Evaluating Top-k Queries over Incomplete Data Streams

Parisa Haghani
EPFL
Lausanne, Switzerland
parisa.haghani@epfl.ch

Sebastian Michel
EPFL
Lausanne, Switzerland
sebastian.michel@epfl.ch

Karl Aberer
EPFL
Lausanne, Switzerland
karl.aberer@epfl.ch

ABSTRACT

We study the problem of continuous monitoring of top-k queries over multiple non-synchronized streams. Assuming a sliding window model, this general problem has been a well addressed research topic in recent years. Most approaches, however, assume synchronized streams where all attributes of an object are known simultaneously to the query processing engine. In many streaming scenarios though, different attributes of an item are reported in separate non-synchronized streams which do not allow for exact score calculations. We present how the traditional notion of object *dominance* changes in this case such that the k dominance set still includes all and only those objects which have a chance of being among the top-k results in their life time. Based on this, we propose an exact algorithm which builds on generating multiple instances of the same object in a way that enables efficient object pruning. We show that even with object pruning the necessary storage for exact evaluation of top-k queries is linear in the size of the sliding window. As data should reside in main memory to provide fast answers in an online fashion and cope with high stream rates, storing all this data may not be possible with limited resources. We present an approximate algorithm which leverages correlation statistics of pairs of streams to evict more objects while maintaining accuracy. We evaluate the efficiency of our proposed algorithms with extensive experiments.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.4.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Experimentation

Keywords

top-k queries, incomplete streams, skyline, dominance set

CIKM'09, November 2–6, 2009, Hong Kong, China.

1. INTRODUCTION

Research on data streams has gained a lot of interest in the past few years [2, 26, 9, 24, 22, 6]. Many data and query characteristics of modern applications are best captured by this computational model, where data streams in continuously at high rates and each tuple has the chance of being observed only once. Memory constraints usually force the eviction of old tuples to let new tuples arrive and be processed. In this paper, we consider tracking top-k items over multiple data streams in a sliding window. Each stream represents one particular dimension of interest, for instance, a particular attribute of the observed item w.r.t. a sensor location. We address a broad area of application scenarios, like network monitoring and sensor network data processing such as observing local natural phenomena, a common task in environmental sciences.

While top-k processing over sliding window data streams has been the focus of several recent papers, [6, 24, 21, 12], all these works assume complete information over the arriving object's attributes or desired scores. In this model tuples arrive in *one* stream where all attributes of the object have been measured and projected in this tuple, or, different attributes arrive in *several* streams but with an *unlikely* assumption that all attributes of an object arrive at the same time in all streams. Therefore, in this model it is possible to calculate the exact score of each object with regard to a desired query instantly as the object arrives. On the contrary, we observe that in many streaming scenarios this is not the case.

For example consider an Internet Service Provider that monitors traffic at different routers in a network. Each router sends detailed traffic logs of different flows to a central server. These logs can for example consist of the source and destination IP addresses, the number and size of packets corresponding to them and a timestamp. The central server receives this information from different routers (i.e., in multiple streams) and can process various queries to better estimate and control traffic over the network or prevent security attacks. Top-k queries are commonly used in these applications, e.g., to *continuously report the top-20 flows with largest total size*. Since each flow can appear in several routers at different timestamps, the exact score of each flow (in this example sum of *size*s measured in all routers) can not be computed unless a tuple representing this flow arrives in *all* of the streams.

As another example, imagine a network of cameras on highways with detectors of the number plates, each reporting the observed vehicles in a stream to a central unit. A query would involve certain camera streams in a given area and the goal is to report based on (timestamp, platenumber, speed)-triples the fastest drivers. Naturally, a vehicle is captured by different cameras at different times, so its attributes

(i.e., speed at various points) arrive at the central unit with some delay. Not *all vehicles* are observed by *all cameras*, as there are various alternative paths. In similar scenarios where moving objects are tracked by stationed sensors, for example in supply chain management, the objects are incomplete by nature as they do not necessarily pass through all sensors, therefore do not explicitly possess values for all attributes. Unless all attributes of an object have been observed, or sufficient time has passed since its last observed attribute, it is not certain that the object's score will not change. Additionally, this incomplete view can be due to measurement noises, lossy transmissions, or delayed arrivals as a result of network characteristics, as opposed to the nature of attributes and monitored objects. Environmental monitoring scenarios can serve as an example to this.

The sliding window model, adds up to this uncertainty, as different attributes of the same object may be valid for different amounts of time. As a result, a system favoring exact results should maintain several aggregation scores, with regard to different expiration times of its attributes. This, significantly changes the properties of the system, especially with regard to the necessary storage.

Motivated by these scenarios, where complete observations are rare, we choose incomplete streams as our underlying streaming model. We show that extensions of existing approaches to the uncertain score scenario do not perform well in practice, particularly with regard to storage which poses fundamental restrictions in stream processing engines.

1.1 Problem Statement and System Model

Let $O = \{p, q, ...\}$ be the set of objects we are monitoring. We consider d incoming streams $s_1, s_2, ..., s_d$, each corresponding to one attribute of the objects. Each stream s_i contains tuples of the form $\langle p.id, p.value(i), p.t_i \rangle$, where $p.id$ uniquely identifies p, $p.value(i)$ is the value of attribute i of p and $p.t_i$ is the arrival time of this tuple. We assume all attribute values $p.value(i)$ are normalized to $[0, 1]$. Objects do not necessarily appear in all streams and can arrive in different streams at different timestamps. Tuples continuously stream in and they are considered *valid* while they belong to a sliding window W. Sliding windows can be either count or time based. Our algorithms can naturally handle both kinds of sliding windows. For simplicity, we assume each object appears in each stream at most once. We later show how our methods are extendable to the case where this assumption is not necessary.

At each instant of time, we can calculate the scores of *valid* objects given a monotonically increasing aggregation function: $score(p) = f(p.value(1), ..., p.value(d))$. An object is considered valid if it has at least one valid attribute. In calculating $score(p)$, the value of an unseen or expired attribute is considered to be the smallest possible, in our case where values are normalized to $[0, 1]$ this is 0. The score of an object can increase over time as some of its unseen attributes arrive or it can decrease as some of its attributes expire.

Given the above, we are interested in continuous monitoring of the top-k valid objects with regard to their scores. We consider the tradeoff of minimizing storage consumption and accuracy of top-k results. Our goal is to sum-up two contradictory conditions: keep less than necessary, but maintain the accuracy of the top-k results.

Figure 1 shows an example of our model in the network monitoring scenario. In this case, objects are *flows*, therefore id can be the concatenation of the source and destination IP addresses. Attribute i of each flow is its size measured at router i. Objects do not arrive in the same order in different streams. The aggregation function is sum and a time based window of size 100 is assumed. As can be seen the score of objects increase and decrease over time as new attributes arrive or old ones expire.

