Optimal Load Shedding with Aggregates and Mining Queries

Barzan Mozafari, Carlo Zaniolo
Computer Science Department, University of California at Los Angeles
California, USA
{barzan,zaniolo}@cs.ucla.edu

***Abstract*— To cope with bursty arrivals of high-volume data, a DSMS has to shed load while minimizing the degradation of Quality of Service (QoS). In this paper, we show that this problem can be formalized as a classical optimization task from operations research, in ways that accommodate different requirements for multiple users, different query sensitivities to load shedding, and different penalty functions. Standard non-linear programming algorithms are adequate for non-critical situations, but for severe overloads, we propose a more efficient algorithm that runs in linear time, without compromising optimality. Our approach is applicable to a large class of queries including traditional SQL aggregates, statistical aggregates (e.g., quantiles), and data mining functions, such as k-means, naive Bayesian classifiers, decision trees, and frequent pattern discovery (where we can even specify a different error bound for each pattern). In fact, we show that these aggregate queries are special instances of a broader class of functions, that we call reciprocal-error aggregates, for which the proposed methods apply with full generality.**

Finally, we propose a novel architecture for supporting load shedding in an extensible system, where users can write arbitrary User Defined Aggregates (UDA), and thus confirm our analytical findings with several experiments executed on an actual DSMS.

I. Introduction

Important applications, such as live traffic monitoring, tracking of stock prices, credit card fraud detection, and network monitoring for intrusion detection, must process massive volumes of data streams with real-time or quasi real-time response. To support these applications, a new generation of data management systems, called Data Stream Management Systems (DSMS) is being developed. The continuous query languages of such DSMS are often similar to those of traditional SQL-compliant DBMSs [4], [2], [27]. But DSMS must also provide QoS in the presence of high-arrival rates, bursty arrivals and many other technical challenges not faced by traditional DBMS. In fact, data stream arrival rates can be high and unpredictable. If the arrival rates significantly exceed the system's capacity, queues build up and the processing latency increases without bound. Therefore, one of the fundamental tasks of a DSMS is to (i) constantly monitor the current load, and detect circumstances, where shedding some of the load becomes inevitable. This of course, will degrade the quality of the queries' answers. The question then becomes (i) when, (ii) where and (iii) how much load to shed. Graceful load shedding is desired in order to minimize the accuracy loss. The state-of-the-art on load shedding is still lacking a general model. Previous work on the problem delivered effective solutions under simplified conditions. For instance, most current techniques differentiate between running queries only based on their processing costs [5], [26], [7], [21], [22]; only a few of them also try to discard the less important parts of the load, with techniques that are application-specific [7] or assume that load shedders can be placed at the very source of the data streams rather than in the DSMS as needed by many applications [9], [12]. Therefore, the need for more general load shedding models and algorithms remains acute—inasmuch as current approaches do not support well the conditions that occur in many application scenarios, including the following:

1) Many users may share the same DSMS, demanding different QoS guarantees. Also, the same user may weigh her own queries differently. In fact, the QoS specification itself can be in the form of an aggregate metric of multiple queries. Thus, we must accept higher level quality specifications, e.g. to minimize the total relative error, summed up over a user's queries.
2) Each query may seek a different goal under load shedding, and that can potentially lead to even different treatment of the `PARTITION BY` (a.k.a. `GROUP BY`) keys within the same query.
3) The load shedding algorithm itself must incur no or little overhead to the system. It also has to guarantee an optimal solution for a large class of aggregate queries.
4) Load shedding techniques should be added easily to DSMSs, without compromising the openness and extensibility of the system. Thus, simple primitives are needed to provide load shedding capabilities for arbitrary aggregates.

The following examples illustrate how addressing the above points is not trivial.

Example 1. Consider a data stream where each tuple represents a transaction of basket items[1]. Suppose that the system is running two queries, Q_A and Q_B, counting the occurrence of patterns A and B, respectively, on a window size of $W = 24$. (Thus, if $A = \{milk, butter\}$, Q_A is the number of transactions containing both milk and butter.)If checking whether a pattern is contained in a tuple takes c processing units, determining the exact frequency of all the patterns over all the transactions takes $2 \times 24 = 48c$. Now, if the system's capacity only allowed $30c$, we must skip counting some transactions, some patterns, or both. If the user expresses

[1]In the literature, terms 'pattern' and 'itemset' are often used interchangeably.

no preference, any arbitrary load shedding scheme leading to $30c$ would be acceptable. But let us instead assume that our user wants to minimize $G = \sum_p \frac{1-r_p}{r_p} f_p$ where p ranges over the patterns, r_p is the fraction of the transactions considered in processing Q_p, and f_p is the true frequency of pattern p[2]. Let us now assume that $f_A = 1, f_B = 4$ (in reality, these values are not known and will have to be estimated as discussed in Sections II,III) and compare three different load shedding policies, as follows:

1. Uniform: We spend the same amount of resources for each of the queries, namely $15c$ each. We have $r_A = r_B = \frac{15}{24}$ and:

$$G = \frac{1 - 15/24}{15/24} \times 1 + \frac{1 - 15/24}{15/24} \times 4 = 3$$

2. Proportional: We allocate our resources in proportion to the frequency of each pattern. Thus, since $\frac{f_B}{f_A} = 4$, we spend $6c$ and $24c$ units on Q_A and Q_B, respectively. Therefore, $r_A = 6/24$, $r_B = 24/24$, and thus:

$$G = \frac{1 - 6/24}{6/24} \times 1 + \frac{1 - 24/24}{24/24} \times 4 = 3$$

3. Optimal: The optimal load shedding plan (one that minimizes G) could be achieved (discussed in Section V-C) if we allocated $10c$ and $20c$ units to Q_A and Q_B, respectively. Thus, we have $r_A = 10/24$, $r_B = 20/24$, and thus:

$$G = \frac{1 - 10/24}{10/24} \times 1 + \frac{1 - 20/24}{20/24} \times 4 = 2.2$$

While all three policies meet the maximum load requirement of $30c$ processing units, not all of them produce the same value for our goal function. The current literature on load shedding has so far only applied the first method, namely *uniform* [5], [26], [7], [21], [22]. However, as illustrated by this simplified example, depending on the users' criteria and the specific application needs, a uniform load shedding may not be best.

Example 2. Consider the following data stream and the two continuous queries running upon (written in ESL [6]):

```
STREAM OpenAuction (itemID int, price real, ts timestamp)
  ORDER BY ts SOURCE 'port4561';
SELECT itemID, sum(price)
    OVER(ROWS 49 PRECEDING SLIDE 10
          PARTITION BY itemID)
   FROM OpenAuction;
SELECT itemID, max(price)
    OVER(ROWS 49 PRECEDING SLIDE 10
          PARTITION BY itemID)
   FROM OpenAuction;
```

Since both `max` and `sum` are built-in aggregates, the system can automatically correct the query answers once load shedding is applied. For `sum`, the answer needs to be scaled up by the inverse of the shedding ratio, whereas for `max`, the result can be left intact[3]. Second, different users/applications may prefer to minimize different types of error, which will require different correction policies. Finally, due to the black box semantics of User-Defined Aggregates (UDA), the system is not able to shed data from its input[4]. UDAs have proven effective in providing expressiveness[5], extensibility and mining functionalities in a DSMS [24], [23]. Thus, when overloaded, shedding input from UDAs provides a significant source of efficiency for a DSMS. In this paper, we propose a novel architecture (implemented in our own DSMS) to enable a flexible load shedding framework that can be applied to both built-in and user defined aggregates, and can suit different correction policies.

Problem Definition. In this paper, we tackle the problem of optimal load shedding for aggregates over data streams, when queries have different processing costs, different importance and the users have provided their own arbitrary error functions, which may require different treatment of the keys even within the same query. Therefore, customers provide their business needs in terms of QoS specifications (e.g., stating the maximum error tolerated), and our work translates such guarantees into concrete amounts of load to be shed from each query (or its keys). We also propose and implement a novel architecture, that allows the system to apply our optimal load shedding over a large class of arbitrary UDAs (e.g., complex mining tasks), which are treated as black boxes.

