A Thin Monitoring Layer for Top-k Aggregation Queries over a Database[*]

Foteini Alvanaki
Saarland University
Saarbrücken, Germany
alvanaki@mmci.uni-saarland.de

Sebastian Michel
Saarland University
Saarbrücken, Germany
smichel@mmci.uni-saarland.de

ABSTRACT

We consider the problem of maintaining a large set of top-k rankings over the update stream of a database. The rankings stem from top-k aggregation queries that are given a-priori based on the application scenario, for instance created along dimensions of a traditional data warehouse, for efficient automated reporting/detection of changes. The focus on only the top part of a ranking enables efficient buffering techniques to limit expensive interactions with the underlying database, while still guaranteeing correct top-k rankings at all times. This is achieved by employing conservative rank (score) estimates of previously unseen items that are not in the top-k result so far. The proposed family of maintenance algorithms further exploits the relations between the monitored rankings known from multi query optimisation. We present results of a preliminary experimental evaluation using TPC-H data to study the performance of our algorithms.

1. INTRODUCTION

Making sense out of massive amounts of data for business intelligence and similar tasks is crucial for decision making processes. Data is gathered in database systems that enable efficient ad-hoc queries or is periodically loaded into data warehouses that enable exploration of performance measures along multiple dimensions. As interesting data is not assumed to be static, a live (near realtime) monitoring of interesting aspects of data is required, hence posing the problem of continuously maintaining queries against arriving updates, for (semi)-automated reporting or reacting to specific events. To render the maintenance feasible in presence of large amounts of dynamic information, the objective is to focus only on the essence of information and to maintain it – instead of wasting resources in the aim of keeping everything up to date. Arguably the most natural way to condense large amounts of information into a conceivable form is the computation of rankings, where users can focus on the top-k portion of the results, that show an outstanding (in a good or bad sense) performance. Among the most prominent examples on how top-k results can be efficiently *computed* is the work on threshold queries for aggregation queries (c.f., [2, 5, 3]). In contrast to the computation task, we consider the *continuous maintenance* of top-k rankings using a thin monitoring layer of algorithms that aim at *avoiding recomputations* of the rankings inside the underlying database. We show that the restriction to the top-k portion of a ranking enables efficient ranking maintenance as the majority of updates is expected not to affect the top-k result. The queries/rankings we consider in this work stem from top-k OLAP queries over the various dimensions of the data, what is commonly inspected in a traditional data warehouse.

We introduce a generic framework that maintains the top-k results produced by the queries using multi-query optimisation and view maintenance techniques. This framework has two important characteristics, (i) It limits the interaction with the data back end, aiming at handling OLTP and top-k maintenance of OLAP queries at the same time. (ii) It guarantees exact top-k results at any time by conservatively estimating the score of instances that do not belong in the top of the ranking.

The paper is organised as follows. In the remainder of this section we introduce the data and query model and present the problem statement. In Section 2 we discuss related work. In Section 3 we describe our algorithms and present and discuss experimental results in Section 4. Section 5 concludes the paper and gives an outlook on ongoing and future work.

1.1 Query Model

We consider monitoring queries that stem from traditional OLAP-style aggregation queries, like the following over product and sales information, computing the aggregated sales volume (price*quantity) for groups of product type, brand and country.

```
SELECT P.type, P.brand, S.country,
   SUM(P.price*S.quantity)
FROM products P, sales S
WHERE P.id = S.pid
GROUP BY P.type, P.brand, S.country
ORDER BY SUM(P.price*S.quantity)
```

To compute the top-k product types, with the lowest revenue, of each brand in each country several top-k aggregate queries will stem from the above data cube query. These queries will look like the following

```
SELECT P.type, SUM(P.price*S.quantity)
FROM products P, sales S
WHERE P.id = S.pid
   AND P.brand=X AND S.country=Y
GROUP BY P.type
ORDER BY SUM(P.price*S.quantity)
LIMIT K
```

[*]This work has been supported by the Excellence Cluster on Multimodal Computing and Interaction (MMCI).

7th International Workshop on Ranking in Databases (DBRank) 2013
Riva del Garda, Trento, Italy.

The attribute P.type is the attribute for which the top-k results are tracked and we call it primary attribute. The attributes P.brand and S.country are used to create a filtering condition and we call them secondary attributes. The values X and Y are instances of these attributes. One such query is created for every combination of (P.brand, S.country) instances. The combinations having any number of attributes bound to ANY are considered also valid. ANY is a term used to symbolise that the attribute is not bound to a specific instance for example, the product types with the least revenue in each country that are of ANY brand.