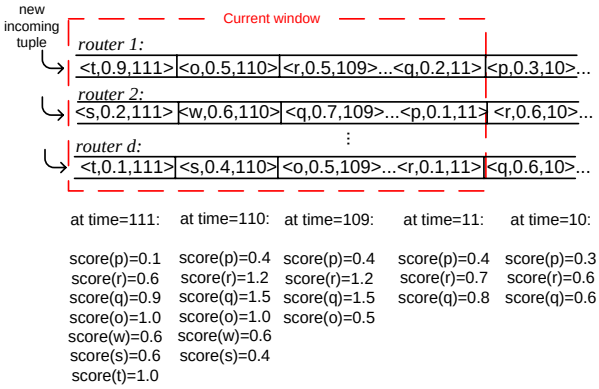


Figure 1: An example: streams are generated at various routers and flows are the monitored objects

1.2 Contribution and Outline

In this paper we make the following contributions.

(*i*) We consider continuous monitoring of aggregation queries over incomplete data streams in a sliding window model which has not been, to our knowledge, considered in existing works.

(*ii*) We show how the notion of *dominance check* should be changed in this case to enable retaining only those objects which are necessary for providing exact results. We theoretically show that the necessary memory usage increases from previously logarithmic (in window size) to linear, compared to the case where exact score calculations are possible.

(*iii*) Leveraging the statistics collected from the data streams and their correlations, we propose an approximate algorithm which allows for early-drops of objects with little loss in the accuracy of the returned results.

(*iv*) We report on results from a comprehensive performance and accuracy study using real-world and synthetic datasets. Our main focus is on the accuracy and storage consumptions of our algorithms.

Section 2 presents the related work. Section 3 presents two exact algorithms and discusses the new notion of dominance in order to enable retaining only the necessary objects for providing exact results. The increase in memory consumption is also discussed in this section which motivates our approximate algorithm. Section 4 presents our approximate approach to deal with partial knowledge imposed by the incomplete data streams. Section 5 shows how our proposed algorithms can be extendable to the case where each object may appear several times in each stream. Section 6 presents the experimental evaluation. Section 7 concludes the paper and gives an outlook on ongoing and future work.

2. RELATED WORK

Motivated by the growing need to develop new techniques to cope with high-speed data streams, there has been a wealth of research on stream processing over the past few years (for comprehensive surveys see [2, 26]). Early works mostly consider one-pass algorithms in limited space over

the whole stream where all tuples are considered valid at all times. In *sampling* based methods, a sample of the whole stream is maintained to provide fast approximate answers to queries such as distinct value estimation [17] or keeping statistical summaries of the stream such as histograms [18]. FM sketches [16] which are among hashing sketches are used for distinct item counting over streams. AMS [1] sketches, based on linear projections, are used to estimate the self-join size of one stream or join-size of several streams. We use a variant of FM sketches to capture the necessary statistics for our approximate algorithm. Reporting on *quantiles* or *heavy hitters* in streams is another important problem studied in the literature: Solutions often apply techniques such as the mentioned AMS sketches, sampling, or more recently group testing, see for example [8, 10] and the references within. The Exponential Histogram technique [13] and deterministic waves [19] maintain stream statistics such as bit counting or sum over a sliding window in streams. In this model only tuples within a sliding window are considered valid. In a more general model, Cormode et al. [30, 9] consider time decaying aggregates in out-of-order streams. This asynchronous model of data streams is the closest to ours, since tuples' arrival time does not necessarily follow tuples' observation time (or time of birth). However they assume objects have one attribute and consider aggregates such as heavy hitters or quantiles. We consider multi-dimensional objects with an arbitrary monotonically increasing aggregation function.

Top-k query evaluation over data streams has been a hot topic in the previous years, too. Mouratidis et al. [24] maintain a skyline [7] which represents the possible top-k candidates, i.e., those items that have, due to the sliding window (timeouts) still a chance to get in the top-k at some point. They focus on efficiency of evaluations while we consider memory limitations. They assume all attributes of a data point are seen together which means exact score calculation is possible. In a more general setting, [12] proposes indexing methods for answering adhoc top-k queries utilizing arrangements. Complete information over object attributes is also assumed here. Also related to our work is continuous k nearest neighbors (kNN) queries on data streams which is considered in [22, 6]. Koudas et al. present Disc [22] for indexing high dimensional points using space filling curves to give approximate answers to kNN queries. On the other hand [6] considers a fixed number of queries and indices queries instead of incoming tuples in a structure similar to VA files [31] to continuously provide exact answers in a sliding window model. They also maintain a skyline to decide which tuples should be kept, therefore minimizing the needed storage. Note that a kNN query can be regarded as top-k query when the query point is fixed. They also assume complete knowledge of the score once a tuple arrives in the stream. In [28] the authors consider the top-k problem in a publish/subscribe context. The goal is to return the top-k most relevant publication to each subscriber. Similar to other approaches, they assume each publication contains the complete information to calculate its score and rank against each subscription. Babcock and Olston [3] consider a distributed setting and aim at minimizing the network overhead such that stream sources do not report all their incoming tuples to the processing unit. Different to that, we consider a centralized setting where all streams arrive at the same node for evaluation and the goal is to minimize storage instead. Jin et al. [21] consider top-k queries on uncertain streams. The idea is to maintain a compact set which is necessary and sufficient for answering a top-k query and update it as the window slides. In this work tuples are associated with existential probabilities, however, attributes of objects are certain and scores can be determined once a tuple is seen.

Another line of related research to our problem is load shedding and approximate join processing in data streams. Das et al. [11] consider approximate join processing in a sliding window model with limited resources. They propose an optimal offline algorithm for evicting tuples when fixed amount of memory is available and two online algorithms, *PROB* which leverages the probability distribution of objects appearing in streams, and *LIFE* which considers the lifetime of objects as well. We also focus on memory limitations, however we aim at maximizing the accuracy (precision) of a given top-k query as opposed to maximizing the number of generated results (*MAX-subset* measure). [29] considers the same problem but also introduces the *age-based* model in which objects do not repeat in streams, therefore *PROB* is not applicable. [32] generalizes the setting to stochastic streams. In [4] the authors introduce the notion of k-constraints and exploit that to reduce the run-time state of continuous queries. Li et al. [23] exploit reference locality to reduce the cost of stream operators, such as joins. Processing multi-joins in a sliding window is considered in [20] and adaptations of nested loop and hash joins are proposed and evaluated.

