Big Data Research 2 (2015) 127–144

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Practical Identification of Dynamic Precedence Criteria to Produce Critical Results from Big Data Streams

Karen Works [a,*,1], Elke A. Rundensteiner [b,*]

[a] Westfield State University, Westfield, MA, USA
[b] Worcester Polytechnic Institute, Worcester, MA, USA

ARTICLE INFO

Article history:
Received 29 March 2015
Received in revised form 13 September 2015
Accepted 17 September 2015
Available online 30 September 2015

Keywords:
Big data streams
Critical result production
Rapid online adaption

ABSTRACT

During periods of high volume, big data stream applications may not have enough resources to process all incoming tuples. To maximize the production of the most critical results under such resource shortages, a recent solution, *PR* (short for Preferential Result), utilizes both static criteria (defined at compile-time) and dynamic criteria (identified online at run-time) to prioritize the processing of tuples throughout the query pipeline. Unfortunately, locating the optimal criteria placement (i.e., where in the query pipeline to evaluate each prioritization criteria) is extremely compute-intensive and runs in exponential time. This makes PR impractical for complex big data stream systems. Our proposed criteria selection and placement approach, *PR-Prune* (short for Preferential Result-Pruning), is practical. PR-Prune prunes ineffective dynamic criteria and combines multiple criteria along the same pipeline. To achieve this, PR-Prune seeks to expand the duration in the query pipeline that tuples identified as critical are pulled forward. Our experiments use a real data stream from the S&P 500 stocks, synthetic data streams, and a diverse set of queries. The results substantiate that PR-Prune increases the production of the most critical results compared to the state-of-the-art approaches. In addition, PR-Prune significantly lowers the optimization search time compared to PR.

1. Introduction

1.1. Preferential result applications

Big data streams process large volumes of incoming tuples to answer continuous queries. At times they may be unable to process all incoming tuples within the response time required for the application [1]. Yet it often is imperative for applications to assure the production of results from certain objects that are the most critical for the application. Under resource duress, Preferential Result big data streams (or *PR*) utilize both static application-specific preference criteria as well as dynamic criteria identified online to determine which tuples should be allocated resources ahead of other tuples throughout the query pipeline [2].

1.2. Examples of systems with preferential results

Outpatient health care: Data stream systems are used to track people with dementia [3]. In these systems, monitoring people away from their proper location for an extended time (i.e., likely lost) is critical. While monitoring people who live on their own (i.e., need help) may be reduced based on whether or not resources remain after processing more critical tuples, i.e., tuples from people likely to be lost. Periodically when resources are scare, monitoring tuples from other people could be temporarily skipped. Overloads have been experienced in these systems [4].

Law enforcement: Data stream systems are used to monitor prisoners assigned to home arrest [5]. Consider a system that reports any prisoner at an improper location who is within 3 miles of an officer. At the highest level of urgency, escaped violent prisoners (i.e., may cause harm) must be monitored. Next to be monitored are prisoners at an improper location (i.e., likely to be in violation). Finally, prisoners known to be flight risks ought to be monitored when resources are sufficient. These systems get overloaded, e.g., in October 2010 an application that monitors released sex offenders across 49 states shutdown for 12 hours [6].

* Corresponding authors.
E-mail addresses: kworks@westfield.ma.edu (K. Works), rundenst@cs.wpi.edu (E.A. Rundensteiner).

[1] This work was started during Karen's Ph.D. study at WPI.

http://dx.doi.org/10.1016/j.bdr.2015.09.001
2214-5796/

Table 1
Desired result precedence order.

System load	Desired processing order
System not overloaded	all results processed
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

1.3. Running example: stock market

Mutual fund companies often determine what to buy or sell by monitoring the social buzz on different business sectors, i.e., business sectors mentioned in recent news and blogs.

```
(Stock Market Query)                                  /*Operators*/
SELECT S.company_name, S.symbol, S.price
  FROM Stock as S, News as N, Blogs as B
  WHERE contains(S.BusinessSector,                    /*op1*/
    N.BusinessSector)
  AND contains(S.BusinessSector,                      /*op2*/
    B.BusinessSector)
  WINDOW 30 sec;
```

Consider the following desired processing order of tuples in such a system (Table 1) defined by the user at compile-time. First, tuples from aggressive investments should be processed. If resources remain, then tuples from conservative investments should be processed. Until there are adequate resources to process all other tuples, the processing of certain tuples can be temporarily skipped altogether (e.g., tuples from stocks under evaluation).

1.4. Critical tuples

Resources should be allocated to particular tuples based upon the application's desired processing order (Table 1) and the amount of available resources. When the *Stock Market Application* is extremely overloaded, the CPU resources should be dedicated to the tuples most critical for the application. The most critical results are generated from these tuples. In the Stock Market example, the 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.

These most critical results are created by two classes of tuples. The first class are 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 criteria selected by the user (Table 1) [7,8].

The second class are *promising tuples*. Promising tuples are tuples estimated to be *highly likely to produce critical 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 critical results due to their association with their join partners. The criteria to identify promising tuples are *dynamic*. It requires knowledge of which join attributes of the current significant stock tuples are also prevalent in tuples in both the news and blog streams. The identification of such dynamic criteria is accomplished at *run-time* [2].

1.5. How PR adapts the allocation of resources

PR adapts which tuples are preferentially allocated resources when due to system load changes the data stream system is unable to process all incoming critical tuples. Many things can change the system load. It could be changed by the number of incoming tuples that are significant. Namely, increasing the percentage of significant tuples in the pipeline increases the chance that some critical tuples will not be processed due to limited resources. It could be changed by the distribution of promising tuples varying over time as this will cause the number of significant tuples that have join partners to vary as well. Regardless of what causes the system load changes, PR adapts how resources are allocated accordingly.

The goal of PR is to ensure that given the available memory that the most critical tuples are processed. When resources are sufficient, PR will process all tuples. When they are not, PR will process the most critical tuples first. If resources remain then they are dedicated to ensuring that the next most critical tuples are processed (and so on).

To achieve this, criteria of critical results are identified for each join operator. These criteria are then pushed backwards through the query pipeline to operators before their respective join operator that identify and pull forward significant and promising tuples. The query plan optimizer seeks to find the best query plan by adjusting the cost of precedence determination. The cost of precedence determination is adjusted by modifying both which precedence criteria are evaluated and where each of these criteria are evaluated in the query plan.

1.6. State-of-the-art & shortcomings

As we show in Section 3.3, the time complexity of the PR optimizer is exponential in the number of criteria that identify promising tuples and the number of operators in the query plan where such criteria could be evaluated. Thus, it is costly to determine the most effective combination of precedence criteria and where such criteria should be evaluated. When optimization takes a long time, it may delay or even worse yet prevents the production of some critical query results. In the Stock Market application this could result in the company losing money or, in the worse case, going bankrupt. No existing approach addresses this critical problem. It is now the focus of our work.

1.7. Our PR approach & contributions

We now propose a new a criteria selection and placement approach that provides an efficient optimization algorithm by pruning the query plan search space, named *PR-Prune*. To prune the query plan search space is challenging. Namely, we want to eliminate some options but never to prune the best query plan. KEW: Challenges

Our contributions include:

1) We outline the design of **PR-Prune**. We describe how **PR-Prune** utilizes a statistics reduction methodology to eliminate inferior statistics used to find prioritization criteria and how **PR-Prune** reduces the number join operators that pull promising tuples by combining the needs of multiple consecutive join operators.

2) We show that the complexity of PR-Prune is significantly less than the standard PR optimization. We summarize the theoretical contribution.

3) Our **experimental study**, using real data, synthetic data sets, and a wide variety of queries, shows that PR-Prune consistently produces more critical results than the state-of-the-art systems. We quantify the improvements in our experimental study.

2. PR model and queries

2.1. 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 2). Each

Table 2
Notations for PR query plans.

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 (operator where promising rank ends)
$t_i.rnk$	rank of t_i (max rank of $t_i.srnk$ or $t_i.prnk$)
q_j	a query
$q_j.SML$	set of static monitoring levels of query q_j
$q_j.NumRnks$	number of possible ranks in 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
pr_m	a PR query plan
$pr_m.A_{SML}$	set of activated static monitoring levels in pr_m
$pr_m.A_{DML}$	set of activated dynamic monitoring levels in pr_m
s_n	a stream
op_o	an operator
$ER_s(pr_m, srnk)$	expiration rate of potential significant tuples at significant rank $srnk$ in pr_m
$ER_p(pr_m, prnk)$	expiration rate of potential promising tuples at promising rank $prnk$ in pr_m

P-CQL query is a CQL query [9] extended to support multi-tiered monitoring criteria.

(P-CQL Extension to Stock Market Queries)

```
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)
```

Each pair of *rank* and *criteria* clauses specify which objects in the tuple stream the user would prefer to produce results from compared to other objects when resources are scarce. At compile-time these clauses are specified by the user as a part of the query. 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 static monitoring level sml_k's significance. Static monitoring level sml_k is more significant than level sml_l if $sml_k.srnk < sml_l.srnk$. For example, consider the Stock Market P-CQL query above. Respectively, static monitoring levels sml_1, sml_2, and sml_3 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 tuples to pull forward. We refer to the current set of selected monitoring levels as the set of *activated* significant monitoring levels denoted by A_{sml}. If no resource shortage exists then no monitoring levels would be activated and 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 upon the activated static monitoring levels.

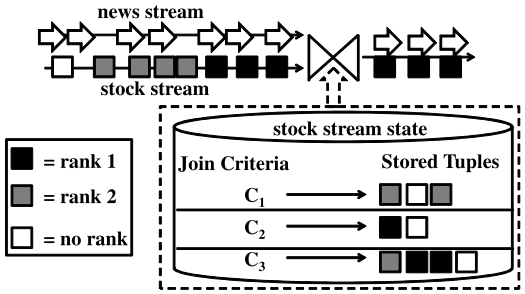

Fig. 1. Estimated significant tuples example.

2.2. Significant tuples

Significant tuples satisfy the membership criteria of an activated static monitoring level [7,8]. A tuple may satisfy the membership criteria of more than one activated static monitoring level.