Contributions. In summary we make the following contributions: 1. We formulate the general load shedding problem as an optimization problem of finding a proper shedding ratio for each query, such that in the end, a weighted error is minimized. We allow the queries in question to have different importance, error functions, processing costs, and maximum tolerated error.
2. We recognize a sub-class of queries based on the relation between their error and the applied shedding ratio, called *reciprocal-error queries*, and show that most common aggregate functions (and thus, mining tasks) fall into this class.
3. For a collection of N reciprocal-error queries, standard algorithms from operations research can find an optimal solution in time $O(N \cdot logN)$. However, we propose a more efficient algorithm for severe overload conditions, that runs in $O(N)$, without losing optimality.
4. We propose a novel architecture that can deliver our optimal policy, even in the presence of a large class of UDAs. We provide our users with an API to export their keys and their weights, and use query rewritings that can suit different execution environments, i.e. sequential DSMSs and parallel/distributed ones.
5. We present an extensive case study for the applicability and effectiveness of our approach, using frequent pattern mining and monitoring. We discuss several optimization opportunities in the implementation of adaptive load shedding.

[2]If the f_p estimates are accurate, this minimizes the total variance.

[3]There are more sophisticated methods for even correcting the min results [17].

[4]Assuming a window-based aggregate, one may still shed the input via WinDrop operators [22]. However, it will result in missing output for several windows instead of providing approximate results within some error guarantees.

[5]A DSMS becomes Turing-complete for data streams if UDAs are allowed [6].

6. We validate our theoretical results by implementing them into a full-fledged DSMS (StreamMill [6]). We present empirical results demonstrating the significant improvements on the quality of mining queries, measured by well-known metrics such as absolute MSE, relative error, and the number of false positives and negatives.

Outline. In §II and §III we review the related work and provide a background on load shedding in a DSMS. In §IV we study the effect of load shedding on answer quality. Adaptive load shedding and our proposed algorithm are introduced in §V. We present our proposed architecture in §VI. Our extensive case study on counting queries and frequent pattern mining in §VII is followed by addressing efficiency concerns in §VIII. Finally, empirical results are presented in §IX, and we conclude in §X.

II. Related Work

The prior work has addressed the processing of join queries under load shedding [10], [11], [16], which usually involves adhoc heuristics. For aggregate queries, which is the focus of this paper, we instead use random load shedding.

In their pioneering paper, Babcock et al. [5] proposed random drop operators carefully inserted along a query plan such that the aggregate results degrade gracefully. Tatbul et al.[22] showed that an arbitrary tuple-based load shedding can cause inconsistency when windowed aggregation is used. They proposed a new operator called *WinDrop* that drops windows completely or keeps them in whole. Inspired by their work, in Section V-D, we discuss how our framework can deliver both subset [22] and approximate results [5], [17], [7], [12], [9]. Also in Section VI, we address a similar concern, where missing output tuples is not an acceptable option for the application, and the system has to be assisted by programmers. Although in this paper we focus on dropping tuples, our techniques can be easily combined with the WinDrop operator, as follows. Once our algorithm decides on the shedding ratio for each query, a separate WinDrop operator can be applied to each group of the queries sharing the same ratio.

Perhaps the most aligned with our line of work is that in Loadstar [7], arguing that for many data mining tasks a more intelligent load shedding scheme for data streams is required. Even though this pioneering work differentiates between different input streams, they still treat all queries equally. Moreover, their adhoc method only focuses on classification tasks and their quality of decision, whereas we propose a general framework for a larger class of queries and support a more customizable setting. Recently, load shedding in sensor and mobile networks has been stated as an optimization problem in [12], [9], where they rely on the very sources of the data stream (i.e., nodes) to perform filtering and shedding. However, we do not make any assumptions about the source nodes, and the load shedding is performed by our centralized DSMS, as required by many streaming applications. Thus, our DSMS does not need to trust, rely upon or communicate with any of the stream generator nodes, in order to deliver more flexibility, reliability and ease of maintenance.

We borrow the existing techniques from [7], [17] as complementary modules of our load shedding architecture. We describe the interaction between such components in Section III. For boosting the quality of the apriori estimations, Law and Zaniolo [17] favor a Bayesian model while Loadstar favors a finite Markov model [7].

There has also been recent work on the application of control theory techniques in detecting the right time and also the 'total amount' of load shedding. In general, in an open-loop control (i.e., traditional load shedders), system output or state information is not used in the controller. On the other hand, closed-loop control provides better quality, less delay and less overshooting [26]. As briefly described in Section III, our method of optimal load shedding can be easily integrated with such control-loops. Whenever the controller determines the need for shedding load, and decides on the total amount of load that needs to be shed (based on monitoring the arrival rates, queue lengths, CPU usage, etc), our component optimally distributes the current resources between the running queries to satisfy the controller, while minimizing the total error.

There is also a close connection between load shedding and random samplers. In [15], streaming operators (analogous to our UDAs) can be used for random sampling of the tuples, which can then be fed into other aggregates. Yi et al. [28] proposed probabilistic algorithms for detecting malicious inconsistencies in answers from continuous queries, run over random synopses of data.

III. Background

As shown in Figure 1(a), in the typical architecture of a DSMS, load shedders are inserted between certain nodes of the query graph, in order to randomly discard a portion of the tuples under overload, and hence, reduce the buffer length and latency [5], [17]. The problem of deciding the optimal locations in the query network for inserting load shedders has been addressed by prior work [21]. Figure 1(b) depicts the major components in our framework. In the following we discuss each component.

The right time (or frequency of) load shedding, and the *total* amount of load to be shed have been addressed by prior work using techniques from control theory [26]. Their method can be integrated with ours, referred to as **Monitor** and **Controller** components in Figure 1(b).

The current load shedding literature, generally speaking[6], affects all queries equally. For instance, if the total load is twice the system's capacity, all the queries will face 50% load shedding. Thus, in this paper we extend the current architecture by adding a novel component, called **Optimal Resource Distributer (ORD)**. As shown in Figure 1(b), once the total amount of needed load shedding is decided by the controller, the ORD calculates the optimal shedding ratio for every load shedder in the query network. As shown later, in order to find the optimal solution, the ORD also needs an estimate of the data distribution/statistics. Prior work [17] has

[6]Except[12], [9] which are source-based load shedding, and [7] that is adhoc to a certain classification task, see §II.

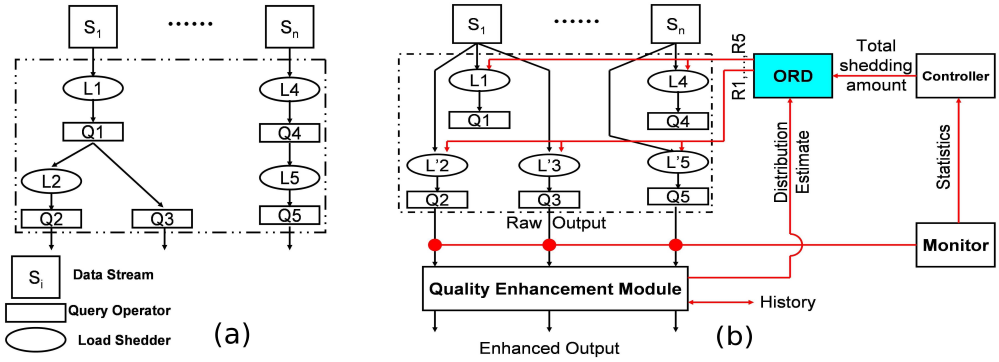


Fig. 1. (a) A query network. (b) The general framework of our load shedding proposal.

also provided means for such estimations from the past data, which is the **Quality Enhancement Module**. In fact, those estimations can also improve the accuracy of the final results significantly.

IV. Load Shedding and Error

Data streams are often divided into windows. In particular, the technique of partitioning large windows into slides to support incremental computations has proven very valuable in DSMS [8], [6]. Windows and slides can be either count based or time based. Continuous queries can be issued over a tumbling window or a sliding window. There is also a third type of continuous queries called decaying that does not require a window definition [6].

Throughout this paper we assume that the arrival rate of the input tuples and the processing load of the system is monitored periodically, based on which load shedding decisions are made for the next W tuples. Let $t_1, \cdots, t_W$ denote the current set of input tuples. The load shedding ratio applied to query q is referred to as r_q, where $0 < r_q \leq 1$. Thus, for processing q we only look at $R_q = r_q \cdot W$ randomly selected tuples ($0 < R_q \leq W$), and then try to estimate the answer from this random sample. We use L to denote the total resource limit in this period. The R_q values should be selected in such a way that the constraint below is satisfied:

$$\sum_{q \in S} R_q \leq L \tag{1}$$

where S is the set of running queries. In the above formulation, we are assuming separate execution of the queries. Later in Section VI, we will incorporate the amortized cost of processing similar queries together.

A query network may contain arbitrary operators (e.g., `SUM`, `AVG`, mining queries). Next, we explore the relationship between the shedding ratio and the error for different types of aggregate queries.