3. EXACT ALGORITHMS

We describe two exact algorithms for evaluating top-k queries over incomplete data streams. The first algorithm is an adaptation of the threshold sorted list mechanism used in traditional databases. The idea is to store the tuples of each stream in a sorted list and use the Threshold Algorithm (TA)[15] in order to evaluate top-k queries. The second algorithm builds on early aggregation of tuples as they stream in. It enables pruning of tuples which do not have a chance of becoming a top-k and can be safely dropped.

3.1 Sorted List Algorithm (SLA)

We assume all valid tuples are sorted in a first-in-first-out list. This provides an efficient mechanism for evicting expired tuples. Newly arriving tuples in each stream are placed at the head of this list and old tuples are dropped from the tail. Note that this is applicable to both count-based and time-based sliding windows. In addition to this list we maintain d sorted lists, one per stream (i.e., for each attribute). Upon receiving $\langle p.id, p.value(i), p.t\rangle$ from the i^{th} stream, $(p.id, p.value(i))$ is inserted in the i^{th} list which is sorted based on the *value* field. When a tuple expires, it is also removed from the sorted list it belongs to.

In order to evaluate a top-k query, the TA algorithm is used. Similar to [24] we use the technique of [33] for efficient maintenance of the top-k results in face of frequent insertions/deletions: we maintain $k_{max} > k$ entries for a top-k query in order to reduce re-computations. When a new tuple $\langle p.id, p.value(i), p.t\rangle$ arrives, p's new score is computed by random access to all other attribute lists. The result list is updated accordingly: if p's score is higher than the least score in this list, p is inserted to the result view. Similarly whenever a tuple expires, the score of its corresponding object decreases. If this object was part of the result view, the result view is updated. Once the size of the result view falls below k the TA algorithm is called to recompute the top-k results.

3.2 Early Aggregation Algorithm (EAA)

In monitoring scenarios where tuples stream in with very high rate, it is often impossible to save all incoming data as in the previous approach. Most of the tuples are not interesting and can be dropped. Therefore with limited resources, only objects which have a chance of becoming a top-k result

should be saved. These include those objects which at their time of arrival are not among the top-k results of the query, but over time as some objects expire qualify as top-k results. This idea is used in [6, 24, 25] to devise efficient methods for processing fixed kNN and top-k queries. In the following we first describe how such objects are identified when instant score evaluation is possible and then show how this can be extended to our model.

If the data arrives in a way that allows for a full evaluation instantly, i.e., all attributes of an object arrive at the same time, the score of each object can be calculated with regard to a fixed query (i.e., a given aggregation function), and this score does not change during the object's life time. Now each object p can be regarded as a point in the score/time space represented by two attributes, its score: $p.score$, and time of arrival: $p.time$. A point p is said to *dominate* another point q if and only if p is preferable to q in all attributes. For our problem this translates to $p.score > q.score$ and $p.time > q.time$. A point is in the *k-skyband* of the dataset if it is dominated by less than k other points in the dataset. *Skyline* constitutes the case for $k = 1$. It is easy to observe, and has been formally proven in [6, 24, 25], that it is sufficient to save only points in the k-skyband over these two attributes of the dataset to answer top-k or kNN queries. This is also the minimum number of points required to answer the top-k query *accurately*.

It should be noted that although the objects are originally d dimensional, the skyband is calculated over only two attributes (score, which is an aggregation of the initial d attributes, and time of arrival).

The shortcoming of these approaches is that they assume the data to arrive in a way that allows for a full evaluation instantly. In case of multiple non-synchronized data streams, the score of an object may change over time as more of its attributes are observed or some expire. As a result, the classic dominance check can not be used to drop objects. However, since we are assuming a monotone aggregation function we can calculate the highest score an object can acquire. This is a standard concept in top-k query processing [15] that allows for pruning items based on their upper bound score. This upper bound can be used in performing a conservative dominance check, which considers possible increases in the score of an object. But as some attributes of an object expire before the rest, its score can also decrease over time, which means the dominance relation between two objects may change over time. We solve this problem by creating several instances of the same object with different expiring times in such a way that the score of these instances can only increase over time. In the following we first explain how these aggregated instances are created and then show how the dominance check condition should be changed in this case such that all objects having a chance of being among the top-k are still in the k-skyband.

3.2.1 Instance Creation

Assume the attributes of an object p have been observed in a subset $I = \{i_1, i_2, ..i_r\} \subseteq \{1, .., d\}$ of streams and w.l.o.g. $p.t_{i_1} < p.t_{i_2} < ... < p.t_{i_r}$. We create r aggregated instances of p in the following way:

$$p^j = \langle p.id, p^j.currentscore, p^j.bestscore, p^j.t \rangle$$

where $p^j.currentscore = f(v_w(1), ..., v_w(d))$ and

$$v_w(x) = \begin{cases} p.value(x) & x \in I \wedge p.t_x \geq p.t_{i_j} \\ 0 & otherwise \end{cases}$$

$p^j.bestscore$ represents the highest score this object can get during its life time and is calculated as $p^j.bestscore = f(v_b(1), ..., v_b(d))$ and

$$v_b(x) = \begin{cases} p.value(x) & x \in I \wedge p.t_x \geq p.t_{i_j} \\ 0 & x \in I \wedge p.t_x < p.t_{i_j} \\ 1 & otherwise \end{cases}$$

where the second condition is based on the fact that each object is observed once per stream. The arrival time of this instance is set to $p^j.t = p.t_{i_j}$, which is the earliest time among all observed attributes which are considered for calculating its *currentscore*. This ensure that *currentscore* can only increase during this instance's life time. If a new attribute i_{r+1} of p is observed, the above instances are updated accordingly: $v_w(i_{r+1}) = v_b(i_{r+1}) = p.value(i_{r+1})$, which results in a larger value for *currentscore* and a smaller one for *bestscore*. As a result, the interval of [*currentscore, bestscore*] can only shrink over time for each aggregated instance. Also a new instance p^{r+1} is created accordingly. Figure 2 shows an example of object instances and their score intervals in case of $W = 100$. p's second attribute is observed earliest at $t = 1$, where p^1 is created. When p's third attribute arrives in stream 3, p^1's scores are updated accordingly and p^2 is created. As can be seen the score interval decreases: $p^1.currentscore$ increases while $p^1.bestscore$ decreases. Upon observing the first attribute, the evaluation engine has full information over the score of this object. p^1 and p^2's scores are updated and p^3 is created.

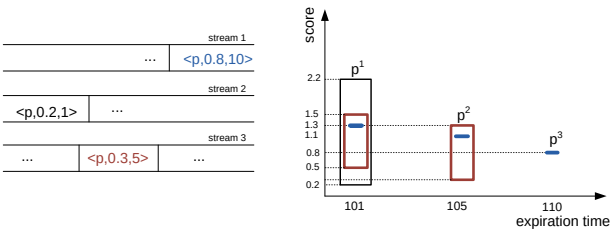


Figure 2: An example of object instance creation

The introduction of currentscore and bestscore follows the scoring introduced by Fagin et al. in [15], in particular in their NRA algorithm that uses only sequential scans of the input data. However, in NRA the best possible score to be considered when calculating the upper bound score (bestscore) decreases with ongoing sequential scan, as the input data per attribute is considered to decrease in score, which would be an unrealistic assumption in the streaming data scenario we consider in this current work.

