Reliable Aggregation over Prioritized Data Streams

Karen Works[1](✉) and Elke A. Rundensteiner[2]

[1] Westfield State University, Westfield, MA, U.S.A.
kworks@westfield.ma.edu
[2] Worcester Polytechnic Institute, Worcester, MA, U.S.A.
rundenst@cs.wpi.edu

Abstract. Under limited resources, targeted prioritized data stream systems (*TP*) adjust the processing order of tuples to produce the most significant results first. In TP, an aggregation operator may not receive all tuples within an aggregation group. Typically, the aggregation operator is unaware of how many and which tuples are missing. As a consequence, computed averages over these streams could be skewed, invalid, and worse yet totally misleading. Such inaccurate results are unacceptable for many applications. *TP-Ag* is a novel aggregate operator for TP that produces reliable average calculations for normally distributed data under adverse conditions. It determines at run-time which results to produce and which subgroups in the aggregate population are used to generate each result. A carefully designed application of Cochran's sample size methodology is used to measure the reliability of results. Each result is annotated with which subgroups were used in its production. Our experimental findings substantiate that TP-Ag increases the reliability of average calculations compared to the state-of-the-art approaches for TP systems (up to 91% more accurate results).

Keywords: Data Streaming · Aggregation · Prioritized Resource Allocation

1 Introduction

1.1 Targeted Prioritized Data Stream Systems (*TP*)

Data stream systems process streams of tuples to answer continuous queries. When CPU resources are limited, targeted prioritized data stream systems (*TP*) cannot always process all incoming tuples [6] as motivated below by several application examples. Yet in spite of such overloads, many DSMS must ensure the production of results from certain critical objects. To address these contradicting requirements, TPs utilize application-specific preference criteria to determine

This work is supported by GAANN and NSF grants: IIS-1018443 & 0917017 & 0414567 & 0551584 (equipment grant).
This work started during Karen's Ph.D. study at WPI.

which tuples should be allocated resources ahead of other tuples throughout the query pipeline [2,9,26,36,42–44].

The state-of-the-art TP, Proactive Promotion [43,44], processes more significant tuples ahead of less significant ones throughout the query pipeline. It is a tuple level scheduling approach. Other TP systems use workload reduction approaches, i.e., shedding [2,9,36] and spilling [26,42]. These methodologies remove less significant tuples at the incoming streams. The tuples not removed are processed in FIFO order.

As shown below, the selection of which tuples are processed in TPs is contrary to the production of reliable average calculations. However, in some systems there can be a need to both ensure the production of results from certain critical objects and to generate reliable average results.

1.2 Motivating Examples of TPs

Outpatient Health Care: TP systems track people with dementia [24]. It is critical to monitor people located at improper locations (i.e., likely lost). While monitoring people who live on their own (i.e., need help) may be reduced based on whether or not resources remain after processing people likely to be lost. Until enough resources are available, monitoring any other people could be temporarily skipped. These systems are known to experience data overloads [19].
Military: TP systems track missiles [21]. It may be critical to ensure that each and every object of a certain class is guaranteed to be monitored (e.g., nuclear missiles). While the monitoring of other objects (e.g., missiles bound for unpopulated areas) may be reduced based on whether or not processing resources remain after processing more significant objects. In addition monitoring of certain objects could in the worst case be temporarily skipped altogether (e.g., missiles sent by our country) until all other objects can be processed within their response time.
Law Enforcement: TP systems monitor prisoners assigned to home arrest [13]. They also get overloaded. In October 2010, an application monitoring released sex offenders across 49 states shut down for 12 hours [32]. With the highest level of urgency, violent prisoners (i.e., may cause harm) must be monitored. Next, prisoners at an improper location (i.e., likely to be in violation) shall be monitored. Finally, if resources permit, prisoners known to be flight risks ought to be monitored.

1.3 Running TP Example: Stock Market

TP systems monitor stocks online [20]. Such applications can get overloaded. In 2012, the London Stock Exchange shut down after a rash of computer-generated orders overwhelmed the system [30].

Consider a data stream application that monitors the average price of stocks by their business sector that appear in recent news and blogs (*Q1* below). Results are formed when news tuples join with stock tuples based upon their business sector (op1). Then these join results are joined with blog tuples based upon their business sector (op2). Finally, the average price for every business sector of these join results are created (op3).

```
(Stock Market Query)                                              /*Operators*/
Q1:SELECT AVG(S.price)                                            /*op3*/
   FROM Stock as S, News as N, StreetResearch as SR
   WHERE contains(S.sector, News[10 min])                         /*op1*/
   AND contains(S.sector, StreetResearch[15 min])                 /*op2*/
   Group by S.sector
   WINDOW 30 sec;
```

Mutual fund companies often invest in diverse stock portfolios. It is critical to ensure that every tuple of a certain class (e.g., their aggressive investments) is processed. While the processing of other tuples (e.g., their conservative investments) may be reduced based on what resources remain after processing the more significant tuples. Until there are enough resources to process all important tuples, monitoring of certain tuples can be temporarily skipped (e.g., stocks under evaluation). When the Stock Market Application is extremely overloaded, the scarce CPU resources will be dedicated only to the tuples most critical for the application, namely, tuples from aggressive investments.

1.4 Running Example: Inaccurate Aggregation Results

Consider the Stock Market Aggregation Example in Figure 1. Tuples from the stock, news and street research streams are respectively depicted by circles, squares, and triangles. The significance of each tuple is represented by its color. Black, grey, and white are respectively the most (i.e., level 1), the average (i.e., level 2), and least (i.e., NA) significance levels. In Figure 1 the system is overloaded. No tuples at level NA arrive at the aggregate operator op_3. The state of aggregate groups g_1 and g_4 thus only contain data from tuples at levels 1 and 2.

This may cause the average price per business sector produced by aggregate operator op_3 in query Q1 to be skewed. Clearly, under limited resources some tuples may not be used to create the aggregate result. Rather in this case, the aggregate result will be generated only from those most significant tuples that reach operator op_3. Unfortunately, this result may not match the aggregate result that would have been produced if all tuples had been processed by the aggregate operator. This makes the result unreliable and worst yet misleading. In our particular example, a broker would thus not know which subset of tuples was used to compute each aggregate value. Without such knowledge, a broker may make poor trades, which may lead to investors losing money.

At the very least we would want to inform application users about which tuples were used to create the result. Consider the example above. The aggregate result should be annotated with the fact that it is formed from aggressive investments only (i.e., a particular subgroup of the overall population) instead of from all investments. However, sometimes there may not even be enough

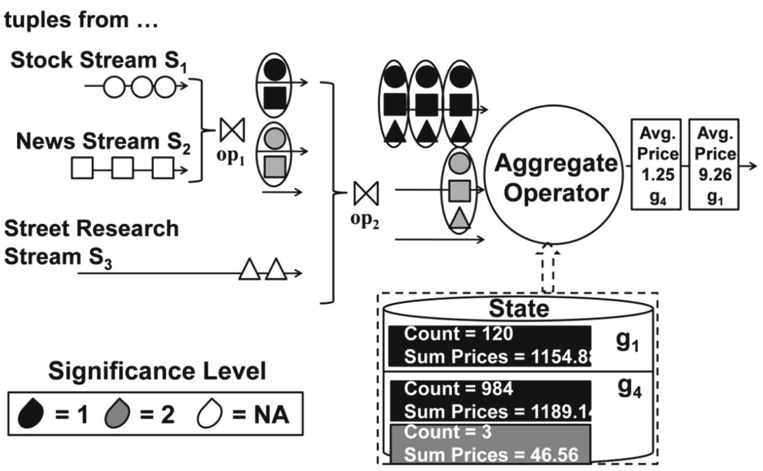


Fig. 1. Stock Market Aggregation Example

resources to process all tuples from a subgroup. That is, the subgroup may be incomplete. Aggregate results produced from such subgroups may again risk to be misleading and faulty. Thus, an aggregate operator must be designed to only generate results from subgroups that reliably represents their actual population.