Definition 1. Significant tuple t_i is designated with one significant rank $t_i.srnk$ which corresponds to the most significant of all the activated static monitoring levels that tuple 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., $sml_3.mem(t_i) = true$). The set of activated static monitoring levels A_{SML} contains static monitoring levels 1, 2, and 3, i.e., $A_{SML} = \{sml_1, sml_2, sml_3\}$. Tuple t_i's significant rank is thus 1, i.e., $t_i.srnk = 1$.

2.3. Promising tuples

Promising tuples are likely to create critical query results by joining with significant tuples at a join operator [2]. Consider a symmetric binary hash join operator op_i [10] that combines tuples from streams s_1 (e.g., news stream) and s_2 (e.g., stock stream). Incoming tuples to this join operator op_i for the news s_1 and stock s_2 streams are stored respectively in the news and stock stream state. Join 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 (i.e., business sector = *Advertising*) (Fig. 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 tuple t_i will be a promising tuple. That is, tuple t_i has a high chance to produce a critical 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 non-critical join results.

Adapting the rank of tuple: Whether or not a tuple is considered to be a promising tuple may *adapt during processing*. Significant tuples can produce critical 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 significant rank throughout the pipeline (Fig. 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 critical query results. Hence after they have been processed by operator op_o, these promising tuples should no longer be preferentially allocated resources. In other words, they are *only promising*

Table 3
Example: ranking query plans.

PR plan	*CPU overhead*	$ER_s(pr_m, 1)$	$ER_s(pr_m, 2)$	$ER_s(pr_m, 3)$	$ER_p(pr_m, 1)$	$ER_p(pr_m, 2)$	$ER_p(pr_m, 3)$
pr_1	1022	0	2	15	0	35	75
pr_2	1256	0	2	15	0	35	75
pr_3	1956	0	2	95	0	80	89
pr_4	1006	9	123	90	89	90	87

for a portion of the query pipeline, namely, until they reach the operator op_o.

Reconsider Fig. 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 at a given join operator 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 Fig. 1.

The set of *dynamic monitoring levels* (*DML*) are constructed at run-time by the optimizer to indicate the criteria and rank of promising tuples at join operators (Section 3.3.1). Each dynamic monitoring level dml_l denoted as (s_n; op_c; *prnk*; *mem* 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 $dml_l.mem$ that identifies such promising tuples. Under limited resources, the optimizer also selectively activates some dynamic monitoring levels (Section 3.3.2 covers how the optimizer chooses which monitoring levels to activate).

Definition 2. A promising tuple t_i has one promising rank $t_i.prnk$ and one designated operator $t_i.op_c$. These attributes are set to the attributes of the activated dynamic monitoring level dml_l (i.e., $t_i.prnk = dml_l.prnk$ and $t_i.op_c = dml_l.op_c$) that satisfies the following criteria. The rank of dml_l is set to the most critical of all the activated dynamic monitoring levels that tuple t_i satisfies the membership criteria of. 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 of dml_l is the furthest down the pipeline of all the activated dynamic monitoring at promising rank $dml_l.prnk$ that tuple t_i satisfies the membership criteria of.

2.4. Tuple rank

Tuple t_i can have both significant and promising rank. Each designation refers to distinct critical 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 is allocated resources based upon the maximum of its significant and promising ranks. Tuple t_i is assigned a *rank* attribute $t_i.rnk$ that is the maximum of tuple t_i's significant and promising ranks.

2.5. Optimal PR plan

A *PR query plan* represents a P-CQL query q_j. Each PR plan pr_m is modeled as a one directional flow network composed of PR algebra operators as nodes and data exchange interfaces that transfer tuples between operators as edges (Section 3.1). The PR query algebra is composed of rank classifier operators and PR augmented standard operators as outlined in Section 3.1.2.

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

An *expired tuple* is a tuple that is no longer processed due to inadequate resources. A tuple may expire at any point along the query pipeline. 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 3. Expiration Rate of Potential Significant Tuples $ER_s(pr_m, srnk)$ is the number of tuples that satisfy static criteria for rank *srnk* and have expired.

The optimal PR plan also improves the flow of tuples that can have promising rank *prnk* until they reach their designated operator. If all tuples that can have promising rank *prnk* are being devoted adequate processing cycles then the number of such tuples that expire before reaching their designated operator (or the expiration rate of potential promising tuples $ER_p(pr_m, prnk)$ at promising rank *prnk*) should be low (ideally zero).

Definition 4. Expiration Rate of Potential Promising Tuples $ER_p(pr_m, prnk)$ is the number of tuples that satisfy dynamic criteria at promising rank *prnk* that expire before they reach their designated operator in PR plan pr_m.

Definition 5. The optimal PR plan compared to all possible PR plans minimizes the expiration rate of both significant (Definition 3) and promising tuples (Definition 4) for each rank starting from the highest rank and remains within the available system capacity.

Consider the example in Table 3. Assume the resources required to execute each PR plan are within the available system capacity. PR plan pr_1 is the best for the following reasons. For the highest rank $rnk = 1$ (i.e., the most critical query results), PR plan pr_4 has an expiration rate of potential significant tuples greater than 0, i.e., $ER_s(pr_4, 1) > 0$. For the next highest rank $rnk = 2$, PR plan pr_3 has an expiration rate of potential promising tuples greater than PR plans pr_1 and pr_2. For all ranks, PR plans pr_1 and pr_2 have equal expiration rates. However, compared to PR plan pr_2, pr_1 has the lowest CPU overhead cost. Thus, PR plan pr_1 is the preferred solution.

3. PR architecture

The online adaptive PR architecture (Fig. 2) is derived from the architecture of self adaptive software [11]. It contains the PR Executor, PR Monitor, PR Optimizer, and PR Adaptor. The PR Executor (Section 3.1) runs the current PR plan and produces query results. The PR Monitor (Section 3.2) gathers statistics to locate the optimal PR plan at runtime. The PR Optimizer (Section 3.3) uses the statistics collected to select a new optimal PR plan for the current system load. This "new" PR plan is forwarded to the PR Adaptor (Section 3.4) which in turn adapts the current PR plan to the new plan. The PR architecture is designed such that online

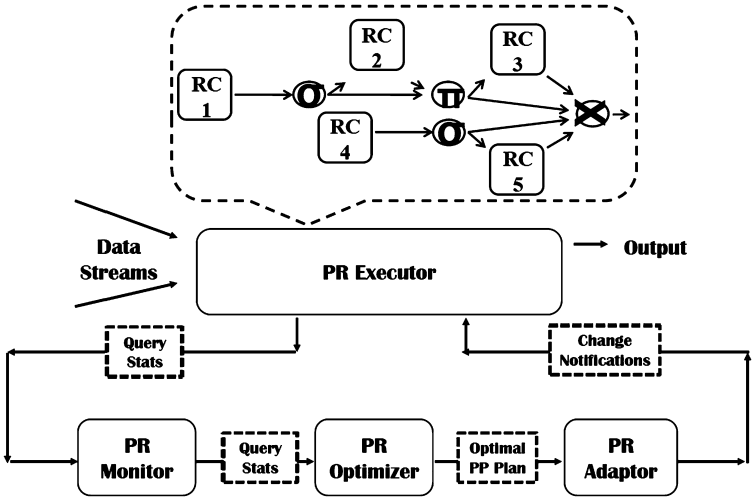

Fig. 2. PR architecture.

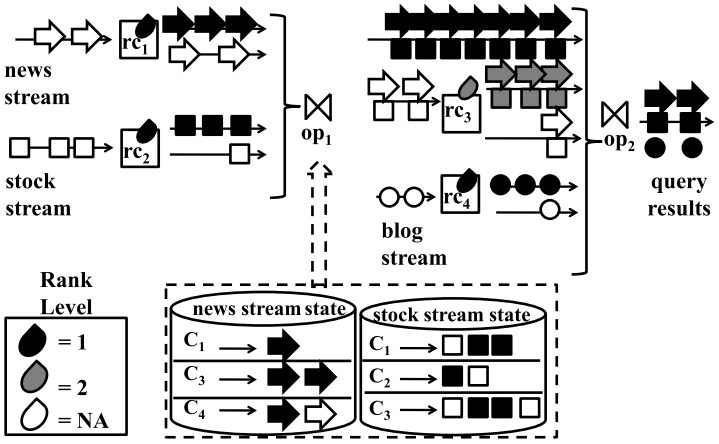

Fig. 3. Stock market PR plan.

PR plan adaption requires no expensive infrastructure changes [2]. Namely, each query operator is designed to be able to adapt online how they allocate resources. The PR Adaptor notifies each operator of their required changes via notifications sent along a control exchange interface. To not delay adaption, PR supports a control exchange interface between each operator and the PR Adaptor dedicated to handling these notifications. This is similar to how interrupts are handled in real-time operating systems [12]. Our PR-Prune approach is part of the PR Optimizer (Section 3.3). The other components are part of PR [2].

3.1. PR executor infrastructure

We now outline the PR Executor and how it executes a PR plan efficiently. The design of the PR Executor addresses the challenge of how to efficiently pull certain tuples ahead of others in the query plan. PR pulls some tuples ahead of others causing tuples to not be processed in arrival time order. Thus PR Executor must also address the challenge of supporting out-of-order processing.

3.1.1. Pulling tuples ahead of others

The *data exchange interface* transfers tuples between operators. To efficiently process certain tuples before other tuples, PR uses multiple queues. 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 tuple resides in the queue that corresponds to their rank. In this case, operators process tuples in rank order. Operator op_o starts processing tuples from the most critical queue. When this is empty and resources remain, operator op_o moves to the second most critical queue. Each result (i.e., tuple t_i) is placed into the incoming queue for tuple t_i's rank of the next down stream operator.

Consider the queues for the news stream in Fig. 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.

3.1.2. PR query algebra

Our PR algebraic operators support both significant and/or promising tuples where the rank may adapt at run-time (Section 3.4). In PR algebra, traditional operators [13] process tuples as usual and propagate the appropriate rank related metadata to the results. *Rank classifier* operators assign preference related metadata 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 to their rank related metadata to the next operator.