A. Counting and Frequent Pattern Mining

Let us consider an aggregation query q that counts the occurrence (a.k.a., frequency) of a pattern p. This query can be a mining task (to verify whether p is frequent enough), or just a simple `COUNT` query. In the context of pattern mining, each t_i is a transaction[7] and the frequency of p is defined to be the number of tuples t_i where $p \subseteq t_i$. For scalar `COUNT` queries, the frequency will simply be the number of those tuples where $p = t_i$.

In the presence of a load shedder with sampling rate r_q, every tuple of the window will get included in the sample with probability r_q. Let f_p be the true frequency of p over this window. Based on this random sample, we can set an approximate answer $\hat{f}_p$ to be the frequency of p in the tuples that get included, scaled by $1/r_q$. Thus, using Bernoulli sampling theory, one can prove that $\hat{f}_p$ is in fact an (unbiased) estimator, as follows.

An unbiased estimator. For each given query q (with a `WHERE` clause looking for pattern p), we can define[8] $b_i = 1$ when $p \subseteq t_i$ and $b_i = 0$ otherwise. Clearly, we have $f_p = \sum_{i=1}^{W} b_i$.

Since each tuple t_i is processed with a probability r_q and discarded with probability $1 - r_q$, we can define a random variable X_i such that $X_i = \frac{b_i}{r_q}$ with a probability of r_q and $X_i = 0$ with a probability of $1 - r_q$. In terms of these random variables, our estimator of p's frequency (which is based on the randomly selected R_q transactions) will be: $\hat{f}_p = \sum_{i=1}^{W} X_i$. The bias and the variance of this estimator can be derived as follows.

$$E[\hat{f}_p] = E[\sum_{i=1}^{W} X_i] = \sum_{i=1}^{W} b_i = f_p \tag{2}$$

Thus, we have an unbiased estimator, and:

$$\begin{aligned} Var[\hat{f}_p] &= E[\hat{f}_p^2] - E[\hat{f}_p]^2 = E[\hat{f}_p^2] - f_p^2 \\ &= \sum_{i=1}^{W} E[X_i^2] + \sum_{1 \leq i \neq j \leq W} E[X_i] \cdot E[X_j] - f_p^2 \\ &= \sum_{i=1}^{W} \frac{b_i^2}{r_q} + (\sum_{i=1}^{W} b_i)^2 - (\sum_{i=1}^{W} b_i^2) - f_p^2 \\ &= \frac{1 - r_q}{r_q} \sum_{i=1}^{W} b_i^2 = \frac{1 - r_q}{r_q} \sum_{i=1}^{W} b_i \quad \text{(Since } b_i \in \{0, 1\}) \\ &= \frac{1 - r_q}{r_q} f_p \end{aligned} \tag{3}$$

Note that Eq. (3) relates the variance (relative error) of the estimator to the applied sampling (shedding) rate r_q. The larger

[7] For simplicity, in this paper we assume fix length transactions that can fit in a tuple.

[8] A more precise notation would be $b_{i,p}$ since each pattern has its own estimator variables.

the r_q the better, e.g. if $r_q = 1$ (i.e. $R_q = W$) the estimator will be perfect, confirmed by (3) as a zero variance.

Computing the variance. Note that the ultimate goal is to estimate the f_p values as accurately as possible, but equation (3) is itself written in terms of this unknown variable. Thus in practice, instead of f_p values, we use their approximations (denoted by f_p^*) in Eq. (3) which can be derived from past data (see the distribution estimates in Figure 1(b)). However, note that these approximations are solely used in the process of estimating the variances which will then be used in Section V, to find the optimal r_q values. Once this optimal load shedding policy has been applied, we compute the $\hat{f}_p$ estimators which are now more reliable than our first approximations, namely f_p^*. This is due to the fact that for $\hat{f}_p$, we at least have an analysis of the variance and expectation values. Moreover, our experimental studies in Section IX have validated that even using the simplest approximation for f_p^* values, we still achieve significantly more accurate estimators in the end.

Our proposed techniques in later sections are independent from the specific approximation techniques, except that more accurate approximations yield better final results. Therefore, while in our experiments (Section IX) we simply use the f_p counts from the previous window as the f_p^* values for the next window, any other approximation method could be used here too. In fact, more sophisticated methods can be easily adopted in our framework, such as a weighted (decaying) sum of the counts from several past windows or the accuracy boosting framework introduced by Law and Zaniolo [17] where a Bayesian model is exploited in correcting the errors and detecting concept shifts.

B. Reciprocal-Error Queries

In this section we formally define a class of operators, named *reciprocal-error* queries, which include a large range of conventional aggregate operators and which are also key components of many important mining tasks. As shown later, the proposed algorithms in Section V work with any collection of operators from this class.

Definition 1 (Reciprocal-Error Query): A query q is reciprocal-error with respect to error function e, if under load shedding with sampling rate r_q, one can find an unbiased estimator for its answer from the random sample, such that its error e_q grows reciprocally with r_q. In other words, there should be a_q and b_q (both independent of r_p and e_q) such that for any $0 < r_q \leq 1$:

$$e_q = \frac{b_q}{r_q} - a_q \tag{4}$$

Note that in this definition we do not restrict the type of error. Thus, a query can be reciprocal-error with respect to a certain type of error while not so with respect to other error types. Therefore, once an error function e is chosen, our proposed algorithms are guaranteed to optimally minimize e for all the queries that are reciprocal-error with respect to e. However, in practice more commonly-used error functions in the context of estimators include variance[9], relative error and accuracy error.

According to Eq. (3), COUNT (and therefore frequent pattern mining) are reciprocal-error with respect to variance, where $a_q = f_q$ and $b_q = f_q$. Also, dividing this variance by the true frequency gives the relative error as $\frac{1}{r_p} - 1$. Thus, COUNT and frequent pattern queries are also reciprocal-error with respect to relative error.

We can use results from [17] for SUM and AVG queries, for which two unbiased estimators are introduced. Denoting the true value of SUM as s_q, and its estimator as $\hat{s_q}$, we have:

$$Var[\hat{s_q}] = -\frac{s_q^2(\sigma^2+\mu^2)}{N\mu^2} + \frac{s_q^2(\sigma^2+\mu^2)}{N\mu^2} \times \frac{1}{r_q} \tag{5}$$

Also denoting the AVG and its estimators by m_q and $\hat{m}_q$ respectively, the variance can be derived as:

$$Var[\hat{m_q}] = -\frac{\sigma^2+\mu^2}{N} + \frac{\sigma^2+\mu^2}{N} \times \frac{1}{r_q} \tag{6}$$

where in both equations σ and μ represent the standard deviation and the mean of the original data respectively. Eq. (5) and (6) prove that SUM and AVG are also reciprocal-error with respect to variance.

In addition to frequent pattern mining and monitoring [18], many other mining tasks can be expressed using the above reciprocal-error queries. For instance, AVG clustering algorithms such as K-means and K-nearest neighbors are implemented using AVG and COUNT, respectively. The main building blocks of many classification algorithms are also reciprocal-error aggregates. For instance, a Naïve Bayesian Classifier consists of several COUNT queries. Other examples of mining tasks expressible in terms of reciprocal-error queries include frequent pattern mining and k-means. More such algorithms can be found in [17].

Another important class of queries that are reciprocal-error with respect to variance are quantiles. According to [20], the p-th sample quantile is asymptotically normal with mean $F^{-1}(p)$ and variance:

$$Var[p\text{-th quantile of the sample}] = \frac{p(1-p)}{(f(F^{-1}(p)))^2} \cdot \frac{W}{r_q}$$

where W is the population (window) size, F is the cumulative distribution function and $f = F'$ is the density function. Note that Median, MAX, MIN are special cases of quantiles.

V. Adaptive Load Shedding

As simplified in Example 1, the general idea behind adaptive load shedding is to treat different keys of each query differently, in order to achieve optimality. Using different shedding ratios for different queries always pays off when the aggregates occur in different paths of the query graph, as shown in Figure 1. However, different shedding ratios for

[9]In this paper, we use variance and absolute error interchangeably since for unbiased estimators, their variance decides the magnitude of uncertainty. Thus, variance divided by the true value will be our relative error. For accuracy error see Section VII.

different PARTITION BY keys of the same aggregate, can also be beneficial. This may not hold for aggregates with simple PARTITION BY clauses. For instance, let us compare the following two queries:

```
Q1: SELECT itemID, sum(price)
    OVER(ROWS 49 PRECEDING SLIDE 10
       PARTITION BY itemID)
    FROM OpenAuction;

Q2: SELECT patID, count(transaction) AS freq
    OVER(ROWS 49 PRECEDING SLIDE 10
       PARTITION BY itemID)
    FROM PatternTable, TransStream
    WHERE contained(patID, transaction)
    HAVING freq > 1000;
```

Due to the unbounded nature of data streams, most DSMSs use a hash-based implementation for the PARTITION BY keys of aggregates in order to make the execution non-blocking (whereas in a DBMS, a sort-merge implementation could be applied, as a blocking operator). Thus, when executing Q1, the processing of each tuple requires constant-time (i.e., independent of the total number of *itemID*'s) to look up the value of its *itemID* in the hash table. Therefore, having different ratios for different keys will not save much computation. In other words, as long as a tuple is going to be considered for a particular key, considering it for all other keys will not incur additional overhead. We refer to such queries as 'flat-cost' queries. For example, if the above Q1 query involves three keys for *itemID*, say i, j, k, to which we need to apply $50\%, 30\%$ and 10% shedding ratios respectively, one can use the same 50% of the tuples to update the sum of all three items without extra overhead.

