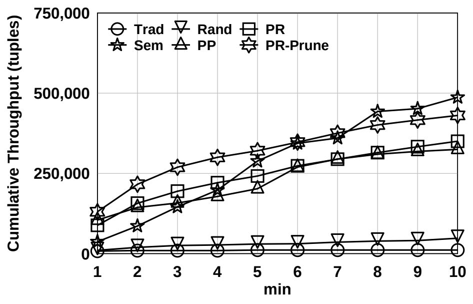
Most experiments use the *News/Blog Data Set 1* which mimics days when no particular ICB subsector dominates the news. In News/Blog Data Set 1, the ICB subsectors were randomly placed into the news and blog streams with the constraint that 15% of tuples in each window in the news and blog streams are promisimg tuples, i.e., from ICB subsectors of the significant tuples in the Stock Data Set 1.

Hardware. Our experiments were conducted in a compute cluster. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory. Each solution (i.e., PR-Prune, PR, PP, Sem, or Trad) was distributed across 2 processing nodes. The query executor was run on one node. The system monitoring, optimizing, and plan adaption components were run on the other node.

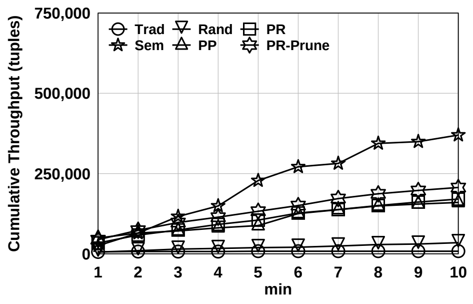
Metrics and Measurements. All experiments were run 3 times for 10 minutes. The results are averaged over these runs. The majority of our experiments measure separately for each monitoring level the cumulative throughput of significant query results.



a) Throughput for Level 1



b) Throughput for Level 2



c) Throughput for Level 3

Figure 6.4: Effective at Increasing Throughput of the Most Significant Results

6.7.2 Experimental Results

Effectiveness at Increasing Throughput of the Most Significant Results. First, we compare the throughput of each approach. This experiment uses Stock and News/Blog Data Set 1, and the Stock Market Join Query. The query lifespan = 1 billion ns. The window size = 1,000 tuples.

Figures 7.4 a-c show the average cumulative throughput for monitoring levels 1 - 3 respectively as it changes over 10 minutes. Overall compared to the alternative approaches PR-Prune produced more of most significant results (i.e., level 1) (Fig. 7.4 a). In addition, for the second most significant results (i.e., level 2), PR-Prune produced more results than PR, PP, Rand, and Trad (Fig. 7.4 b). Only Sem produced slightly more significant results at level 2 than PR-Prune. Finally, for the least significant monitoring level, (i.e., level 3) PR-Prune produced more results than PR, PP, Rand, and Trad (Fig. 7.4 c). Again, only Sem produced more of the least significant results than PR-Prune.

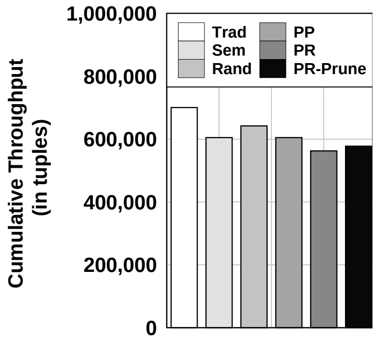
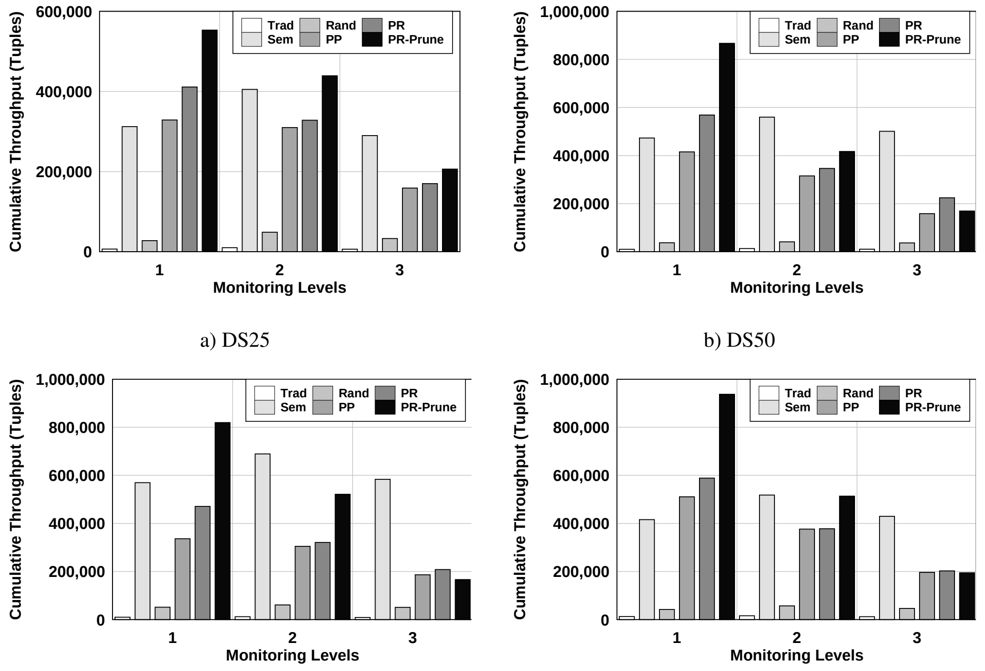


Figure 6.5: Overall Throughput of Correct Results

This is in line with how we expect PR-Prune to function. Namely, the goal of PR-Prune is to dedicate the resources to produce the most significant results, followed by the next most significant results (and so on) until no resources remain. PR-Prune is indeed effective at increasing the throughput of the most significant results. This is due to PR-Prune's ability to pull forward both significant and promising tuples. In addition, although Sem produces more significant results at monitoring levels 2 and 3, it produces significantly less of the most significant results. In the Stock Market Example this could have severe financial consequences.

Figure 6.7.2 shows the overall throughout of all query results (regardless of significance) using the experimental set up outlined above. Overall compared to the alternatives, *PR* achieved the lowest overall throughput. PR produced 2% fewer results than PR-Prune. It produced 4% fewer results than PP and Sem. Compared to Rand, it produced 10% fewer results. Finally, compared to Trad it produced 15 % fewer results. This is by design. PR is designed for EMAs where they is a need to ensure that at all costs certain tuples are processed. The overhead to support PR takes away resources from processing insignificant tuples.



a) DS25 b) DS50

c) DS75 d) DS100

Figure 6.6: Varying the Number of Promising Tuples

Varying the Number of Promising Tuples. We now explore how the number of promising tuples in the streams affects the throughput of significant results. This experiment uses the setup described above. However in this experiment, we use four distinct News/Blog Data Sets. These datasets emulate days where particular ICB subsectors dominate the news.

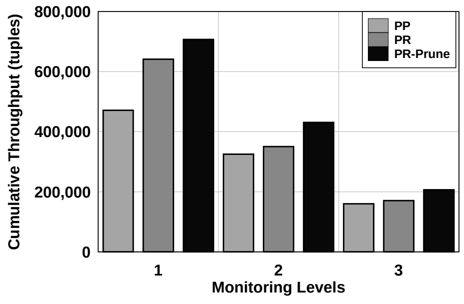
In the 25% News/Blog Data Set (*DS25*), 25% of ICB subsectors of the significant tuples in the Stock Data Set 1 were randomly selected and placed into tuples in each window in the news and blog streams. In other words, only 25% of all the significant tuples have corresponding join partners and 75% have no join partners. Similarly, in the 50% (*DS50*), 75% (*DS75*), and 100% (*DS100*) News/Blog Data Sets, respectively 50%, 75%, and 100% of ICB subsectors of the significant tuples in the Stock Data Set 1 were randomly selected and placed into tuples in each window in the news and blog streams. In DS50, DS75, and DS100, respectively only 50%, 75%, and 100% have corresponding join partners, i.e., 50%, 25%, and 0% have no join partners. Figures 7.8 a-d show the average cumulative throughput for monitoring levels 1 - 3 respectively after 10 minutes respectively for the DS25, DS50, DS75, and DS100 Data Sets.

In this experiment, there is no connection between the number of significant results produced by each scenario as each scenario uses distinct News and Blog streams. Therefore they cannot be directly compared to each other. However, we can observe trends in how PR-Prune performs in scenarios where there are few versus many promising tuples.

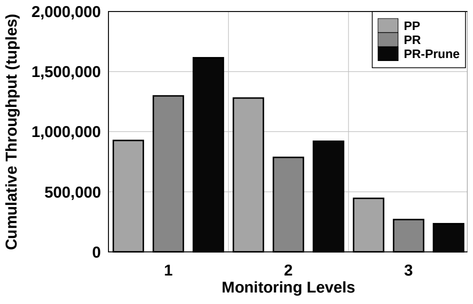
Regardless of the data set used, PR-Prune produced more of the most significant results than the other approaches. The gains achieved by PR-Prune increases when the stream contains fewer promising tuples (e.g., DS25) compared to when the stream contains more promising tuples (e.g., DS100).

In all scenarios, compared to Trad, PR-Prune between 71 and 85 fold more of the most significant results (i.e., level 1). Respectively for DS25, DS50, DS75 and DS100, PR-Prune produced between 1.3 and 19.8 fold, 1.5 and 23 fold, 1.7 and 15.9 fold, and 1.6 and 22 fold more of the most significant results (i.e., level 1) than PR, PP, Sem, and Rand. This is as expected. Namely, PR-Prune is designed to pull the promising tuples forward that have the estimated highest potential of producing significant query results.

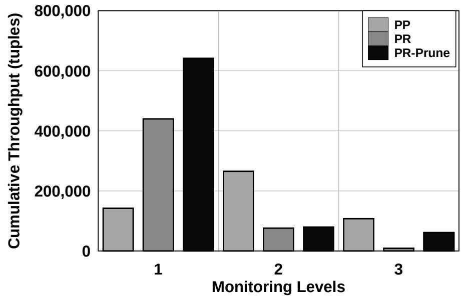
Locating promising tuples adds overhead. PR-Prune efficiently determines which dynamic criteria to use to locate promising tuples while taking the overhead of locating promising tuples. Compare the results in the DS100 scenario to the DS25 scenario. DS100 has more possible dynamic criteria to evaluate than DS25. However, even with these additional dynamic criteria in the DS100 scenario PR-Prune still produced more of the most significant query results than the other approaches.



a) 2 join operators



b) 4 join operators



c) 8 join operators

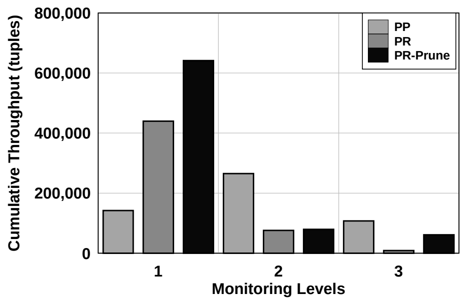
Figure 6.7: Varying the Number of Join Operators

Varying the Workload and Number of Dynamic Evaluation Paths. We now compare how varying the workload and number of dynamic evaluation paths affects PR-Prune compared to PP and PR. Comparing PR-Prune to PP demonstrates the benefit to pulling promising tuples forward. The comparison to PR highlights the advantage of reducing the optimization time and thus quickly pulling promising tuples forward. Similar to the experiment above, there is no connection between the number of significant results produced by each scenario in these experiments.

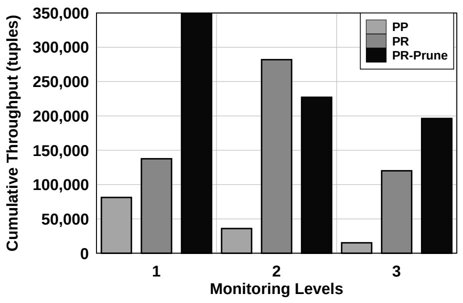
Varying the Number of Join Operators. This experiment again uses Stock and News/Blog Data Set 1 where the query lifespan = 1 billion ns, and the window size = 1,000 tuples. However in this experiment we vary the number of join operators from 2, 4, to 8 operators. The 2 join operator experiment uses the Stock Market Join Query. While the 4 and 8 join operators query plans respectively extend the Stock Market Join Query by 2 and 6 join operators. Each join operator added combines the current results respectively with an additional news or blog stream. Such queries are used to locate hot news trends across multiple news sources. Figures 7.6 a-c show the average cumulative throughput for monitoring levels 1 - 3 respectively after 10 minutes. Overall PR-Prune consistently produced more of the most significant results (i.e., level 1) than PR and PP. Clearly, PR-Prune is better at increasing the throughput of the most significant results in complex the queries than the competitors.

Varying the Number of Dynamic Evaluation Paths. This experiment uses the Stock and News/Blog Data Set 1 and the 8 join query outlined above where the query lifespan = 1 billion ns, and the window size = 1,000 tuples. The number of dynamic evaluation paths is varied by adjusting how many of the consecutive join operators shared the same join attribute.