Join [10] creates results (t_i, t_j) by matching tuples from streams s_1 and s_2. Tuple t_i is taken from an input queue and processed as fol-

lows. First, tuple t_i is stored with its rank related metadata in the state of tuple t_i's stream s_1. Then, join results (t_i, t_j) are created by joining tuple t_i with tuples t_j stored in stream s_2's state. Next, result (t_i, t_j)'s rank related meta data are set. That is, join result (t_i, t_j) is assigned the highest rank among the rank related meta data 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. Then join 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 critical tuple at a lower rank.

Rank classifier (or *RC*) is a special-purpose operator with static (S_{AS}) and dynamic assessment set (D_{AS}) parameters. It creates results by assigning rank related meta data to tuples in its input queue 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 RC operator op_o.

RCs process each tuple t_i by comparing the criteria in S_{AS} and D_{AS} to tuple t_i in rank order starting from the most critical 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. The reason for this is that if tuple t_i is a significant tuple at rank *rnk* then it is guaranteed to be a critical tuple at rank *rnk* for the duration of tuple t_i's processing. Assigning tuple t_i to be a promising tuple at rank *rnk* or a rank lower than *rnk* does not improve how tuple t_i is preferentially processed. However, even if tuple t_i satisfies a dynamic criteria no static criteria comparisons are eliminated. That is, how tuple t_i is preferentially processed will be affected by whether or not tuple t_i is a significant tuple at rank *rnk* or a rank lower than *rnk*.

Before each standard operator in the PR plan pr_m an RC operator is placed. The optimizer determines which monitoring levels are activated and the static S_{AS} and dynamic D_{AS} assessment set of each RC (Section 3.3). Then the optimizer notifies each RC of changes to their assessment sets.

Each RC in the PR plan assigns rank related metadata to particular tuples. The PR optimizer may select a PR plan in which some RCs may not evaluate the rank related metadata of any monitoring levels. In this case, the assessment sets of these RC will be empty and at runtime all tuples will skip being sent to these RCs (Section 3.4).

Consider the PR Plan in Fig. 3. Incoming news and blog tuples are respectively evaluated by rank classifier operators rc_1 and rc_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 RC in Fig. 3 by the color of the tear drop in the top of the RC. First, incoming stock tuples are evaluated by rank classifier operator rc_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, join results from op_1 are routed to either rc_3 and then to join operator op_2 or directly to join operator op_2 (Section 3.4). RC rc_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.

3.1.3. 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 RC.

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 (Section 3.3). 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 operator op_o places the resulting tuple t_i created from the *elevated* or *degraded* tuple t_j into a different queue than the incoming queue of operator op_o that held tuple t_j. Operator op_o places result t_i into the appropriate queue based upon tuple 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 tuple t_i. If tuple t_i is placed into the insignificant queue then henceforth operators will not preferentially allocate resources to tuple t_i. Tuple t_i retains the values of its rank related metadata in case tuple t_i is elevated or degraded in the future.

3.1.4. Out-of-order handling

PR pulls some tuples ahead of others causing tuples to not be processed in arrival time order. Thus PR operators support out-of-order processing. Strategies have been proposed in the literature to address out-of-order issues due to external factors such as network transmission delays [14]. PR can use similar methods to assure completeness and correctness of results produced. To safely purge tuples from states, similar to [15], each leaf operator op_o periodically sends indicator punctuations when operator op_o will no longer process any tuples from a set period of time. When operator op_o receives such a punctuation, first it assures that no more tuples are waiting to be processed whose query window is this period of time. Then it sends a punctuation to its next operator down stream when it has no more tuples waiting to be processed whose query window is also this period of time. This progressively continues until the punctuation reaches the last operator in the query.

3.2. PR monitor

The PR Monitor gathers statistics to track the progress of tuples that have the potential to be significant and/or promising tuples. 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. One optimization used by the PR Monitor is to reduce the number of statistics collected by removing statistics of attributes that rarely occur in the tuple streams.

3.2.1. Monitoring potential static tuples

Static membership criteria are defined in the P-CQL extension at compile-time (Section 2.1). 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*.

Each join operator op_i tracks how many 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 tracks how many 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* $F_{Sig(op_o, s_n, rnk, c_p)}$, or 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.

3.2.2. Monitoring potential promising tuples

Dynamic membership criteria that identifies promising tuples in the current system are unknown at compile-time. Thus, the PR

Algorithm ***General Create-a-PR-plan***
Input: PCQL query q_j
Input: estimated available resources C_{avail}
Input: initial PR plan pr_m
Input: Frequency of Potential Significant Tuples collected by PR Monitor F_{Sig}
Input: Frequency of Potential Promising Tuples collected by PR Monitor F_{Prom}
Output: a generated PR plan pr_g

1: Create pr_g by copying pr_m
2: Create the set of possible dynamic monitoring levels *DML* using F_{Sig} and F_{Prom} data
3: $Rank = 1$
4: $C_{used} = 0$

5: **while** (($Cavail > Cused$) and ($Rank <= q_j.NumRnks$)) **do**
6: add static monitoring levels at rank *Rank* from $q_j.SML$ the set of static monitoring levels of query q_j to the set of activated static monitoring levels $pr_g.A_{SML}$ in the generated PR plan pr_g
7: add dynamic monitoring levels at rank *Rank* from *DML* (created above in Step 2) to the set of activated dynamic monitoring levels $pr_g.A_{DML}$ in the generated PR plan pr_g
8: **for** each monitoring level ml_l in the sets of activated static and dynamic monitoring levels $pr_g.A_{SML}$ U $pr_g.A_{DML}$ at rank *Rank* **do**
9: Determine which rank classifier in the plan will evaluate monitoring level ml_l
10: **end for**

11: $Rank = Rank + 1$
12: $Cused$ = Estimated CPU used by current version of pr_g
13: **end while**
14: return pr_g

Fig. 4. The general steps to locating one possible PR plan.

Monitor gathers statistics that the PR Optimizer will use to identify such criteria. To locate potential dynamic membership criteria, 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. 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. 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 join criteria c_p identifies significant join partners at join operator op_o.

Consider the PR Plan (Fig. 3) for the Stock Market Example (Section *Running example: stock market*). 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.

In the PR Plan (Fig. 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.

The number of possible join criteria is exponential given the possible domains and range of join criteria values. Thus, PR-Prune reduces the number of frequencies collected by using a heavy hitter algorithm [16]. 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 are gathered on a range of values. This ensures that the join criteria monitored are from a discrete domain.

3.3. PR optimizer

Upon receiving the statistics, the PR Optimizer selects the optimal order of operators within the query plan and then generates a new PR plan. First the *initial PR plan* is created by placing an RC with empty assessment sets before each standard operator in p_m. Only one RC is required as the static and dynamic assessment sets of multiple adjacent RCs 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.

As outlined below, this is challenging as the complexity of dynamic priority determination in locating the optimal PR plan is exponential in the number of dynamic criteria and the number of designated operators. Our PR-Prune addresses this by reducing dynamic criteria and designated operators.

Fig. 4 contans the general steps to locating one possible PR plan. We now explore the details behind this algorithm.

3.3.1. Creating the set of 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. 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 the 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 with different ranks. 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 partners identified by criteria c_p.

Consider PR Plan (Fig. 3). 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.

3.3.2. Selecting which static and dynamic monitoring levels to activate

Some of the potential dynamic monitoring levels in *DML* may reference join operators that will not have any incoming signifi-

Algorithm ***Detailed Create-a-PR-plan***
Input: PCQL query q_j
Input: estimated available resources C_{avail}
Input: initial PR plan pr_m
Input: Frequency of Potential Significant Tuples collected by PR Monitor F_{Sig}
Input: Frequency of Potential Promising Tuples collected by PR Monitor F_{Prom}
Output: a generated PR plan pr_g

1: Create pr_g by copying pr_m
2: Create the set of possible dynamic monitoring levels *DML* using F_{Sig} and F_{Prom} data
3: $Rank = 1$
4: $C_{used} = 0$

5: **while** ((*Cavail* > *Cused*) and (*Rank* <= q_j.*NumRnks*)) **do**
6: add static monitoring levels at rank *Rank* from q_j.*SML* the set of static monitoring levels of query q_j to the set of activated static monitoring levels $pr_g.A_{SML}$ in the generated PR plan pr_g
7: **for** each monitoring level ml_l in the sets of activated static monitoring levels $pr_g.A_{SML}$ at rank *Rank* **do**
8: Determine which rank classifier in the plan will evaluate monitoring level ml_l
9: **end for**

10: Identify *designated operators*, i.e., join operators where significant tuples at rank *Rank*

11: add dynamic monitoring levels at rank *Rank* from *DML* (created above in Step 2) to the set of activated dynamic monitoring levels $pr_g.A_{DML}$ in the generated PR plan pr_g
12: **for** each monitoring level ml_l in the sets of activated dynamic monitoring levels $pr_g.A_{DML}$ at rank *Rank* **do**
13: Determine which rank classifier in the plan will evaluate monitoring level ml_l
14: **end for**

15: $Rank = Rank + 1$
16: *Cused* = Estimated CPU used by current version of pr_g
17: **end while**
18: return pr_g

Fig. 5. The detailed steps to locating one possible PR plan.

cant tuples. A join operator op_o is said to be *designated* if and only if it has incoming tuples that are significant tuples. Whether or not incoming tuples to join operator op_o are significant (or not) 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, the PR optimizer firsts select which static monitoring levels to activate and determines where in the plan to evaluate each static criteria before considering where to evaluate the dynamic monitoring levels.

Generating one possible PR plan involves the following detailed steps. (See Fig. 5.)

3.3.3. Determining where to evaluate each static and dynamic criteria

The PR Optimizer must determine which RC(s) should evaluate which criteria of the activated monitoring levels. We now discuss how many RCs 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 rank classifier (RC) 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 RC to assess each static criteria.

Each s-crit s-$crit_k$ has a *static evaluation path*, i.e., an ordered set of operators that respectively begins and ends at the first and last operators in the plan that can evaluate s-$crit_k$, or roughly, the operators where incoming tuples contain all the attributes required to evaluate s-$crit_k$.

Dynamic evaluation path: In contrast, 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, namely the join operator where t_i is estimated to join with a significant tuple. Further along the pipeline, the tuple t_i may be assigned another promising rank. That is, during its processing, a tuple may be pulled forward to different designated operators along the query pipeline. The evaluation paths of criteria may overlap. In addition, 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 RCs, namely, after any of its designated operator. This is because tuples may lose their current promising rank at each designated operator. This is the first place to check if a tuple should be assigned the promising rank of a designated operator further along the query pipeline.