3.2.2 Interval Dominance

Given a set S of such object instances we are interested in keeping only those which have a chance of becoming a top-k in future. We define the interval dominance as follows.

DEFINITION 1. *[Interval Dominance] Given two object instances p^i and q^j we say p^i dominates q^j and denote this by $p^i \succcurlyeq q^j$ iff $p^i.t > q^j.t$ and $p^i.currentscore > q^j.bestscore$.*

The **k dominance set** of S, denoted as S_k, consists of all instances which are dominated by less than k other distinct object instances. Two object instances are said to be *distinct* if they possess non equal *id*s. Interval Dominance has the following desirable properties:

P1-Persistence: Dominance is persistent during an instance's life time: if $p^i \succcurlyeq q^j$ at time τ, $p^i \succcurlyeq q^j$ for all times $t > \tau$. This is because $p^i.currentscore_\tau > q^j.bestscore_\tau$ and *currentscore* can only increase over time while *bestscore*

can only decrease, so $p^i.currentscore_t > q^j.bestscore_t$ for $t > \tau$, where $p^i.bestscore_t$ denotes p^i's bestscore at time t. Also the *time* attribute t of instance objects does not change over time.

P2-Transitivity: If $p^i \succcurlyeq q^j$ and $q^j \succcurlyeq r^k$ then $p^i \succcurlyeq r^k$. The definition of interval dominance directly results in this property. As a result, if $p^i \succcurlyeq q^j$ and $p^i \notin S_k$ then also $q^j \notin S_k$.

THEOREM 1. *Assume S is the set of all valid instances. It is necessary and sufficient to retain the object instances in S_k to provide exact top-k results.*

Proof. We first show that if $p^i \notin S_k$ then p^i cannot be a top-k result. If $p^i \notin S_k$ then there are at least k distinct instances which dominate p^i. From the interval dominance definition it follows that at least k distinct instances exist which have higher *currentscore* than the best score p^i can ever get and live longer than p^i. Since dominance is persistence, p^i cannot be part of top-k anytime during its life time, as there are at least k preferred instances all its life. For showing the necessity of keeping S_k, assume $p^i \in S_k$. We describe a case where p^i is part of top-k results. There are at most $k-1$ other distinct objects which live longer than p^i and their *currentscore* is higher than $p^i.bestscore$. Let τ be the time when all instances which have better scores than p^i and are older, expire. Assume no new tuples arrive at any of the streams until τ. p^i will be a top-k result at τ. □

3.2.3 Structure and Maintenance

Our *Early Aggregation Algorithm* (EAA) is based on maintaining the k dominance set of valid instances as new tuples arrive and old ones expire. We keep the list of valid objects in a hashtable based on their *id*s. As described in Section 3.2.1, for each valid object we have r aggregated instances where r is the number of valid attributes with distinct arrival times. We keep a sorted list of these instances based on their *time* attribute and keep pointers to them from the corresponding hashtable entry. Upon arrival of a new tuple $\langle p.id, p.value(i), p.t\rangle$ from the i^{th} stream, we first look up $p.id$ in the validity hashtable. If $p.id$ exists in the hashtable, a number of other attributes of p have been observed before. Assume r distinct attributes were observed for p before the new tuple. We have created r instances of p as described previously. These instances are updated by taking $p.value(i)$ into account instead of 0 in calculating *currentscore* or 1 in calculating *bestscore*. We also create a new instance p^{r+1} as described in Section 3.2.1 and insert it in the *time* sorted list. A pointer to this new object is also kept in the validity hashtable in the corresponding entry.

So far we described the insertion of new tuples and creation/modification of object instances. In order to prune unnecessary instances, we maintain the number of objects which dominate each instance. We keep a dominance counter *dc* for each instance, which shows the number of distinct instances that dominate this instance, and maintain this value during the instance's updates. To facilitate maintaining *dc* and avoid scanning all instances each time an update occurs, we keep also two sorted lists of score values: one for *bestscores* denoted by *bsorted* and one for *currentscores* shown by *csorted*. For an object instance p^i, we create two entries of the form $\langle id, v\rangle$ with the following values: $\langle p^i.id, p^i.currentscore\rangle$ and $\langle p^i.id, p^i.bestscore\rangle$ and insert them in the *csorted* and *bsorted* lists respectively. Figure 3 illustrates the data structures we use.

The dominance counter of each instance can be calculated utilizing the three described lists. When p^1 is created, it has the last expiration time, as its time attribute t is set

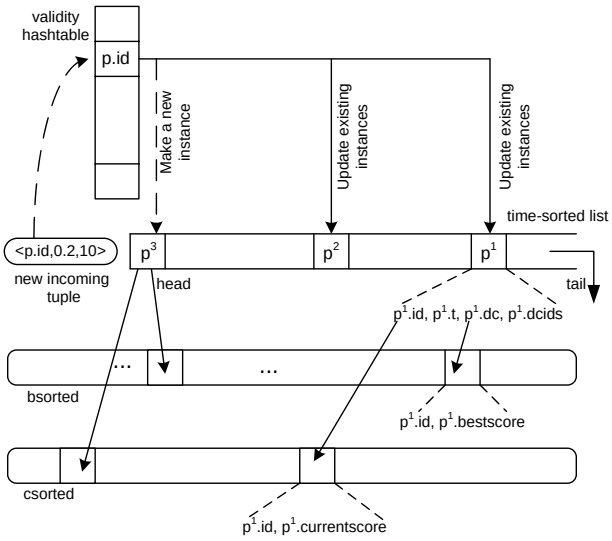


Figure 3: Structures for maintaining k-skyband

to the arrival time of the last received tuple. So it can not be dominated by any other instances. However it dominates those instances whose *bestscore* is less than $p^1.currentscore$. These instances are easily accessed by first identifying the position of $p^i.currentscore$ in *bsorted*, which is possible by a binary search in that list, and from that point scanning all entries in a descending order and increasing their *dc* values. The steps necessary for this operation are shown in Algorithm 1. For each instance object we also keep a list of *id*s of instances which dominate this one: *dcids* and use it to avoid recounting non-distinct dominating instances.