1.2 Problem Statement

We assume a large number of top-k aggregate queries that produce a set of top-k rankings. Each ranking orders instances from a primary attribute according to a score computed as the aggregation of the values of other (one or more) attributes (numeric). The rankings are limiting their qualifying results using conditions created by binding secondary attributes to their values (see above).

We consider a stream of incoming updates that insert new tuples in the database. For simplicity we assume that each update contains information about all attributes involved in the rankings (primary, secondary and numeric). This assumption can be easily withdrawn by creating a preprocessing step which will take the update and will obtain the missing information from the database. Thus, the updates we consider are of the following form

$$(\textit{updated instance}, \textit{properties}, \textit{added value})$$

where the *updated instance* is the instance of the primary attribute, *properties* are the instances of the secondary attributes and *added value* is the change in the score of the updated instance imposed by the update.

Our goal is to maintain the exact top-k results for each ranking at any time by limiting the interaction with the database.

2. RELATED WORK

The work we consider in this paper is related to the problem of processing continuous top-k queries over a data stream (e.g. [6], [13]). However, work in this area considers that the aggregated score is computed using different attributes of a single tuple, and using sliding window semantics to determine its lifetime. In our work, the aggregated score is computed with respect to multiple previously seen tuples for a specific instance, and no sliding window semantics are assumed.

Zhang et al. [17] compute aggregated scores using the past tuples. They have multiple aggregate queries that differ only in the group-by condition. Our queries differ in the filtering condition. To some extent the two sets of queries are related to each other since the filtering condition can be transformed to a group by condition and vice versa. The main difference to our work is that they consider the accessed elements to be clustered in time. This allows them to track a small number of elements in each time period and forget about the others until they see them again.

Metwally et al. [11] compute the top-k elements in a stream by tracking N elements in a way that resembles our estimates algorithm. In this work, the position of an element in the ranking is determined by the number of occurrences of the element in the stream. In our work, the position of an element in the ranking is determined by aggregating a numeric value that accompanies the element in each of its occurrences. This results in elements with few occurrences having big aggregated scores and vice versa. In this case the method in [11] has no way to find the correct top-k results.

A lot of work also exists on how to evaluate top-k rankings accessing as few raw data as possible. Most of these works are based on the family of threshold algorithms by Fagin et al. [3]. Despite all the available work on efficiently creating top-k rankings (see the survey of Ilyas et al. [8] for an overview) the top-k aggregate rankings having group-by conditions have not attracted enough attention. To the best of our knowledge the only existing approaches are [9] and [16]. The main difference to our work is that they compute top-k rankings on demand while we maintain them continuously.

The works in [4] and [10] attempt to create systems that unify stream processing techniques with techniques from traditional database systems (row store and column store respectively). Both of these systems could be used in combination with our methods. Any of them could be used to replace the current RDBMS and then our algorithms could be applied on top.

Athanasoulis et al. in [1] develop a method to make possible online updates in a data warehouse. In contrast to our work their focus is on how to exploit new available hardware in order to achieve it.

Nagaraj et al. in [14] have a setting very similar to ours. They organise the queries in a tree structure very similar to the lattice we create. In this work they try to make queries execution faster by sharing the aggregate computations. Our top-k queries do not allow for such sharing since top-k results do not necessarily overlap.

Techniques to select the views to materialise in order to make query execution faster have been proposed in many works (e.g. [12], [7]). These works exploit the shared parts of the queries to be maintained. Although our queries are very similar with each other their top-k nature does not allow us to use these techniques.

3. APPROACH

We now present algorithms that maintain the top-k portion of the rankings of interest. Note that rankings are trivially maintained in the presence of updates performing insertions to tables that are not used by the corresponding aggregate queries by ignoring the updates. All other updates can potentially affect the top-k results and, thus, need to be handled by the algorithms.

3.1 Naive Approach

In a naive approach each ranking that receives an update checks whether the updated instance exists in its top-k ranking. In case it does, it updates its score. Otherwise, a query is issued to the database to obtain the aggregated value for the missing instance and decide whether it should enter the top-k results. Since the top-k results are assumed to contain a rather small portion of all instances most updates will need the execution of a query resulting in a significant degradation of the system's performance.