As shown below, state-of-the-art aggregation methods for TPs [4,39] do not tackle this problem. This critical problem of reliable average calculation from incomplete populations is now the focus of our work.

1.5 State-of-the-Art Aggregation Operators and Their Shortcomings

The state-of-the-art aggregation methodologies [4,39] designed to produce reliable results in TPs address the problem by trying to solve the issue of system overload. They adapt the selection of which tuples are processed and which are not. [39] only processes tuples from windows where the population is guaranteed to be complete. Tuples from other windows are not processed. [4] adapts how many tuples are randomly dropped to ensure that all aggregate results produced are bound by a given error rate. These approaches adapt which tuples are allocated resources. By changing how resources are allocated, they may not abide by the user defined resource allocation preferences. This adaption assumes that accuracy is more important than the user-defined resource allocation preferences which may not always be the case.

Olston et al. [31] proposed a system that creates an aggregate result from cached results and the actual data set with the goal of creating an aggregate result within a high confidence interval range as quickly as possible. They propose a methodology to determine which results to cache that considers the trade off between precision and performance. Their algorithm delivers an answer that is bound by a specified precision constraint. They do not address that the data may be skewed by the set of tuples that are selected to be processed by the TP. Their approach assumes that all tuples within a population are given equal chance at reaching the aggregate operator.

It is challenging to construct reliable aggregate results from only the tuples that arrive at the operator. Consider the stream aggregate operator op_3 [10] in a TP under duress that implements the Stock Market query Q1 above (Fig. 1). The average price for business sector group g_4 could be produced from the tuples at levels 1 and 2. There are only 3 tuples at level 2. Such few tuples may not be representative of the actual population of tuples at level 2. TP requires an aggregate operator than can selectively control which subgroups are used to generate each result. This raises the question of how to determine which subgroups best represent their actual populations.

1.6 Our Approach and Contributions

We propose the TP aggregate operator *TP-Ag* to address the issues described above. TP-Ag produces reliable aggregate average results from incomplete subgroups. It uses an efficient estimation model to determine how representative each subgroup is of their actual population. In addition, TP-Ag annotates each aggregate result with the subgroup population(s) that the result is generated from.

The design of an aggregate operator that meets the requirements above is challenging. TPs pulls specific significant tuples ahead of other tuples. Some TPs only identify significant tuples upon arrival at the data stream system [2,9,26,36,42], while others can identify significant tuples at operators further down the query pipeline [43,44]. This makes it more complicated to determine which tuples from specific populations never reach the aggregate operator. It is equally challenging to estimate what contribution these tuples would have had on any aggregate result produced.

Our Contributions Include

- We formulate the TP aggregate operator problem of generating reliable annotated aggregate results from incomplete populations.
- We propose a carefully designed estimation model and application of Cochran's sample size methodology to measure if any subset of the actual population is large enough to generate a reliable aggregate result. This requires knowledge of how many tuples fail to reach the aggregate operator due to limited resources. We propose a method to track this information.
- We carefully outline the logical design of our novel TP-Ag operator, namely, how TP-Ag generates reliable aggregate results by selectively controlling which subsets in the aggregate group population are used to generate each result.

- We carefully outline the physical design of our novel TP-Ag operator. The data structures and infrastructure behind our TP-Ag operator are designed to allow efficient retrieval of subgroups within an aggregate population.
- Our experiments show that TP-Ag produces up to 90% more accurate aggregate results compared to the state-of-the-art methodologies on a wide variety of data sets and workloads.

2 TR Query Model and Plan Definitions

2.1 TP Queries

A query q_j for a TP is a CQL query [3] extended to allow the specification of multi-tiered monitoring criteria to identify significant tuples in TP systems. Below is the *Stock Market Query Q1* with the extension to support TPs.

```
(Stock Market Query with Extension)                                      /*Operators*/
Q1:SELECT AVG(S.price)                                                   /*op3*/
   FROM Stock as S, News as N, StreetResearch as SR
   WHERE contains(S.sector, News[10 min])                                /*op1*/
   AND contains(S.sector, StreetResearch[15 min])                        /*op2*/
   Group by S.sector
   WINDOW 30 sec;     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)
```

Monitoring levels $q_j.ML$ are predicates defined at compile-time that state the user's desired result production order. When resources are scarce, the monitoring levels are used to identify which tuples will be processed (i.e., not shed). Each monitoring level m_k states m_k's *rank* $m_k.rnk$ and *membership criteria* $m_k.mem$. The rank $m_k.rnk$ denotes how significant monitoring level m_k is compared to the other monitoring levels. When resources are limited, TP optimizers [2,9,26,36,42–44] identify which types of tuples to process based upon the monitoring levels. These tuple are called *significant tuple*s.

The *query window* denotes the finite set of tuples from stream s_n used to create results. *Window bounds* are specified as time (e.g., within a 30 second time span) or count (e.g., last 30 tuples) ranges. In *sliding windows*, tuples from consecutive windows may overlap. While we assume time-based sliding windows, our proposed techniques equally apply to count-based window semantics.

2.2 TP Query Plans

A TP query plan is a directed acyclic graph composed of *TP query operator*s as nodes and *data exchange interface*s that transfer tuples between operators as edges. The *data exchange interface* transfers tuples between operators.

TP operators include enhanced traditional operators and significance classifiers. Traditional operators from the continuous query algebra [14] are enhanced to allocate resources to incoming tuples in significance order and to propagate significance properties assigned to tuples. Significance classifiers (or *SC*s) are special-purpose operators that compute and assign significance properties to tuples [43,44].

3 Background of Stream Aggregation

3.1 Basic Aggregate Operator

The *Basic Aggregate Operator* [8] computes a function over the set of tuples that belong to the same aggregate group within the current query window w_p of the stream. In Stock Market Query Q1, the aggregate group is a business sector. Incoming tuples are stored in the state and associated with their aggregate group. When it is determined that no future incoming tuples for window w_p will arrive then the aggregate result(s) are generated for each aggregate group in w_p.

3.2 Aggregate Operator Supporting Out-of-Order Data Streams

Some TPs [43,44] cause tuples to become out-of-order. TP aggregate operators require special support to know when all tuples from a window have been processed to produce their results. An aggregate operator should produce results for window w_p when no tuples from window w_p remain to be processed. Prior out-of-order work [23] proposed to use punctuations to trigger the creation of aggregate results whenever a window is complete. To quickly identify tuples within a given window, windows are divided into groups of tuples (a.k.a. *panes*) [22]. Each pane p_q is assigned a pane number. When a tuple arrives, it is associated with a pane number which determines the windows the tuple will produce results for. Each aggregate result is only generated from tuples whose panes compose its query window. Each operator contains a punctuation queue that stores notifications described below.

To discern when no tuples from pane p_q remain to be processed, the progress of tuples is tracked. Each leaf operator op_o periodically sends a punctuation into when it has processed all tuples from pane p_q. To send a punctuation requires simply placing the punctuation into the punctuation queue of the next down stream operator. Upon receiving such a notification, the aggregate operator produces all results for the window that ends with pane p_q. The Out-of-Order Aggregate Operator assumes that tuples are processed. It can produce unreliable results.