Whenever an object instance p^i is modified (due to the observation of a new attribute of its corresponding object) it may dominate more instances, as its *currentscore* increases, at the same time it may be dominated by more instances, since its *bestscore* has decreased. The two score sorted lists are utilized to identify and update these involved instances. We only need to check an instance q^j, if $q^j.bestscore$ is larger than p^i's old *currentscore* and is smaller than its new value. Such instances are easily identified by looking up p^i's old *currentscore* in *bsorted* and performing a scan from the found position in an ascending order until the value of the entry is larger than $p^i.currentscore$. We perform similarly for p^i's *bestscore*, looking its old value up in *csorted* and scanning in descending order until the value of the entry is smaller than $p^i.bestscore$. Algorithm 2 shows the details.

The object instances whose dominance counter *dc* hits k can be safely discarded. They are removed from the three sorted lists and their corresponding pointer is also eliminated from the validity hashtable. Note that removal of such instances does not effect the dominance counters of other instances. This is due to the *transitivity* property of dominance: all those instance which were dominated by p^i are also dominated by instances which dominate p^i. As a result if p^i is removed because it is dominated by k distinct instances, all other instances which were dominated by p^i have been removed before (because they were dominated by k instances earlier). Also objects which expire fall off the tail of the time sorted list and are also removed from the two score sorted lists and the validity hashtable. Their removal also doesn't effect other instances, as all instances which they could have dominated have expired before.

The top-k elements are identified by scanning *csorted* list in a descending fashion, until *currentscore*s of k distinct in-

```
Input: p.id,p.value(i),p.t,tsorted,csorted,bsorted
r=1;
if validityhashtable.contains(p.id) then
    forall E=validityhashtable.entry do
        update(E,p.value(i),tsorted,csorted,bsorted) ;
        r++;
    end
end
p^r=createInstance(p.id,p.value(i),p.t,r);
tsorted.append(p^r,0,null);
csorted.insert(p.id,p^r.currentscore);
bsorted.insert(p.id,p^r.bestscore);
foreach object instance q in bsorted do
    if q.bestscore < p^r.currentscore and p^r.id ∉ q.dcids
    then
        q.dc++;
        q.dcids.add(p.id);
    end
end
```

Algorithm 1: Algorithm for inserting a new tuple

stances are identified. Similar to the Sorted List Algorithm (SLA) we can materialize k_{max} instances to avoid the frequency of evaluations, though due to the availability of the score sorted list they are much faster than the evaluations done in the SLA algorithm.

3.2.4 On Skies and Linear Growth

It has been shown in [5] that for a set of d dimensional objects of size n, under uniform and independent attribute selection, the skyline size is $O((ln\ n)^{d-1})$. As a result, in case of synchronous streams when exact score computation is possible, the number of objects which should be stored to provide exact top-k evaluation, is $O((ln\ n))$, as the skyline is computed over two attributes: score and time. In the following we show that with the interval dominance check, the size of the dominance set grows linearly in the size of the original set. As a result, although the expected storage consumption of EAA is less than SLA, still, it is linear in size of W. As we will see later, this gives the basis for our proposed approximate algorithm which aims at minimizing the memory consumption.

LEMMA 1. *[Linear Growth] Given a set S consisting of n elements of the form $\langle currentscore_i, bestscore_i, time_i\rangle$, where $currentscore_i$ are chosen uniformly at random from $[0,1]$ and $bestscore_i = min(currentscore_i + \epsilon, 1)$. The* **expected size of the dominance set** *S^* is in $\Omega(n)$ where the constant depends on ϵ.*

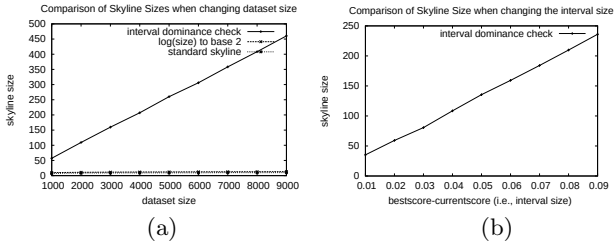
Proof. Let $X_1, X_2, ..., X_n$ be iid random variables following a probability distribution function $f(x)$, denoting the n *currentscore*'s we are considering. Note that the iid assumption is valid, since we are not considering several instances of the same object, but objects with distinct *id*s. We can re-order them

$$S_1 \leq S_2 \leq ... \leq S_n$$

where S_κ is called the κ-th order statistic. We are interested in the maximum, which is the S_n-th order statistic. Since the difference between $currentscore_i$ and $bestcore_i$ is at most ϵ, points which are not dominated by any other point are those whose *bestscore* is at least $max_{1\leq i\leq n} currentscore_i$ which can be estimated by $E[S_n]$. Hence, we have the following lower bound for the expected size of the dominance set:

```
Input: E,p.value(i),tsorted,csorted,bsorted
p^r=E.getInstance;
oldcurrentscore= p^r.currentscore;
oldbestscore = p^r.bestscore;
p^r.updateCurrentscore(p.value(i));
p^r.updateBestscore(p.value(i));
foreach object instance q^j in bsorted do
    if
    oldcurrentscore < q^j.bestscore < p^r.currentscore
    then
        if p^r.t > q.t and p^r.id ∉ q^j.dcids then
            q^j.dc++;
            q^j.dcids.add(p^r.id);
        end
    end
end
foreach object instance q^j in csorted do
    if p^r.bestscore < q^j.currentscore < oldbestscore
    then
        if q^j.t > p^r.t and q^j.id ∉ p^r.dcids then
            p^r.dc++;
            p^r.dcids.add(q^j.id);
        end
    end
end
```

Algorithm 2: Algorithm for updating an existing instance object



(a) (b)

Figure 4: Skyline size when changing (a) data set size (b) interval size

$$E[|S^*|] \geq n * \int_{E[S_n]-\epsilon}^{E[S_n]} f(x)dx$$

This is a lower bound as some objects whose *bestscore* is smaller than S_n are part of the dominance set due to their *time* attribute. In case of standard uniform distribution $f(x) = 1$ and $E[|S^*|] \geq n * \epsilon$ and is independent of $E[S_n]$. □

Figure 4 reports on the results of a simple preliminary experiment to analyze the behavior of the skyline size with changing dataset size (a) and changing the distance between best and current scores (i.e., the interval size) (b). As can be seen the size of the standard skyline, where only the current scores are taken into account, is logarithmic in the dataset size. The size of skyline points with interval dominance check grows linearly with the number of data points. We also observe the linear growth with increasing the interval size. The experiment were conducted 100 times and we report on the average sizes values.