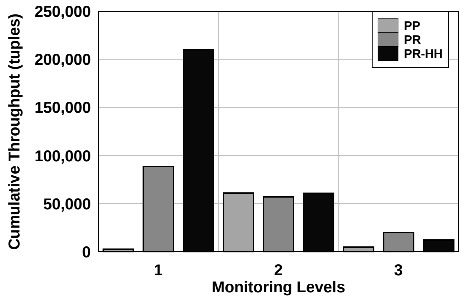
In the 1 path query, all 8 join operators share the same join attribute. PR-Prune



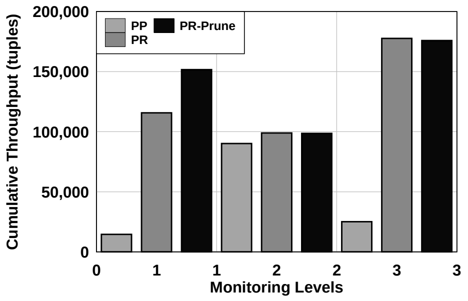
a) Throughput for 1 path



b) Throughput for 2 paths



c) Throughput for 4 paths



d) Throughput for 8 paths

Figure 6.8: Varying the Number of Dynamic Evaluation Paths

will seek to pull tuples forward across the entire query path. In the 2, 4, and 8 path queries respectively 4, 2, to 0 (no) join operators share the same join attribute. In the 2 path query PR-Prune will seek to pull tuples forward across the first and the last four consecutive join operators in the query pipeline. In the 4 path query PR-Prune will seek to pull tuples forward across the first, second, third, and last two consecutive join operators in the query pipeline. In the 8 path query PR-Prune will seek to pull tuples forward across each join operator individually.

To achieve this we vary which incoming news and blog streams are generated from the GICS sectors and which streams are generated from ICB subsectors. When there is 1 path, all 8 news streams are generated from ICB subsectors, i.e., all

8 join operators share the same join attribute. Thus PR-Prune would pull promising tuples forward across all 8 join operators. In the 8 paths case, every other news streams is generated from ICB subsectors or GICS sectors, i.e., no consecutive join operators share the same join attribute. In this case, PR-Prune would pull promising tuples only across one individual join operator at a time.

Figures 7.9 a-c show the average cumulative throughput for monitoring levels 1 - 3 respectively over 10 minutes. Overall PR-Prune consistently produced more of the most significant results (i.e., level 1) than PR and PP. It can be seen that PR-Prune produces more highly significant results than PR and PP regardless of the number of dynamic evaluation paths. Clearly, the gains in the throughput of the most significant results achieved by PR-Prune are also not affected by the number of dynamic evaluation paths.

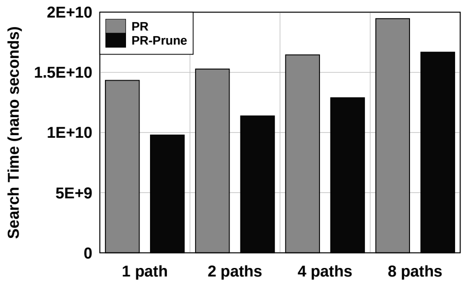
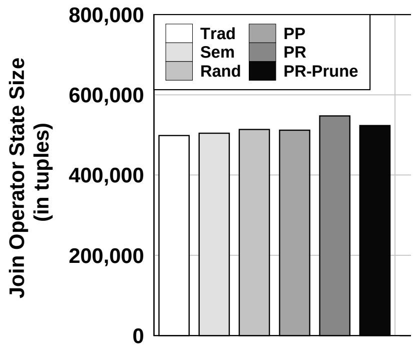


Figure 6.9: Optimization Search Time

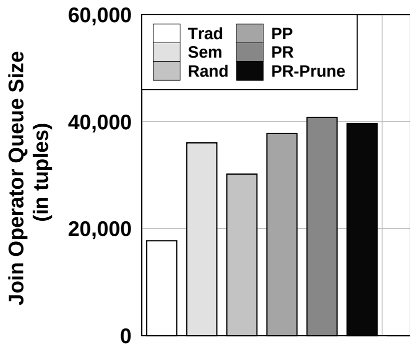
Optimization Search Time. We now compare the optimization search time of PR-Prune to PR to locate the "best" or optimal PR plan for queries that contain a varied number of dynamic evaluation paths (Figure 6.9). Namely, we analyze the optimizer search time for the 1, 2, 4, and 8 path experiment outlined above. In the 1 path case, PR-Prune takes 31.6. % less time time than PR to search for the "best"

PR plan, while in the 8 paths case, PR-Prune takes 14.3 % less time.

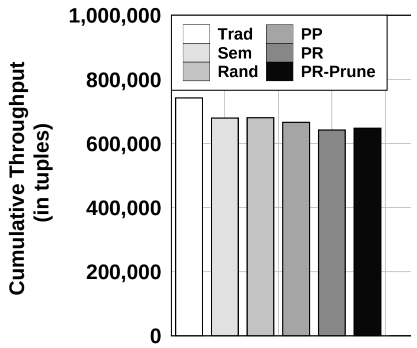
This is as expected. In the 1 path case, PR Prune will eliminate potential designated operators and combine all join operators into a single dynamic evaluation path (Sec. 6.5.5). In the 8 path case, PR Prune will not be able to eliminate any potential designated operators and a dynamic evaluation path will be created for each join operator. However, in both cases PR Prune reduces the optimizer search time by eliminating inferior dynamic criteria (Sec. 6.5.5).



a) Join Operator State Size (in number of tuples)



b) Join Operator Queue Size (in number of tuples)



c) Cumulative Throughput

Figure 6.10: Execution-Runtime CPU and Memory Overhead

Execution-Runtime CPU Overhead. To measure the runtime overhead we evaluate the cumulative throughput using the worst case scenario for PR-Prune (Fig.

c), namely, when no static monitoring levels are defined.This experiment uses the Stock Market Join Query (Sec. 1.3.2) without the P-CQL extension. In addition, no tuples expire (i.e., query lifespan $=\infty$). Thus, all tuples are processed in FIFO order. The overhead of the state-of-the-art resource allocation methodologies (Sem, Rand, PP, PR, PR-Prune) here corresponds to the cost to collect and evaluate runtime statistics. This is the worst case scenario as although these systems will never be overloaded, they will continue to evaluate how to allocate resources. This experiment uses the Stock and News/Blog Data Set 1 where the window size is 1,000 tuples.

As shown by our results (Fig. 7.7 c), the overhead of PR-Prune compared to the overhead of the alternative state-of-the-art resource allocation methodologies is minimal. PR and PR-Prune require more detailed statistics than PP and Sem (Sec. 6.4). That is, PP and Sem only collect statistics to activate static monitoring levels. In contrast, PR-Prune and PR collect statistics to identify and activate static and dynamic monitoring levels. Rand and Trad have respectively significantly less and no statistics gathering overhead than the other approaches. However, as shown in the experiments above, this minimal overhead is well worth it in systems that require preferential resource allocation.

Memory Overhead. We now evaluate the average number of tuples in the state and input queues of the last join operator (i.e., operator op_2 in the Stock Market Join Query) in the query pipeline using the scenario outline above (Fig. 7.7 a & b). As per our results, the memory overhead of PR-Prune is comparable to the current state-of-the-art approaches.

All approaches process tuples in FIFO order. In PP, PR, and PR-Prune, all tuples will reside in the insignificant queue. The number of tuples in the queue of PR-Prune and PR is slightly higher than the other approaches. This is mainly due to the extra overhead to support additional monitoring statistics which leaves PR-Prune and PR less resources to process tuples.

Sem, Rand, and Trad process tuples in FIFO order. Their join operators internally determine when to purge their states by tracking when it processes the last tuple from a given window. PR does not necessarily process tuples in arrival time order. It uses punctuations to signal when to purge their states (Sec. 4.2.2). This causes the purge to be delayed longer than for other methods and subsequently the state to be slightly larger.

6.7.3 Summary of Experimental Findings

We now summarize our key findings.

1) PR-Prune increases the throughput of significant results (between 1.3 to 23 fold).

2) Regardless of the number of promising tuples in the streams, the query complexity, or the number of dynamic evaluation paths, PR-Prune successfully produced a larger quantity of highly significant results than the state-of-the-art competitors.

3) The optimization search time of PR-Prune is significantly lower than PR especially for scenarios with dynamic evaluation paths that contain multiple operators (roughly 32% reduction in search time).

4) PR-Prune does require a modest CPU and memory overhead. As shown in our experiments above, this overhead is justified in systems that must ensure that resources are dedicated to producing the most significant query results first.

Chapter 7

Join Processing in Preferential Result Streams

7.1 Join Operators

This chapter provides details on our work towards Task 4 (i.e., Preferential Result Join Operator). The main goal of Task 4 is to design a join operator that utilizes the available CPU resources to maximize the production of significant results in precedence order.

7.1.1 A General Stream Join Operator

A symmetric binary hash join maintains a state for each input stream that it combines. For example in Figure 1.4, the join operator maintains one state for stock stream s_1 and news stream s_2 respectively. The join operator processes tuples by performing the following set of subtasks as one single atomic process:

1. *Select* a tuple t_i from an input queue to process next.

2. *Insert* this tuple t_i into the state for tuple t_i's stream to serve as future join partners for tuples that may arrive later.

3. *Search* the state of the opposite stream for the set of tuples JP_i that match tuple t_i's join attribute and fall into t_i's window. We refer to tuple t_i as the *probe tuple* as it probes the state, while we call the set of tuples JP_i the *set of join partners for tuple t_i*.

4. *Join* probe tuple t_i with each join partner t_j in the set of join partners JP_i to produce join results (t_i,t_j).

The join operator continues to execute this set of subtasks until either no resources remain or all queues are empty.

The sequence of four subtasks executed by this join for a given probe tuple constitutes one execution pass. We introduce the term *atomic result production* to denote the process of producing all join results for a probe tuple in a single execution pass.

7.1.2 PR Join Operator

We now review the enhanced symmetric hash *join operator adopted by PR*. The special purpose enhancements include assigning significance properties to join results and scheduling to process incoming tuples in significance order. These enhancements are underlined in the list of subtasks below. Each join result (t_i,t_j) is assigned the highest significance properties of those of its probe tuple t_i and its join partner t_j. This join operator processes tuples by performing the following set of subtasks (See logic in Fig. 7.1).

1. *Select* a tuple t_i to process next from the <u>most significant input</u> <u>queue that contains</u>

tuples (line 6 in Fig. 7.1).

2. *Change tuple t_i's significance properties.* If tuple t_i is a promising tuples tuple and this join operator is tuple t_i's designated operator then tuple t_i's potential significance properties are set to null. (lines 7-9 in Fig. 7.1).

3. *Insert* tuple t_i into the state for tuple t_i's stream (line 10 in Fig. 7.1).

4. *Search* the state of the opposite stream for the set of join partners JP_i for tuple t_i (line 11 in Fig. 7.1).

5. *Join* probe tuple t_i with each join partner t_j in JP_i to produce join results (t_i,t_j). Each result tuple (t_i,t_j) is assigned the highest significance properties of the probe tuple t_i and join partner t_j (line 12 in Fig. 7.1). Output the join results (t_i,t_j) (line 13 in Fig. 7.1).

Once all incoming tuples from one queue have been processed, then the incoming tuples in the next highest significance queue are processed (lines 13-18 in Fig. 7.1). This continues until either no resources remain or all queues are empty (line 4 in Fig. 7.1).

Clearly, this enhanced symmetric hash join adopted by PR also conforms to atomic result production. Recall that this approach is *result significance-agnostic* (Sec. 1.7.3), i.e., each tuple produces all possible results regardless of their significance.

7.2 Principles of PR-Join Strategy

7.2.1 Significance of a Join Result

The significance of a join result (t_i,t_j) produced when processing tuple t_i is affected by the significance of tuple t_i itself at the time the join result is formed.

Algorithm ***State-of-the-Art Join***($Q_{inc}(s_1)$ & $Q_{inc}(s_2)$ incoming queues for streams s_1 & s_2, C_{avail} - available resources)

```
1: for $x = 1$ to 2 do
2:   $C_{avail}(s_x) \leftarrow$ avail. res. to process tuples from $s_x$
3:   $LevelProcessed \leftarrow 1$
4:   while (($C_{avail}(s_x) > 0$) and ($Q_{inc}(s_x)$ is not empty)) do
5:     if ($Q_{inc}(s_x, LevelProcessed)$ contains a tuple) then
6:       select Tup $\leftarrow$ first tuple in $Q_{inc}(s_x, LevelProcessed)$
7:       if (Tups designated operator is this join operator) then
8:         set Tup's potential significance to null
9:       end if
10:      insert Tup in the state for $s_x$
11:      search for join partners in the state of the opposite stream
12:      join Tup with all join partners in the state of the opposite stream
13:      place join results into input queue of next query plan operator
14:    else
15:      $LevelProcessed \leftarrow$ LevelProcessed + 1
16:    end if
17:    if $LevelProcessed >$ max(Monitoring Level) then
18:      $LevelProcessed \leftarrow$ insignificant tuples;
19:    end if
20:  end while
21: end for
```

Figure 7.1: State-of-the-Art Join operator

Observation 1: A tuple that arrives at a join operator may lose its significance properties before it creates join results.