Each d-crit d-$crit_i$ has a *dynamic evaluation path*, i.e., an ordered set of operators that respectively begins and ends at the first and last operators in the plan that can evaluate d-$crit_i$.

Static vs dynamic evaluation paths: A static evaluation path ends at the last RC operator in the query plan as significant tuples remain significant for the entire query pipeline. In contrast, a dynamic evaluation path ends at the RC 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.

3.3.4. PR plan search space

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

Static priority determination: Recall that each static criteria is evaluated in one RC in its static evaluation path. In the PR Plan (Fig. 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 RC_2 (i.e., before join operator op_1) or by RC_3 (i.e., after join operator op_1 and before join operator op_2).

Complexity of static priority determination: Assume that in PR plan pr_m, there are $|SEP|$ static evaluation paths. Each static evaluation path sep_k contains $|sep_k.rc|$ RC 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 $|sep_k.rc|^{|SCrit(sep_k,rnk)|}$ possible combinations of which RC evaluates each static criteria in $SCrit(sep_k, rnk)$. Hence, for rank rnk, there are $\prod_{k=1}^{|SEP|} |sep_k.rc|^{|SCrit(sep_k,rnk)|}$ 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) RCs in its dynamic evaluation path. In the PR Plan (Fig. 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. Rank Classifier RC_1 would be the only operator that need to try to identify these tuples using the promising criteria (i.e., before join operator op_1). This is because there are no designated operators between RC_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 RCs before join operator op_2, namely, RC_1 and RC_3.

If only RC_3 evaluates join criteria c_1 then such tuples only need to be located by RC_3. This is because there are no designated operators between RC_3 and the designated operator for join criteria c_1, i.e., designated operator op_2. However, this is not true if RC_1 evaluates join criteria c_1 and operator op_1 is a designated operator between RC_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, RC_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 RC_3. This is because RC_3 is the first RC after designated operator op_1 and the last RC before the next designated operator op_2. The optimizer must select one RC in each dynamic re-evaluation path to evaluate the dynamic criteria.

Complexity of dynamic priority determination: For each dynamic criteria c_p whose dynamic evaluation path is dep_l, there are $\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|DREP(dep_l,rc_x)|} |drep_m.rc|$ possible combinations of which RCs evaluate dynamic criteria c_p. Rank Classifier rc_x is the RC in dynamic evaluation path dep_l selected by the optimizer. $|DREP(rc_x, dep_l)|$ denotes the number of dynamic re-evaluation paths that proceed rc_x in dynamic evaluation path dep_l. $|drep_m.rc|$ denotes the number of RCs 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 $\prod_{l=1}^{|DEP|} (\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|DREP(rc_x)|} |drep_m.rc|)^{|DCrit(dep_l,rnk)|}$ possible PR plans where the dynamic criteria at rank rnk can be evaluated.

The PR search space thus is: $\sum_{rnk=1}^{|q_j.SML|} \prod_{k=1}^{|SEP|} |sep_k.rc|^{|SCrit(sep_k,rnk)|} * \prod_{l=1}^{|DEP|} (\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|DREP(dep_l,rc_x)|} |drep_m.rc|)^{|DCrit(dep_l,rnk)|}$.

We notice that the complexity of dynamic priority determination in the PR problem is exponential in the number of dynamic criteria $|DEP|$ and the number of designated operators $|DCrit(dep_l, rnk)|$. It is impractical to exhaustively search for the optimal PR plan with many dynamic criteria and designated operators.

3.3.5. PR prune optimization strategy

We now introduce PR-Prune, an 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 the number of dynamic criteria $|DEP|$ and the number of designated operators $|DCrit(dep_l, rnk)|$. 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 $|DREP(dep_l, rc_x)|$).

Roughly, locating a single PR Plan in PR-Prune consists of the following steps. (See Fig. 6.)

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. 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. While 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: $(\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|DREP(dep_l,rc_x)|} |drep_m.rc|)$ 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 statistics-based reduction method. Namely, 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 critical 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. (See Fig. 7.)

Cost savings: There is clearly a cost savings in reducing the number of designated operator and the number of dynamic evaluation paths. Each designated operator eliminated reduces the number of possible PR plans that need to be generated by $(\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|DREP(dep_l,rc_x)|} |drep_m.rc|)^{|DCrit(dep_l,rnk)|}$ plans. While each dynamic re-evaluation path eliminated reduces the number of possible PR plans that need to be generated by $(\sum_{x=1}^{|dep_l.rc|} |drep_m.rc|)^{|DCrit(dep_l,rnk)|}$ plans.

Reducing the dynamic evaluation paths: In the PR Plan (Fig. 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 RC 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 RC 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 RC against dynamic criteria c_2. In this case, tuple t_i would a promising tuple at rank rnk with designated operator op_2.

Algorithm ***General PR-Prune Create-a-PR-plan***
Input: PCQL query q_j
Input: estimated available resources C_{avail}
Input: initial PR plan pr_m
Input: Frequency of Potential Significant Tuples collected by PR Monitor F_{Sig}
Input: Frequency of Potential Promising Tuples collected by PR Monitor F_{Prom}
Output: a generated PR plan pr_g

Create pr_g by copying pr_m
Create the set of possible dynamic monitoring levels *DML* using F_{Sig} and F_{Prom} data
Eliminate inferior dynamic criteria from the set of possible dynamic monitoring levels *DML*
$Rank = 1$
$C_{used} = 0$

while (($Cavail > Cused$) and ($Rank <= q_j.NumRnks$)) **do**
  add static monitoring levels at rank *Rank* from $q_j.SML$ the set of static monitoring levels of query q_j to the set of activated static monitoring levels $pr_g.A_{SML}$ in the generated PR plan pr_g
  for each monitoring level ml_l in the sets of activated static monitoring levels $pr_g.A_{SML}$ at rank *Rank* **do**
    Determine which rank classifier in the plan will evaluate monitoring level ml_l
  end for

  Identify *designated operators*, i.e., join operators where significant tuples at rank *Rank*
  Reduce the set of *designated operators* found in the previous step

  add dynamic monitoring levels at rank *Rank* from *DML* (created above in Step 2) to the set of activated dynamic monitoring levels $pr_g.A_{DML}$ in the generated PR plan pr_g
  for each monitoring level ml_l in the sets of activated dynamic monitoring levels $pr_g.A_{DML}$ at rank *Rank* **do**
    Determine which rank classifier in the plan will evaluate monitoring level ml_l
  end for

  $Rank = Rank + 1$
  $Cused$ = Estimated CPU used by current version of pr_g
end while
return pr_g

Fig. 6. The general PR-Prune steps to locating one possible PR plan.

Algorithm ***Detailed PR-Prune Create-a-PR-plan***
Input: PCQL query q_j
Input: estimated available resources C_{avail}
Input: initial PR plan pr_m
Input: Frequency of Potential Significant Tuples collected by PR Monitor F_{Sig}
Input: Frequency of Potential Promising Tuples collected by PR Monitor F_{Prom}
Output: a generated PR plan pr_g

Create pr_g by copying pr_m
Create the set of possible dynamic monitoring levels *DML* using F_{Sig} and F_{Prom} data
Eliminate inferior dynamic criteria from the set of possible dynamic monitoring levels *DML*
$Rank = 1$
$C_{used} = 0$

while (($Cavail > Cused$) and ($Rank <= q_j.NumRnks$)) **do**
  add static monitoring levels at rank *Rank* from $q_j.SML$ the set of static monitoring levels of query q_j to the set of activated static monitoring levels $pr_g.A_{SML}$ in the generated PR plan pr_g
  for each monitoring level ml_l in the sets of activated static monitoring levels $pr_g.A_{SML}$ at rank *Rank* **do**
    Determine which rank classifier in the plan will evaluate monitoring level ml_l
  end for

  Identify *designated operators*, i.e., join operators where significant tuples at rank *Rank*
  Reduce the set of *designated operators* found in the previous step

  add dynamic monitoring levels at rank *Rank* from *DML* (created above in Step 2) to the set of activated dynamic monitoring levels $pr_g.A_{DML}$ in the generated PR plan pr_g
  for each monitoring level ml_l in the sets of activated dynamic monitoring levels $pr_g.A_{DML}$ at rank *Rank* in order of highest to lowest product of the frequencies of potential significant and promising tuples **do**
    Determine which rank classifier in the plan will evaluate monitoring level ml_l
  end for

  $Rank = Rank + 1$
  $Cused$ = Estimated CPU used by current version of pr_g
end while
return pr_g

Fig. 7. The detailed PR-Prune steps to locating one possible PR plan.

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. 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, a critical 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 this join operator, namely, the join operator furthest along the query pipeline.

PR-prune plan search space: The **PR-Prune search space** for plan p_i is thus: $\sum_{rnk=1}^{|q_j.SML|} \prod_{k=1}^{|SEP|} |sep_k.rc|^{|SCrit(sep_k,rnk)|} \times \prod_{l=1}^{|RDEP|} (\sum_{x=1}^{|dep_l.rc|} \prod_{m=1}^{|RDREP(dep_l,rc_x)|} |drep_m.rc|)^{|RDCrit(dep_l,rnk)|}$. *RDEP* is the reduced set of dynamic evaluation paths. $RDREP(dep_l, rc_x)$ is the reduced set of dynamic re-evaluation paths that follow rank classifier rc_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*.

3.3.6. Optimal order of operators in PR plan

Selecting the optimal ordering of operators within a plan is NP-hard [17]. 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 [18], the PR optimizer locates the optimized PR plan pr_m for a given traditional query plan p_m.

3.4. PR adaptor

After the optimizer selects a new optimized PR plan, the PR Adaptor adapts the current PR plan to the new one. A challenge to any online query plan adaption is how to do so efficiently so as to require the least amount of resources and the shortest amount of time. A key to how PR efficiently supports the adaption of PR plans is the retention of a rank classifier (a.k.a., *RC*) before each standard operator. As we explain in further detail below, simply changing the attributes of the RC operators allows the PR current plan to change to a new PR plan. This allows PR to adapt the PR plan 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 RC a notification about their new static and dynamic assessment sets.