4. SCORE ESTIMATION BASED ON APPEARANCE CORRELATION

As seen in the previous section, the number of objects which should be retained in order to provide exact answers to a top-k query in our streaming model is linear in the size of the sliding window. Such storage may not be available, especially when objects are stored in main memory to enable fast and online evaluation of the top-k query. Returning approximate query answers instead of exact answers is a graceful way of dealing with limited resources and has been applied in many streaming problems before [11, 32, 29]. In this section we describe an algorithm which uses smaller storage at the expense of providing approximate results as opposed to exact ones. Our algorithm can be seen as a *semantic load shedding* algorithm which aims at maximizing the quality of the reported top-k. As in other classical work concerning top-k evaluation our measure of quality is the average precision obtained during all evaluations.

As shown in the proof of Lemma 1 and observed in Figure 4, the size of the dominance set is directly influenced by the score interval size: $(bestscore - currentscore)$. Our approximate scheme leverages this fact to provide better space efficiency. *bestscore* for each object instance gives an upper bound of the score this instance can acquire during its life time. Previously, we consider the maximum value for unseen attributes in calculating *bestscore*. However, in most streaming scenarios that are best captured under our non-synchronized multiple stream model such as network or vehicle monitoring, not all attributes of an object are observed. This conceptually means that the value considered for such attributes in evaluating the object's *bestscore* can be set as the minimum value an attribute can get. Pairs of streams usually have different correlations among them. For example in a vehicle monitoring scenario, cars which are observed in path i maybe more likely to be observed in path j than path k. We utilize the correlation of appearance between different streams in order to better estimate *bestscore*.

Given an object p we define the random variable $X_i \in \{0,1\}$ such that $X_i = 1$ if p has been observed in stream s_i in the current window. Similarly $X_i = 0$ if p has not been seen in stream s_i. We are interested in calculating the conditional probability of observing p in s_i if p has been already seen in stream s_j: $Pr(X_i = 1|X_j = 1)$. In the following we first show how this information is utilized and then present how it can be computed efficiently.

Given $I = \{i_1, i_2, ..i_r\} \subseteq \{1, .., d\}$ for an object p which indicates the streams where p has been observed in, for each $j \notin I$ assume we know the conditional probability of observing p in s_j : $Pr(X_j = 1|X_k = 1; k \in I)$. As before, $p^l.\widetilde{bestscore} = f(\tilde{v}_b(1), ..., \tilde{v}_b(d))$ where $\tilde{v}_b(y)$ is estimated as follows:

$$\tilde{v}_b(y) = \begin{cases} p.value(y) & y \in I \wedge p.t_y > p.t_{i_j} \\ 0 & y \in I \wedge p.t_y < p.t_{i_j} \\ Pr(X_y = 1|X_k = 1; k \in I) & otherwise \end{cases}$$

$\tilde{v}_b(y)$ is naturally smaller than or equal to $v_b(y)$. Since the aggregation function f is monotonically increasing $p^l.\widetilde{bestscore}$ in a lot of cases will be much smaller than $p^l.bestscore$. As a result of this the interval size would be smaller. Decreasing the interval size results in a smaller dominance set. At the same time, since we are estimating *bestscore* realistically the quality of results should remain high.

We now describe how the required correlation statistics can be computed. Since $Pr(X_i = 1|X_j = 1) = \frac{Pr(X_i=1 \wedge X_j=1)}{Pr(X_j=1)}$, $Pr(X_i = 1|X_j = 1)$ can be statistically estimated as the join size of s_i and s_j, $|s_i \bowtie s_j|$ weighted by $1/|s_j|$ where $|s_j|$ is the number of distinct elements observed in a window in stream s_j. This is because each object appears in each stream at most once during an active window. This can be generalized to the case where an object is seen in several streams and we are interested to know with what probability it will appear in the rest of the streams.

We use the FM sketches to estimate these probabilities. FM sketches were first proposed in [16] to probabilistically estimate the cardinality of a multiset M. These hash sketches use a pseudo-uniform hash function $h() : M \rightarrow [0, 1, \ldots, 2^L)$. In [14], Durand and Flajolet presented a similar algorithm (*super-LogLog counting*) which reduces the space complexity and relaxes the required statistical properties of the hash function.

Briefly, hash sketches work as follows. Let $\rho(y) : [0, 2^L) \rightarrow [0, L)$ be the position of the least significant (leftmost) 1-bit in the binary representation of y; that is,

$$\rho(y) = \min_{k \geq 0} bit(y, k) \neq 0,\ y > 0$$

and $\rho(0) = L$. $bit(y, k)$ denotes the k-th bit in the binary representation of y (bit-position 0 corresponds to the least significant bit). In order to estimate the number n of distinct elements in a multiset S we apply $\rho(h(d))$ to all $d \in S$ and record the least-significant 1-bits in a bitmap vector $B[0 \ldots L-1]$. Since $h()$ distributes values uniformly over $[0, 2^L)$, it follows that $P(\rho(h(d)) = k) = 2^{-k-1}$.

Thus, when counting elements in an n-item multiset, $B[0]$ will be set to 1 approximately $\frac{n}{2}$ times, $B[1]$ approximately $\frac{n}{4}$ times, etc. Hence, the quantity $R(S) = max_{d \in S}\rho(d)$ provides an estimation of the value of $\log_2 n$. Techniques which provably reduce the statistical estimation error typically rely on employing multiple bitmaps for each hash sketch, instead of only one. The overall estimation then is an averaging over the individual estimations produced using each bitmap.

In our case we are interested in the cardinality of the intersection of two streams s_i and s_j in a window. Since we know that the number of valid distinct elements in s_i is equal to the number of elements arriving in a window, we can use the fact that $|s_i \bowtie s_j| = |s_i \cup s_j| - (|s_i| + |s_j|)$ to estimate $|s_i \bowtie s_j|$. If we keep an FM sketch for each of s_i and s_j, the union of these two sketches can be used to estimate $|s_i \cup s_j|$. Since $(|s_i| + |s_j|)$ is known, it is then possible to estimate $|s_i \bowtie s_j|$. Note that we can similarly estimate $|s_i \bowtie s_j \bowtie s_k|$ or the cardinality of higher number of joins by only keeping FM sketches per stream.

We still need to take care of the sliding window factor: if an object o is seen in s_i at time t_i and in s_j at time t_j and t_i and t_j do not belong to the same window, o shouldn't appear in $s_i \bowtie s_j$. As mentioned in [13], the FM sketches can be adapted to estimate the number of distinct elements in a sliding window by associating a bitmap of size $O(logW)$ with each of the bits in the sketch. Whenever a bit is (re)set by an object in the stream, its associated timestamp is updated to that of the object. In this way when evaluating the number of distinct elements in the current window, only those bits which could have been set in the current window are considered. This increases the storage requirement of the FM sketch with a logarithmic factor.