3.2 Estimates Algorithm

An optimisation can be achieved exploiting the top-k nature of the rankings. Each ranking needs to have exact scores

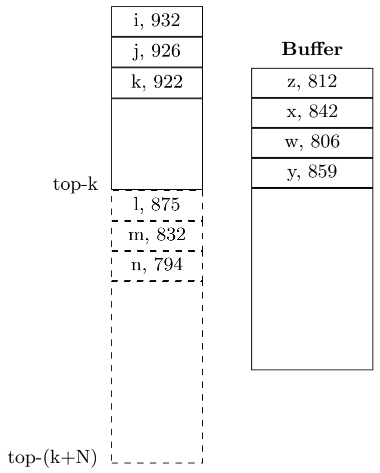


Figure 1: In-Memory structures: the actual top-(k+N) ranking (left) and the estimates for the previously unseen entities (right).

only for the top-k instances. Hence, the rest of them can have an estimated score. This is the idea in the Estimates Algorithm (EA). If a score for an updated instance is not already known, the algorithm assigns to it an initial estimated score, called basic score. To this basic score the additional amount contained in the update is added (aggregated). For the selection of the basic score each ranking tracks a number of N extra instances, i.e. each ranking stores the exact scores for top-(k+N) instances. The worst score of any instance in these N extra instances is selected to be the basic score. This conservative estimate assures that an instance belonging in the top-k part is never missed. When the estimated score (the basic score + the additional quantity) of an instance qualifies for the top-k results, its real score does not necessarily qualify too. In order for this to be verified a query that returns the real score of the instance is executed.

All instances being assigned an estimated score are stored in a structure, called Buffer, until their scores are verified against the database. This is done such that the rankings can consider the cumulative increase in an instances estimated score when it is updated multiple times. If the Buffer becomes full no more instances can be added to it and the algorithm falls back to the Naive Approach. To avoid that, when the Buffer is found to be full a query is issued that verifies all instances currently in it. After that, the Buffer is reset to empty allowing again the addition of new instances.

The in-memory structures necessary for the EA are shown in Figure 1. The idea of EA is described by the pseudocode of Algorithm 1. Every time we transfer an instance to the top-k or to the N extra instances another instance is removed to keep the sizes of these sets unchanged.

The reduction in queries achieved by EA is dependant on the score difference (gap) between the worst score of any instance in the top-k and the worst score of any instance in the N extra. The bigger this gap, the more updates an instance can get without qualifying for the top-k instances. It also depends on the number of instances with estimated scores that can be stored in memory. The more instances stored the fewer times the Buffer needs to be reset.

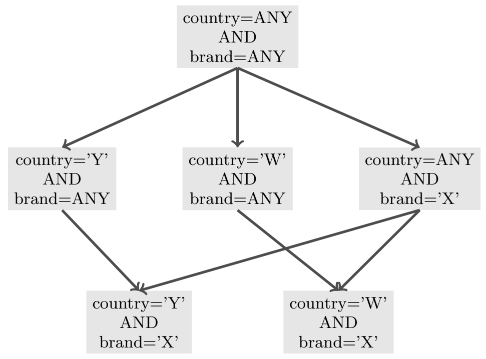
Algorithm 1: Estimates Algorithm (EA)

```
input: i: the instance that is affected
       change: the change in the instance's score

       if $i \in$ top-k then
           Update score
       else if $i \in$ Extra then
           Update score;
           if score exceeds top-k threshold then
               Transfer i to top-k
           end
       else if $i \in$ Buffer then
           Update score;
           if score exceeds top-k threshold then
               Execute query ;
               if score exceeds top-k threshold then
                   Transfer i to top-k
               else if score exceeds extra threshold then
                   Transfer i to N extra
               end
           end
       else
           Compute estimation;
           Add i to Buffer
       end
```

3.3 Groups Algorithm

The queries we examine, apart from focusing on the top-k portion of the results, have a special relation with each other. Bunches of them stem from the same data cube query i.e. they use the same primary and secondary attributes, and the same aggregation. These queries can be organised in a subgroups lattice according to the tuples qualifying to their filtering condition. For the products-revenue example query, introduced in Section 1.1, assuming that there are only two instances of countries, e.g. {'Y', 'W'} and a single instance of brand e.g. {'X'} the lattice will be the one in Figure 2.



country=ANY AND brand=ANY
country='Y' AND brand=ANY
country='W' AND brand=ANY
country=ANY AND brand='X'
country='Y' AND brand='X'
country='W' AND brand='X'

Figure 2: Subgroups lattice organising the top-k aggregate queries using the same primary attribute, and the attributes country and brand in the filtering condition

The basic characteristic of the queries organised in a lattice is that they share the same tuples. Each query lying in a join in the lattice is satisfied by the union of all tuples of the queries lying below it and each query lying in a meet point is satisfied by the intersection of the tuples satisfying queries lying above it. This partial order relation between the aggregate queries is an immediate consequence of the filtering conditions of the queries and can help decreasing the interaction with the underlying database.