Since our design requires that an RC exist before each operator, it is possible that some RCs in the PR plan do not evaluate any static or dynamic monitoring levels and thus, have empty assessment sets. Operators do not send tuples to such RCs. Such RCs are simply skipped. To skip extraneous RCs, standard operators control where they send results. They send results to either: 1) the next down stream RC rc_p and then to the next down stream operator op_o or 2) directly to the next down stream operator op_o, i.e., skipping RC rc_p. To adapt where results are sent, the PR Adaptor notifies each operator of the whether or not to send their results to the next down stream RC.

In short, PR quickly adapts the PR plan online. The adaption does not require any infrastructure changes. Instead, each operator locally adapts online how they allocate resources to any future in process tuples (Section 3.1.3).

4. Experimental evaluation

4.1. Experimental setup

Alternative solutions. We compare PR with the PR-Prune Optimizer (or *PR-Prune*) to PR with the basic PR Optimizer solution (or *PR*) [2]. Both approaches identify and pull promising tuples forward. However, PR-Prune reduces the optimization time for criteria identification and placement which allows PR-Prune to pull promising tuples forward sooner than PR. We also study the traditional data stream management system 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 critical results. We also analyze the state-of-the-art resource allocation methodologies for data streams (Section 5), namely, semantic (or *Sem*), random (or *Rand*) [19], and Proactive Promotion (or *PP*) [7,8].

PP [7,8] only pulls significant tuples forward. It does not identify nor pull forward promising tuples. As shown in [2], this limits the number of critical results produced by PP as compared to PR. 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 RC 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 efficiently pulling promising tuples forward (i.e., PR-Prune). Trad simply processes all tuples in FIFO order. It neither sheds nor allocates resources to specific tuples based upon their rank.

All systems were implemented in the same data stream management system, in our case, CAPE [20]. 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 critical query 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-Prune. The quantity of promising tuples affects the number of tuples in the workload that may benefit from being pulled forward. PR-Prune's effectiveness is based upon efficiently adapting the PR plan to allow it to identify promising tuples at their designated join operator and creating critical 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.

In addition, this demonstrates PR-Prunes ability across a diverse set of queries, namely, adding additional join operators creates unique distinct query plans. 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 Query (Section 1.3) with the P-CQL extension. This query has 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 was created from the pricing information on the S&P 500 stocks gathered over July 18, 2012 via Yahoo Finance [21]. 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 metric of how well the U.S. economy is doing.

Stock Data Set 1 mimics the monitoring levels of an investment company. The stock market can change rapidly. Thus 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). 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.

News and blog data streams were created by randomly selecting from either the sectors in the Global Industry Classification Standard (GICS) or the subsectors in the Industrial Classification Benchmark (ICB). These streams represent the industries or business sectors mentioned in current postings. GICS was developed by Morgan Stanley Capital International (MSCI) and Standard & Poor's. It contains 10 sectors that categorize the industries of companies in the S&P 500 by their stock symbol. ICB was created by Dow Jones and FTSE. It contain 100 subsectors that classify the business sector of the companies in the S&P 500 by their stock symbol.

Most experiments use the *News/Blog Data Set 1* which mimics the average stock market day 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 promising tuples, i.e., contain the same the ICB subsectors as the significant tuples in the Stock Data Set 1.

Hardware. Our experiments were conducted in a compute cluster. Each host has two AMD 2.6 GHz Dual Core Opteron CPUs and 1 GB 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. Most of our experiments measure separately for each monitoring level the cumulative throughput of critical query results.

4.2. 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 Query. The window size = 1000 tuples.

Figs. 6a–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 critical results (i.e., level 1) (Fig. 6a). Quantitatively, PR-Prune produced 98%, 95%, 45%, 33%, and 9% more of the most critical results than Trad, Rand, Sem, PP, and Pr respectively. In addition, for the second most critical results (i.e., level 2), PR-Prune produced more results than PR, PP, Rand, and Trad (Fig. 6b). Quantitatively, PR-Prune produced 97%, 88%, 24%, and 18% more of the second most critical results than Trad, Rand, PP, and Pr respectively. Only Sem produced slightly more critical results at level 2 than PR-Prune, namely, 13% more. Rand, Sem, and Trad all process tuples in FIFO order. Thus they cannot guarantee that resources are dedicated to the most critical tuples first. Finally, for the least critical monitoring level, (i.e., level 3) PR-Prune produced more results than PR, PP, Rand, and Trad (Fig. 6c). Quantitatively, PR-Prune produced 96%, 83%, 22%, and 17% more of the least most critical results than Trad, Rand, PP, and Pr respectively. Again, only Sem produced more of the least critical results than PR-Prune, namely, 79% more.

Even though Sem produced significantly more of the least critical results, PR-Prune was most efficient at dedicating resources to producing the most critical query results. Namely, the goal of PR-Prune is to efficiently dedicate the resources to produce the most critical results, followed by the next most critical results (and so on) until no resources remain. PR-Prune is indeed effective at increasing the throughput of the most critical results. This is due to PR-Prune's ability to efficiently pull forward both significant and promising tuples. In addition, although Sem produces more critical results at monitoring levels 2 and 3, it produces notably less of the most critical results. In the Stock Market Example this could have severe financial consequences.

Varying the number of promising tuples. We now explore how the number of promising tuples in the streams affects the throughput of critical results. This experiment uses Stock and News/Blog Data Set 1, and the Stock Market Query. The window size = 1000 tuples. 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. Figs. 5a–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, we did not control the number of critical results that can be produced by each scenario. The data streams was randomly produced for each Data Set. Therefore there is no correlation between the results from different Data Sets. We can view the results independently to 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 critical results than the other approaches (Figs. 7a–d). 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 critical 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 critical 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 the most critical query results.

Locating promising tuples adds overhead. PR-Prune efficiently determines which dynamic criteria to use to locate promising tu-

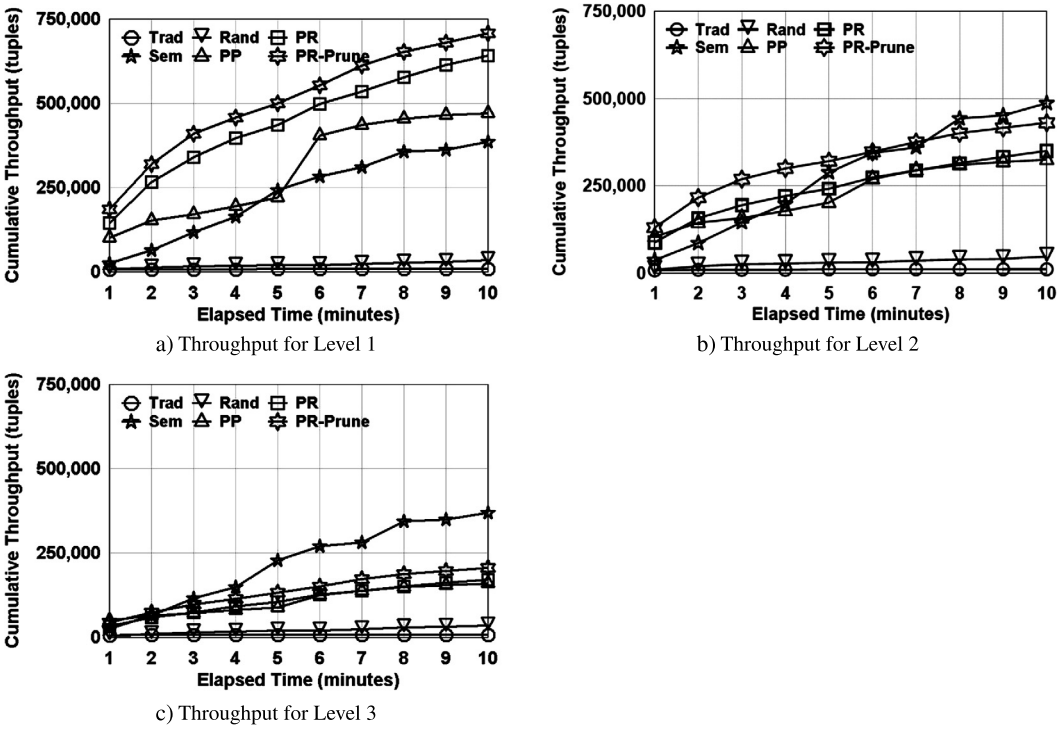

Fig. 8. Effective at increasing throughput of the most significant results.

ples 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 is still more efficient than the other approaches. This is demonstrated by the fact that PR-Prune produced more of the most critical query results than the other approaches.

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. While the comparison to PR highlights the advantage of reducing the optimization time and thus quickly pulling promising tuples forward. The data streams was again randomly produced for each Data Set. Therefore there is no correlation between the results from different Data Sets. We can view the results independently to observe trends in how PR-Prune performs in scenarios.

Varying the number of join operators. This experiment again uses Stock and News/Blog Data Set 1 where the window size = 1000 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. Figs. 8a–c show the average cumulative throughput for monitoring levels 1–3 respectively after 10 minutes. Overall PR-Prune consistently produced more of the most critical results (i.e., level 1) than PR and PP.

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 window size = 1000 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 will seek to pull promising tuples forward across the entire query path. While respectively in the 2, 4, and 8 path queries, 4, 2, to 0 (no) join operators share the same join attribute. In the 2 path query PR-Prune will seek to pull promising 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 promising 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 respectively seek to pull promising 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. While 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.

Figs. 9a–c show the average cumulative throughput for monitoring levels 1–3 respectively over 10 minutes. Overall PR-Prune consistently produced more of the most critical results (i.e., level 1) than PR and PP. It can be seen that PR-Prune produces more of the most critical results than PR and PP regardless of the number of dynamic evaluation paths.