5. MULTIPLE OCCURRENCES

So far in all our described algorithms we assumed that each object is observed in each stream at most once. In some real world scenarios, such as the flow example, this assumption may not hold. Here, we shortly describe how each algorithm is extendable to the case when reoccurrences

happen. In SLA, if an object reoccurs in a stream, its previously observed values are updated with aggregating the old seen value with the new one, and the newly seen value is also inserted in the corresponding sorted list. When performing the TA algorithm, a lookup in each list may return several instances, in such a case the largest value is used. When a tuple expires, its corresponding item in the sorted list is also removes. Assume the maximum number of reoccurrences in a stream is m. The currentscore and bestscore of an instance object are updated with regard to the number of times this object has been observed in a stream as well as in which streams it has been observed. In particular we use $m - m_i$ as the maximum value if an object has been observed m_i times in stream i (assuming *sum* for aggregating multiple occurrences). Also, similar to SLA, the values of existing observations are updated by aggregating a newly seen value and a new instance object is produced. Details are omitted due to space limitations but we mention that all calculations are straight forward by keeping the number of times an object has been observed. In our approximate algorithm, instead of using m, we estimate the maximum number of reoccurrences of an object by measuring the self join size of a stream using one of the existing approaches such as AMS sketches [1].

6. EXPERIMENTS

We have implemented a light-weight stream processing engine in Java 1.6 providing standard operators (project, select, join), data source wrappers that replay existing benchmark data, and data generators for synthetic data. The real world dataset is stored in an Oracle 11g database, the synthetic data is created on the fly.

6.1 Setup

6.1.1 Datasets

Synthetic Dataset: To obtain a better understanding of the impact of data characteristics on the performance and accuracy behavior of the algorithms under comparison, we first start by employing a data generator to produce streams with tunable inter-stream correlation. We use the following methodology: to generate n streams of each distinct items but with controlled correlation, the data generator keeps track of each item sent in one of the n streams separately. In addition, it keeps a sliding window of the last C values for each stream. To produce a new item to be inserted in a stream, the generator picks with probability ξ a value that is currently in the sliding windows of the other streams, and with probability $(1 - \xi)$ draws a fresh item from a random number generator, ignoring those items that have already been sent. The stream is not materialized in a database (or file); it is created on demand.

Real World Dataset: We use the WorldCup98 [1] dataset as our real dataset. This dataset consists of requests made to the 1998 World Cup Web site between April 30 and July 26, 1998. During this period the site received 1,352,804,107 requests. Each tuple consists of several fields. Each tuple has a *clientID* field which uniquely identifies the client which issued the request. The *server* field indicates which server handled the request and specifies the stream ID for us. *Size* shows the number of bytes in the response. The query asks for the clients who have downloaded the highest number of bytes in a given time frame. The data is stored in an Oracle database with a B+ tree index on (server, time ASC, client, bytes). The stream per server consists of multiple requests for each user (e.g., each HTTP request). We have implemented a stream operator that pre-aggregates these fine grained events in the following way. As is commonly done in this scenario, the preprocessing operator pre-aggregates chunks of a certain time period (δ_t) and sends the pre-aggregated values to the consuming stream operator. For each stream, client ids occurring in the same pre-aggregated stream (streams of pre-aggregated chunks) multiple times will be broken in different clients. This reflects in particular changes of IP addresses. These chunked streams are then processed by our top-k operators.

[1]http://ita.ee.lbl.gov/html/contrib/WorldCup.html

6.1.2 Algorithms under Comparison

We evaluate the following proposed algorithms:

SLA: This is the algorithm based on multiple sorted lists as explained in Section 3.1. It provides the baseline for assessing the quality of results as explained later.

EAA: Our proposed algorithm as described in Section 3.2. It uses the interval dominance check to discard instances which are not part of the k dominance set. Similar to SLA, EAA provides exact results and retains the least number of objects which can guarantee this.

approxAlgo: This is the approximate algorithm as proposed in Section 4. It estimates the best possible score of each instance more realistically based on stream inter correlations.

6.1.3 Measures of Interest

Memory Consumption: The number of retained objects is the dominant factor in storage. FM sketches generally consume very little space which is constant for each stream and negligible compared to the number of data points which should be stored. We report on the number of items retained for each of the algorithms under comparison as a measure of storage. Items are tuples in case of SLA, as we keep tuples separately without aggregating them. We ignore the k_{max} materialized aggregated results, as $k_{max} << W$. For EAA and approxAlgo, an item is an aggregated object instance.

Precision: We report on the precision, i.e., the number of relevant data points among returned top-k results as the effectiveness metric. The relevance is defined by the SLA method which keeps all valid tuples. Assume SLA reports A as the set of top-k results, this set is the ground truth. So if set B is returned as the top-k in another algorithm, this algorithm's precision is calculated as: $precision = \frac{|A \cap B|}{k}$

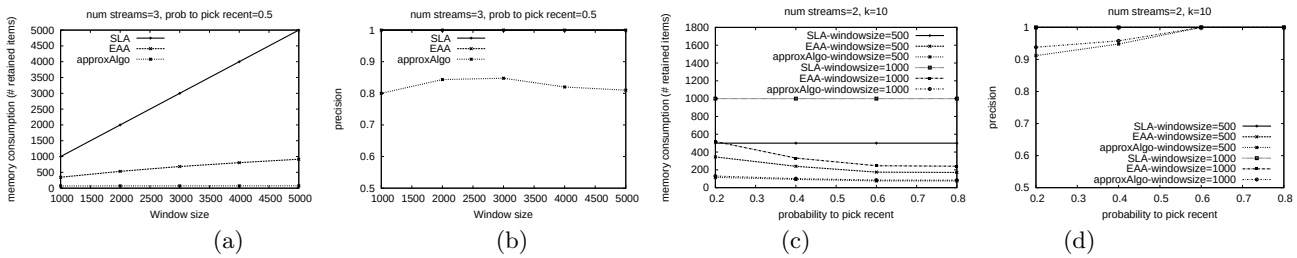
Relative Error: Since one of our proposed algorithms provides approximate results, we are interested in measuring the quality of the approximate results: how far from the true result is a returned element. Assume A is the ground truth set and B is the set of top-k results returned. The rank i element in set X is denoted by x_i. Then the relative error is calculated as: $\frac{1}{k} \sum_{i=1}^{k} |a_i - b_i|$

All reported measurements are averaged over 50 evaluations.

6.2 Experimental Results

6.2.1 Synthetic Dataset

In this section we describe the experimental results for the synthetic dataset. In the first set of experiments we investigate the effect of the sliding window size on memory consumption and precision. SLA keeps all valid tuples, therefore has a memory consumption equal to size of the sliding window. EAA decreases this by dropping objects which will never be part of the top-k. EAA's memory consumption is therefore the minimum required to guarantee exact



(a) (b) (c) (d)

Figure 5: synthetic dataset: (a) Memory consumption when varying the window size W (b) Precision when varying the window size W (c) Memory consumption when varying the probability to pick recent (d) Precision when varying the probability to pick recent

results. Figure 5(a) shows the results of this experiment. As expected, EAA has smaller memory consumption compared to SLA. *approxAlgo* reduces the memory significantly and shows very little increase as opposed to EAA which has a linear growth with the window size. Figure 5(b) presents the achieved precision in this case. EAA has precision 1 as it returns exact results. *approxAlgo* shows very small variations in precision (between 0.80 and 0.84) as the window size changes, indicating its robustness.