However, for more involved queries such as Q2, the cost of processing each tuple of the input stream depends on the number of patterns stored in *PatternTable*. We refer to this class of queries as 'variable-cost' ones. These two classes of queries can be easily detected syntactically. In our system, joins, and function-based selections mark a query as variable-cost [19].

In its most general form, we formulate the load shedding problem as that of minimizing a weighted error of our estimations, once a certain amount of load has to be shed. Let S be the set of all keys of all the running queries[10], and assume $|S| = N$. We denote the errors by vector $\vec{E} = [e_{k_1}, \cdots, e_{k_N}]$, where e_{k_i} is the error in our approximate answer for key k_i, for all keys in S. Similarly, we denote the keys' importance by $\vec{V} = [v_{k_1}, \cdots, v_{k_N}]$, and their resource cost by $\vec{C} = [c_{k_1}, \cdots, c_{k_N}]$. Thus, for each key k, we have a triple: e_k, v_k and c_k. Now, the problem of adaptive load shedding can be formally stated as choosing r_k values such that they minimize the weighted error (the scalar product of $\vec{E} \cdot \vec{V}$) as the goal function (7), subject to the resource constraint (8).

$$\text{Minimize:} \quad G = \vec{E} \cdot \vec{V} = \sum_{k \in S} e_k \cdot v_k \tag{7}$$

[10]In Section VI we discuss how these keys are extracted from the past results.

while according to (1) and $R_k = W \cdot r_k$:

$$\vec{r} \cdot \vec{C} = \sum_{k \in S} r_k \cdot c_k \leq \frac{L}{W} \tag{8}$$

If all the queries in S are reciprocal-error with respect to the given error function $\vec{E}$, one can use Eq. (4) to simplify the above optimization goal as follows:

$$\text{Minimize:} \quad G = -\sum_{k \in S} a_k \cdot v_k + \sum_{k \in S} \frac{b_k \cdot v_k}{r_k} = -\sum_{k \in S} a_k \cdot v_k + G_1$$

where:

$$G_1 = \sum_{k \in S} \frac{b_k \cdot v_k}{r_k} \tag{9}$$

Note that to minimize G, it suffices to minimize G_1 while satisfying (8). Also, notice that a_k and b_k values differ from one query type and key to another, and also from one error function to another. In each case, the proper formula should be applied, as described in Section IV-B.

For the keys extracted from flat-cost queries, the costs for all the keys from the same query are equal to the processing cost of that query for one tuple, divided by the number of keys. However, we also add extra equality constraints to enforce that the solution to the above optimization problem sets the same shedding ratio for all the keys in the same flat-cost query. On the other hand, for variable-cost queries, the c_k values represent the average processing factor of their queries, per-tuple-per-key. Also, the shedding ratios of their keys can be different here.

Next, we discuss three different solutions for the above mentioned optimization problem. First, we explain the uniform approach, which is the state of the art method in centralized load shedding methods [17], [5], [26], [21], [2]. We also present an alternative method, called proportional, that takes the weights into consideration when deciding the r_k values. Both of these methods will be later used as baselines to compare with our proposed solution. We argue that our formulation of a centralized load shedding is flexible, can be efficiently implemented (see Section V-C), and can be easily extended to arbitrary UDAs (see Section VI).

A. Uniform Resource Allocation

In current centralized load shedders [17], [5], [26], [21], [2], a single shedding rate is selected to discard some of the transactions. The remaining transactions will then be used to process all the queries. In particular, when the queries share the same query plan (e.g. all perform counting), they exhibit the same processing cost too, and thus, the same shedding ratio will be applied to all of them. In other words, all queries are treated equally, as they are processed against the same (number of) transactions. Since the same shedding ratio is uniformly applied to all the queries regardless of their error functions and importance (and sometime even their costs), we refer to this method as *uniform*. Thus, given L, W and $\vec{C}$ the global load shedding ratio r can be derived as:

$$r = \frac{L}{W. \sum_{k \in S} c_k} \tag{10}$$

Therefore, in uniform load shedding, for all $k \in S$, we have $r_k = r$. When all the the running queries are homogeneous, one can assume $c_k = 1$ for all $k \in S$.

B. Proportional Resource Allocation

Another heuristic to cope with different importance is to distribute the available resources between different queries proportional to their importance v_p. More formally, the share for each query key k from the total resource is determined using the formula below[11].

$$r_k = \frac{v_k}{\sum_{k' \in S} v_{k'}} \cdot \frac{L}{c_k.W} \tag{11}$$

Depending on the application requirements, query types involved, and the error function, one may find one of the above methods (i.e., uniform and proportional) more favorable (e.g., see Section VII).

C. Optimal Resource Allocation

As mentioned previously, a solution consists of a set of positive r_k values (for all keys $k \in S$) that satisfy constraint (8). An optimal solution with respect to a given error function is one that minimizes the goal function G_1—see Eq. (9). Our adaptive load shedding problem becomes a special case of a subclass of non-linear programming, named separable and convex resource allocation, and thus, can be optimally solved using classic operations research algorithms [25]. However, all these algorithms involve sorting [25], and therefore, their best time-complexity is, in general, $O(|S| \cdot log(|S|))$.

However, under severe loads, even a linear-logarithmic time complexity can be too expensive. In the rest of this section, we first formulate certain overloaded settings. Then, we propose a linear-time algorithm for finding the optimal solution under such settings. Later, in Section VIII, we discuss further optimization techniques.

Definition 2: For a given resource limit L, window size W and vectors $\vec{C}, \vec{E}$ and $\vec{V}$, we call the situation a '*critical setting*' if the following condition holds for all $k \in S$:

$$\frac{L}{W \cdot \sqrt{c_k}} \cdot \frac{\sqrt{b_k \cdot v_k}}{\sum_{k \in S} \sqrt{b_k \cdot v_k \cdot c_k}} \leq 1 \tag{12}$$

Roughly speaking, a critical setting refers to a situation in which the available resources make us apply load shedding to most keys if we seek an optimal solution. Before presenting an efficient algorithm for critical settings, we need to show that any optimal solution must satisfy the monotonicity property. All the omitted proofs can be found in our technical report [19].

Lemma 1: If $b_q \cdot \frac{v_q}{c_q} = b_{q'} \cdot \frac{v_{q'}}{c_{q'}}$, in any optimal solution $r_q = r_{q'}$. Also, in such solutions, when $b_q \cdot \frac{v_q}{c_q} < b_{q'} \cdot \frac{v_{q'}}{c_{q'}}$ we have:

1) $r_q < r_{q'}$ if $r_q < 1$.
2) $r_q = r_{q'}$ if $r_q = 1$.

[11]When the righthand side is larger than one, r_k is set to 1.

This lemma, in conjunction with the following results, leads us to an efficient algorithm as long as the system is in a critical setting (i.e., the resources are below a certain threshold).

Theorem 2: In any optimal solution for a critical setting, if $b_k \cdot \frac{v_k}{c_k} < b_{k'} \cdot \frac{v_{k'}}{c_{k'}}$, we have:

$$r_k = \sqrt{\frac{b_k \cdot v_k}{b_{k'} \cdot v_{k'}}} \cdot \sqrt{\frac{c_{k'}}{c_k}} \cdot r_{k'} \tag{13}$$

Lemma 3: Under a critical setting, if r_k values for $k \in S$ are an optimal solution with respect to a given $L > 0$, then an optimal solution for the same set of vectors $\vec{C}, \vec{E}$ and $\vec{V}$ with respect to any other $L' > 0$, consists of $r'_k = \frac{L'}{L} \cdot r_k$ for all $k \in S$.

According to Lemma 3, we do not even need to examine all values for the maximum ratio before applying Theorem 2. The pseudo code for finding an optimal solution in linear time is presented next.

Algorithm 1.