Optimization search time. We now compare the optimization search time of PR-Prune to PR to respectively locate the “best” or optimal PR plan for queries that contain a varied number of dynamic evaluation paths (Fig. 10). 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 took 31.6% less time than PR to search for the “best” PR plan. While in the 8 paths case, PR-Prune took 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

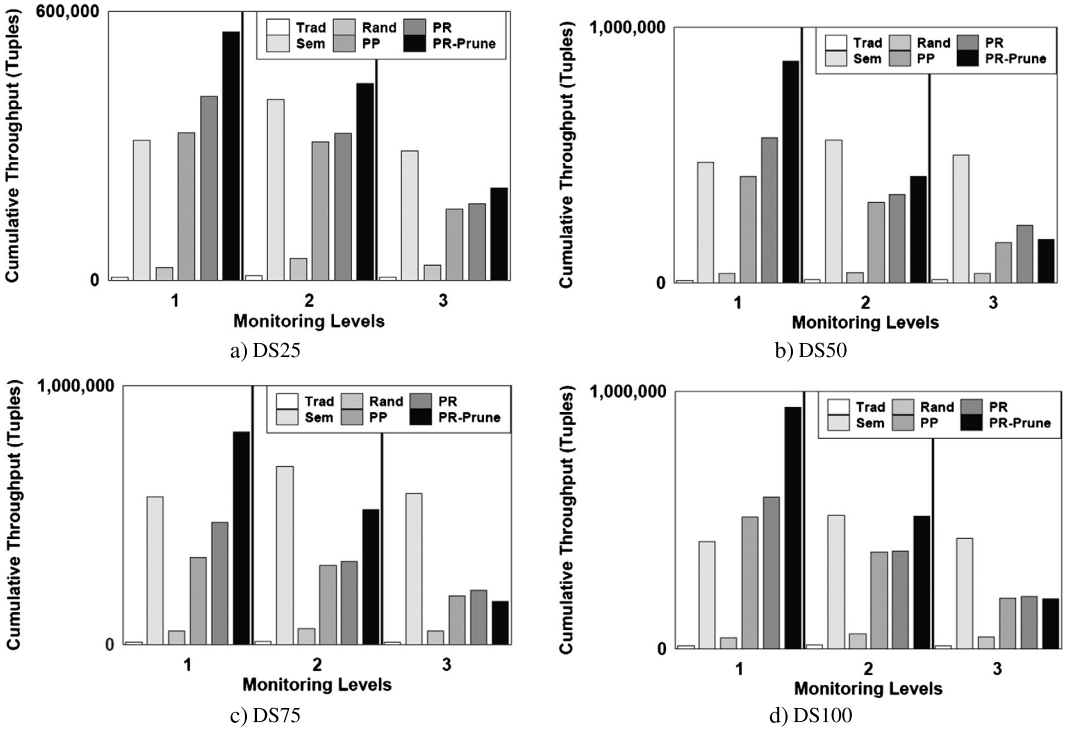

Fig. 9. Varying the number of promising tuples.

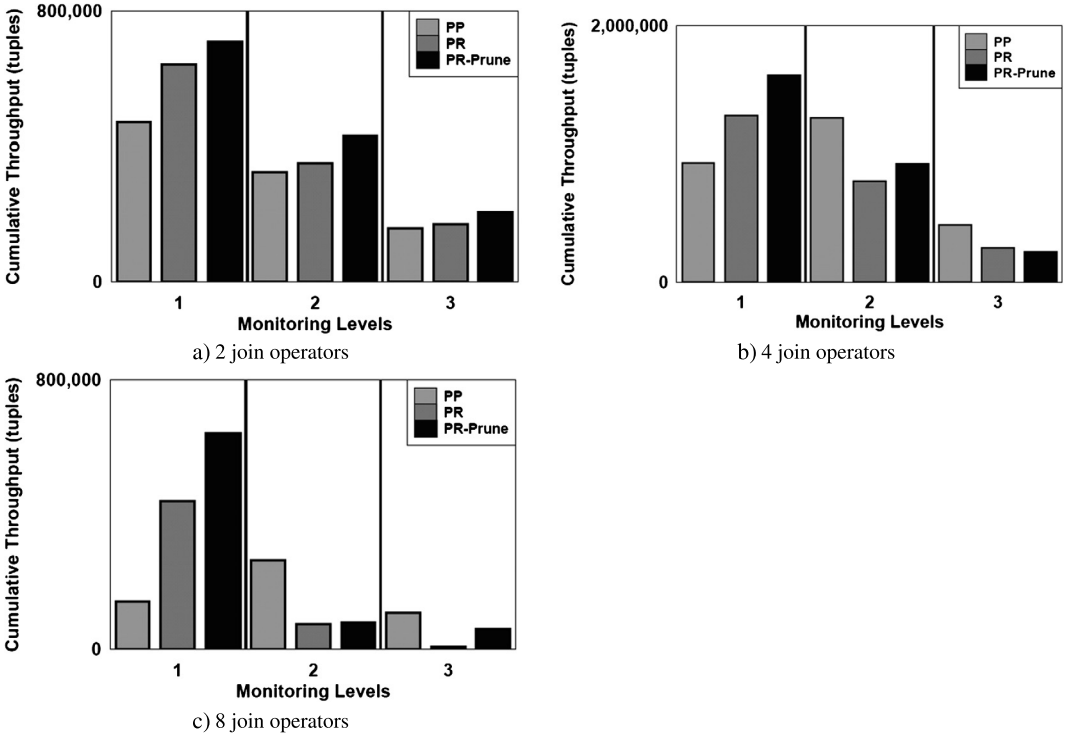

Fig. 10. Varying the number of join operators.

into a single dynamic evaluation path (Section 3.3.5). While 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 (Section 3.3.5).

Execution-runtime CPU overhead. To measure the runtime overhead we evaluate the cumulative throughput using the worst case scenario for PR-Prune (Fig. 11c), namely, when no static monitoring levels are defined. This experiment uses the Stock Market Query (Section 1.3) 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 resource allocation methodologies (Sem, Rand, PP, PR, PR-Prune) here correspond 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

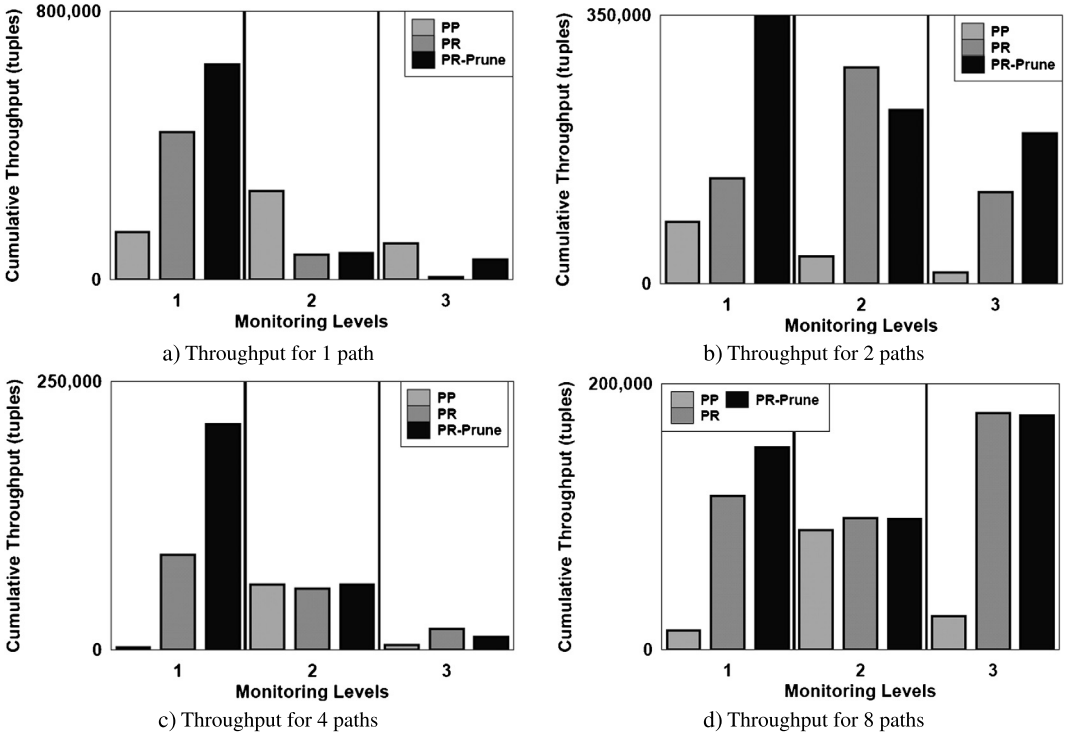

Fig. 11. Varying the number of dynamic evaluation paths.

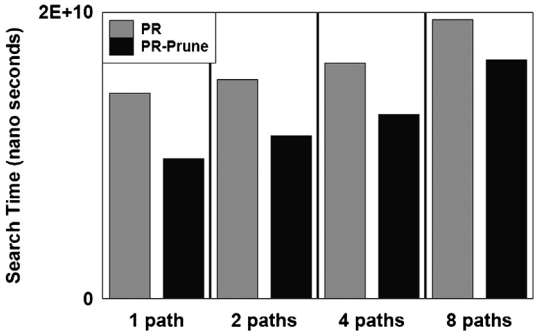

Fig. 12. Optimization search time.

continue to utilize resources to evaluate how to allocate resources. This experiment uses the Stock and News/Blog Data Set 1 where the window size is 1000 tuples.

As shown by our results (Fig. 13c), 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 (Section 3.2). That is, PP and Sem only collect statistics regarding static monitoring levels. In contrast, PR-Prune and PR collect statistics regarding both 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 Query) in the query pipeline using the scenario outline above (Fig. 13a and 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 (Section 3.1.4). This causes the purge to be delayed longer than for other methods and subsequently the state to be slightly larger.

4.3. Summary of experimental findings

We now summarize our key findings.

1) PR-Prune increases the throughput of the most critical results.

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 critical 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; see Fig. 12).

4) PR-Prune does require some fairly negligible 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 critical query results first (see Fig. 13).

5. Related work

We now review the related work beyond that in Section 1.6.