In Figures 5(c) and (d) we observe the effect of varying the correlation parameter between streams. The number of streams is fixed to 2 in this case and we show the results for different algorithms for window sizes 500 and 1000. SLA has memory consumption equal to the sliding window. As the correlation between streams increases (which is the result of increasing ξ) the memory consumption for EAA decreases. This is in accordance with our results from Lemma 1: by increasing ξ the average score interval decreases, as more objects are seen in all streams. *approxAlgo*'s memory consumption shows small decrease (from 130 to 84 for W=1000 and from 115 to 75 for W=500). Figure 5(d) reports the precision when changing ξ. EAA and SLA have precision 1. For *approxAlgo*, precision values increase as ξ increases, which is due to decreasing the uncertainty. However the variations between precision values are small for both window sizes indicating the good quality of our correlation estimation technique.

We present the results of changing the number of streams, which corresponds to dimensionality of the objects monitored in Table 1. We do not show the results for SLA, as its memory consumption is fixed (1000 which is equal to the window size). The memory consumption increases with increasing dimensionality for both EAA and *approxAlgo*. Again this is due to the increase in interval size. Precision and relative error are shown for *approxAlgo*. Note that these values are respectively 1 and 0 for EAA.

Table 2 reports the effect of parameter k. Increasing k clearly increases the storage as the number of elements which are not dominated by more than k elements naturally increases. This is apparent for both EAA and *approxAlgo*. We observe better precision for bigger values of k for *approxAlgo*: equal number of missing objects from the top-k results has less effect for larger values of k. Relative error shows the same trend (decreases as k increases).

6.2.2 Worldcup Dataset

Table 3 reports on the average performance for the worldcup dataset when varying the number of streams which represents the number of servers in this scenario. Similar to the synthetic dataset described above, our approximate algorithm (*algoApprox*) causes drastic performance gains in memory consumption with only minor losses in result accuracy, yielding a precision between 0.92 and 0.95.

streams	sizeEAA	sizeApprox	precApprox	errorApprox
3	345	69	0.80	0.0174
4	450	92	0.81	0.0242
6	495	115	0.76	0.0408

Table 1: Results when changing number of streams for the synthetic dataset. Windowsize =1000, k=10 and ξ =0.5

k	sizeEAA	sizeApprox	precApprox	errorApprox
10	345.3	69.26	0.80	0.01743
20	396.3	105.6	0.86	0.0103
50	450.0	198.7	0.96	0.0027
100	481.7	315.1	0.99	0.0009

Table 2: Results when changing k for the synthetic dataset. Windowsize =1000, numstr=3 and ξ=0.5

Similarly, Table 4 reports on the performance and accuracy numbers when changing the window size W, showing major decrease in memory consumption with minor losses in accuracy.

Given the above results, we observe the effectiveness of the proposed algorithms. The pruning of dominated items in our EAA algorithm successfully decreases the memory consumption while still guaranteeing exactness of the result set. It gives the basis for the optimization steps introduced by our approximate algorithm, that further decreases memory consumption. The penalties in result quality is almost negligible given the achieved reduction of items to keep in memory. The insights learned, in particular from running experiments using the more controllable synthetic dataset, is that the smaller the correlation of the involved streams the higher the impact of our approximate method.

7. CONCLUSION

We have addressed the problem of processing continuous top-k queries over multiple non-synchronized data streams where exact score computation is seldom possible. We have extended the notion of *dominance*, proposing an early aggregation scheme which enables efficient pruning of objects. However, we have theoretically shown and experimentally confirmed, that the necessary number of elements which have to be kept to guarantee exact results, grows linearly with the window size. Our approximate approach is based on the observation that the size of dominance set is a direct factor of the difference between best and current scores. We leverage the correlation appearance between different streams, which is usual in real world scenarios, to estimate bestscore in a less optimistic way than considering the best

streams	sizeEAA	sizeApprox	precApprox	errorApprox
2	217.46	80.1	0.92	0.0113
3	250.0	110.9	0.93	0.0136
4	250.0	139.8	0.95	0.0097

Table 3: Results when changing number of streams for the Worldcup dataset. Window = 500 and δ_t=5000ms and k=20

W	sizeEAA	sizeApprox	precApprox	errorApprox
500	217.46	80.1	0.923	0.0113
750	310.66	80.9	0.959	0.0082
1000	407.09	80.86	0.974	0.0084

Table 4: Results when changing the window size W for the Worldcup dataset. number of streams = 2 and δ_t=5000ms and k=20

possible scores for unseen attributes. As seen in the experiments, this method provides highly accurate results while reducing the number of retained objects dramatically. We will explore the design of efficient data structures tailored to maintaining the dominance set of a dynamic database in case of interval dominance check, as part of future work. Performance comparison of our approach to adaptation of existing progressive skylines such as [27] will be a starting point.

Acknowledgment This work is partially supported by FP7 EU Project OKKAM (contract no.ICT-215032).

8. REFERENCES

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.*, 64(3), 2002.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, 2002.

[3] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *SIGMOD Conference*, 2003.

[4] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3), 2004.

[5] Jon Louis Bentley, H. T. Kung, Mario Schkolnick, and Clark D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4), 1978.

[6] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, 2007.

[7] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, 2001.

[8] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1), 2004.

[9] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Time-decaying aggregates in out-of-order streams. In *PODS*, 2008.

[10] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *PODS*, 2003.

[11] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, 2003.

[12] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, 2007.

[13] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6), 2002.

[14] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *ESA*, 2003.

[15] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.

[16] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2), 1985.

[17] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, 2001.

[18] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.

[19] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, 2002.

[20] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[21] Cheqing Jin, Ke Yi, Lei Chen 0002, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1), 2008.

[22] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang 0003. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, 2004.

[23] Feifei Li, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE*, 2006.

[24] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, 2006.

[25] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6), 2007.

[26] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[27] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1), 2005.

[28] Kresimir Pripuzic, Ivana Podnar Zarko, and Karl Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS*, 2008.

[29] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, 2004.

[30] Srikanta Tirthapura, Bojian Xu, and Costas Busch. Sketching asynchronous streams over a sliding window. In *PODC*, 2006.

[31] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[32] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *SIGMOD Conference*, 2005.

[33] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *ICDE*, 2003.