3.3 State-Of-the-Art Aggregate Operator for TPs

The *State-Of-the-Art TP Aggregate Operator* (Sec. 1.5) also produces unreliable results. The operator contains a punctuation queue that stores notifications (See above) and incoming queues that store tuples by significance level (Sec. 2). It works as follows.

Producing Aggregate results: Aggregate results are created for the first notification in the punctuation queue and are sent to the next operator (line 4). Finally, tuples stored in the state within panes that will not produce any future results are purged (line 5). This continues until either no resources or incoming punctuations remain (line 1).
Processing tuples: If resources remain then starting from the most significant incoming queue tuple t_i is stored and associated with the proper aggregate group and pane (line 11). After all tuples from one queue have been processed, tuples in the next significant incoming queue are processed (lines 13-16). This continues until either no resources remain or all queues are empty (line 8).

```
Algorithm State-Of-the-Art TP Aggregate Operator(
        Qp        /pane complete punct. queue/,
        Cavail    /avail. res./,
        Qinc(s1) /queues for stream s1/)
        Cavail    /avail. res./,
        Qinc(s1) /queues for stream s1/)
1:  WHILE ((Cavail > 0) and (Qp is not empty))
2:    Punc = first pane complete punctuation in Qp
3:    create aggregate results that for window that ends with pane in Punc
4:    place aggregate results into input queue of next query plan operator
5:    purge state of tuples within panes that will not produce any future results
6:  ENDWHILE
7:  LevelProcessed= 1
8:  WHILE((Cavail > 0) and (Qinc(s_1) is not empty))
9:    IF(Qinc(s_1,LevelProcessed) contains a tuple)
10:    Tup = first tuple in Qinc(s_1,LevelProcessed)
11:    store Tup in state and associate with the aggregate group and pane
12:  ELSE
13:    LevelProcessed = LevelProcessed + 1
14:    IF (LevelProcessed > max(Monitoring Level))
15:       LevelProcessed= Insignificant Tuple
16:    ENDIF
17:  ENDIF
18: ENDWHILE
```

4 TP-Ag Problem Definition

Our TP-Ag operator must meet the following requirements.

1. It must produce the most reliable aggregate result from the tuples that arrived at the aggregate operator and belong to selective subgroup populations. The set of tuples used to create each result must be estimated to represent the actual selected subgroup populations of tuples that would have arrived at the aggregate operator if resources were available.
2. It must annotate each aggregate result with the subgroup population(s) that the result is generated from.
3. It cannot ignore the desired resource allocation order specified by the user and adjust which tuples the TP optimizer chose to process (Sec. 2). It must build reliable aggregate results from the significant tuples already pulled forward. Data streams are often shared to efficiently produce more than one output. Adjusting the desired resource allocation order specified by the user assumes that the aggregate result is the most important query result produced (which may not be the case).

5 TP Aggregation Foundation

5.1 Aggregate Result Annotation

Per the requirements (Sec. 4), each aggregate result must be annotated with which tuples it is generated from. In TP, tuples chosen to be processed satisfy the membership criteria of an activated monitoring level (Sec. 2). Thus we propose to logically divide the population of tuples within an aggregate group and pane into subsets based upon their significance levels. Each population is divided into subsets, namely, one subset for each significance level and one for insignificant tuples.

Each aggregate result a_i is associated with a population generation subset flag or ss_l that annotates which tuples the aggregate result a_i is generated from. ss_l is represented as a bit vector with one bit for each monitoring level lvl_1, lvl_2, ..., lvl_n and one bit for insignificant tuples. For instance, subsets flag $ss_1 = 100$ signifies that the aggregate result a_i is generated from only tuples at significance level 1. While subsets flag $ss_2 = 110$ signifies that the aggregate result a_i is generated from only tuples at significance levels 1 and 2. Finally, subsets flag $ss_3 = 111$ signifies that the aggregate result a_i is generated from all tuples (both significant and insignificant).

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

5.2 Evaluation Strategies for Sample Population

Result Accuracy: The actual population pop_m is the set of tuples in the aggregate group g_l and window w_k that given adequate resources would have reached aggregate operator op_o. Aggregate answer a_i^* is generated from an actual population pop_m. Aggregate result a_i is generated from a sample population $spop_m$, i.e., the set of tuples that reached aggregate operator op_o. The sample population is a subset of the actual population. Aggregate result a_i may be incorrect, i.e., not match the aggregate answer a_i^*. Reconsider Figure 1. A sample population $spop_1$ for aggregate group g_4 includes 987 tuples where 984 tuples have significance level 1 and 3 tuples have significance level 2.

Determining Sample Population Accuracy: One method to determine if a sample population accurately portrays the actual population is to compare the mean of the sample and actual population via the Hoeffding equation [17]. Many aggregation operators [16,23,31] that process aggregate results from most (if not all) tuples in a given query window (Sec. 8) use this approach. TP-Ag seeks to build reliable aggregate results solely from the tuples processed by controlling which of the available subgroup(s) are used to create an aggregate result. However, TP-Ag cannot use the Hoeffding equation to determine the accuracy of the sample population. The Hoeffding equation requires that the actual mean μ be precisely measured. TP cannot ensure that all tuples reach TP-Ag op_o. Thus to calculate the actual mean μ requires knowledge of which tuples could have reached TP-Ag op_o but did not. Such an evaluation amounts to running the full query. Clearly, this is prohibitively costly.

Thus lead us to look at other statistics methods. One method to determine if a sample population accurately portrays the actual population is to estimate the sample size required to determine the actual mean within a given error threshold via Cochran's sample size formula [7]. If the size of sample population, denoted by $|spop_m|$, is less than the estimated required sample size $|spop^{est}|$, then the sample population $spop_m$ may not accurately represent the actual population pop_m. No aggregate results should be generated from $spop_m$. TP-Ag can use Cochran's sample size formula [7]. It determines the sample size by considering the limits of the errors in the mean values of items in the sample population. $|pop_m|$ is the size of the actual population. ϵ is the user selected error rate. σ is the standard deviation of the actual population. z is the user selected confidence level or the estimated percentage of the values in the sample population within two standard deviations of the mean of the actual population. The Cochran's sample size formula [7] is $|spop^{est}| = (z^2 * \sigma^2 * (|pop_m|/(|pop_m| - 1)))/(\epsilon^2 + ((z^2 * (\sigma^2)/(|pop_m| - 1))))$. Roughly, $z^2 * \sigma^2 * (|pop_m|/(|pop_m| - 1)))$ represents the percentage of tuples from the sample population that are within the confidence interval of the estimated mean. $(\epsilon^2 + ((z^2 * (\sigma^2)/(|pop_m| - 1))))$ represents the percentage of tuples from the sample population that per the error rate must be within the confidence interval of the estimated mean.

This approach requires that the aggregate values in the population follow a normal distribution. While not all data has this distribution, many practical streams do. One example is stock market prices which can be mapped to the normal distribution [11].