1) Let m be one of the keys with maximum $\frac{b_m \cdot v_m}{c_m}$. Assign an arbitrary value to r_m. For all other keys $k \neq m$, if $f_k = f_m$ then let $r_k := r_m$; otherwise choose r_k based on Theorem 2.
2) Let $u = sum_{k \in S} r_k \cdot c_k$. Return $r'_k = \frac{L \cdot u}{W} \cdot r_k$ for all $k \in S$ as an optimal solution satisfying constraint (8).

D. Flexibility of the framework

In general, load shedders follow one of the following two paradigms: (i) they drop a fraction of the input, and try their best to provide approximate answers in the output [5], [17], [7], [12], [9]. (ii) they drop/keep windows entirely such that the output of the aggregates for the kept windows remains unaffected while no output is produced for the missing windows. This latter method is called 'subset results', as the output is always guaranteed to be a subset of the actual answer [22], and comes at the expense of missing output tuples. Our framework can naturally combine the two paradigms, to provide a broad spectrum of applications with more flexibility, as described next.

The user can provide a maximum-tolerable error for each of his queries (or even for certain keys within his queries), above which he would not be willing to see our approximate results. Thus, once we solve our optimization problem, we simply revisit[12] all the queries that will not meet the required QoS specs. By ignoring such queries and distributing their resources among other queries we are able to further boost the quality of their answers. Thus, a user who always prefers subset results over inaccurate ones, can simply set all those maximum tolerable errors to zero. In such a case, the optimization problem determines an optimal solution in which less affordable/important queries will be ignored in the interest of providing subset results for other queries.

Implications on designing future systems. Having access to an optimal load shedding algorithm can also be beneficial

[12]In fact, we do not need to solve the equation iteratively. We only need to add appropriate inequality constraints on the shedding ratios. More details can be found in our technical report [19].

from a design point of view. For a given QoS requirement, and an upper bound on overload, one can pose the following question: What amount of resources would we need to guarantee the QoS requirements under a worst-case overload scenario? The optimal solution will then effectively determine the minimum amount of resources which would need to be allocated at the time of designing the system.

VI. Architecture

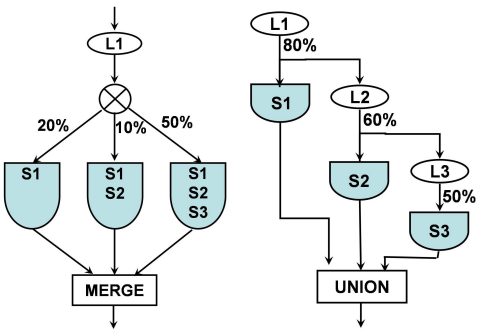
In this section, we present our extensible architecture that can deliver optimal load shedding for aggregates including arbitrary UDAs. The main difficulty in shedding input for arbitrary UDA lies in its black box nature which causes the following issue:

1) Their internal semantics is not known to the system, and therefore dropping random input tuples can lead to unexpected, and unacceptable results.
2) The system cannot automatically make the appropriate corrections to the results returned by the UDA—as opposed to built-in aggregates such as `MIN` and `SUM` where the results require no correction or are simple to scale up.
3) The keys involved in a UDA (to be used in a `PARTITION BY` clause) may be unknown to the system, and therefore, the load shedder cannot decide on different shedding ratios for each key.

While the first issue above, has been addressed in [22], their solution seeks to maximize subset results and assumes that several windows can be ignored, entirely. Our method instead, deals with general UDAs (whether they are running over a window or are decaying). We take a middle-road approach, where we provide the users with an API to export their keys from the UDAs in a certain format, but the rest of the load shedding and query re-writing are taken care of by the system. The user can also specify another built-in or user defined aggregate to perform the result correction, which will be invoked by the system upon application of load shedding. We first discuss the API through which each UDA can export its internal keys. Then in Section VI-B, we present our execution model based on the rewriting of aggregate queries.

A. Key Extraction

Each UDA can call our API to export a number of triples $\langle k, a_k, b_k \rangle$, where k is a key value and a_k and b_k are its corresponding coefficients from Eq. (4), assuming that the UDA seeks a reciprocal-error function in a load shedding situation. These triples exported from UDAs are the only information that need to be fed into our adaptive load shedder module (i.e., ORD) for finding an optimal policy. Note that in most practical cases these coefficients do not impose any burden on the UDA, inasmuch as these coefficients are either identical to the results from the previous window, or are easily computable from those results. For instance, in frequent pattern mining, according to Eq. (3), we have $a_k = f_k$ and $b_k = f_k$, where f_k is the UDA's output for pattern (key) k. Thus, we can simply use the results from the previous window to estimate these coefficients.



(a) Fork-Merge. (b) Cascade.

Fig. 2. Two different alternatives for rewriting UDA queries.

B. Query Rewriting

Once the load shedder (ORD) decides on the shedding ratios for individual keys involved in a query, the system will enforce these ratios as follows:

Flat-cost queries: For each incoming tuple, the scheduler can easily determine the involved key, and apply the appropriate shedding ratio. The implementation in this case is quite straightforward, as each tuple will only be used for one of the keys.

Variable-cost queries: For this type of query, enforcing different shedding ratios for different keys is not trivial. The source of this complexity is either complicated selection clause (e.g., see Q2 in Section V), or implicit keys (i.e., keys not mentioned in the query expression). Thus, a simple separation of the incoming tuples based on their key value will not be feasible, and in some cases can even lead to logical inconsistencies. In particular, for UDAs, we cannot simply drop some seemingly irrelevant tuples from their inputs, as it can interfere with their internal semantics, e.g. the UDA may be building a histogram (Remember that we treat UDAs as black boxes). Detailed examples can be found in [19]. Hence, to address this situation, we rewrite the variable-cost queries that contain a UDA, into provably equivalent forms, as described next.

1) Fork-Merge Operation: We create multiple instances of the UDA. In Figure 2(a) these instances are shown as blue (shaded) semi-ellipses, filled with S_i's. As a running example, assume that we group all the keys based on their optimal shedding ratio. Say that we have three groups, $S1$ with a ratio of 80%, $S2$ with 60%, and $S3$ with 50%. As depicted, we initialize the first UDA instance with all three sets of the keys, the second one with the keys in $S1$ and $S2$ and the last one with only $S1$. For each tuple, the load shedder $L1$, draws a random number r ($0 < r < 1$), and the tuple gets routed to the appropriate UDA instance, based on r. Thus, if $0 < r \leq 0.5$, it will get routed to the rightmost instance, if $0.5 < r \leq 0.6$ it will reach the middle UDA, and if $0.6 < r \leq 0.8$ it will be sent to the leftmost UDA. One can easily verify that due to the delta shedding ratios, at the end, all the keys will experience their own shedding ratio. Thus, the original query will be rewritten as follows, where `RANDOM()` returns a random scaler value, and will be evaluated only once per each tuple, and `MAXRATIO` is set to 0.8 by the load shedder.

```
CREATE STREAM PatternStreamForkMerge AS
 SELECT FIS(tid)
   OVER (ROWS 1000000 PRECEDING SLIDE 1000)
   PARTITION BY ROUTE(RANDOM())
```

```
FROM InputStream WHERE RANDOM() < MAXRATIO;
```

However, due to common keys among different instances, we cannot simply take a UNION of the outputs. Instead, we ask the user to specify the appropriate merge operation for eliminating/combining the duplicate keys, as shown in Figure 2(a). For instance, we use MAX for MAX, MIN for MIN, SUM for COUNT, weighted sum for SUM, or a customized UDA for a UDA with obscure semantics.

This operation is well-suited to multi-processor servers, or distributed DSMSs. However, when the key space is too large, the rewritten query increases the total number of keys in the running UDAs, and can suffer from many large hash-tables, causing memory issues. The more important limitation of the Fork-Merge is its dependence on the user to specify an appropriate merge operation. Thus, in our system, we have designed an alternative operation, presented next.

2) Cascade Operation: Similar to Fork-Merge, we again create multiple instances of the running UDA, shown as blue semi-ellipses in Figure 2(b). Using the same running example, this time we initialize each UDA instance with only one key group, as shown in Figure 2(b). Here, we cascade a series of load shedder operator[13] such that each UDA instance is fed according to its shedding ratio. For example, a tuple with a random label $r = 0.55$ will be forwarded to all the UDAs except the one with a ratio of 0.5.