An incoming tuple t_i to a join operator can have one of six types of significance properties. Tuple t_i can be either 1) only a significant tuple, 2) only a promising tuple where the join operator is not the designated operator, 3) only a promising tuple where the join operator is the designated operator, 4) both a significant tuple and a promising tuple where the join operator is not the designated operator, 5) both a significant tuples and a promising tuple where the join operator is the designated operator, or 6) an insignificant tuple.

A tuple t_i's potential significance properties can be set to null before tuple t_i creates join results (Sec. 7.1.2 subtask 2). In particular, this occurs when an incoming tuple t_i is only a promising tuple and the join operator is the designated operator (i.e., type 3 above) or both a significant tuple and a promising tuple where the join operator is the designated operator (i.e., type 5 above). In the former case, tuple t_i will be an insignificant tuple when tuple t_i creates join results. In the later case, tuple t_i will be a significant tuple when tuple t_i creates join results.

Thus there are only four types of significance that tuple t_i may have when it creates join results (subtask 5 in Sec. 7.1.2). Tuple t_i can be either 1) only a significant tuple, 2) only a promising tuple where the join operator is not the designated operator, 3) both a significant tuple and a promising tuple where the join operator is not the designated operator, or 4) an insignificant tuple.

Observation 2: Any join result created from a significant tuple t_i at level lvl_z is as or more significant than the significance of the tuple t_i itself when the join result is formed.

This follows directly from the join operator definition (Sec. 7.1.2) as it assigns the more significant rank of the two join partners to the join result at the time the join result is formed.

Lemma 1 Consider that probe tuple t_i and join partner t_j create join result (t_i,t_j) at join operator op_o.
Case 1: If probe tuple t_i is a *significant tuple* when join operator op_o creates join result (t_i,t_j) then the result is guaranteed to be as or more significant than probe tuple t_i.
Case 2: If probe tuple t_i is an *insignificant tuple* when join operator op_o creates join results (t_i,t_j) then the significance of the result solely depends upon the significance of join partner t_j.

Proof *In case 1, any join result (t_i,t_j) generated by probe tuple t_i will be as or more significant than level lvl_z (per Observation 2). On the other hand, in case 2, the* *significance of join result (t_i,t_j) solely depends upon whether or not join partner t_j is significant (per Observation 2) as tuple t_i is an* insignificant tuple *at the time the join results (t_i,t_j) are formed.*

7.2.2 Processing Probe Depth

When the available CPU resources are limited, only the most significant join results should be produced. We now consider which results should be produced by a given probe tuple to ensure that only the significant join results at significance level lvl_z are produced first. The *processing probe depth* of probe tuple t_i defines which tuples that probe tuple t_i should join with during a single execution pass of the join subtasks (Sec. 7.1.2).

Lemma 2 Assume all join results more significant than level lvl_z have been created. Join operator op_o is dedicating resources to produce as many join results at level lvl_z as possible.
Case 1: If probe tuple t_i is a *significant tuple at level* lvl_z then the processing probe depth when operator op_o processes probe tuple t_i should be a full scan, i.e. *atomic result production* (Sec. 7.1.1).
Case 2: If probe tuple t_i is an *insignificant tuple* at level lvl_z then the processing probe depth when operator op_o processes probe tuple t_i should correspond to the join results at significance level lvl_z, i.e. *non-atomic result production*.

Proof *In case 1, any join result (t_i,t_j) generated by probe tuple t_i will be as or more significant than level lvl_z (per Lemma 1). Assume that not all join results (t_i,t_j) produced by probe tuple t_i are created. At some point, resources may be allocated to producing join results less significant than level lvl_z. If few resources* *are available then some join results at level lvl_z will not be produced. The resources allocated to producing the less significant join results could have been allocated to producing such tuples. Thus, the probe tuple t_i should be processed in its entirety.*

In case 2, not all join results generated by probe tuple t_i will be as significant as level lvl_z (per Lemma 1). Assume that all join results (t_i,t_j) produced by probe tuple t_i are created. In this case, resources may be allocated to producing join results less significant than level lvl_z. If few resources are available then some join results at level lvl_z will not be produced. The resources allocated to producing the less significant join results could have been allocated to producing such tuples. Thus, the probe tuple t_i should not be processed in its entirety. Probe tuple t_i should only join with significant join partners t_j at level lvl_z.

We now outline the process of *non-atomic result production*. In a nutshell, a probe tuple t_i only joins with join partners at a given significance level each time a state scan occurs. To produce all join results for a given probe tuple requires multiple executions of the join operator search and join subtasks. Each execution pass results in the creation of join results only at one given significance level.

Non-atomic result production is beneficial when promising tuples arrive at their designated operator and the significance of their join partners varies. Assume that promising tuple t_i arrives at its designated operator. Consider the following scenarios. Case 1: All tuple t_i's join partners are significant tuples at level lvl_z. In this case, once probe tuple t_i joins with all its join partners at significance level lvl_z then no join partners are left for probe tuple t_i to join with. Thus, there is no need to run multiple execution passes (i.e., non-atomic result production) as a single pass will results in the creation of all join results. Case 2: Some of tuple t_i's join partners are less significant than level lvl_z. In this case, this execution pass would only produce join results at significance level lvl_z. This would delay the production of tuple t_i's less significant join results and free resources to produce results at significance level lvl_z from other tuples, i.e., it provides a benefit.

7.2.3 Processing Order of Probe Tuples

Beyond the processing probe depth of each probe tuple, we also consider the design choice of which probe tuples should be processed first. In particular, we analyze if we should process a significant tuples probe tuple at level lvl_z versus a promising tuples probe tuple at level lvl_z first (or visa versa).

Observation 3: Unless the join operator is the last operator in the query pipeline, there is no guarantee that a join result (t_i,t_j) regardless of the type of it's significance properties will satisfy all query constraints remaining in the pipeline and thus produce a significant final query result.

Hence we propose to process significant tuples and promising tuples at the same level in arrival time order.

7.2.4 Atomic & Non-Atomic Result Production

Theorem 6 If probe tuple t_i is significant at level lvl_z and *atomic result production* is used to produce all of tuple t_i's join results then all join results (t_i,t_j) will be as or more significant than level lvl_z.

Proof *This is directly derived from Lemma 1. Any join result of probe tuple t_i is guaranteed to be as or more significant than level lvl_z. Thus, if join operator op_i performs atomic result production on probe tuple t_i, then all results produced will also be as or more significant than level lvl_z.*

Theorem 7 If probe tuple t_i is insignificant and *non-atomic result production* is used to join tuple t_i only with significant join partners t_j at level lvl_z then all their join results (t_i,t_j) will be at significant at level lvl_z.

Proof *By Lemma 1, join result (t_i,t_j) is guaranteed to be as significant as level lvl_z. Thus if join operator op_i performs non-atomic result production on probe tuple t_i (i.e., only joins tuple t_i with significant join partners t_j whose level is lvl_z) then each result produced will be at significance level lvl_z.*

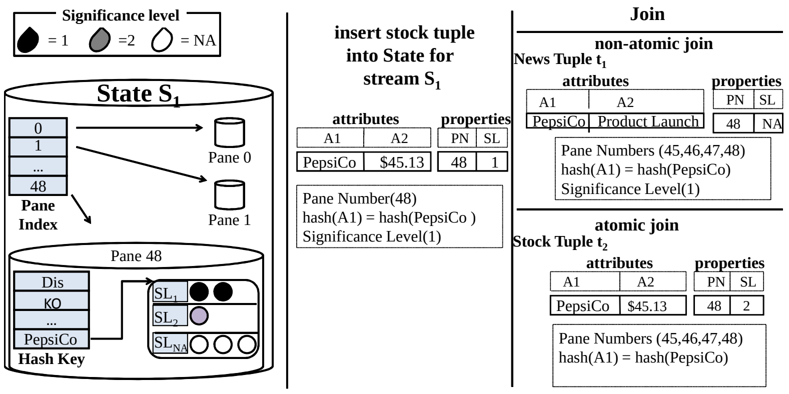


Figure 7.2: Join State Example

7.3 PR-Join Design & Algorithm

To maximize the production of significant join results in precedence order, we now design a preferential result join operator (or PR-Join) that supports both atomic and non-atomic result production based on Thms. 6 - 7.

7.3.1 PR-Join Run-Time Infrastructure Design

We first describe the data structure we design to efficiently support both atomic and non-atomic result production.

Infrastructure Design to Store Probe Tuples: PR-Join must manage the interruption and resumption of probe runs. It must offer a light-weight mechanism to correctly maintain their partial join states and overall join status of progression to assure that inprocess probe tuples can produce results in the future. This includes not only the management of unprocessed incoming tuples but also the status of the partially processed probe tuples due to non-atomic result production. Such *in-process probe tuples* may not have produced all their join results yet. Recall that non-atomic result production joins a probe tuple with join partners at a given significance level. To efficiently locate probe tuples by the significance levels that they need to produce join results, we thus propose to create an *inprocess queue* for each significance level and one for insignificant tuples. Each inprocess probe tuple t_i is stored in the *inprocess queue* for the next significance level at which it next should create results for if it were to be awoken again due to remaining resources. Each unprocessed incoming tuple t_i is stored in the *incoming queue* according to its own significance level.

State Design to Store Join Partners: To support atomic result production, tuples stored in the state must be efficiently locatable by join criteria. To support non-atomic result production, tuples stored in the state must be found by both their significance level and join criteria. To complicate matters, PR does not always process tuples in arrival time order. As a consequence, states may maintain tuples that belong to multiple windows. Hence the state design must support flexible access to stored tuples by join criteria, window, and significance level. Beyond search requests, each purge state scan requires locating tuples by its window. Section 7.3.2 covers how PR-Join efficiently purges tuples from the join states. Both search requests and purge processes share the requirement of locating stored state tuples by the windows they belong to.

To efficiently support both operations, stored tuples are first grouped by the windows they belong to. To quickly identify stored tuples that fall within a given window, states are divided into groups of tuples (a.k.a. *panes*) of a constant length (Sec. 4.2.2). When creating results, only panes who compose the query window of the result being generated are searched.

The goal of the indexing by significance levels is to identify the few significant tuples at a given desired significance level amongst the multitude of tuples in the stream. Thus, we estimate that there are likely to be more tuples that have the same join criteria as compared to the number of tuples that have the same significance level. Hence, to limit the length of the search scan, we next index stored tuples on join criteria. Namely, within each pane, tuples are hashed based on their join criteria. Lastly, tuples with the same join criteria must be efficiently located by their significance level. The number of significance levels is narrow, countable, and a known constant. Thus rather than creating a hash key for each significance level, an array of linked lists is created. The number of elements in the array is equal to the number of monitoring levels plus 1. Namely, there is one linked list in the array for each significance level and one for insignificant tuples. Tuples are stored within the linked list that represents their significance.

Consider the insertion of a stock tuple $t_1 < PepsiCo, \$45.13 >$ that belongs to pane 48 and significance level 1 into the state for stream s_1 (Fig. 7.2). First, the group of tuples for pane 48 is located by locating the hash key for pane 48. Next, the hash bucket applied to the join attribute (i.e., $A1 = PepsoCo$) is found. Finally tuple t_1 is appended to the linked list for significance level 1.

Algorithm ***PR-Join***$(Q_{inc}(s_1)\&Q_{inc}(s_2)$ incom. Qs. for $s_1\&s_2, C_{avail}$ avail. res., $Q_{inp}(s_1)\&Q_{inp}(s_2)$ inpro. Qs. for $s_1\&s_2)$

```
1: for $x = 1$ to 2 do
2:     $C_{avail}(s_x) \leftarrow$ avail. res. to process tuples from $s_x$
3:     LevelProcessed $\leftarrow 1$
4:     while (($C_{avail}(s_x) > 0$) and ($Q_{inc}(s_x)$ and $Q_{inp}(s_x)$ are not empty)) do
5:         if ($Q_{inc}(s_x, LevelProcessed)$ or $Q_{inp}(s_x, LevelProcessed)$ contains a tuple) then
6:             select Tup $\leftarrow$ oldest tuple of the first tuple in $Q_{inc}(s_x, LevelProcessed)$ or
               $Q_{inp}(s_x, LevelProcessed)$
7:             if (Tups poten. designated operator is this PR-Join Op) then
8:                 set Tup's potential significance to null
9:             end if
10:            if (Tup is from an incoming queue) then
11:                insert Tup in the state for $s_x$
12:            end if
13:            if (Tup is insignificant or from inprocess queue) then
14:                search for join partners in the state of the opposite stream for level LevelProcessed
15:                join Tup with only the join partners in the state of the opposite stream for level LevelProcessed
16:                if (LevelProcessed $\neq$ insignificant) then
17:                    place Tup into the queue into the next level more significant than $LevelProcessed$ that
                       contains tuples
18:                end if
19:            else
20:                search for join partners in the state of the opposite stream
21:                join Tup with all join partners in the state of the opposite stream
22:            end if
23:            place join results into input queue of next query plan operator
24:        else
25:            LevelProcessed $\leftarrow$ LevelProcessed + 1
26:        end if

27:        if LevelProcessed $>$ max(Monitoring Level) then
28:            LevelProcessed $\leftarrow$ insignificant tuples;
29:        end if
30:    end while
31: end for
```

Figure 7.3: Preferential Result Join Operator *PR-Join*

7.3.2 PR-Join Algorithm

PR-Join Logic

We now outline our proposed PR-Join algorithm (Fig. 7.3). PR-Join processes tuples using the following set of subtasks. Any enhancements to the state-of-the-art join [WA91] in Section 7.1.2 are underlined.