We must calculate the standard deviation of the actual population σ and the size of the actual population $|pop_m|$. As common practice, we propose to calculate the standard deviation of the actual population σ using the standard deviation of the sample population and Bessel's correction [18]. To estimate the actual population size $|pop_m^{est}|$, we must measure *the estimated number of expired tuple* or how many tuples would have reached TP-Ag operator op_i if given adequate resources. To calculate the estimated number of expired tuple the number of tuples that expire at each operator in the query path before aggregate operator op_o (i.e., $|exp(op, g_l, lvl_p)|$) and the probability of expired tuples reaching aggregate operator op_o (i.e., $P(op, op_o, lvl_p)$) is tracked (Sec. 6). The estimated number of expired tuple is the product of these two values (i.e., $P(op, op_o, lvl_p) * |exp(op, g_l, lvl_p)|$).

5.3 Policy for Selecting the Sample Population

There are many ways of selecting which subgroup(s) are used to generate a result. We propose to include the largest number of reliable subgroups in consecutive significance order in the sample population. For example, aggregate result a_i could be created from only tuples at significance level 1, at significance levels 1 or 2, or all levels.

Reconsider the aggregate operator op_3 (Fig. 1). An aggregate result for aggregate group g_4 could be created from the two subgroups, namely, the 984 tuples at significance level 1 and 3 tuples at significance level 2. Assume that the estimated actual population size $|pop_m^{est}|$ for this sample population is 1984 tuples. The estimated standard deviation σ is 7.9. The error rate ϵ is .05. The critical standard score z is 1.96 (95% confidence level). Then the estimated required sample size $|spop^{est}|$ (i.e., $= (1.96^2 * 7.9^2 * (1984/(1984-1)))/(.05^2 + ((1.96^2 * (7.9^2)/(1984-1)))$) is 1943 tuples. This sample population is not large enough to create a reliable aggregate result.

However, an aggregate result could be created for group g_4 that only using the 984 tuples at significance level 1. Assume that the estimated actual population size $|pop_m^{est}|$ for this sample population is 1000 tuples. The estimated standard deviation σ is 5.9. The error rate ϵ is .05. The critical standard score z is 1.96 (95% confidence level). Then the estimated required sample size $|spop^{est}|$ (i.e., $= (1.96^2 * 5.9^2 * (1000/(1000-1)))/(.05^2 + ((1.96^2 * (5.9^2)/(1000-1)))$) is 981 tuples. This sample population is large enough to produce a reliable aggregate result.

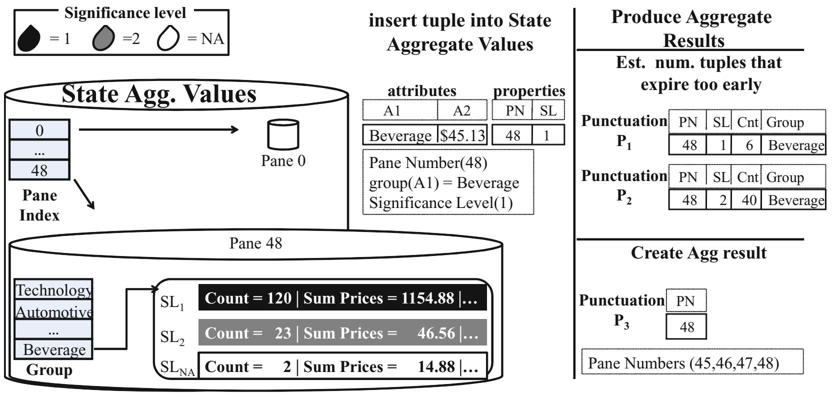


Fig. 2. TP-Ag State Example

6 Design of the TP Aggregation Operator

6.1 Tracking Expired Tuples

We designed the traditional operators (Sec. 2) to periodically send an *expiration count punctuation* to the TP-Ag operator. The expiration count punctuation contains the number of expired tuples for each aggregate group and significance level. TP-Ag uses these values to estimate the actual population size (Sec. 5.2).

Operators track the number of tuples that expire too early over a window by aggregate group and significance level (Sec. 5.2) and send this count to the TR-Ag operator. TR-Ag will use this count to estimate the actual population size when deciding whether or not a sample population should produce a result. To achieve this, TR-Ag uses an *expiration count punctuation*. Each operator tracks the estimated number of tuples that expire too early by pane, aggregate group, and significance level. When no tuples in a pane remain to be processed by an operator, the operator sends an expiration punctuation for each aggregate group and significance level where tuples expired. Upon receiving an expiration punctuation, each operator adds on their estimated number of tuples that expire too early. The expiration punctuation moves along the pipeline until it reaches the TP-Ag (Sec. 6.2). When, the expiration punctuations reach the TP-Ag they contain the number of tuples that expire too early over a window for each aggregate group and significance level.

Considerexpirationpunctuations$p_1 < 48, 1, 6, Beverage >andp_2 < 48, 2, 40, Beverage >$ in Figure 2. Expiration punctuation p_1 states that 6 tuples from pane 48, significance level 1, and group Beverage are estimated to have expired too early. While expiration punctuation p_2 states that 40 tuples from pane 48, significance level 2, and group Beverage are estimated to have expired too early.

6.2 TP-Ag Physical Design

To support the production of aggregate results from certain subset(s) of the sample populations, TP-Ag must support the efficient look-up and purging of stored aggregate variables by pane (Sec. 3), group, and significance level. The number of tuples within each pane is limited. Thus tuples maintained by the operator are first grouped by the panes they belong to. Next, tuples are indexed by their aggregate group. Lastly, tuples are stored by their significance level.

To support the production of aggregate results from certain subset(s) of the sample populations, TP-Ag must support the efficient look-up and purging (Sec. 6.4) of stored aggregate variables by pane (Sec. 3), group, and significance level. **State Design:** TP does not always process tuples in arrival time order (Sec. 3). Thus, the state can contain tuples from more than one query window. That is, a multitude of tuples with the same aggregate group and significance level are likely to exist across multiple panes. However, the number of tuples within each pane is limited. Thus stored tuples are first grouped by the panes they belong to. Next, tuples are indexed by their aggregate group. Finally, tuples are stored by their significance level.

Consider the insertion of stock tuple $t_1 < Beverages, \$45.13 >$ with pane 48 and significance level 1 into state aggregate values (Fig. 2). Stock tuple t_1 is stored in the state for pane 48, business sector $Beverages$, and significance level 1.

6.3 TP-Ag Operator

The major difference between TP-Ag and the State-Of-the-Art TP Aggregate Operator is that TP-Ag upon receiving a notification punctuation (Sec. 3) determines the largest number of reliable subgroups in consecutive significance order in the sample population that can produce a result (if any). Consider the production of aggregate results triggered by punctuation $p_2 < 48 >$ (Fig. 2). p_2 signals that all tuples from pane 48 that arrived at the operators that reside in the query pipeline prior to the aggregate operator have been processed. First, TP-Ag locates the group attributes for pane 48 which are Technology, Automotive, ..., and Beverage. Then for each group attribute (e.g., Beverage), it creates a sample population from all tuples from the window (composed of 4 panes) that ends with pane 48 (i.e., pane $45-48$). If the size of the sample population is greater than or equal to the estimated required sample size then TP-Ag creates an aggregate result for tuples from this sample population. Otherwise, TP-Ag creates a new sample population by removing the least significant subset from the current sample population. The process of testing the current sample population and creating a new sample population continues until either a result is generated or the sample population is empty. In the latter case, no aggregate result will be produced. Then TP-Ag moves to the next group for pane 48 (and so on...).