Note that in this operation, we are duplicating the tuples instead of the keys. Therefore, the total size of required hash-tables will be comparable to that of the original one. Thus, it is preferable over Fork-Merge operation when the key space is too large (i.e., many distinct keys). Sharing of the tuples prevents parallel execution of the query graph, and duplicating the tuples (as implemented in our system) can cause extra load to the system. Thus, we only fall back on the Cascade operation, when the user refuses to specify a merge operator with his UDA. Note that for Cascade we do not need an adhoc merge operation; due to the separate keys in each output, we can use a simple UNION operator, that is the same for all UDAs (see Figure 2(b)).

C. Result Correction

For built-in aggregates, the system can automatically correct the output answers once load shedding is applied. For instance, for MIN and MAX, the answer does not need[14] a correction, but for SUM, the answer has to be scaled up by the inverse of the shedding ratio.

The correction phase, however, becomes another challenge in providing load shedding for arbitrary UDAs, as the system is unaware of their internal semantics. The simplest solution could be dropping tuples after the UDAs (i.e., from their outputs). But this late shedding of tuples will not save much computation, as all the tuples have been already processed by the UDA. To overcome this problem, we allow our users to specify a correction function/aggregate for their own UDA, if they want a system provided load shedding. At run-time, the current load shedding ratio applied, can be accessed by invoking a built-in function, called $shedratio()$. For simple corrections, it can be directly called in the query expression; e.g., to scale up the frequency counts the user will write[15]:

```
SELECT FIS(tid) * 1/shedratio()
  OVER (ROWS 9999 PRECEDING SLIDE 100)
  FROM InputStream;
```

For more involved answer corrections, the user can implement yet another UDA. This correction function (specified in the ON SHEDDING clause) must have a signature that is compatible with the SELECT clause of the query. Namely, it has to both take as input and return as output, tuples from the SELECT clause. The following provides a simple example.

```
SELECT patID, MyCount(transaction) AS freq
  OVER(ROWS 9999 PRECEDING SLIDE 100
     PARTITION BY patID)
  FROM PatternTable, TransStream
  WHERE contained(patID, transaction)
  ON SHEDDING Corrector(patID, freq);
```

VII. Case Study: Frequent Pattern Mining

While our results hold for any error function under which queries become reciprocal-error, in this section, we provide a few concrete examples from the field of frequent pattern mining. We show how one can seek different objectives by choosing appropriate error functions with different b_q coefficients. In general, different applications that consume frequent patterns may prefer minimum absolute error or minimum relative error. Moreover, some applications only need to predict which patterns are frequent/infrequent, without knowing their exact frequencies. In other words, if a solution provides poor estimates it might still be acceptable as long as the frequency error does not cross the minimum support threshold, i.e. the error does not make a frequent pattern infrequent or vice versa. In the following, we briefly discuss how each of these popular goals can be achieved within our framework.

A. Minimizing the absolute error.

By choosing the coefficients $a_p = f_p$ and $b_p = f_p$, according to Eq. (3), minimizing Eq. (7) will effectively minimize the total (or average) Mean Squared Error (MSE) or variance. Next, we use the general results obtained in Section V, for analyzing the special case of frequent pattern mining, and for different resource allocation policies.

B. Minimizing the relative error.

The relative error of the estimator is its variance normalized by its frequency. Thus, dividing Eq. (3) by f_p gives the relative error as $-1+\frac{1}{r_p}$. Similarly, by choosing the coefficients $a_p = 1$ and $b_p = 1$, minimizing Eq. (7) will effectively minimize the total (or average) relative error. Due to the symmetry of the

[13]In the actual implementation, we only use one operator at the top, but annotate the tuples with their random number, to allow a simpler filtering along the query graph.

[14]Here, correction refers to making the estimator unbiased. However, the accuracy of the results can always be enhanced using other techniques to reduce the estimator's variance, see [17].

[15]Note that this is different from a built-in count query, where the system can automatically correct the results.

patterns in terms of these coefficients, the optimal solution uses the same shedding ratio for all patterns. Thus, we have the following lemma.

Lemma 4: If the error function is relative error, and all queries are counting queries, both uniform and optimal approaches lead to the same solution.

Uniform Policy. Since all the queries are counting, we can assume that $c_k = 1$ for all $k \in S$. We can use Eq. (3) and also assume that $v_k = 1$ for all $k \in S$. This simplifies the total weighted error of this special case as follows.

$$G^{uni} = \frac{1-r}{r} \sum_{k \in S} f_k = \frac{N.W}{L} \sum_{k \in S} f_k - \sum_{k \in S} f_k \qquad (14)$$

Proportional Policy. Similar to the uniform case, we can further simplify the total weighted error for frequent pattern mining. When for all $k \in S$, we have $r_k < 1$, $c_k = 1$, and $v_k = 1$, one can derive the following:

$$G^{prop} = \sum_{k \in S} (\frac{W \cdot \sum_{k \in S} f_k}{L \cdot f_k} - 1) \cdot f_k = \frac{N \cdot W \cdot \sum_{k \in S} f_k}{L} - \sum_{k \in S} f_k \qquad (15)$$

Thus, in the context of frequent pattern mining, pattern verification [18], and any other application that consists of only counting queries, we make the following observation: from (14) and (15) we notice that both the uniform and proportional[16] load shedding policies produce the same total variance ($G^{uni} = G^{prop}$). However, the uniform approach is still more favorable since it does not require knowing the f_k values, while the proportional method does. Based on the quality of the approximation used, the analysis for the proportional approach will vary, and the result in (15) may not necessarily be achievable.

Both uniform and proportional approaches can be implemented in time linear in the number of queries, but none of the two produce the optimal solution for G_1 (as demonstrated by Example 1).

Optimal Policy. Similar to uniform and proportional policies, we can calculate the total error (here, variance) for the special case of frequent pattern mining:

Lemma 5: For frequent pattern mining, under a critical setting, the minimum variance is the following:

$$G^{opt} = -\sum_{k \in S} f_k + \frac{W}{L} (\sum_{k \in S} \sqrt{f_k})^2 \qquad (16)$$

C. Maximizing the classification confidence

For a given minimum support α, if the objective is to determine whether $f_p \geq \alpha$ or $f_p < \alpha$ as confidently as possible, one can seek to maximize the following goal function:

$$G_\alpha = \sum_p \left| Pr[f_p \geq \alpha | \hat{f}_p] - Pr[f_p \leq \alpha | \hat{f}_p] \right| \qquad (17)$$

Using the central limit theorem, we can prove that for a given set of patterns with f_p and $\hat{f}_p$ being the true frequency and the estimate for pattern p respectively, we have:

$$G_\alpha = \sum_p \frac{2}{\sqrt{\pi}} \cdot \int_0^{\sqrt{2 \cdot \frac{1-r_p}{r_p}}} e^{-t^2} dt sinv \qquad (18)$$

By minimizing the negated goal, namely $-G_\alpha$, we will maximize the confidence of our classification. Even though our counting queries are not reciprocal-error with respect to Eq. (18), we can use an approximation of this integration for which counting becomes reciprocal-error. For instance, for $0.1 < r_p < 0.9$, a simple approximation can be the following:s

$$G_\alpha \approx \sum_p \frac{1}{2} - \frac{(f_p - \alpha)^2}{\pi} \frac{1}{r}$$

Our experiments in Section IX show that even this simplified goal function leads to significant improvements in terms of false positive and false negative percentages.

VIII. Optimization Opportunities

As analyzed in Section V-C and validated by our experiments in Section IX, the overhead of finding an optimal solution itself is negligible. However, there are circumstances where having the same load shedding ratio allows for execution optimizations. One important such circumstance is in frequent pattern mining. In frequent pattern mining, all the patterns that share the same shedding ratio can be batched together in a single *pattern tree* which is a compact data structure allowing fast mining and counting of transactional data [13], [18]. Thus, while the uniform approach does not deliver an optimal solution, it can be implemented more efficiently. In the rest of the section, we address this issue.

A. Verification and fast counting

Mozafari et al. have recently shown that the well known *fp-tree* data structure is not only efficient for mining but is even more so for conditional counting (called *verification* in [18]). For a given set of patterns and a set of transactions, the verification task is to accurately count the occurrence of those patterns against the transactions if their frequency is above a given threshold. In other words, patterns that are guaranteed to be infrequent need not be counted and can be skipped for efficiency. The authors have proposed a fast *verifier* (i.e., an algorithm for verification) that outperforms the traditional counting methods such as hash trees, even with a threshold of zero. Thus, we use these verifiers to perform our optimal load shedding solution to address the aforementioned efficiency concerns that arise in a pattern mining/monitoring scenario, as described next.