1. *Select* which tuple t_i to process next from the most significant input or inprocess queue that contains tuples (line 6 in Fig. 7.3).

2. *Change tuple t_i's significance properties.* If tuple t_i is a promising tuple and this join operator is t_i's designated operator then t_i's potential significance properties are set to null. (lines 7-9 in Fig. 7.3).

3. *Insert* tuple t_i into the state for tuple t_i's stream if tuple t_i is from an incoming queue (lines 10-12 in Fig. 7.3).

4. *Search* the state of the opposite stream depending upon the significance of tuple t_i.

4. a) if tuple t_i is insignificant or from an inprocess queue then *search* the state of the opposite stream for the set of join partners JP_i for tuple t_i with their significance level equal to that of the queue that tuple t_i originated from (line 14 in Fig. 7.3). (*Non-Atomic Result Production*)

4. b) Otherwise if tuple t_i is significant and from an incoming queue then *search* the state of the opposite stream for the set of all join partners JP_i for tuple t_i (line 20 in Fig. 7.3). (*Atomic Result Production*)

5. *Place* probe tuple t_i into the inprocess queue of the next significance level to be

processed that contains tuples (lines 16-18 in Fig. 7.3).

6. *Join* probe tuple t_i with each join partner t_j in JP_i to produce join results (t_i,t_j). Each result tuple (t_i,t_j) is assigned the highest significance properties of the probe tuple t_i and its join partner t_j (line 15 or 21 in Fig. 7.3). Output the join results (t_i,t_j) (line 23 in Fig. 7.3).

Once all tuples in the input and inprocess queues from one significance level have been processed, then all tuples in queues for the next highest significance level are processed (lines 25-29 in Fig. 7.3). This continues until either no resources remain or all queues are empty (Fig. 7.3 line 4).

Discussion: During atomic result production the entire set of subtasks is executed once for a single probe tuple and all join results are produced. When PR-Join utilizes non-atomic result production some of the subtasks are executed multiple times by a single probe tuple. During the first probe, the tuple is selected (subtask 1). Its significance properties are changed (subtask 2). The tuple is inserted into the state (subtask 3). Then join results are created at the most significant level (subtasks 4a, 5, and 6). In subsequent probes, the tuple is again selected (subtask 1). Then join results are created at the next most significant level (subtasks 4a, 5, and 6).

Duplicate Handling

We now demonstrate that care must be taken to avoid the generation of duplicate results. Consider the processing of two tuples that can create a join result at operator op_1 (Fig. 1.4). One tuple t_1 is a significant tuple from stock stream s_1 assigned to pane 48 and level 2. The other tuple is a promising tuple t_2 from the news stream s_2 also assigned to the same pane 48 with operator op_1 is its designated operator.

Let us assume that promising tuple t_2 is processed first using *non-atomic result production* because tuple t_2 is not significant when its join results are created. First, tuple t_2's potential significance properties are set to null because operator op_1 is its designated operator. Next, tuple t_2 is added to the state for news stream s_2. Then tuple t_2 creates partial join results by scanning the stock stream state for join partners within the query window (composed of 4 panes, i.e., panes $45, 46, 47, 48$), join criteria (i.e., $company\_name = PepsiCo$), and significance level $= 1$. Under pane 48, tuple t_2 will create 3 results at significance level 1. Finally the tuple t_2 is placed in the inprocess queue for significance level 2.

Next, the significant tuple t_1 is processed using *atomic result production* because tuple t_1 is significant when its join results are created. First, tuple t_1 is added to the state for stock stream s_1. Then tuple t_1 scans the news stream state for any join partners within the window (i.e., panes $45, 46, 47, 48$) and join criteria (i.e., $company\_name = PepsiCo$). Join result (t_2,t_1) will be created.

Later on, after all join results at significance level 1 driven by processing of other tuples have been created, PR-Join allocates resources next to produce join results at significance level 2. PR-Join selects tuple t_2 from the inprocess queue for significance level 2, then tuple t_2 resumes its processing. Now, tuple t_2 creates partial join results by scanning the stock stream state for join partners within the query window (composed of 4 panes, i.e., panes $45, 46, 47, 48$), join criteria (i.e., $company\_name = PepsiCo$), and significance level $= 2$. During this scan, join result (t_2,t_1) will be created again because t_1 is stored in the state within t_2's query window, join criteria, and under significance level $= 2$. Thus, join result (t_2,t_1) could be produced more than once (i.e, duplicated result).

The issue of duplicate results is neither unique nor new. Rather, it is also experienced by adaptive query processing techniques (i.e., *Eddies* [AH00] and *QMesh* [NRB09]) whenever the join operator subtasks are also not performed atomically. In the case of *Eddies* [AH00] and *QMesh* [NRB09], all incoming tuples are stored in the states before they are selectively routed to operators in different orders. That is, the join operator *Insert* subtask (Sec. 7.3.2 subtask 3) is executed for possibly several tuples before all join results for these tuples are created. In our case, a probe tuple may not complete the *Search* (Sec. 7.3.2 subtask 4) and *Join* subtasks (Sec. 7.3.2 subtask 5) for all join results before another tuple is selected to be processed, i.e., *Select* subtask (Sec. 7.3.2 subtask 1) is executed. These approaches solve this issue by utilizing a last push method where the last arriving join partner (i.e., with the latest time stamp) pushes the results out [AH00, NRB09]. That is, the arrival time is the determining factor for join result production.

We propose to use a similar approach in PR-Join. Assume that tuples t_i and t_j create a join result (t_i,t_j). PR-Join only allows the tuple that is the last to begin probing amongst tuples t_i and t_j to create the join result (t_i,t_j). PR-Join assigns each incoming tuple a *start probing time*, namely, the time PR-Join begins searching for join partners, i.e., executes (Sec. 7.1.2 task 4). PR-Join only joins probe tuple t_i with stored join partners who began probing before probe tuple t_i. This eliminates two tuples from creating the same results regardless of their significance or the order in which the tuples produce results.

Theorem 8 PR-Join creates no duplicate join results.

Proof *Consider join result (t_i, t_j) produced by a PR-Join operator. If tuple t_i begins probing before tuple t_j, then only when tuple t_j is probing will the join result (t_i, t_j) be created. If tuple t_j uses* atomic result production *to create join results, then* *the opposite state will only be scanned once by tuple t_j and the join result (t_i, t_j) will be produced once by the probing of tuple t_j. If tuple t_j uses* non-atomic result production *to create join results, then the opposite state may be scanned more than once by tuple t_j. However, each scan will target join partners with a particular significance level. Each tuple t_i will be stored exactly once in the state according to its significance level. The significance level of a stored tuple in the state will never change. Thus, the PR-Join operator op_o will produce join result (t_i, t_j) exactly once, namely, during the probing of tuple t_j. Hence PR-Join will not produce any duplicate results.*

Theorem 9 PR-Join ensures the production of join results in significance order based upon the estimated significance of each probe tuple's join results.

Proof *Consider join result (t_i, t_j) that can be produced by a PR-Join operator. Assume that tuple t_j will create join result (t_i, t_j), namely, tuple t_i begins probing before tuple t_j. There are only two possible types of significance properties that tuple t_j can have when join result (t_i, t_j) is created, namely, either tuple t_j is 1) significant or 2) insignificant.*

In either case, tuple t_j is selected (subtask 1) when resources are being allocated to producing join results at the significance level of input queue in which tuple t_j resides.

In case 1, join result (t_i, t_j) will be at least as significant as the input queue in which tuple t_j resides (Thm. 6). Join result (t_i, t_j) will be created immediately after tuple t_j is taken from the input queue (Thm. 6).

In case 2, join result (t_i, t_j) will only be as significant as tuple t_is significance level (Thm. 7). Join result (t_i, t_j) will be created when PR-Join is allocating re-

sources to produce tuples at tuple t_is significance level and tuple t_j resides in the inprocess queue of tuple t_is significance level.

Hence join result (t_i, t_j) will be produced when resources are being allocated to produce results at the estimated significance of tuples t_i and t_j, i.e., PR-Join produces results in significance order.

Purging Policies. Beyond limited CPU resources, PR must also determine which tuples to store when limited memory resources do not allow all tuples to be stored. To achieve this, PR-Join utilizes the pane structure and purging policies from Section 5.4.4.

7.4 Experimental Evaluation of PR-Join

7.4.1 Experimental Set Up

Alternative Solutions. We compare PR using PR-Join (or PR-Join) to the state-of-the-art PR join strategy (or PR) (Sec. 7.1.2) (Ch. 4 & 6). The key difference between these two solutions is that only PR-Join supports non-atomic result production. That is, both pull the same significant tuples forward. However, only PR-Join is join result significant aware. In essence, we evaluate if and when non-atomic result production (PR-Join) is better than atomic result production (PR) (Sec. 1.7.3). We also study the differences between PR-Join and symmetric binary hash joins implemented in the state-of-the-art resource allocation methodologies, namely, semantic (or sem), and random (or rand) shedding. We analyze the state-of-the-art resource allocation methodologies to demonstrate that a tuple significance-agnostic technique will not resolve the problems addressed by PR-Join (Sec. 1.7.3). In addition, we compare PR-Join to symmetric binary hash joins implemented in a traditional DSMS with no preferential resource allocation optimization (or trad). We analyze trad to show that our experimental scenarios require a resource allocation methodology (such as PR) to ensure the production of the most significant query results.

Experimental Methodology. We first determine the key factors that most directly affect PR-Join. The quantity of significant tuples at different levels affects the number of tuples in the workload that may benefit from non-atomic result production. The window size affects the amount of work that is performed by a join operator. The larger the window size the more tuples that each join operator will need to probe to create results. Similarly, increasing the number of join operators in a query increases the complexity of the pipeline. We aim to demonstrate that PR-Join is effective in both simple (containing one join) and complex (containing many joins) query pipelines. Increasing the window size and query complexity variables puts more pressure upon the system by increasing the workload of significant tuples to process.

Given this analysis, we explore the following: 1) Is our proposed PR-Join join strategy more effective at increasing the throughput of the most significant results than the state-of-the-art solutions? 2) What effect does the number of significant tuples have on the effectiveness of the PR-Join strategy? 3) How does the size of the window or the number of join operators in a query affect the throughput of the most significant results produced by non-atomic (PR-Join) vs atomic result production (PR)? 4) What is PR-Join's runtime CPU and memory overhead in the worst case scenario?

Data Streams and Queries. Unless noted we utilize the Stock Market Join Query in Section 1.3.2. Similar to the monitoring levels defined in Section 6.1.1, there are three monitoring levels that define which tuples in the stock stream are significant.

The stock market stream used is the same stock market stream generated in Section 5.5.

News and blog data streams used is the same news and blog streams generated in Section 5.5.

Hardware. All experiments are conducted on nodes in a cluster that consists of 20 processing nodes. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory.

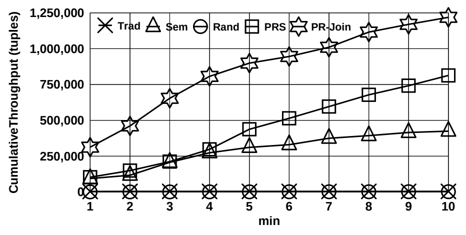
Metrics and Measurements. Our experiments measure throughput separately for each monitoring level. All experiments were run 3 times for 10 minutes. The results are averaged over these runs.

7.4.2 Experiments

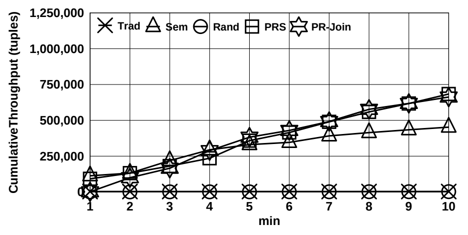
Effectiveness at Increasing Throughput of Most Significant Results

First, we compare the throughput achieved by each of the alternative approaches (Sec. 7.4.1) using Stock Market Join Query and Data Set 1 where the query lifespan is 1,000,000 ns and the window size is 500 tuples. Figure 7.4 a shows the average cumulative throughput for monitoring level 1 over a 10 minute run.

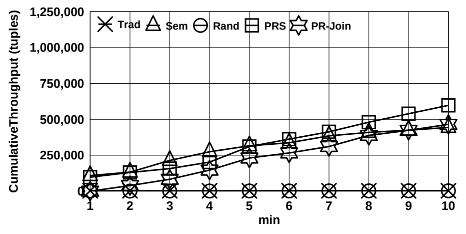
Overall PR-Join compared to all alternatives consistently produces the largest number of most significant results for monitoring level 1. As the overall cumulative throughput in Figure 7.4 d shows, PR-Join produced substantially more of the most significant results (i.e., level 1) than the other methods. PR-Join produced between 1.5 and 190 fold more of the most significant results than the other methods. Now consider the second most significant results (i.e., level 2). PR-Join compared to



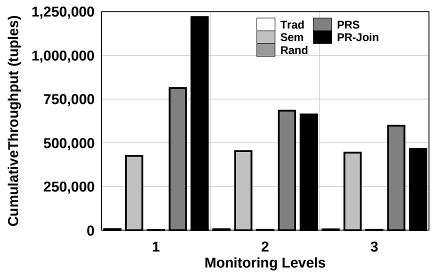
a) Throughput for Monitoring Level 1



b) Throughput for Monitoring Level 2



c) Throughput for Monitoring Level 3



d) Cumulative Throughput

Figure 7.4: Effectiveness at Increasing Throughput of Most Significant Results

PR achieves roughly the same number of results. While compared to sem, rand, and trad, PR-Join produced significantly more results. Finally, consider the least significant results, (i.e., level 3). PR-Join produced less results than PR, while compared to sem, rand, and trad, PR-Join still continues to produce significantly more results. This is exactly how PR-Join should perform. Namely, the goal of PR-Join is to first dedicate the resources to producing the most significant results, followed by the next most significant results, and so on until no resources remain.