It works as follows. First, the incoming punctuations are processed.
Processing Expiration Punctuations: Any incoming expiration punctuation is stored in the state (lines 3-6).
Producing Aggregate Results: Any incoming notification punctuation is processed as follows. First, the groups in the window pane are identified (line 7). Next, for each group, we test to see if the sample population for all tuples represents the actual populations (line 10). If so, results are created and sent to the next operator (lines 11-12). Otherwise, the sample population is reduced by tuples that belong to the least significant subset (line 14) and the test starts over (line 10). This continues until either a result is produced or all sample populations have been explored (line 8). Then, results for the next group are produced. This continues until either no resources remain or there are no more incoming punctuations (line 1). Finally, tuples stored in the state within panes that will not produce any future results are purged (line 18).
Processing tuples: This is the same as the processing tuples logic for State-Of-the-Art TP Aggregate Operator (Sec. 3) (lines 18-29)

```
Algorithm TP-Ag Operator( Qep      /exp. punct. queue/,
                          Qnp      /not. punct. queue/,
                          Cavail   /avail. res./,
                          Qinc(s_1) /queues for stream s1/)

1: WHILE((Cavail > 0) and (Qnp is not empty))
2:   Punc = first punctuation in Qnp
3:   WHILE((Qep is not empty) and (first punctuation in Qep = pane in Punc))
4:     ExpPunc = first punctuation in Qep
5:     store the values in ExpPunc
6:   ENDWHILE
7:   FOR each group gl in defined by the pane in Punc
8:      WHILE(no result has been produced) and (the sample population is not empty)
9:      sample population = population defined by the pane in Punc and group gl
10:      IF (sample population represents actual population)
11:        create aggregate results for sample population
12:        place aggregate results into input queue of next query plan operator
13:      ELSE
```

```
       reduce the sample population by the least significant subgroup
     ENDIF
    ENDWHILE
  ENDFOR
  purge state of tuples within panes that will not produce any future results
ENDWHILE
LevelProcessed= 1
WHILE((Cavail > 0) and (Qinc(s_1) is not empty))
 IF(Qinc(s_1,LevelProcessed) contains a tuple)
    Tup = first tuple in Qinc(s_1,LevelProcessed)
    store Tup in state and associate with the pane, aggregate group, and significance level
  ELSE
    LevelProcessed = LevelProcessed + 1
    IF (LevelProcessed > max(Monitoring Level))
      LevelProcessed = Insignificant tuples
    ENDIF
  ENDIF
ENDWHILE
```

6.4 Memory Resource Management

Beyond CPU resources, memory resources may also be limited.
State Management: To ensure complete results, tuples stored in states are not purged if they may create aggregate results in the future (Sec. 3). However, this purging method assumes that sufficient memory is available to store all tuples that will create future aggregate results. This may not always be the case. In the case of insufficient memory, we propose to purge tuples from the oldest panes in the state first. Our approach is based upon the fact that the majority of aggregate results generated from the oldest tuples would have already been produced. Memory resources are allocated to storing the freshest tuples.
Queue Management: In the case of insufficient memory, the incoming queues must also be purged. We also utilize the oldest pane method defined above to purge the queues.

7 Experimental Evaluation

7.1 Experimental Setup

Alternative Solutions. We compare *TP-Ag* (or *TP w/ TP-Ag*) to the state-of-the-art aggregate operators in TPs. That is, we compare to the out-of-order aggregate operator [23] implemented in the state-of-the-art TP tuple level scheduling approach (or *PP*) [43,44] and the stream aggregation operator [8] implemented the state-of-the-art TP workload reduction approaches, namely, semantic (or *sem*) and random (or *rand*) [2] (Sec. 8). PP requires the out-of-order aggregate operator (Sec. 3). We also compare TP w/ TP-Ag to state-of-the-art aggregate operators for TPs [4,39] that limit which tuples are dropped from specific windows (or *Shed Window Ag*). Finally, we compare to the traditional aggregate operator [8] implemented in a non-targeted prioritized data stream systems (or *trad*). TP-Ag uses the critical standard score $z = 1.96$ (95% confidence level). To ensure fairness, all systems are implemented in the same data stream system with appropriate extensions to implement the methods, in our case, CAPE [34].

TP w/ TP-Ag, PP, and sem use the same criteria to select the tuples processed. Rand randomly selects tuples to process in FIFO order based upon the estimated number of tuples that can be processed within their lifespan. Trad simply processes all tuples in FIFO order.

Data Streams and Query. Most experiments use Stock Market Query Q1 with the extension (Sec. 2) where the window size = 500 tuples.

The stock market stream was created from stock ticker information on the S&P 500 stocks gathered over July 18, 2012 via Yahoo Finance [12].

News and blog data streams were created by randomly selecting sectors from the global industry classification standard (GICS). GICS, developed by Morgan Stanley Capital International (MSCI) and Standard & Poorś, contains 10 sectors that categorize the S&P 500 stocks.

Data Set 1 (or *DS1*) mimics a financial company monitoring diversified mutual funds. That is, the stocks chosen are distributed across different business sectors and investment types (i.e., aggressive versus conservative investments). In DS1, 5% of the 500 stocks (or 25 stocks) were randomly selected to be at each of the three monitoring level (Sec. 2.2).

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

Metrics and Measurements. TP w/ TP-Ag produces an aggregate result from a subset of the tuples that arrive at the aggregate operator. The other approaches generate aggregate results from all tuples that arrive at the aggregate operator. To be able to validate whether or not the results are correct, each aggregate result produced is annotated with the significant levels of the tuples in the sample population. For each experiment, the actual aggregate answers (Sec. 5.2) for each possible sample population was found.

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

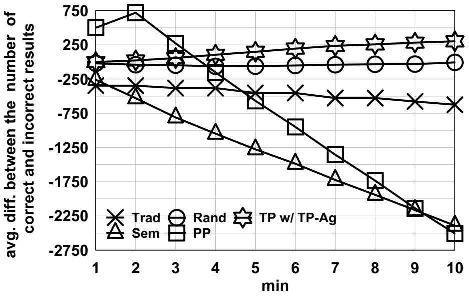
7.2 Experimental Methodology

We explore the following: 1) Is TP-Ag more effective at producing a larger percentage of correct significant aggregate results than the state-of-the-art solutions? 2) What effect does the number of significant tuples that belong to each aggregate group have on the effectiveness of the TP-Ag strategy compared to the state-of-the-art solutions? 3) How do changes in the error rate (Sec. 5.2) affect the percentage of correct results produced by TP-Ag? 4) What is TP-Ag's runtime CPU and memory overhead in the worst case scenario compared to the state-of-the-art solutions?

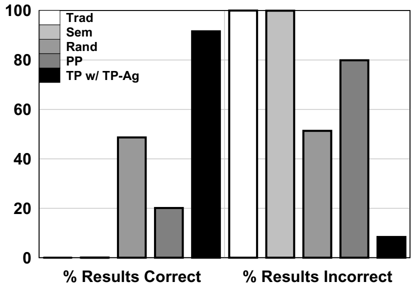
We vary the *number of significant tuples that belong to each aggregate group* and the *error rate* as they directly affect TP-Ag. When the *number of significant* *tuples that belong to each aggregate group* decreases, this reduces the number of tuples in each sample population. The *smaller the sample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

7.3 Experimental Findings

Effectiveness at Increasing the Percentage of Correct Aggregate Results Produced. First, we compare the percentage of correct aggregate results produced by each approach. This experiment uses DS1. Figure 3 a shows the average difference between the number of the correct and incorrect aggregate results produced at each minute. This measures whether more correct (positive number) or incorrect results were produced (negative number). Overall TP w/ TP-Ag compared to sem, rand, and trad consistently produces more correct aggregate results.