Result ordering methods adapt the order in which the results are produced [22–24]. These approaches produce all results regardless of how long it takes. The *Juggle operator* [22,23], built for exploratory queries, allows the users to adjust which rows or

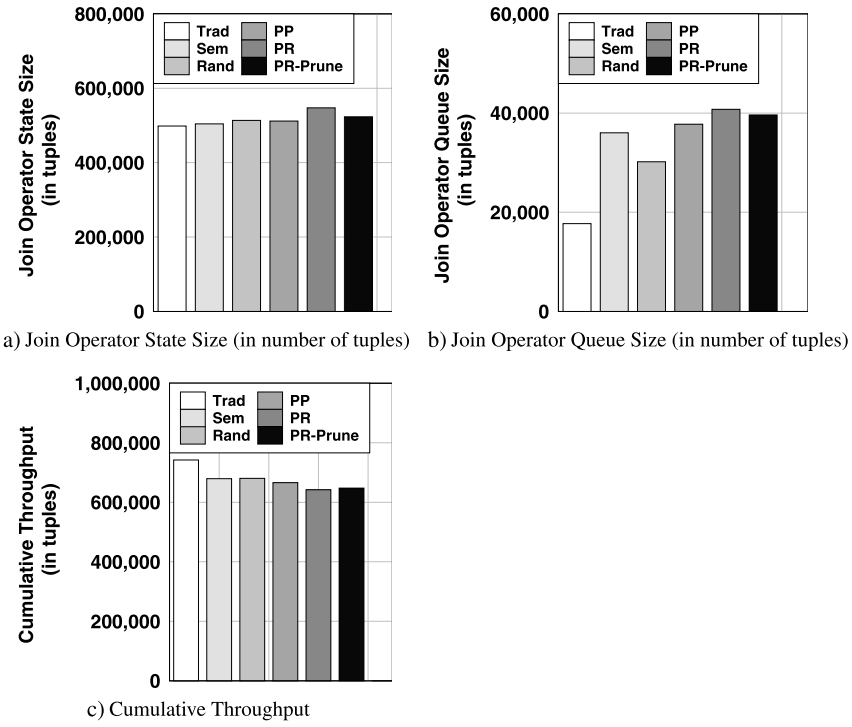

Fig. 13. Execution-runtime CPU and memory overhead.

columns in the results are generated first. [24] reorders tuples in data stream management systems per user preferences. In contrast, PR may not always produce all results. That is, some tuples may expire. When resources are scarce, PR uses the available resources to reorder tuples identified as central to producing the most critical query results.

Result selection methods restrict which results are produced [25–28]. *Top k* [25,26], *web search* [27], and *preference queries* [28] limit the number of results produced, often a fixed small cardinality, by ranking tuples based upon user preferences. For instance, they may return only the set of most interesting results that will fit on a screen. In contrast, PR uses the available resources to process as many tuples identified as central to producing as many of the critical query results as possible.

There are many resource allocation approaches that *reduce the workload* [29–39]. One approach is load shedding [29–35]. Load shedding drops less significant tuples. It only allocates resources to the tuples not dropped, specifically, the most significant tuples. Once a tuple is chosen to be processed, the tuple will not be shed at any point along the query pipeline. [33] described a load shedding scheme that dynamically determines when, how many, and where in the query plan resources will be allocated to the most significant tuples. While [40] introduced a load shedding algorithm for mining data streams.

Another resource allocation approach that *reduces the workload* is spilling [36–39]. Spilling temporarily moves less significant tuples to memory to be processed later and allocates resources to the tuples not spilled, specifically, the most significant tuples. Many approaches, such as XJoin [36], Hash-Merge Join [37], and MJoin [38], push tuples temporarily to disk when no available memory remains. [39] considers how spilling tuples in one operator affects the other operators in the same plan.

There are also many resource allocation approaches where optimization decisions concern *locally reordering the workload* [41–43]. [41] makes localized decisions about subsets of interest in the data stream between neighboring operators in the query pipeline and allocates resources to these subsets of interest. 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 overall counter productive for other operators. Ultimately, such an uncoordinated approach may produce fewer rather than more significant query results. In comparison, [42] extended join operators to makes localized decisions about how to allocate resources to process the most profitable segments of the join windows. As another direction, [43] makes localized decisions about which tuples that join operations are performed on based upon the input stream rates, time correlation between the streams and properties of the tuples stored in the join state.

PR-Prune instead requires a *global coordinated approach* that considers the impact of pulling promising tuples forward on the production of critical query results. PR-Prune seeks to efficiently and adaptively adjust resource allocation throughout the query pipeline. Namely, PR-Prune makes centralized decisions about how it is best to allocate resources at each operator in the query pipeline.

Other query optimization approaches that consider processing preferences have been introduced by Babcock et al. [44] and Raman et al. [45]. In Babcock et al. [44], using the probability distribution of the approach selected, the 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 predictability and performance (which could be at odds sometimes) to find the appropriate query plan. While Raman et al. [45] introduce the STAIRS operator, an extension to the ripple join operator. Beyond the standard join processing, the STAIRS operator allows the system to dynamically adjust the intermediate tuples stored in the cache to optimally process tuples based upon changes in the selectivity, data arrival rates, and/or performance of operators in the

query plan. Our PR Optimizer focuses on discovering criteria to identify promising tuples based on some preference scheme, selecting which monitoring levels to activate, and deciding where in the plan to evaluate each criteria of the activated monitoring levels.

Another area is data stream system scheduling. Broadly, scheduling methods determine which operator to run and for how long to improve different metrics. There are many efforts that introduce methods whose focus is to improve the metrics of a single query [46–49]. [46] proposed a rate based scheduling policy based upon a response time metric for single a query. Aurora [50] first uses Round Robin to schedule which query to run and a scheduling policy based upon the average tuple latency metric to schedule operators within the query to run. [48] proposed a scheduling policy based upon improving the quality of data as a metric. Tick scheduling [49] strategy is based upon maximizing throughput and minimizing deadline miss ratio by reducing overheads.

Similarly, there are many efforts that introduce methods whose focus is to improve the metrics across multiple queries [51–61]. [51] explores the multi-query scheduling as a job-scheduling problem and utilizes real-time computing approaches. [52] explored task schedulers that consider the hardware statistics in parallel processing systems that execute multiple streaming queries. [53] proposes an adaption to chain scheduling strategy to minimize the memory overhead and output latency. The Golden Mean scheduling strategy was introduced by [54] that seeks to minimize the memory overhead and output latency by considering the future workload. [55] proposed a multi-query scheduling method that uses a metric based upon reducing the average response time per query by prioritizing shared operators (i.e., operators that process tuples for more than one query) and window constraints. [56] introduces a scheduling policy that uses the slowdown metric to balance the performance and fairness needs of multiple queries. [57] introduced a multi-query scheduling strategy that recasts the problem into a job scheduling problem. Chain is a multi-query scheduling policy that uses a memory usage metric [59]. The work on [60] extended Chain to use a combined memory usage and response time metric. While, [61] extend the Chain scheduling for complex DAG plans. [58] seeks to maximize the output of all query results, regardless of their significance, by adjusting the scheduling of query join operators.

Other efforts look into adapting the scheduling method used [62–64]. [62] uses machine learning unit to adapt systems parameters to improve the output latency. [63] periodically selects a scheduler to support multiple metric objectives. [64] proposed an adaptive scheduling algorithm that changes the scheduling algorithm online to meet multiple metric objectives.

[65–68] use real time deadlines metrics to schedule operators. [65,66] proposed a real-time scheduling strategies based upon the earliest deadline metric. While [67] uses an earliest deadline first strategy to schedule the periodical queries. [68] 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.

In contrast to these approaches, PR-Prune requires a system that seeks to schedule which tuples are processed. Namely, PR-Prune seeks to select the order in which tuples are processed and control the amount of CPU resources dedicated to processing tuples based upon their significance. Traditional operator scheduling approaches make coarse grained decisions on how to utilize the CPU resources available. PR-Prune requires fine grained decisions on how to utilize the CPU resources available.

Compared to the current scheduling research, PR-Prune requires a new scheduling metric. Namely, PR-Prune seeks to schedule operators to ensure that the most critical results progress the furthest along the query pipeline before less critical tuples. Beyond the current scheduling research which looks at determining which operator to run and for how long to improve different metrics, PR-Prune must also determine which tuples are allocated resources.

6. Conclusions

Our innovative preferential resource allocation optimization strategy, PR-Prune, efficiently locates online dynamic criteria that are central for the production of critical query results by pruning ineffective dynamic criteria and combining multiple criteria along the same pipeline. Our experimental study confirms that for applications where priority resource allocation matters and promising tuples exist, PR-Prune consistently increases the throughput of the most critical query results compared to the state-of-the-art approaches. In addition, our experimental study confirms that the optimization search time of PR-Prune is significantly lower than its closest competitor, namely, PR.

Acknowledgements

We thank GAANN and NSF grants: IIS-1018443, 0917017, 0414567, and 0551584 for financial support. I personally cannot thank Dr. Rundensteiner enough for her continued support of my research and academic career well beyond my graduation from Worcester Polytechnic Institute.