We observe that compared to the state-of-the-art join operator using the same PR system, PR-Join increases the throughput of the most significant query results. This is due to the key feature of PR-Join, namely, non-atomic result production. Recall that PR and PR-Join both use the same preferential resource allocation optimization, i.e., PR. However, PR-Join supports non-atomic result production while

PR only supports atomic result production. Our results support that non-atomic result production (i.e., PR-Join) more effectively dedicates resources to producing the most significant results than atomic result production (i.e., PR).

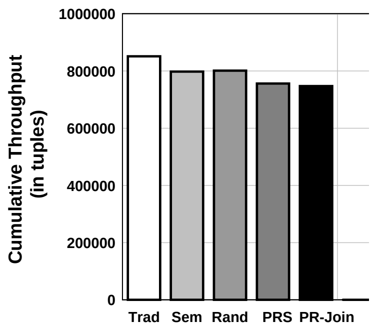
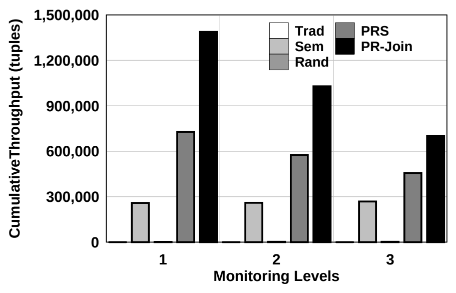


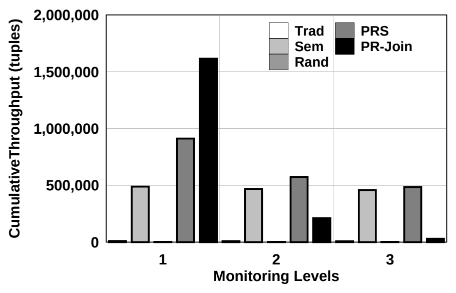
Figure 7.5: Overall Throughput

At the other extreme, trad produced very few significant query results. In this experiment, there are too many incoming tuples to process all of them within the query lifespan limit. Trad does not select which tuples are processed using any notion of significance. As resources are scarce, this leads to few significant results being produced. This demonstrates that in this overloaded scenario a resource allocation methodology is required.

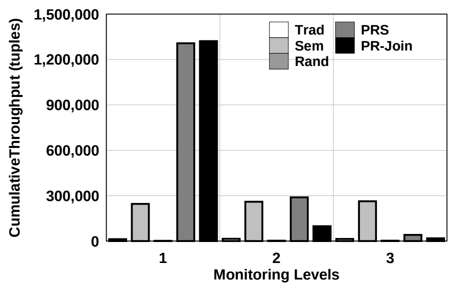
Figure 7.4.2 shows the overall throughout of all query results (regardless of significance) using the experimental set up outlined above. Overall compared to the alternatives, *PR-Join* achieved the lowest overall throughput. PR-Join produced 1% less results than PRS. It produced 6% less results than the resource allocation approaches (sem and rand). Compared to trad, it produced 12% less results. This is by design. Both PR and PR-Join are designed for EMAs where they is a need to ensure that at all costs certain tuples are processed. The overhead to support these approaches takes away resources from processing insignificant tuples.



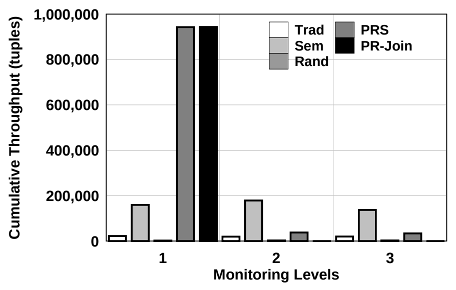
a) Throughput for DS25



b) Throughput for DS50



c) Throughput for DS75



d) Throughput for DS100

Figure 7.6: Varying the Number of Significant Tuple Join Partners at a Designator Operator

Varying the Number of Significant Tuples Join Partners at a Designator Operator

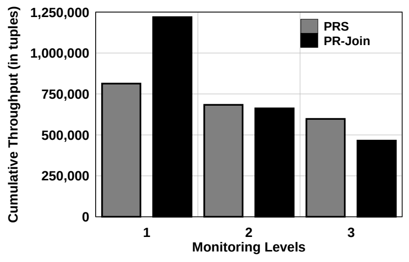
This experiment again uses Stock Market Join Query where the query lifespan = 1,000,000 ns and the window size = 500 tuples. However, we now use four distinct Data Sets, namely, DS25, DS50, DS75, and DS100. In every Data Set, the news and blog stream contains 2 sectors that have potential significance at level 1. These promising tuples news and blog tuples may join with significant tuples stock tuples at levels 1, 2, 3, and NA (or no significance). Each data set adjusts the number of join partners (i.e., stock tuples) for the incoming promising tuples (i.e., promising tuples news and blog tuples) that are insignificant. Non-atomic result production may skip (or delay) producing results from such insignificant join partners. In DS25, of all possible join partners for the promising tuples, 25% are significant tuples and 75% are insignificant tuples. That is, for each promising tuple 75% of all possible join partners have no significance. The production of results from these tuples may be skipped. Similarly, of all possible join partners for the promising tuples in DS50, DS75, and DS100 respectively have 50%, 25%, and 0% have no significance. That is, in DS100, all such join partners are significant.

Figures 7.6 a-d show the average cumulative throughput for monitoring levels 1 - 3 respectively after 10 minutes for DS25, DS50, DS75, and DS100. For DS25, PR-Join produced the largest quantity of all significant results (i.e., levels 1, 2, 3). While in the other data sets, PR-Join only produced the largest quantity of the most significant results (i.e., level 1), up to 614 fold more results. PR-Join's closest competitor is PR. For DS50, DS75, and DS100, PR produced more significant results at levels 2 and 3 than PR-Join. This is as expected. PR-Join achieves the most gains when the non-atomic result production delays or halts the production of less significant or insignificant results. DS25 offers the greatest opportunity for skipping the production of such results. In contrast, DS100 has no such opportunity. Clearly, PR-Join is best suited in situations where there are opportunities to skip or delay the production of certain results. These opportunities only occur when promising tuples exist whose join partners at the designator operator come from a range of significance levels.

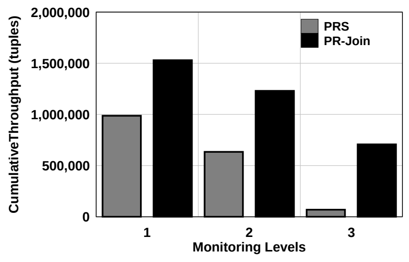
Varying the Size of the Workload

We now focus on the pros and cons of using non-atomic versus atomic result production in the PR system. For the next two, we compare PR-Join to PR across varying workloads

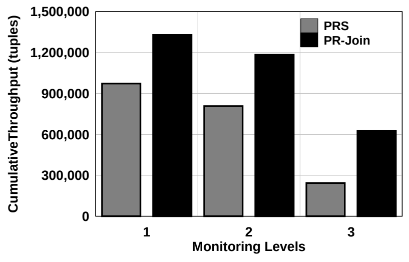
Increasing the workload does not guarantee an increase in the number of significant query results. All experiments are allocated the same amount of CPU resources. Each tuple processed is bound by the same query lifespan (i.e., 1,000,000 ns). There is a threshold that when too many significant join results are produced by operator op_1 then operator op_2 may no longer be able to process all of them within the query lifespan. Hence, there is no correlation between the query window size or complexity and the cumulative throughput of significant query results.



a) Throughput when window size = 500 tuples



b) Throughput when window size = 1000 tuples



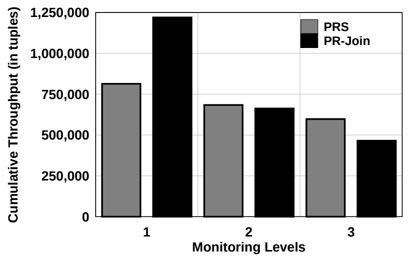
c) Throughput when window size = 2000 tuples

Figure 7.7: Varying the Query Window Size

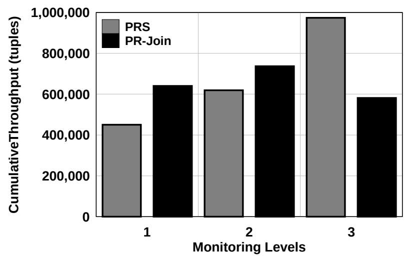
Varying Query Window Sizes. This experiment uses the Stock Market Join Query and Data Set 1 where the query lifespan = 1,000,000 ns. In this experiment we now vary the window size from 500, 1,000, to 2,000 tuples. Figures 7.7 a-c show the average cumulative throughput produced for monitoring levels 1 - 3 respectively over 10 minutes. Increasing the window size in turn increases the number of significant join results produced by operator op_1 (Fig. 1.4).

This experiment assesses how PR-Join performs when the join probe workload size varies due to having to scan longer and longer states. Clearly, the join operators have fewer incoming significant tuples to probe against for a 500 versus 2,000 query window. Our proposed PR-Join produced more of the most significant results (i.e., level 1) than PR. PR-Join still produced as many or more of the second most significant results (i.e., level 2) than PR. In the 500 window size experiment, PR-Join dedicates the resources to producing the significant results from the two highest significance levels (i.e., levels 1 and 2). It thus outperforms PR. However, there are inadequate resources for PR-Join to also produce more of the least significant results (i.e., level 3 in this case) than PR. The number of most significant results produced by PR-Join is not affected by the join probe workload size.

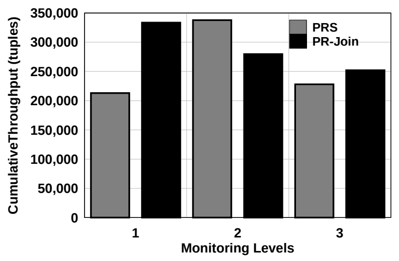
Varying the Query Complexity. This experiment uses Data Set 1 where the query lifespan = 1,000,000 ns and the window size is 500. We now vary the number of join operators in the query from 2, 4, to 8 join operators. More precisely, the 2 join operator experiment uses the Stock Market Join Query. The 4 join query adds two more joins to the Stock Market Join Query where each join combines the current query result with a news stream. Similarly, the 8 join query adds six more joins to the Stock Market Join Query. The join probe workload size varies when there are 2, 4, and 8 join operators in the query. This experiment evaluates if non-atomic



a) Throughput when query contains 2 joins



b) Throughput when query contains 4 joins



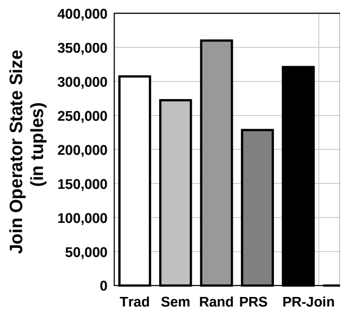
c) Throughput when query contains 8 joins

Figure 7.8: Varying the Query Complexity

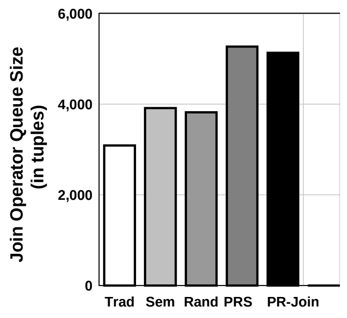
result production continues to be effective in producing the most significant query results first even for complex query pipelines.

Figures 7.8 a-c show the average cumulative throughput for monitoring levels 1 - 3 respectively produced over 10 minutes. Overall PR-Join consistently produced many more significant results at level 1 than PR. For the second most significant results (i.e., level 2), PR-Join produced slightly less results than PR. Finally, compared to PR for the least significant results (i.e., level 3) PR-Join produced less results. In summary, regardless of the query complexity (i.e, number of PR-Joins in a query) PR-Join effectively produces the most significant join results first.

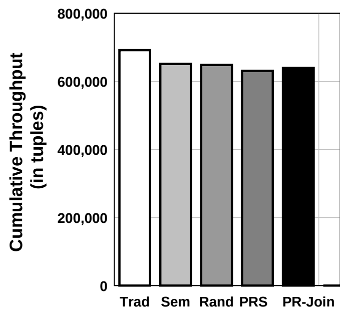
Overhead. These experiments use Stock Market Join Query and Data Set 1 where the query lifespan = ∞, and the window size = 250 tuples and the worst case for PR-Join. The worst case arises when no monitoring levels are defined and no tuples



a) Join Operator State Size



b) Join Operator Queue Size



c) Cumulative Throughput

Figure 7.9: Memory and Execution-Runtime CPU Overhead

expire (i.e., query lifespan =∞). As a consequence, no tuples are significant and all tuples are ultimately processed in FIFO order by all methods. This is the worst case for PR-Join as it still continues to carry the overhead of evaluating whether or not to perform non-atomic result production. Yet, no non-atomic join result production will occur. In addition, PR-Join also carries the statistics collection overhead required by all the other resource allocation methodologies, namely, the cost to gather and evaluate runtime statistics. The statistics collection overhead takes both CPU and memory resources away from the processing of tuples. Even though in these experiments the system is not overloaded, these methodologies would continue to evaluate how to best allocate resources.