B. k-means for coarsening different ratios

An optimal load shedding solution can potentially lead to a different shedding ratio for each pattern (or query). Since applying a different shedding (sampling) rate for counting each pattern's frequency is not practical, we group the patterns according to the proximity of their sampling rate. Then, each group will be stored in a separate pattern tree that will undergo the same shedding ratio in the counting process. By choosing the ratio of each group to be the mean of its members' ratios,

[16]The requirement of $r_q < 1$ for proportional method, will be formalized in Definition 2.

Algorithm k-optimal(k)
Input: k is the allowed number of shedding groups.
Output: Frequency estimates of the given patterns.
0: For each window:
1: $\vec{r} \leftarrow$ Optimal Solution from [25]
2: $(g_1, \bar{r_1}), \cdots, (g_k, \bar{r_k}) \leftarrow$ k-means($\vec{r}$)
3: For each group $0 \geq i \geq k$:
4: Insert patterns of group g_i into pattern tree PT_i
5: For each transaction t in the current window:
6: draw a random number ρ
7: Consider t in counting of all pattern trees PT_i, where $\rho \leq \bar{r_i}$
Return count estimates for the given patterns.

Fig. 3. The pseudo code for k-optimal algorithm.

the total resource usage will not increase. Note that the larger the groups, the fewer the different ratios, which would improve efficiency at the expense of optimality. In an extreme case, when we group all the patterns into one group, the final solution turns into a uniform one. At the other extreme, when each group only consists of one pattern, the solution remains optimal. A high-level pseudo code for this algorithm is given in Figure 3, named *k-optimal*.

The *k-optimal* algorithm employs the k-means clustering algorithm [14] to group the patterns, and therefore a proper k should be provided as input to the algorithm. The best trade-off between efficiency (smaller k) and optimality (larger k) can be determined according to the application requirements and through empirical comparisons to measure the overhead of adding each extra pattern tree. In our experiments in Section IX-F, we show that in practice even a few pattern trees can significantly improve on the uniform approach without compromising either accuracy or efficiency. Since we are dealing with one dimensional data (i.e., the ideal shedding ratio of each pattern) we can perform k-means in time $O(N \cdot log(N))$ where N is the total number of patterns. We first sort the patterns according to their shedding ratio which is determined by the optimal solution. We use a disjoint set data structure to represent the groups. Initially, each pattern is a group by itself. All the groups are inserted into a min-heap data structure according to their closest distance from their neighbors. Since group ratios are numbers and they are kept sorted, each group will always have (at most) two neighbors. By performing delete-min on the heap, and merging the top-element of the heap with its closest neighbor, we will have one fewer group. This operation will be repeated until there are only k groups left in the heap. Due to space limitations, we omit a pseudo code for performing k-means on 1-D data.

IX. Experiments

This section presents empirical evidence, demonstrating (i) improvements on the results' quality, and (ii) the efficiency aspects of our proposed techniques. All experiments were conducted on a P4 machine running Linux, with 1GB of RAM. All the algorithms are implemented in C, and integrated into StreamMill [6] which is an existing DSMS.

Quality of mining results. The first goal of our experiments is to compare the proposed load shedding algorithm with its state-of-the-art counterpart, namely the uniform approach. We will study the effect of different load shedding policies under different quality metrics and under different overloading settings (§IX-A, §IX-B,§IX-C). We used both synthetic (IBM QUEST [3]) and real-world datasets (Kosarak [1]), but due to the similarity of the results and lack of space, we only report the experiments achieved on the Kosarak dataset. Unless stated otherwise, in most of the following experiments we used a window size of $10,000$ tuples, a minimum support of 1%, and almost 400 patterns.

Efficiency. The second sets of our experiments, study the efficiency of our proposed framework, in §IX-D,§IX-E,§IX-F.

A. Absolute error

We measured the absolute MSE (i.e., variance) of different shedding policies for a wide range of overloading ratios. We separately investigated slightly overloaded and highly overloaded situations, respectively in Figures 4(a) and 4(b). The horizontal axis demonstrates the amount of available processing resources normalized by the ideal amount needed to process the entire window. The vertical axis (shown in log-scale) is the variance summed up over all the patterns. For the Kosarak dataset, according to Definition 2, the critical setting was any setting in which the available resource was less than 4.5% of the current load.

As shown in Figure 4(a), both optimal and proportional methods significantly outperform the uniform approach when the resource to load ratio is comparable (e.g., above 50%). While in such situations the optimal method is only slightly better than the proportional method, their distance becomes more dramatic for highly overloaded settings as shown in Figure 4(b). Also, the more overloaded the closer the uniform and proportional methods are. In particular, they produce exactly the same total variance for all critical settings (confirmed by Eq. (14) and Eq. (15)), namely ratios below 4.5%.

B. Relative error

When the goal is to minimize the relative error, the uniform method is identical to our optimal solution (as confirmed by Lemma 4). As shown in Figure 4(c), proportional load shedding on average causes 1.3 times more relative error than the optimal (or uniform) solution. Note that the vertical axis is in log-scale.

C. Classification Confidence

As discussed in Section VII, minimizing $-G_\alpha$ would maximize the confidence of our estimators in classifying frequent patterns from infrequent ones. However, even using a simple approximation of Eq. (18) our optimal algorithm was able to significantly outperform the uniform approach. A false negative occurs when a pattern's frequency is falsely underestimated to be below the threshold, and a false positive refers to the false overestimation of an infrequent pattern with a frequency that is above the threshold.

Figure 5(a) compares the average number of false negatives for the optimal and the uniform approach. In particular, when

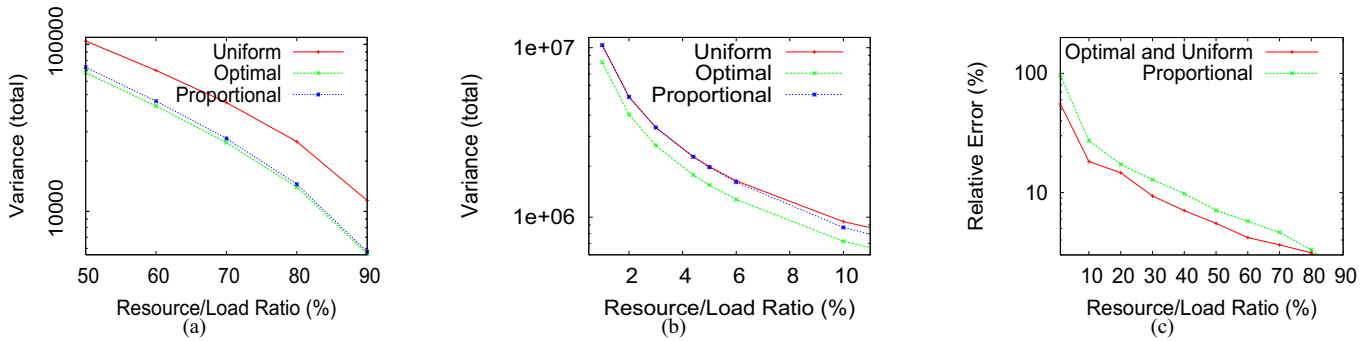


Fig. 4. (a) Variance in different load shedding policies under non-critical settings. (b) Variance in different load shedding policies. All ratios below 4.4% are critical settings for kosarak dataset. (c) Relative error under load shedding.

the resource was at least 50% of the ideal amount, the optimal method introduced no false negatives, while the uniform approach still produced a significant number of false negatives. Consistently, Figure 5(b) demonstrates the superiority of the optimal approach in terms of false positives. While the two curves become closer for highly overloaded settings, they are more distant for other settings (e.g., ratios above 30%).

D. Algorithmic Improvements

As discussed in Section V-C, finding an optimal solution takes $N \cdot log(N)$ time in the worst case scenario, where N is the total number of keys. However, our proposed algorithm for solving the equation under critical settings, runs in time $O(N)$. As shown in Figure 5(c), the actual outperformance delivered by our algorithm for 10000 keys, is at least 5 times. In fact, due to the sorting operation that is required by the standard algorithm [25], our outperformance becomes more dramatic for larger number of keys. The run time of both algorithms is independent of the window size and the number of queries, and only depends on the total number of keys involved. Note that our method is only applicable to critical settings, where finding an optimal solution in linear time is guaranteed. In fact, in critical settings the system's resources become even more valuable.

E. Load Shedding Overhead on the System

For this experiments, we use the California's Realtime Freeway Speed data[17], stored as an offline stream with a window size of $200K$ tuples, and a collection of randomly generated continuous queries, each containing one simple algebraic aggregate, involving 1000 distinct key values. The average processing time of each query was 17.713 secs per window. Due to space limitations and similarity of the results, we report a few different settings in Table I. The cost of finding the optimal solution (shown under LS Time column) is negligible compared to the time spent on processing the actual queries (this ratio is shown in the Overhead column). Thus, our system allows for supporting hundreds of aggregate queries with hundreds of thousands of different keys, without spending more than a few percents of the resources. In many real applications, the key space is much smaller. Rows marked with an asterisk represent critical settings.