References

[1] D. Carney, U. Çetintemel, M. Cherniack, et al., Monitoring streams: a new class of data management applications, VLDB J. (2002) 215–226.
[2] K. Works, E.A. Rundensteiner, Utilizing Dynamic Precedence Criteria to Ensure the Production of Critical Results from High Volume Big Data Streams, Academy of Science and Engineering, USA, 2015.
[3] C.-C. Lin, et al., Wireless health care service system for elderly with dementia, IEEE Trans. Inf. Technol. Biomed. (2006) 696–704.
[4] H.G. Kang, D.F. Mahoney, H. Hoenig, V.A. Hirth, P. Bonato, I. Hajjar, L.A. Lipsitz, In situ monitoring of health in older adults: technologies and issues, J. Am. Geriatr. Soc. 58 (8) (2010) 1579–1586.
[5] R.R. Gainey, B.K. Payne, Understanding the experience of house arrest with electronic monitoring: an analysis of quantitative and qualitative data, Int. J. Offender Ther. Comp. Criminol. (2000) 84–96.
[6] A. Press, Officials lose track of 16,000 sex offenders after GPS fails, http://www.foxnews.com.
[7] K. Works, E. Rundensteiner, The proactive promotion engine, in: ICDE, Software Demonstration, 2011, pp. 1340–1343.
[8] K. Works, E.A. Rundensteiner, Preferential resource allocation in stream processing systems, Int. J. Coop. Inf. Syst. 23 (04) (2014) 1450006.
[9] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, VLDB J. (2006) 121–142.
[10] A.N. Wilschut, P.M.G. Apers, Dataflow query execution in a parallel main-memory environment, in: PDIS, 1991, pp. 68–77.
[11] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intell. Syst. 3 (1999) 54–62.
[12] J.A. Stankovic, R. Rajkumar, Real-time operating systems, Real-Time Syst. 28 (2–3) (2004) 237–253.
[13] L. Golab, M.T. Özsu, Update-pattern-aware modeling and processing of cont. queries, in: SIGMOD, 2005, pp. 658–669.
[14] M. Liu, M. Li, D. Golovnya, et al., Sequence pattern query processing over out-of-order event streams, in: ICDE, 2009, pp. 784–795.
[15] J. Li, D. Maier, K. Tufte, et al., No pane, no gain: efficient evaluation of sliding-window aggregates over data streams, SIGMOD Rec. 34 (2005) 39–44.
[16] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: VLDB, VLDB Endowment, 2002, pp. 346–357.
[17] S. Babu, R. Motwani, K. Munagala, et al., Adaptive ordering of pipelined stream filters, in: SIGMOD, 2004, pp. 407–418.
[18] R. Ramakrishnan, et al., Database Management Systems, McGraw-Hill, 2000.
[19] D. Abadi, D. Carney, U. Çetintemel, et al., Aurora: a data stream management system, in: SIGMOD, 2003, 666 pp.
[20] E.A. Rundensteiner, L. Ding, T. Sutherland, et al., CAPE: continuous query engine with heterogeneous-grained adaptivity, in: VLDB, 2004, pp. 1353–1356.
[21] Y. Finance, http://finance.yahoo.com/.
[22] V. Raman, B. Raman, J.M. Hellerstein, Online dynamic reordering for interactive data processing, Tech. rep., Berkeley, CA, USA, 1999.

[23] V. Raman, J.M. Hellerstein, Partial results for online query processing, in: SIGMOD, 2002, pp. 275–286.
[24] J. Jacobi, A. Bolles, M. Grawunder, et al., A physical operator algebra for prioritized elements in data streams, Comput. Sci. Res. Dev. 25 (2010) 235–246.
[25] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-*k* query processing techniques in relational database systems, ACM Comput. Surv. (2008) 1–58.
[26] K. Mouratidis, et al., Continuous monitoring of top-*k* queries over sliding windows, in: SIGMOD, 2006, pp. 635–646.
[27] J. Teevan, et al., Discovering and using groups to improve personalized search, in: WSDM, 2009, pp. 15–24.
[28] J. Chomicki, Semantic optimization techniques for preference queries, Inf. Syst. (2007) 670–684.
[29] A.L. Thao, N. Pham, Panos K. Chrysanthis, Self-managing load shedding for data stream management systems, in: IEEE International Conference on Data Engineering Workshops, 2013, pp. 1–7.
[30] C. Basaran, K.-D. Kang, Y. Zhou, M.H. Suzer, Adaptive load shedding via fuzzy control in data stream management systems, in: IEEE International Conference on Service-Oriented Computing and Applications, 2012, pp. 1–8.
[31] S. Senthamilarasu, M. Hemalatha, Load shedding techniques based on windows in data stream systems, in: International Conference on Emerging Trends in Science, Engineering and Technology, 2012, pp. 68–73.
[32] N. Tatbul, U. Çetintemel, S. Zdonik, Staying fit: efficient load shedding techniques for distributed stream processing, in: International Conference on Very Large Data Bases, 2007, pp. 159–170.
[33] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in: VLDB, 2003, pp. 309–320.
[34] B. Babcock, et al., Load shedding for aggregation queries over data streams, in: ICDE, 2004, p. 350.
[35] F. Reiss, J. Hellerstein, Data triage: an adaptive architecture for load shedding in telegraphCQ, in: ICDE, 2005, pp. 155–156.
[36] T. Urhan, M. Franklin, XJoin: a reactively scheduled pipelined join operator, IEEE Data Eng. Bull. 23 (2) (2000) 27–33.
[37] M. Mokbel, M. Lu, W. Aref, Hash-merge join: a non-blocking join algorithm for producing fast and early join results, in: 20th International Conference on Data Engineering, 2004. Proceedings, 2004, pp. 251–262.
[38] L. Ding, E.A. Rundensteiner, G.T. Heineman, Mjoin: a metadata-aware stream join operator, in: DEBS, 2003, pp. 1–8.
[39] B. Liu, Y. Zhu, E. Rundensteiner, Run-time operator state spilling for memory intensive long-running queries, in: SIGMOD, 2006, pp. 347–358.
[40] Y. Chi, et al., Loadstar: a load shedding scheme for classifying data streams, in: SIAM Conf. on Data Mining, 2005.
[41] R. Fernández-Moctezuma, K. Tufte, J. Li, Inter-operator feedback in data stream management systems via punctuation, in: CIDR, 2009.
[42] B. Gedik, K.-L. Wu, P.S. Yu, L. Liu, Grubjoin: an adaptive, multi-way, windowed stream join with time correlation-aware CPU load shedding, IEEE Trans. Knowl. Data Eng. (2007) 1363–1380.
[43] B. Gedik, et al., Adaptive load shedding for windowed stream joins, in: CIKM, 2005, pp. 171–178.
[44] B. Babcock, S. Chaudhuri, Towards a robust query optimizer: a principled and practical approach, in: SIGMOD, 2005, pp. 119–130.
[45] V. Raman, A. Deshpande, J. Hellerstein, Using state modules for adaptive query processing, in: ICDE, 2003, pp. 353–365.
[46] T. Urhan, M.J. Franklin, Dynamic pipeline scheduling for improving interactive query performance, in: International Conference on Very Large Data Bases, 2001, pp. 501–510.
[47] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, M. Stonebraker, Operator scheduling in a data stream manager, in: VLDB, 2003, pp. 838–849.
[48] M.A. Sharaf, R. Labrinidis, P.K. Chrysanthis, K. Pruhs, Freshness-aware scheduling of continuous queries in the dynamic web, in: Proc. Int. Workshop on the Web and Databases (WebDB), 2005, pp. 73–78.
[49] Z. Ou, G. Yu, Y. Yu, S. Wu, X. Yang, Q. Deng, Tick scheduling: a deadline based optimal task scheduling approach for real-time data stream systems, in: W. Fan, Z. Wu, J. Yang (Eds.), Advances in Web-Age Information Management, in: Lecture Notes in Computer Science, vol. 3739, Springer, Berlin, Heidelberg, 2005, pp. 725–730.
[50] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, VLDB J. 12 (2003) 120–139.
[51] Y. Zhou, J. Wu, A.K. Leghari, Multi-query scheduling for time-critical data stream applications, in: International Conference on Scientific and Statistical Database Management, 2013, p. 15.
[52] Z. Falt, J. Yaghob, Task scheduling in data stream processing, in: International Workshop on Databases, Texts, Specifications, and Objects, 2011, pp. 85–96.
[53] S. Qian, Y. Lu, A modified chain scheduling algorithm in data stream system, in: IEEE International Conference on Computer and Automation Engineering, vol. 4, 2010, pp. 568–570.
[54] H. Deng, Y. Liu, Y. Xiao, The golden mean operator scheduling strategy in data stream systems, 2007, pp. 186–191.
[55] M.A. Hammad, M.J. Franklin, W.G. Aref, A.K. Elmagarmid, Scheduling for shared window joins over data streams, in: VLDB, 2003, pp. 297–308.
[56] M.A. Sharaf, P.K. Chrysanthis, A. Labrinidis, K. Pruhs, Efficient scheduling of heterogeneous continuous queries, in: VLDB, 2006, pp. 511–522.
[57] J. Wu, K.-L. Tan, Y. Zhou, QoS-oriented multi-query scheduling over data streams, in: DASFAA, 2009, pp. 215–229.
[58] Y. Tao, M.L. Yiu, D. Papadias, M. Hadjieleftheriou, N. Mamoulis, RPJ: producing fast join results on streams through rate-based optimization, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, ACM, 2005, pp. 371–382.
[59] B. Babcock, S. Babu, R. Motwani, M. Datar, Chain: operator scheduling for memory minimization in data stream systems, in: SIGMOD, 2003, pp. 253–264.
[60] B. Babcock, S. Babu, M. Datar, R. Motwani, D. Thomas, Operator scheduling in data stream systems, VLDB J. 13 (2004) 333–353.
[61] S. Wang, C. Gupta, A. Mehta, Vpipe: virtual pipelining for scheduling of dag stream query plans, in: W. Aalst, J. Mylopoulos, M. Rosemann, M.J. Shaw, C. Szyperski, M. Castellanos, U. Dayal, R.J. Miller (Eds.), Enabling Real-Time Business Intelligence, in: Lecture Notes in Business Information Processing, vol. 41, Springer, Berlin, Heidelberg, 2010, pp. 32–49.
[62] S. Mohammadi, A.A. Safaei, F. Abdi, M.S. Haghjoo, Adaptive data stream management system using learning automata, Comput. Res. Repos. J., arXiv:abs/1110.1700.
[63] M. Ghalambor, A. Safaeei, M. Azgomi, DSMS scheduling regarding complex QoS metrics, in: AICCSA, 2009, pp. 587–594.
[64] T. Sutherland, B. Pielech, Y. Zhu, L. Ding, E.A. Rundensteiner, An adaptive multi-objective scheduling selection framework for continuous query processing, in: International Database Engineering and Applications Symposium, 2005, pp. 445–454.
[65] L. Ma, X. Li, Y. Wang, H. Wang, Real-time scheduling for continuous queries with deadlines, in: SAC, 2009, pp. 1516–1517.
[66] X. Li, Z. Jia, L. Ma, R. Zhang, H. Wang, Earliest deadline scheduling for continuous queries over data streams, in: International Conference on Embedded Software and Systems, 2009, pp. 57–64.
[67] Y. Wei, S. Son, J. Stankovic, RTSTREAM: real-time query processing for data streams, 2006, pp. 141–150.
[68] S. Schmidt, T. Legler, D. Schaller, et al., Real-time scheduling for data stream management systems, Real-Time Syst. (2005) 167–176.