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



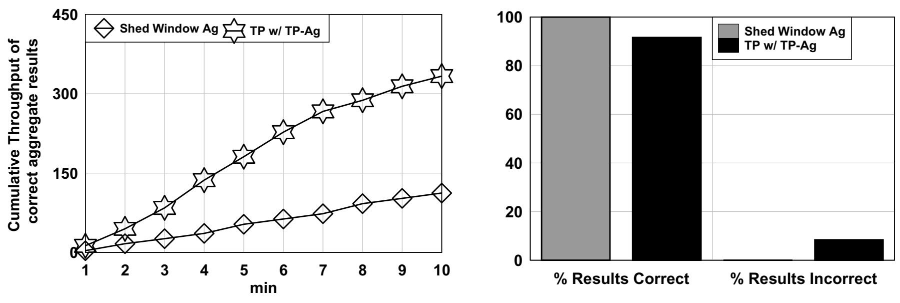
b) Average % Correct Significant Results Over 10 Min

Fig. 3. Effectiveness at Increasing the % of Correct Aggregate Results

PP produced more correct aggregate results than TP w/ TP-Ag at startup (minutes 1 through 3). However, after the system start-up (minutes 4 through 10) PP produced more incorrect than correct aggregate results. This is as expected. Namely, PP has less overhead than TP w/ TP-Ag. In addition, the aggregate results produced by PP will only be incorrect when the system is overloaded and many significant tuples fail to reach the aggregate operator.

As seen in Figure 3 b shows, compared to all alternative solutions, TP w/ TP-Ag produced a much higher percentage of correct aggregate results. Of all the aggregate results produced by TP w/ TP-Ag, 91.5% were correct. Our results support that TP w/ TP-Ag is effective at increasing the percentage of correct aggregate results produced compared to competitor solutions.

TP w/ TP-Ag Versus State-of-the-art Reliable Aggregation Operators for TPs. We now compare TP w/ TP-Ag to state-of-the-art aggregate operators designed to produce reliable results in TPs [4,4,39] (Sec. 1.5). These systems limit which tuples are dropped from specific windows. We refer to these systems as *Shed Window Ag*. First, we compare the percentage of correct aggregate results produced by each approach. This experiment also uses DS1.



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

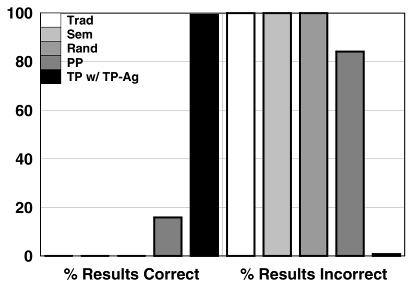
Fig. 4. TP w/ TP-Ag versus State-of-the-art

As the overall percentage of correct and incorrect significant results in Figure 4 b shows, all aggregate results produced by Shed Window Ag were correct. While, of the aggregate results produced by TP w/ TP-Ag produced 91.5% were correct. Clearly, Shed Window Ag will always produce correct aggregate results. Recall that Shed Window Ag will ensure that no tuples from specific windows are dropped or expire. As a result, Shed Window Ag will only produce correct aggregate results. In contrast, TP w/ TP-Ag seeks to produce results that are estimated to be correct from incomplete windows of tuples.

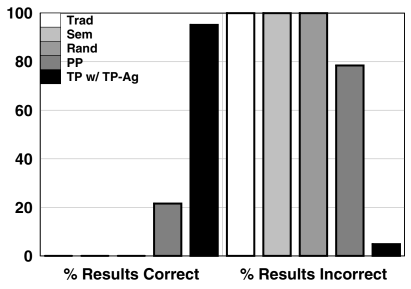
However, Shed Window Ag may not produce as many aggregate results as TP w/ TP-Ag. As Figure 4 a shows, TP w/ TP-Ag produced roughly 2.9 fold more correct aggregate results than Shed Window Ag. Shed Window Ag will process all tuples (both significant and insignificant) from selected windows. This requires a significant amount of CPU overhead. Hence, Shed Window Ag will not produce as many correct aggregate results as TP w/ TP-Ag.

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

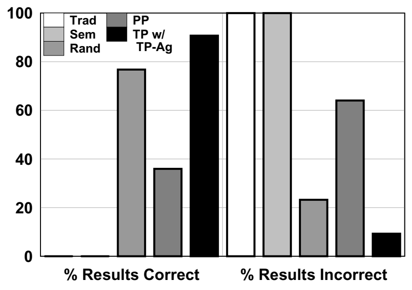
Varying the Sample Population Size. We now explore how the number of the significant tuples in each aggregate group affects TP-Ag. All significant tuples belong to two GICS sector groups. This experiment uses four Data Sets (i.e., DS25, DS50, DS75, and DS100). Each Data Set adapts the percentage of significant tuples that belong to the two GICS sector groups. In DS25, 25% of the stocks in the two sectors are significant (75% of these tuples are insignificant). Similarly, in DS50, DS75, and DS100, respectively 50%, 75% and 100% of the tuples in the two sectors are significant. The sample population size increases from DS25 to DS100.



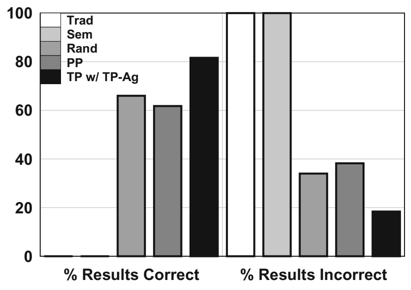
a) Data Set DS25



b) Data Set DS50



c) Data Set DS75



d) Data Set DS100

Fig. 5. Varying the Sample Population Size

Figures 5a-5d show the overall percentage of correct and incorrect aggregate results respectively for DS25, DS50, DS75, and DS100. As can be seen, compared to the alternative solutions, TP w/ TP-Ag produced the highest percentage of correct aggregate results. The closest competitors were rand and PP. In DS25 (the smallest sample populations), TP w/ TP-Ag produced 100% and 84.0% more correct aggregate results than rand and PP. In DS100 (the largest sample populations), TP w/ TP-Ag produced 19.1% and 24.2% more correct aggregate results than rand and PP. This is as expected. Namely, TP-Ag achieves the highest gains when the fewest tuples in the sample population fail to reach the aggregate operator. When the stream is saturated with significant tuples, more significant tuples are likely to fail to reach the aggregate operator.

Varying Error Rate. Now, we compare the percentage of correct aggregate results produced by TP w/ TP-Ag when the error rate (i.e., the desired level of precision ϵ of Cochran's sample size formula (Sec. 5.2)) varies. This experiment also uses DS1. We vary the error rate ϵ from 5%, 10%, to 20%. Figure 6 shows the percentage of correct and incorrect aggregate results produced.

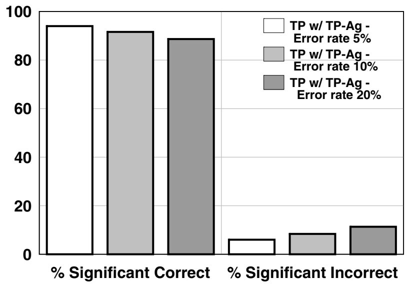


Fig. 6. Varying the error rate

Overall the highest percentage of correct aggregate results was produced when the error rate ϵ is 5%. While the lowest percentage was produced when the error rate ϵ was 20%. The percentage of correct aggregate results produced by TP w/ TP-Ag for the error rate ϵ from 5%, 10%, to 20% was respectively 93.9%, 91.5%, and 88.6%. As expected, decreasing the error rate (i.e., higher level of precision of Cochran's sample size formula) increases the percentage of correct aggregate results achieved by TP w/ TP-Ag (vice versa).