Execution-Runtime CPU Overhead. This experiment evaluates the runtime overhead by measuring the cumulative throughput using the worst case for PR-Join outlined above. As shown by our results (Fig. 7.9 c), the overhead of the PR-Join operator is comparable to the state-of-the-art resource allocation methodologies. Namely, the difference between the throughput of PR-Join and PR is only 1.3%. Similarly, PR-Join produced only slightly fewer results after 10 minutes than sem and rand in this worst case, namely 1.8% and 1.3% respectively. Compared to trad (which has the lowest overhead), PR-Join performed as well as the alternative methodologies. Trad produced roughly 6% - 9% more results than the other approaches.

Of all these solutions, trad has the lowest execution run-time overhead due to not gathering any statistics. Sem and rand have the next highest execution run-time overhead. Their statistics gathering overhead is simpler than that of the PR system (used by both experimental solutions PR-Join and PR). These approaches only gather statistics on significant tuples. In contrast, the PR system gathers statistics on both significant tuples and promising tuples. Thus, PR-Join and PR have the highest execution run-time overhead.

Memory Overhead. This experiment also uses the worst case for PR-Join outlined above. We now measure the memory overhead, namely, the average number of tuples in the state and input queue of the last join operator (i.e., join operator op_2 in the Stock Market Join Query) (Fig. 7.9 a & b).

Trad has the fewest tuples in its queues. Trad has the lowest execution run-time overhead and thus can devote more CPU cycles to processing tuples. Hence, its queue is the smallest. Both sem and rand have the next largest number of tuples in their input queues. This is as expected. Sem and rand have the next highest execution run-time overhead. Finally, both PR-Join and PR have the largest number of tuples in their input queues. They have the highest execution run-time overhead.

Sem, rand, and trad systems store tuples in their join operators by their join attribute. PR (Ch. 4 & 6) stores tuples in their join operators by their window pane and join attribute, while PR-Join stores tuples by their window pane, join attribute, and significance level (Sec. 7.3.1). The PR-Join design relies upon the physical design to support non-atomic result production (Sec. 7.3.1). Thus, given the physical designs, PR-Join takes longer to store tuples than PR. The amount of tuples stored in a state is affected by the overhead to store tuples by the given method. However, our results show that the cumulative number of most significant results produced by PR-Join (Fig. 7.9 c) is not limited by its state sizes.

7.4.3 Summary of Experimental Findings

We now summarize our key findings:

1) When resources are scarce, PR-Join consistently increases the throughput of the most significant query results compared to the alternative solutions.

2) Regardless of the join probe workload size, PR-Join is effective at increasing the throughput of the most significant query results.

3) No matter what the complexity of the query is (number of joins in the query), PR-Join increases the throughput of the most significant query results.

4) PR-Join is most beneficial when there are opportunities to skip or halt the production of less significant join results. This occurs when there exists promising tuples whose join partners at the designator operator come from a range of significance levels.

5) The execution-runtime CPU overhead of the PR-Join operator is comparable to that of the state-of-the-art resource allocation methodologies. The memory overhead of PR-Join is within practical limits of these approaches as well.

Chapter 8

Conclusions of This Dissertation

The goal of this dissertation is to introduce models and specialized operators to support *targeted prioritized processing data stream systems* (*TP*). As illustrated by real events [Pre10, Net12], EMAs may not be able to process all the incoming data within the query lifespan. It is critical to ensure that the EMAs do not shut down and that the monitoring of certain objects continues until additional resources (if available) can be provided. TP supports these requirements. This dissertation addresses the problems of designing TP models that support significance determination using dynamic and/or static criteria. We also proposed new query operator designs to support the efficient production of the most significant correct results generated from subsets of critical tuples.

The key contribution of this dissertation is to introduce a new data stream query optimization approach that efficiently determines how best to allocate resources to ensure the processing of certain subsets of tuples within the query pipeline. Our execution infrastructures support the online adaption of how resources are allocated without requiring any infrastructure changes. The operators we proposed produce accurate results while ensuring that resources are allocated to producing the most significant results first. We introduce cost models that factor in the overhead of precedence determination. Our systems support complex and costly precedence determination.

The specific highlights of the four topics in this dissertation are:

First, Proactive Promotion (PP) takes an innovative approach towards addressing the problem of aligning resource allocation to the significance of tuples and the current system load. Our key contribution is to show that PP is a viable approach towards providing preferential resource allocation based upon user requirements and the current system load. The rank classifier operators and our enhancements to standard operators effectively pull more significant tuples ahead of less significant ones. The PP Optimizer efficiently locates the optimal PP plan and triggers the adaption of which and where precedence criteria are evaluated by sending notifications to operators. Using these notifications, the PP Executor adjusts resource allocation online without requiring any expensive infrastructure changes. Our experiments confirm that when priority resource allocation is needed, PP consistently lowers latency and increases throughput for significant results compared to traditional DSMS and shedding approaches.

Second, the *TP-Ag* operator tackles the unsolved problem of generating reliable results from incomplete aggregation group populations created by TPs. Priority driven, TP-Ag produces non-skewed results by determining at run-time which combination of subset(s) (if any) are used to generate results. To achieve this, TP-Ag uses a carefully designed estimation model and application of Cochran's sample size methodology [Coc77] to measure the accuracy of sample populations from which the results are carefully constructed. Our experimental study confirms that TP-Ag is effective at increasing the percentage of correct aggregate results produced in TPs (TP-Ag produces up to 91% more correct aggregate results).

Third, PR-Prune, an innovative preferential resource allocation methodology, makes the following important contributions. *PR-Prune* efficiently locates online dynamic criteria that are critical for the production of significant query results. Our PR infrastructure allows resources to be quickly shifted to pull critical tuples forward. PR supports the adaption of which, where, and when critical tuples are preferentially allocated resources in the pipeline. Our PR infrastructure supports the online adaption of the rank of each tuple. Our experimental study confirms that when priority resource allocation matters and promising tuples exist, PR-Prune consistently increases the throughput for the most significant results (between 1.3 to 23 fold) compared to the state-of-the-art approaches.

Finally, the *PR-Join* operator is the first to support non-atomic join result production. Namely, PR-Join discerningly interrupts and reinstates the probe operation of specific tuples to control the production of more significant and delay the production of less significant join results. In addition, PR-Join selectively coordinates which join results are produced to ensure that only correct and no duplicate join results are produced. Our experimental study confirms that PR-Join is very effective at increasing the throughput of the most significant results (up to 190 fold more significant query results) in systems that experience resource duress and thus require preferential resource allocation.

Our TP systems have achieved the ability to allow the user to control which tuples are allocated resources throughout the query plan. This approach can be utilized by any DSMS that must ensure that certain tuples are processed. It is helpful to any DSMS where certain results are more important to produce and resources may be limited. Our TP systems and operator designs are well documented. This makes it easy for other developers to implement. Our next critical step is to design a TP system to support the needs of multiple P-CQL queries.

Chapter 9

Ideas for Future Work

The concepts presented in this dissertation have opened possible directions for future research.

9.1 Multi-Query TP Processing

Any stream source of value is likely going to have multiple consumers interested in monitoring its data. Thus the processing of a workload composed of multiple P-CQL queries (Sec. 4.1.1) must be tackled, namely, workloads with two layers of priorities (query priority and a finer tuned tuple priority). In a nutshell, there are three issues that must be investigated, namely, handling of multiple possibly conflicting significance levels by different queries, sharing of processing segments among queries, and the combination of the two.

Modeling of Priority of TP Query Workloads. First consider the simpler case where the queries in the workload are not sharing any computations. In this case, the TP resource allocator must still consider how the allocation of resources

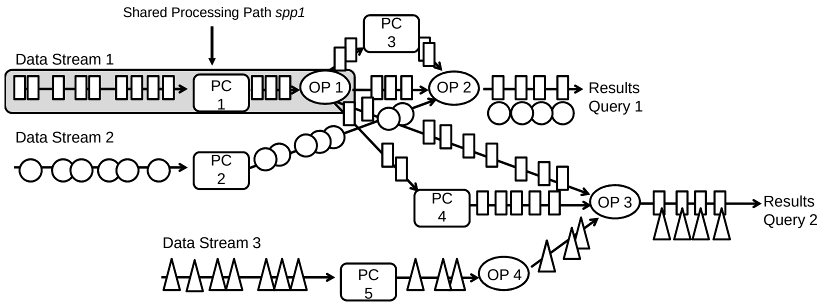


Figure 9.1: Example Global Query Composed of Multiple PP Plans

across queries is guided by the respective priority levels. This will require a calibration amongst the users and their specification models; for example, is a significance level 1 specified by user U1 equal or more important than a significance level 2 specified by user U2. For this, the multi-tiered TP model could be extended to also include the importance of users relative to other users as well as the relative importance of queries by the same user. Once the TP model is extended to support such significance modeling, the different scales of significance could be normalized into one unified scale.

Handling Significance Differences Across Queries. Given such a uniform scale, now consider three different cases. First, consider that one query Q1 is *strictly more significant* than another query Q2, meaning all the priority subsets in Q1 are more significant than any of the priority subsets in Q2. In this case, the resources would be given first to promote tuples for Q1 before resources are allocated to promoting tuples related to Q2. This may starve query Q2. Thus fairness of allocation to assure some minimal level of activity for query Q2 must be addressed. Next queries may be *impartially significant* to each other, in that no query is strictly more significant than any other queries in the set. Then the resources allocated will swap back and forth between the queries as dictated by the priority subsets. That is, the query workload can be treated as one large query graph. The query optimizer and resource adaptor would be applied at the overall multi-query plan level. Lastly, there may be *shared significance* among the queries, i.e., a priority subset in Q1 may have the same significance as a priority subset in Q2. Then equal opportunity is given for that particular significance level across the two queries. However, care must be taken in the case of extremely sparse resource availability, so that both queries at least succeed to produce some of their results at that significance level. Customized scheduling methods of the global query plan will take such optimization of tiered production into account. In summary, a mixed model which integrates query priority and tuple priority must be considered. To make sense of this, a model that determines the combined priority would be devised. Resource allocation would then proceed according to these global-priorities. Proactive Promotion would be utilized as a solution framework there after to place rank classifiers and pull significant tuples forward using the global-priorities (instead of the initial local ones).

Sharing Among PP Queries. Lastly, if queries have common sub-expressions, then the question of if explicit sharing at the physical query plan level is beneficial or not must be addressed. The complication that arises here is that while some subcomputations may be shared at the query specification level, the significance level assigned to the particular sub-paths of the queries respectively may not be matching, and possibly even in conflict. Consider the example in Figure 9.1. The depicted workload consists of the two queries Q1 and Q2.

In this example, while both queries Q1 and Q2 share the same sub-path from stream S1 via the operator PC1 to operator OP1 (marked via the gray shaded rectangle in the figure), it may be that Q1 has specified completely different priority criteria than Q2 for tuples on this stream S1. To handle this case, the query resource allocation optimizer needs to be refined to determine the shared placement of all priority criteria relevant to that shared processing path. Furthermore, we need strategies to assess if it is cost prohibitive to evaluate the criteria of all priority subsets as is versus restricting and possibly integrating the predicates into a simpler set by generalizing some of of predicates.

9.2 On-the-fly Preference Adaption

Another TP challenge is to develop a flexible architecture that allows the user to adaptively and continuously change their resources allocation preferences of any query online. This is a complicated issue. Once the user selects new resource allocation preferences, the current inprocess query plan must be adapted. In such instances, inprocess tuples that have been pulled forward may no longer be considered significant. Conversely, inprocess tuples that have been ignored may need to be rapidly pulled forward. That is, inprocess tuples that were allocated resources according to the old user preferences may need special considerations to ensure that the new user preferences can rapidly be employed. In a nutshell, there are two issues that must be investigated, namely, handling of inflight tuples, and the adaption of the current query plan to the new one.

Handling Inflight Tuples Once a user selects a new resource allocation preferences, there are two possible cases of how inflight tuples in the current query plan must be handled. First, tuples that were significant under the old resource allocation preferences but are now insignificant under the new one must be identified.

Second, tuple that are significant under the new one must be identified. This is complicated. Identifying the new significance of all inflight tuples requires evaluating all tuples throughout the pipeline. This may be prohibitively restrictive. It requires a methodology that optimally determines where to re-evaluate the inflight tuples. In addition, there may be overlapping resource allocation preferences between the old and the new versions. In such a case, the optimizer should consider which inflight significant tuples may not need to be re-evaluated. Lastly, customized scheduling methods to allocate resources to inflight tuples in the current query plan must be considered.