[17] `http://www.dot.ca.gov/traffic/d7/update.txt`

F. Efficiency and Loss of optimality

To address the efficiency concerns discussed in Section VIII, we proposed the k-Optimal Algorithm that groups the patterns into k groups according to the proximity of their shedding ratios. We ran k-Optimal with different values for k to find an appropriate tradeoff. Figure 6(a) shows that while the optimal algorithm incurs some efficiency overhead compared to the uniform method, we can improve our algorithm's efficiency by choosing smaller k values. In the extreme case of $k = 1$, k-optimal exactly matches the uniform method. However, the efficiency overhead becomes negligible for a wide range of k values, here from 1 to 50.

With fewer groups, there are more patterns in each group, and hence ratios within a group tend to be more distant from the optimal value. This is shown in Figure 6(b) where the error (i.e., variance) increases with fewer groups. Again, for a wide range of group sizes the difference in variance is negligible. Thus, by choosing a k value from 50 to 5 (i.e., an average group size between 8 and 80) one can achieve a variance reasonably close to that of the optimal one, without incurring any significant time overhead.

X. Conclusion

In this paper, we have proposed a very general framework that achieves optimal load shedding policies, while accommodating different requirements for different users, different query sensitivities to load shedding, and different penalty functions. The experimental results confirmed the superiority of the proposed algorithm over the state-of-the-art methods. A second advantage of this algorithm is its applicability to a wide spectrum of aggregate functions which we have formally characterized using a newly introduced notion, called reciprocal-error queries. Besides the typical algebraic aggregates, this class also includes sophisticated mining tasks. We propose an extensible architecture that allows UDAs to benefit from the system-provided load shedding functions. In fact,

TABLE I
Load shedding overhead on the system.

#queries	Resource/Load	#keys	LS Time	Overhead
10	90%	$100K$	0.030 sec	0.17%
100	9%	$1M$	0.361 sec	1.99%
500	1.8%	$5M$	0.379 sec *	2.13%
1000	0.9%	$10M$	0.758 sec *	4.27%
2000	0.45%	$20M$	1.515 sec *	8.55%

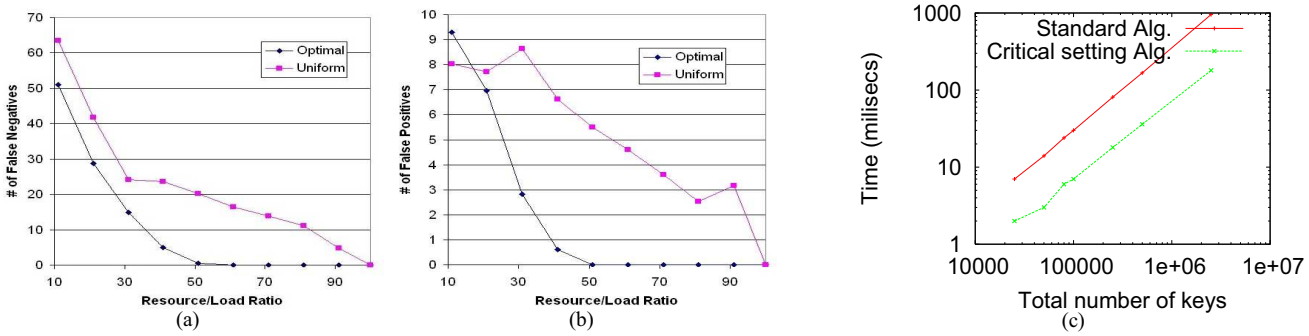


Fig. 5. (a) Effect of optimal load shedding on number of false negatives. (b) Effect of optimal load shedding on number of false positives. (c) Comparing the standard algorithm and our critical-setting method, for finding the optimal solution.

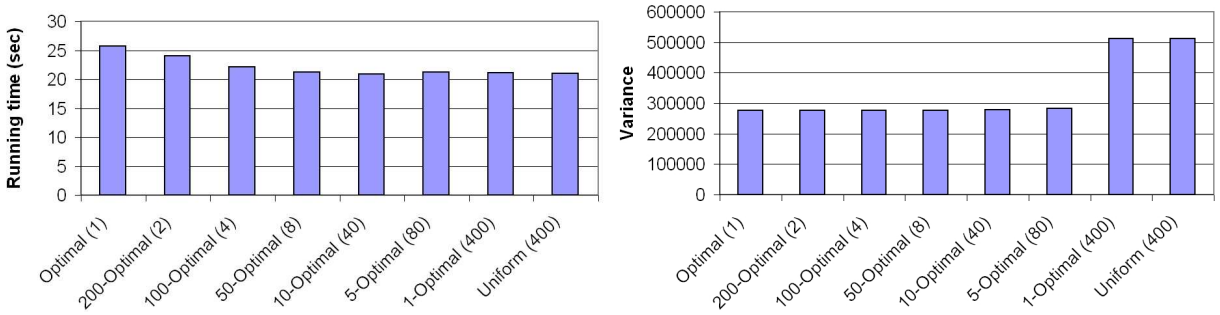


Fig. 6. Efficiency/Accuracy trade-off between different shedding policies (numbers in parentheses are average group size): Run time and Variance.

our experimental studies show significant improvements (in absolute error, false positives, and false negatives) compared to the uniform approach used by previously proposed load shedders. We showed that the load shedding problem reduces to a non-linear optimization problem of operations research, and it is thus solvable, irrespective of window size, in time $N \cdot log(N)$, where N is the total number of queries. We also propose a more efficient algorithm to handle severe overloads, without losing optimality. Our approach has been integrated into an existing DSMS, and has proven to be scalable and without much overhead. Future work includes developing and integrating prediction methods to cope with concept shifts/drifts, by using past statistics [7], [17].

Acknowledgment

This work was supported in part by NSF-IIS award 0742267, "SGER: Efficient Support for Mining Queries in Data Stream Management Systems". We would like to thank Albert Lee, Yan-Nei Law, Alexander Shkapsky and Nikolay Laptev for their insightful comments.

References

[1] Frequent itemset mining dataset repository, http://fimi.cs.helsinki.fi/data/.
[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. Aurora: A data stream management system. In *SIGMOD*, 2003.
[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.
[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD*, 2003.
[5] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, 2004.
[6] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, 2006.
[7] Y. Chi, P. S. Yu, H. Wang, and R. R. Muntz. Loadstar: A load shedding scheme for classifying data streams. In *SDM*, 2005.
[8] J. L. et al. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 2005.
[9] B. Gedik, K. L. Wu, and P. S. Yu. Efficient construction of compact shedding filters for data stream processing. In *ICDE*, 2008.
[10] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *CIKM*, 2005.
[11] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu. A load shedding framework and optimizations for m-way windowed stream joins. *ICDE*, 2007.
[12] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu. Mobiqual: Qos-aware load shedding in mobile cq systems. In *ICDE*, 2008.
[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.
[14] J. A. Hartigan. *Clustering Algorithms*. Wiley, 1975.
[15] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD Conference*, 2005.
[16] Y. N. Law and C. Zaniolo. Load shedding for window joins on multiple data streams. In *ICDE Workshops*, 2007.
[17] Y. N. Law and C. Zaniolo. Improving the accuracy of continuous aggregates and mining queries on data streams under load shedding. *Intl J. Bussiness Intelligence and Data Mining*, 3(1), 2008.
[18] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, 2008.
[19] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. Technical report, UCLA, 2009.
[20] P. K. Sen. On some properties of the asymptotic variance of the sample quantiles and mid-ranges. *J. of Royal Stat. Soc.*, 23(2), 1961.
[21] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.
[22] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, 2006.
[23] H. Thakkar, B. Mozafari, and C. Zaniolo. A data stream mining system. In *ICDM Workshops*, 2008.
[24] H. Thakkar, B. Mozafari, and C. Zaniolo. Designing an inductive data stream management system: the stream mill experience. In *SSPS*, 2008.
[25] I. Toshihide and K. Naoki. *Resource allocation problems: algorithmic approaches*. MIT Press, 1988.
[26] Y. C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, 2006.
[27] H. Wang, C. Zaniolo, and C. Luo. Atlas: A small but complete sql extension for data mining and data streams. In *VLDB*, 2003.
[28] K. Yi, F. Li, M. Hadjieleftheriou, G. Kollios, and D. Srivastava. Randomized synopses for query assurance on data streams. In *ICDE*, 2008.