Execution-Runtime CPU Overhead. To measure the runtime overhead we evaluate the cumulative throughput using the worst case scenario for TP w/ TP-Ag. In the worst case scenario, there is adequate resources to process all tuples (Fig. 7 c). As a consequence, for each aggregate result is produced from a sample population that contains all tuples in a window and aggregate group. The overhead of TP systems is the cost to gather and evaluate runtime statistics. In addition, TP-Ag has the additional overhead of tracking statistics to estimate the actual population, evaluating the required sample size (Sec. 5.2), and determining if there is a sample population for each group and window whose size is comparable to the required sample size (Sec. 5.3). This experiment uses DS1.

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

Memory Overhead. To measure the memory overhead we evaluated the average number of tuples in the state and input queue of the aggregate operator using the worst case scenario outline above (Fig. 7 a & b). As our results demonstrate, the memory overhead of TP w/ TP-Ag is higher than the current state-of-the-art approaches. The state of the aggregate operators in trad, sem, rand, and PP

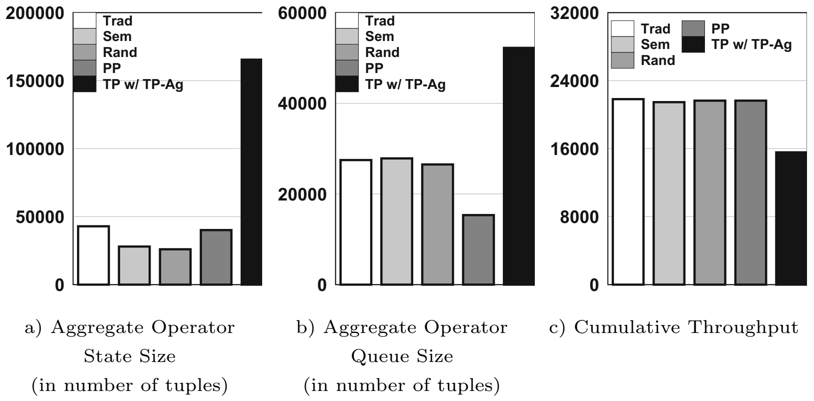


Fig. 7. Memory & Execution-Runtime CPU Overhead

respectively have 74.0%, 83.0%, 84.2%, and 75.7% less tuples in their states than TP w/ TP-Ag. While the queues of the aggregate operators in trad, sem, rand, and PP respectively have 47.4%, 46.6%, 49.2%, and 70.6% less tuples in their queues than TP w/ TP-Ag.

This is as expected. Namely, the TP-Ag design relies upon a memory-intensive physical design to support the production of results from subsets of the actual sample population. Again, TP w/TP-Ag is a great fit for systems that require a TP system and desire reliable accuracy in the aggregate results produced. Ensuring the production of reliable aggregate results however carries an overhead.

7.4 Summary of Experimental Findings

We now summarize our key findings.

- TP-Ag is effective at increasing the percentage of correct aggregate results produced in TPs (TP-Ag produces up to 91% more correct aggregate results).
- Decreasing the error rate increases the percentage of correct aggregate results achieved by TP w/ TP-Ag and vice versa.
- TP-Ag is best suited for environments where the stream is not saturated with significant tuples. When the stream is saturated with significant tuples, more significant tuples are likely to expire.
- TP w/TP-Ag is a great fit for systems that require a targeted prioritized data stream system and desire reliable accuracy in the aggregate results produced.

8 Related Work

Below are related works beyond those already covered in Sections 1.5 and 3.

8.1 Aggregate Operators that Support Tuple Level Resource Reduction and Reorder Systems

Some *aggregation operators* proposed to support data stream systems that utilize tuple level resource allocation and reduction aim to only produce non-skewed aggregate results (Sec. 1.5) by requiring that certain tuples from selective windows are never shed. This is limiting in what tuples will and will not be processed. It does not address the TP systems where the user selects which tuples will and will not be processed. These approaches simplify aggregation because they force a complete set of tuples from these windows to arrive at the aggregate operator.

Hellerstein et al. [16] proposed an online interface that allows users to both observe the progress and halt the execution of their aggregation queries. In their approach, load shedding is initiated by the end user. To help ensure the most accurate aggregate results are produced, their approach returns the output in random order, adjusts the rate at which different aggregates are computed, and computes running confidence intervals. The running confidence intervals are displayed to the user.

Babcock et al. [4] proposed a system that supports load shedding. The goal of the system is drop tuples such that accuracy of the aggregate results produced are within certain limits. They consider the probability that dropping certain tuples has on the accuracy of query answers produced by the multiple queries.

Longbo et al. [27] propose a load shedding system for continuous sliding window join-aggregation queries over data streams. Their load shedding strategy partitions the domain of the join attribute into certain sub-domains. Then they filter out selected input tuples based on their join values.

Guo et al. [15] proposed a load shedding approach for aggregation queries with sliding windows. They analyzed the characteristics of subset model and deficiencies of current load shedding methods. Their load shedding algorithm is based on the strategy of dropping tuples from certain window.

Senthamilarasu et al. [35] proposed load shedding techniques for queries consisting of one or more aggregate operators with sliding windows. Their load shedding method utilizes a window function that divides the input into portions of the windows of the aggregate operators. It then utilizes this function to probabilistically determine which tuple to shed.

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

8.2 Tuple Level Resource Reduction

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

Aurora [1,45] is a system to manage data streams for monitoring applications. It supports real-time requirements. To achieve this, they proposed using load shedding to reduce the system of less critical tuples. Their key idea was to propose load shedding as a means to control the workload.

Tatbul et al. [38] explored a technique for dynamically inserting and removing drop operators into query plans as required by the current load. They considered both semantic and random shedding. Their cost model does not consider the cost of the drop operators to evaluate tuples. It assumes that this cost is low.

Reiss et al. [33] proposed the Data Triage architecture. It supports systems with bursty arrival rates that can fluctuate. During such bursts, Data Triage captures an estimate of the query results that the system did not have time to compute. They combined these results with the query results to generate more accurate statistics. These statistics are used to evaluate which tuples should be shed.

Tatbul et al. [37] proposed load shedding techniques for distributed stream processing environments. They modeled the distributed load shedding problem as a linear optimization problem. They proposed a distributed approach. It was built for dynamic environments in large-scale deployments.

Nehme et al. [29] proposed a load shedding technique for spatio-temporal stream data. Their load shedding model considered spatio-temporal properties by grouping similarly moving objects into clusters. Then they shed selective objects within each cluster. The locations of the objects shed are approximated based upon their associated clusters.

Wang et al. [41] proposed a load shedding technique for real-time data stream applications. The goal of their approach is to reduce the workload while at the same time preserving the system timing constraints. They proposed different modes. These modes define how the load on the stream is adjusted.

Ma et al. [28] proposed a semantic load shedding technique for real-time data stream applications that utilizes a priority table. It considers both the execution costs and tuple attribute values when deciding which tuples are shed.

Basaran et al. [5] proposed a load shedding method that applies distributed fuzzy logic. It considers the per-stream backlog and selectivity of each query operator. Their approach is event-driven. This allows it to react to bursty workloads.

Lin et al. [25] proposed a linear programming based load shedding method for distributed data stream processing systems. It models the system load as a simple query network with network constraints. It considers two factors. These factors are the amount of available CPU and network resources.