PP Plan Adaption Once a user selects new resource allocation preferences, resources should be allocated to process new incoming tuples according to the new preferences. First, a new query plan must be chosen. As preferences are adjusted online, no run-time statistics may be available to determine which significant tuples need to be pulled forward to ensure they are processed. An optimizer must be developed that creates a new optimal query plan using the available statistics. The Query Plan Executor must be modified to support both identifying the significance of inflight tuples as well as identifying the significance of incoming tuples using the new query plan. Lastly, the progression of inflight old tuples must be efficiently monitored to allow the Query Plan Executor to know when to halt the identification of the significance of inflight tuples.

9.3 Other Challenges

Another TP challenge is to develop an optimization approach that would consider both the amount of CPU resources available to each operator as well as the scheduling of operators. This is also a complicated issue. The amount of CPU resources available to each operator as well as the scheduling of operators are strongly dependent on each other. A change in one may cause a modification in the other, subsequently affecting the cost of the overall throughput of significant results. This raises the proverbial chicken and the egg question. Should the scheduling approach be chosen first, and then the the amount of CPU resources available to each operator? Or should it be the other way around? Or is there some hybrid approach of selecting the scheduling and CPU resource allocation for portions of the query pipeline? These issues should be explored.

A final TP challenge is to develop a flexible architecture that allows the system to adaptively and continuously change their resources allocation preferences to ensure that tuples from each object within a given promotable subset is monitored within a given set time period. Currently, TPs allocate resources to *all* tuples in a promotable subset until the query plan changes. We wish to explore developing systems that allocate resources to tuples from only certain objects in a promotable subset for a period of time. Later on, the system would allocate resources to tuples from other objects in a set of promotable tuples. The goal would be to ensure that each tuple in a promotable subset is processed at least once within a given time range. This is a complicated issue. How can we efficiently track and ensure that each object in a promotable subset is monitored within a the time range? How do we select which tuples to monitor at any given moment?

These and other challenges will be the focus of our continued work in this area.

Bibliography

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The International Journal on Very Large Data Bases*, pages 121–142, 2006.

[ACc+03a] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, pages 120–139, 2003.

[ACÇ+03b] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: a data stream management system. In *ACM Special Interest Group on Management of Data Conference*, pages 666–666, 2003.

[AH00] Ron Avnur and Joseph M Hellerstein. Eddies: continuously adaptive query processing. *ACM Special Interest Group on Management of Data Record*, pages 261–272, 2000.

[ALN08] Mohamed Ashour and Tho Le-Ngoc. Priority queuing of long-range dependent traffic. *Computer Communications*, pages 3954–3963, 2008.

[ANWS06] Ahmed Ayad, Jeffrey Naughton, Stephen Wright, and Utkarsh Srivastava. Approximating streaming window joins under cpu limitations. In *IEEE International Conference on Data Engineering*, pages 142–154, 2006.

[ASL04] Tarek F Abdelzaher, Vivek Sharma, and Chenyang Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, pages 334–350, 2004.

[BBD$^+$02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data streams. In *ACM Symposium on Principles of Database Systems*, pages 1–16, 2002.

[BBD$^+$04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases*, pages 333–353, 2004.

[BBDW05] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-based routing: different plans for different data. In *International Conference on Very Large Data Bases*, pages 757–768, 2005.

[BBMD03] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM Special Interest Group on Management of Data Conference*, pages 253–264, 2003.

[BC05] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *ACM Special Interest Group on Management of Data Conference*, pages 119–130, 2005.

[BCM$^+$99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E Strom, and Daniel C Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *IEEE International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *International Conference on Data Engineering*, pages 350–361, 2004.

[BKZS12] Can Basaran, Kyoung-Don Kang, Yan Zhou, and Mehmet H Suzer. Adaptive load shedding via fuzzy control in data stream management systems. In *IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–8, 2012.

[BMM$^+$04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *ACM Special Interest Group on Management of Data Conference*, pages 407–418, 2004.

[BTO13] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. Adaptive input admission and management for parallel stream processing. In *ACM International Conference on Distributed Event-Based Systems*, pages 15–26, 2013.

[CAR05] Nuno Carvalho, Filipe Araujo, and Luis Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *IEEE International Symposium on Network Computing and Applications*, pages 101–108, 2005.

[CCC$^+$01] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *International Conference on Software Engineering*, pages 443–452, 2001.

[CcC$^+$02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *International Conference on Very Large Data Bases*, pages 215–226, 2002.

[CcR$^+$03] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *International Conference on Very Large Data Bases*, pages 838–849, 2003.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM Special Interest Group on Management of Data Conference*, 2013, pages 725–736, 2013.

[Che99] Mitchell Frederic Cherniack. *Building query optimizers with combinators*. PhD thesis, Brown University, Providence, RI, USA, 1999.

[Cho07] Jan Chomicki. Semantic optimization techniques for preference queries. *Information Systems Journal*, pages 670–684, 2007.

[CJ10] Alex King Yeung Cheung and Hans-Arno Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Transactions on Computer Systems*, pages 9–64, 2010.

[CKT08] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Time-decaying aggregates in out-of-order streams. In *ACM Symposium on Principles of Database Systems*, pages 89–98, 2008.

[CLYY92] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, and Honesty C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *International Conference on Very Large Data Bases*, pages 15–26, 1992.

[Coc77] William G. Cochran. *Sampling Techniques*. John Wiley, 3 edition, 1977.

[Cor] Coral8. http://www.coral8.com. Accessed: 2013-11-03.

[DB05] Robert I Davis and Alan Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium*, pages 388–398, 2005.

[DGGR02] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *ACM Special Interest Group on Management of Data Conference*, pages 61–72, 2002.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *International Conference on Very Large Data Bases*, pages 948–959, 2004.

[DL] Virginia Tech:Network Dynamics and Simulation Science Laboratory. http://ndssl.vbi.vt.edu/opendata/index.php.

[DLDC12] Frederico Durao, Ricardo Lage, Peter Dolog, and Nilay Coskun. A multi-factor tag-based personalized search. In *Web Information Systems and Technologies*, pages 192–206. 2012.

[DLX07] Huafeng Deng, Yunsheng Liu, and Yingyuan Xiao. The golden mean operator scheduling strategy in data stream systems. pages 186–191, 2007.

[DMRH04] Luping Ding, Nishant Mehta, Elke A Rundensteiner, and George T Heineman. Joining punctuated streams. In *International Conference on Extending Database Technology*, pages 587–604, 2004.

[DNLR09] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *IEEE Real-Time Systems Symposium*, pages 291–300, 2009.

[DR04] Luping Ding and Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *International Conference on Information and Knowledge Management*, pages 92–101, 2004.

[DRH03] Luping Ding, Elke A. Rundensteiner, and George T. Heineman. Mjoin: a metadata-aware stream join operator. In *ACM International Conference on Distributed Event-Based Systems*, pages 1–8, 2003.

[DTB93] Robert Ian Davis, Ken W Tindell, and Alan Burns. Scheduling slack time in fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium*, pages 222–231, 1993.

[Esp] Esper. http://esper.codehaus.org/. Accessed: 2013-11-03.

[Fam65] Eugene F. Fama. The behavior of stock-market prices. *The Journal of Business*, pages 34–105, 1965.

[FJK$^+$05] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni, and Wei Hong. Design considerations for high fan-in systems: The hifi approach. In *Conference on Innovative Data Systems Research*, pages 290–304, 2005.

[FJL$^+$01] Françoise Fabret, H Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD Record*, pages 115–126, 2001.

[FMTL09] Rafael Fernández-Moctezuma, Kristin Tufte, and Jin. Li. Inter-operator feedback in data stream management systems via punctuation. *Computer Research Repositoy Journal*, pages 1–9, 2009.

[FY11] Zbynek Falt and Jakub Yaghob. Task scheduling in data stream processing. In *International Workshop on Databases, Texts, Specifications, and Objects*, pages 85–96, 2011.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, pages 187–205, 2003.

[GH09] Jing-feng Guo and Chun-liang He. Load shedding for sliding window aggregation queries over data streams. *Application Research of Computers*, pages 1–23, 2009.

[GÖ03] Lukasz Golab and M Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *International Conference on Very Large Data Bases*, pages 500–511, 2003.

[GO05] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *ACM Special Interest Group on Information Retrieval Conference*, pages 658–669, 2005.

[GP00] Randy R. Gainey and Brian K. Payne. Understanding the experience of house arrest with electronic monitoring: An analysis of quantitative and qualitative data. *International Journal of Offender Therapy and Comparative Criminology*, pages 84–96, 2000.

[GRR13] Fernando Guirado, Concepció Roig, and Ana Ripoll. Enhancing throughput for streaming applications running on cluster systems. *Journal of Parallel and Distributed Computing*, pages 1092–1105, 2013.

[GSA09] Mohammad Ghalambor, Ali A Safaeei, and Mohammad Abdollahi Azgomi. Dsms scheduling regarding complex qos metrics. In *International Conference on Computer Systems and Applications*, pages 587 –594, 2009.

[GWYL05] Buğgra Gedik, Kun-Lung Wu, Philip S Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *International Conference on Information and Knowledge Management*, pages 171–178, 2005.

[GWYL07] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE Transactions on Knowledge and Data Engineering Journal*, pages 1363–1380, 2007.

[HFAE03] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *International Conference on Very Large Data Bases*, pages 297–308, 2003.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *ACM Special Interest Group on Management of Data Record*, pages 171–182, 1997.

[HNS94] Peter J Haas, Jeffrey F Naughton, and Arun N Swami. On the relative cost of sampling for join selectivity estimation. In *ACM Symposium on Principles of Database Systems*, pages 14–24, 1994.

[Hoe63] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, pages 13–30, 1963.

[Hoy88] Susan Hoyle. Use and abuse of statistics. *The Association for Information Management*, pages 321–324, 1988.

[HWXZ07] Dong-Hong Han, Guo-Ren Wang, Chuan Xiao, and Rui Zhou. Load shedding for window joins over streams. *Journal of Computer Science and Technology*, pages 182–189, 2007.

[HXZ$^+$06] Donghong Han, Chuan Xiao, Rui Zhou, Guoren Wang, Huan Huo, and Xiaoyun Hui. Load shedding for window joins over streams. In *Advances in Web-Age Information Management*, pages 472–483. 2006.

[IBM] Ibm inc. http://www.ibm.com. Accessed: 2013-11-03.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, pages 1–58, 2008.

[JBG$^+$10] Jonas Jacobi, André Bolles, Marco Grawunder, Daniela Nicklas, and H-Jürgen Appelrath. A physical operator algebra for prioritized elements in data streams. *Computer Science - Research and Development*, pages 235–246, 2010.

[JJ12] Shengming Jiang and Shengming Jiang. Differentiated queueing service (dqs) for granular qos. In *ACM Special Interest Group on Data Communications Review*, pages 73–87. 2012.

[KBB13] Tomáš Kramár, Michal Barla, and Mária Bieliková. Personalizing search using socially enhanced interest model, built from the stream of user's activity. *Journal of Web Engineering*, pages 65–92, 2013.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM Special Interest Group on Management of Data Conference*, pages 106–117, 1998.

[KLJ+09] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. In *International conference on Virtual execution environments*, pages 101–110, 2009.

[KLK11] Tae-Hyung Kwon, Ki Yong Lee, and Myoung Ho Kim. Load shedding for multi-way stream joins based on arrival order patterns. *Journal of Intelligent Information Systems*, pages 245–265, 2011.

[KMH+10] Hyun Gu Kang, Diane F Mahoney, Helen Hoenig, Victor A Hirth, Paolo Bonato, Ihab Hajjar, and Lewis A Lipsitz. In situ monitoring of health in older adults: technologies and issues. *Journal of the American Geriatrics Society*, pages 1579–1586, 2010.

[KMPS04] VS Kumar, Madhav V Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. End-to-end packet-scheduling in wireless ad-hoc networks. In *ACM-SIAM symposium on Discrete algorithms*, pages 1021–1030, 2004.

[KNTA12] Lutful Karim, Nidal Nasser, Tarik Taleb, and Abdullah Alqallaf. An efficient priority packet scheduling algorithm for wireless sensor network. In *IEEE International Conference on Communications*, pages 334–338, 2012.

[KNV03] Jaewoo Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *IEEE International Conference on Data Engineering*, pages 341–352, 2003.

[KPP+02] Hillol Kargupta, Byung-Hoon Park, Sweta Pittie, Lei Liu, Deepali Kushraj, and Kakali Sarkar. Mobimine: monitoring the stock market from a pda. *ACM Special Interest Group on Knowledge Discovery and Data mining Newsletter*, pages 37–46, 2002.

[LCH+06] Chung-Chih Lin, Ming-Jang Chiu, Chun-Chieh Hsiao, Ren-Guey Lee, and Yuh-Show Tsai. Wireless health care service system for elderly with dementia. *IEEE Transactions on Information Technology in Biomedicine*, pages 696–704, 2006.

[LKR10] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *IEEE Real-Time Systems Symposium*, pages 259–268, 2010.

[LL07] Jin-xian Lin and Qin-xian Lin. Adaptive load shedding for slide window joins over data streams. *Journal of Fuzhou University (Natural Science Edition)*, pages 1–11, 2007.

[LLG$^+$09] Mo Liu, Ming Li, Denis Golovnya, Kajal Claypool, and Elke A. Rundensteiner. Sequence pattern query processing over out-of-order event streams. In *IEEE International Conference on Data Engineering*, pages 784–795, 2009.