Labrinidis et al. [40] proposed a load shedding strategy that manages the load shedding without requiring any input from users, namely, any manually tuned parameters. Their approach works with complex query networks containing joins, aggregations or shared operators.

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

9 Conclusions

This paper makes the following important contributions. Our *TP-Ag* operator tackles the open problem of generating reliable average calculations for normally distributed data from incomplete aggregation populations resulting from decisions made by TPs. TP-Ag produces non-skewed average calculations by determining at run-time which combination of subset(s) of an aggregation population (if any) are used to generate a result. A carefully designed application of Cochran's sample size methodology is used to measure the accuracy of possible populations. Our experimental study confirms that TP-Ag is effective at increasing the percentage of reliable results produced in TPs (TP-Ag produces up to 91% more accurate results).

Acknowledgments. We thank our WPI peers for CAPE [34] and feedback. We also thank GAANN and NSF grants: IIS-1018443, 0917017, 0414567, and 0551584 for their support.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. The International Journal on Very Large Data Bases, 120–139 (2003)
2. Abadi, D.J., et al.: Aurora: A new model and architecture for data stream management. VLDB Journal, 120–139 (2003)
3. Arasu, A., et al.: The cql continuous query language: semantic foundations and query execution. VLDB Journal, 121–142 (2006)
4. Babcock, B., et al.: Load shedding for aggregation queries over data streams. In: ICDE, p. 350 (2004)
5. Basaran, C., Kang, K.-D., Zhou, Y., Suzer, M.H.: Adaptive load shedding via fuzzy control in data stream management systems. In: 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8. IEEE (2012)
6. Carney, D., et al.: Monitoring streams: A new class of data management applications. In: VLDB, pp. 215–226 (2002)
7. Cochran, W.G.: Sampling Techniques, 3 edn. John Wiley (1977)
8. Cormode, G., Korn, F., Tirthapura, S.: Time-decaying aggregates in out-of-order streams. PODS, 89–98 (2008)
9. Das, A., et al.: Semantic approximation of data stream joins. IEEE, 44–59 (2005)
10. Dobra, A., et al.: Processing complex aggregate queries over data streams. In: SIGMOD, pp. 61–72 (2002)
11. Fama, E.F.: The behavior of stock-market prices. The Journal of Business **38**(1), 34–105 (1965)

12. Finance, Y.: http://finance.yahoo.com/
13. Gainey, R.R., et al.: Understanding the experience of house arrest with electronic monitoring: An analysis of quantitative and qualitative data. International Journal of Offender Therapy and Comparative Criminology (2000)
14. Golab, L., et al.: Update-pattern-aware modeling and processing of cont. queries. In: SIGMOD, pp. 658–669 (2005)
15. Guo, J.-F., He, C.-L.: Load shedding for sliding window aggregation queries over data streams. Application Research of Computers, 1–23 (2009)
16. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. SIGMOD **26**(2), 171–182 (1997)
17. Hoeffding, W.: Probability Inequalities for Sums of Bounded Random Variables. Journal of the American Statistical Association **58**(301), 13–30 (1963)
18. Hoyle, S.: Use and abuse of statistics. ASLIB Proc. **40**(11–12), 321–324 (1988)
19. Kang, H.G., Mahoney, D.F., Hoenig, H., Hirth, V.A., Bonato, P., Hajjar, I., Lipsitz, L.A.: In situ monitoring of health in older adults: technologies and issues. Journal of the American Geriatrics Society **58**(8), 1579–1586 (2010)
20. Kargupta, H., Park, B.-H., Pittie, S., Liu, L., Kushraj, D., Sarkar, K.: Mobimine: monitoring the stock market from a pda. SIGKDD Explor. Newsl. **3**(2), 37–46 (2002)
21. Katopodis, P., et al.: A hybrid, large-scale wireless sensor network for missile defense. IEEE, 1–5 (2007)
22. Li, J., et al.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD **34**, 39–44 (2005)
23. Li, J., et al.: Semantics and evaluation techniques for window aggregates in data streams. SIGMOD, 311–322 (2005)
24. Lin, C.-C., et al.: Wireless health care service system for elderly with dementia. IEEE, 696–704 (2006)
25. Lin, O., Qin, Z., Jingjing, Q., Qiumei, P.: A new linear programming based load-shedding strategy. In: 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science (DCABES), pp. 260–263. IEEE (2012)
26. Liu, B., et al.: Run-time operator state spilling for memory intensive long-running queries. SIGMOD, 347–358 (2006)
27. Longbo, Z., Zhanhuai, L., Zhenyou, W., Min, Y.: Semantic load shedding for sliding window join-aggregation queries over data streams. In: International Conference on Convergence Information Technology, pp. 2152–2155 (2007)
28. Ma, L., Zhang, Q., Shi, N.: A semantic load shedding algorithm based on priority table in data stream system. In: International Conference on Fuzzy Systems and Knowledge Discovery, pp. 1167–1172 (2010)
29. Nehme, R.V., Rundensteiner, E.A.: Clustersheddy: Load shedding using moving clusters over spatio-temporal data streams. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 637–651. Springer, Heidelberg (2007)
30. Network, M.: Where have all the investors gone? (February 2012). http://money.msn.com
31. Olston, C., Widom, J.: Offering a precision-performance tradeoff for aggregation queries over replicated data. Technical Report 2000–16, Stanford InfoLab (2000)
32. Press, A.: Officials lose track of 16,000 sex offenders after gps fails (2010). http://www.foxnews.com

33. Reiss, F., Hellerstein, J.M.: Data triage: An adaptive architecture for load shedding in telegraphcq. In: IEEE International Conference on Data Engineering, pp. 155–156 (2005)
34. Rundensteiner, E.A., et al.: Cape: Continuous query engine with heterogeneous-grained adaptivity. In: VLDB, pp. 1353–1356 (2004)
35. Senthamilarasu, S., Hemalatha, M.: Load shedding techniques based on windows in data stream systems. In: 2012 International Conference on Emerging Trends in Science, Engineering and Technology (INCOSET), pp. 68–73. IEEE (2012)
36. Tatbul, N.: QoS-driven load shedding on data streams. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 566–576. Springer, Heidelberg (2002)
37. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: Efficient load shedding techniques for distributed stream processing. In: International Conference on Very Large Data Bases, pp. 159–170 (2007)
38. Tatbul, N., et al.: Load shedding in a data stream manager. In: VLDB, pp. 309–320 (2003)
39. Tatbul, N., Zdonik, S.: Window-aware load shedding for aggregation queries over data streams. VLDB, 799–810 (2006)
40. Pham, T.N., Chrysanthis, P.K., Labrinidis, A.: Self-managing load shedding for data stream management systems, 1–7 (2013)
41. Wang, H.-Y., Qin, Z.-D., Li, B.-Y., Cong, J., Wang, Z.-J., Du, M.: Novel load shedding approach for real-time data stream processing. Journal of Chinese Computer Systems, 1–4 (2010)
42. Wei, M., et al.: Achieving high output quality under limited resources through structure-based spilling in xml streams. PVLDB, 1267–1278 (2010)
43. Works, K., Rundensteiner, E.: Preferential resource allocation in stream processing systems. International Journal of Cooperative Information Systems (2014)
44. Works, K., Rundensteiner, E.A.: The proactive promotion engine. In: ICDE, pp. 1340–1343 (2011)
45. Zdonik, S.B., et al.: The aurora and medusa projects. IEEE, 3–10 (2003)