[LM97] Dong Lin and Robert Morris. Dynamics of random early detection. In *ACM Special Interest Group on Data Communication Conference*, pages 127–137, 1997.

[LMT$^+$05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM Special Interest Group on Management of Data Record*, pages 39–44, 2005.

[LQJQ12] Ouyang Lin, Zhou Qin, Qi Jingjing, and Pu Qiumei. A new linear programming based load-shedding strategy. In *International Symposium on Distributed Computing and Applications to Business, Engineering & Science*, pages 260–263, 2012.

[LS98] TV Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM Special Interest Group on Data Communication Review*, pages 203–214, 1998.

[LZ07] Yan-Nei Law and Carlo Zaniolo. Load shedding for window joins on multiple data streams. In *IEEE International Conference on Data Engineering Workshop*, pages 674–683, 2007.

[LZR06] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *ACM Special Interest Group on Data Communication Conference*, pages 347–358, 2006.

[LZZM07] Zhang Longbo, Li Zhanhuai, Wang Zhenyou, and Yu Min. Semantic load shedding for sliding window join-aggregation queries over data streams. In *International Conference on Convergence Information Technology*, pages 2152–2155, 2007.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *ACM Special Interest Group on Management of Data Conference*, pages 635–646, 2006.

[MBT07] Mirella M Moro, Petko Bakalov, and Vassilis J Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *the international conference on Very large data bases*, pages 866–877, 2007.

[MLA04] Mohamed Mokbel, Ming Lu, and Walid Aref. Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In *IEEE International Conference on Data Engineering*, pages 251 – 262, 2004.

[MLWW09] Li Ma, Xin Li, Yongyan Wang, and Hongan Wang. Real-time scheduling for continuous queries with deadlines. In *ACM symposium on Applied Computing*, pages 1516–1517, 2009.

[MLZ$^+$09] Li Ma, Dangwei Liang, Qiongsheng Zhang, Xin Li, and Hongan Wang. Load shedding for shared window join over real-time data streams. In *Advances in Data and Web Management*, pages 590–596. 2009.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Data Bases*, pages 346–357, 2002.

[MSAH11] Shirin Mohammadi, Ali A. Safaei, Fatemeh Abdi, and Mostafa S. Haghjoo. Adaptive data stream management system using learning automata. *Computer Research Repositoy Journal*, pages 1–14, 2011.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *ACM Special Interest Group on Management of Data Conference*, pages 49–60, 2002.

[MZS10] Li Ma, Qiongsheng Zhang, and Nianyun Shi. A semantic load shedding algorithm based on priority table in data stream system. In *International Conference on Fuzzy Systems and Knowledge Discovery*, pages 1167–1172, 2010.

[Net12] Microsoft Network. Where have all the investors gone? *http://money.msn.com*, Feb. 2012.

[NR07] Rimma V. Nehme and Elke A. Rundensteiner. Clustersheddy: Load shedding using moving clusters over spatio-temporal data streams. In *International Conference on Database Systems for Advanced Applications*, pages 637–651, 2007.

[NRB09] R. V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *International Conference on Extending Database Technology*, pages 803–814, 2009.

[NWL$^+$12] Rimma Nehme, Karen Works, Chuan Lei, Elisa Bertino, and Elke A. Rundensteiner. Multi-route query processing and optimization. *Journal of Computer and System Sciences*, pages 312–329, 2012.

[NWRB09] Rimma V. Nehme, Karen Works, Elke A. Rundensteiner, and Elisa Bertino. Query mesh: Multi-route query processing technology. *International Conference on Very Large Data Bases*, pages 1530–1533, 2009.

[Ora] Oracle. http://www.oracle.com/index.html.

[OW00] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. Technical Report 2000-16, Stanford InfoLab, 2000.

[OYY$^+$05] Zhengyu Ou, Ge Yu, Yaxin Yu, Shanshan Wu, Xiaochun Yang, and Qingxu Deng. Tick scheduling: A deadline based optimal task scheduling approach for real-time data stream systems. In *Advances in Web-Age Information Management*, pages 725–730. 2005.

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *International Conference on Very Large Data Bases*, pages 486–495, 1997.

[PJS99] Mark Parris, Kevin Jeffay, and F. Donelson Smith. Lightweight active router-queue management for multimedia networking. In *Electronic Imaging*, pages 162–174, 1999.

[Pre10] Associated Press. Officials lose track of 16,000 sex offenders after gps fails. *http://www.foxnews.com*, 2010.

[PRGK09] Jay A Patel, Étienne Rivière, Indranil Gupta, and Anne-Marie Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, pages 2304–2320, 2009.

[QL10] Shao Qian and YiLi Lu. A modified chain scheduling algorithm in data stream system. In *IEEE International Conference on Computer and Automation Engineering*, pages 568–570, 2010.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *IEEE International Conference on Data Engineering*, pages 353–364, 2003.

[RDS$^+$04] Elke A Rundensteiner, Luping Ding, Timothy Sutherland, Yali Zhu, Brad Pielech, and Nishant Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *International Conference on Very Large Data Bases*, pages 1353–1356, 2004.

[Ree07] David Reed. *Balanced Introduction to Computer Science*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2007.

[REH$^+$12] Patrick Roocks, Markus Endres, Alfons Huhn, Werner Kießling, and Stefan Mandl. Design and implementation of a framework for context-aware preference queries. *Journal of Computing Science and Engineering*, pages 243–256, 2012.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[RH02] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *ACM Special Interest Group on Management of Data Conference*, pages 275–286, 2002.

[RH05] Frederick Reiss and Joseph M. Hellerstein. Data triage: an adaptive architecture for load shedding in telegraphcq. In *IEEE International Conference on Data Engineering*, pages 155 – 156, 2005.

[RJH07] Jiadong Ren, Wanchang Jiang, and Cong Huo. Index-based load shedding for streaming sliding window joins. In *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*, pages 162–170. 2007.

[RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. Technical report, Berkeley, CA, USA, 1999.

[SB03] Heiko Schuldt and Gert Brettlecker. Sensor data stream processing in health monitoring. Technical report, UMIT, 2003.

[SB13] Kai-Uwe Sattler and Felix Beier. Towards elastic stream processing: Patterns and infrastructure. In *First International Workshop on Big Dynamic Distributed Data*, pages 49–54, 2013.

[SCL^{+}12] Zhitao Shen, M Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. A generic framework for top-k pairs and top-k objects queries over sliding windows. *IEEE Transactions on Knowledge and Data Engineering*, page 1, 2012.

[SCLP06] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Efficient scheduling of heterogeneous continuous queries. In *International Conference on Very Large Data Bases*, pages 511–522, 2006.

[SD13] Sanjay Agrawal Sonal Dubey. Qos driven task scheduling in cloud computing. *International Journal of Computer Applications Technology and Research*, pages 595 – 600, 2013.

[SG86] Jeffrey C. Schlimmer and Richard H. Granger. Beyond incremental processing: Tracking concept drift. In *National Conference on Artificial Intelligence*, pages 502–507, 1986.

[SH10] Ali A Safaei and Mostafa S Haghjoo. Parallel processing of continuous queries over data streams. *Distributed and Parallel Databases*, pages 93–118, 2010.

[SH12] S Senthamilarasu and M Hemalatha. Load shedding techniques based on windows in data stream systems. In *International Conference on Emerging Trends in Science, Engineering and Technology*, pages 68–73, 2012.

[SLCP05] Mohamed A. Sharaf, Ros Labrinidis, Panos K. Chrysanthis, and Kirk Pruhs. Freshness-aware scheduling of continuous queries in the dynamic web. In *International Workshop on the Web and Databases*, pages 73–78, 2005.

[SLMK01] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *International Conference on Very Large Data Bases*, pages 19–28, 2001.

[SLSL05] Sven Schmidt, Thomas Legler, Daniel Schaller, and Wolfgang Lehner. Real-time scheduling for data stream management systems. *Real-Time Systems*, pages 167–176, 2005.

[SPZ$^+$05] Tim Sutherland, Brad Pielech, Yali Zhu, Luping Ding, and Elke A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *International Database Engineering and Applications Symposium*, pages 445–454, 2005.

[Stra] Microsoft sql server streaminsight. http://www.Microsoft.com. Accessed: 2013-11-03.

[Strb] Streambase systems inc. http://www.streambase.com/. Accessed: 2013-11-03.

[SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *International Conference on Very Large Data Bases*, pages 324–335, 2004.

[TAJ04] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Databases, Information Systems, and Peer-to-Peer Computing*, pages 138–152. 2004.

[TÇZ$^+$03] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *International Conference on Very Large Data Bases*, pages 309–320, 2003.

[TÇZ07] Nesime Tatbul, Ugur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *International Conference on Very Large Data Bases*, pages 159–170, 2007.

[TD03] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *International Conference on Very Large Data Bases*, pages 333–344, 2003.

[TMB09] Jaime Teevan, Meredith Ringel Morris, and Steve Bush. Discovering and using groups to improve personalized search. In *ACM International Conference on Web Search and Data Mining*, pages 15–24, 2009.

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering Journal*, pages 555–568, 2003.

[TNP13] Alexandros Labrinidis Thao N. Pham, Panos K. Chrysanthis. Self-managing load shedding for data stream management systems. *IEEE International Conference on Data Engineering Workshops*, pages 1–7, 2013.

[TTPM03] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark - a benchmark for querying data streams. Technical report, OGI School of Science & Engineering at OHSU, 2003.

[TZ06] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *International Conference on Very Large Data Bases*, pages 799–810, 2006.

[UF99] Tolga Urhan and Michael J Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, 1999.

[UF00] Tolga Urhan and Michael J. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000.

[UF01] Tolga Urhan and Michael J Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *International Conference on Very Large Data Bases*, pages 501–510, 2001.

[Uni02] Stanford University. Stream query repository. http://www-db.stanford.edu/stream/sqr/, Dec 2002.

[VNB03] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *International Conference on Very Large Data Bases*, pages 285–296, 2003.

[VSL$^+$13] Sergio Vavassori, Javier Soriano, David Lizcano, et al. Explicit context matching in content-based publish/subscribe systems. *Sensors*, pages 2945–2966, 2013.

[WA91] Annita N. Wilschut and Peter M. G Apers. Dataflow query execution in a parallel main-memory environment. In *Conference on Parallel and Distributed Information Systems*, pages 68–77, 1991.

[WGM10] Song Wang, Chetan Gupta, and Abhay Mehta. Vpipe: Virtual pipelining for scheduling of dag stream query plans. *Enabling Real-Time Business Intelligence*, pages 32–49, 2010.

[WPSS06] Yuan Wei, Vibha Prasad, Sang H Son, and John A Stankovic. Prediction-based qos management for real-time data streams. In *Real-Time Systems Symposium*, pages 344 –358, 2006.

[WQL$^+$10] Hong-ya Wang, Zhi-dong Qin, Bai-yan Li, Jing Cong, Zhi-jun Wang, and Ming Du. Novel load shedding approach for real-time data stream processing. *Journal of Chinese Computer Systems*, pages 1–4, 2010.

[WR11] Karen Works and Elke A Rundensteiner. The proactive promotion engine. In *IEEE International Conference on Data Engineering*, pages 1340–1343, 2011.

[WRGB06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *International Conference on Very Large Data Bases*, pages 619–630, 2006.

[WRM10] M. Wei, Elke A. Rundensteiner, and Murali Mani. Achieving high output quality under limited resources through structure-based spilling in xml streams. *The International Journal on Very Large Data Bases*, pages 1267–1278, 2010.

[WSS01] Haining Wang, Chia Shen, and Kang G Shin. Adaptive-weighted packet scheduling for premium service. In *IEEE International Conference on Communications*, pages 1846–1850, 2001.

[WSS06] Yuan Wei, S.H. Son, and J.A. Stankovic. Rtstream: Real-time query processing for data streams. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 141–150, 2006.

[WTZ09] Ji Wu, Kian-Lee Tan, and Yongluan Zhou. Qos-oriented multi-query scheduling over data streams. In *International Conference on Database Systems for Advanced Applications*, pages 215–229, 2009.

[WXL$^+$09] Yan Wang, Weihong Xuan, Wei Li, Baoyan Song, and Xiaoguang Li. A real-time scheduling strategy based on priority in data stream system. In *International Conference on Hybrid Intelligent Systems*, pages 268–272, 2009.

[Yah] Yahoo finance. http://finance.yahoo.com/. Accessed: 2012-07-18.

[YU12] Jing Yang and Sennur Ulukus. Optimal packet scheduling in an energy harvesting communication system. *IEEE Transactions on Communications*, pages 220–230, 2012.

[ZHC02] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z Chen. Task scheduling and voltage selection for energy minimization. In *Design Automation Conference*, pages 183–188, 2002.

[ZMJ12] Kaiwen Zhang, Vinod Muthusamy, and H Jacobsen. Total order in content-based publish/subscribe systems. In *IEEE International Conference on Distributed Computing Systems*, pages 335–344, 2012.

[ZSC$^+$03] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, pages 3–10, 2003.

[ZWL13] Yongluan Zhou, Ji Wu, and Ahmed Khan Leghari. Multi-query scheduling for time-critical data stream applications. In *International Conference on Scientific and Statistical Database Management*, page 15, 2013.