Coming again to the products-revenue example, consider an update that causes an increase of x units in the quantity sold for product type t_u. The supremum ranking of the lattice (i.e. the ranking having the filtering condition *country=ANY AND brand=ANY*) is the first ranking to know about this update. The product type t_u does not exist in the top-(k+N) results and its estimated score qualifies for the top-k results. In this case a query is issued for the product type t_u. Instead of executing a query that will return the aggregated score of t_u the ranking asks for the tuples that compose the final score executing the following query

SELECT P.type, P.brand, S.country, P.price*S.quantity
FROM products P, sales S
WHERE P.id = S.pid **AND** P.type=t_u

The ranking uses the results to compute the score of t_u and forwards them to the rankings lying in lower levels in the lattice. Each ranking uses the qualifying received tuples to compute the score of the instance. This way a query can be saved and/or an instance can be removed from the Buffer. In case the ranking issuing the query has any filtering condition it will use it when issuing the query to filter any tuples not qualifying to it, these tuples are of no use to it or the rankings connected to it. If the aggregation function used by the rankings is distributive, e.g. **SUM** or **COUNT**, the ranking can query the score of each individual group instead of the single tuples saving the cost of computing the score in each ranking separately.

We call this algorithm Groups Algorithm (GA). The idea of GA is described by the pseudocode of Algorithm 2. The re-use of the results is also attempted when a query that verifies all instances in the Buffer is executed. This allows other rankings to free space in their Buffer without executing any queries.

Algorithm 2: Groups Algorithm (GA)

```
input: i: the instance that is affected
       change: the change in the instance's score
       tuples: the tuples comprising the various group for the
affected or removed from estimates instances as sent from an
upper level node

       Use sent tuples to remove from estimates as many
instances as possible
       if i ∈ top-k then
           Update score
       else if i ∈ Extra then
           Update score;
           if score exceeds top-k threshold then
               Transfer i to top-k
           end
       else if tuples ≠ ∅ then
           Compute score using sent tuples;
           if score exceeds top-k threshold then
               Transfer i to top-k
           else if score exceeds extra threshold then
               Transfer i to N extra
           end
       else if i ∈ Buffer then
           Update score;
           if score exceeds top-k threshold then
               Execute query ;
               if score exceeds top-k threshold then
                   Transfer i to top-k
               else if score exceeds extra threshold then
                   Transfer i to N extra
               end
           end
       else
           Compute estimation;
           Add i to Estimates
       end
```

4. EXPERIMENTS

We conducted experiments using the TPC-H dataset [15], We used *part.p_partkey* as the primary attribute and *customer.c_mktsegment*, *orders.o_orderpriority* and *region.r_name* as the secondary attributes. The selected aggregated function was *sum* and the numeric attribute over which we computed the scores was *lineitem.l_quantity*. In total 216 related queries have been created and organised in a lattice. The queries involved five relations. All methods are multi-threaded and implemented in Java 1.6.

We measure the average time needed to process each update and the number of queries executed. We divide the queries in two groups. In the first group belong the queries executed to verify an instance when its estimated score qualifies for the top-k results, called *Verification Queries*. In the second group belong the queries executed when the buffer is found to be full. We call these queries *Buffer Reset Queries*. The value that can be added in each update varies between 1 and 50.

EA and GA were tested under different configurations varying the gap between the worst score of any instance in the top-k instances and the worst score of any instance in the N extra instances, and the size of the Buffer. For the gap we tested the values 100% and 200%, where a value of 100% means that the gap is at least equal to the maximum value that can be added in any update, i.e. 50. The Buffer size in each ranking was set to 100, 500, 1000, 5000 and 10000.

In all experiments we performed 30,000 updates. We created two groups of updates. In the first group each update affects (i.e. increases the score) a random entity. In the second group the updates follow the 80-20 rule. According to this rule, 80% of the updates affect 20% of the instances. The goal of using two different sets of updates is to get an initial understanding of the kind of workloads our methods are more effective.

4.1 Updates following the 80-20 rule

In the left plot shown in Figure 4 we can see the change in the number of queries executed as the size of the Buffer and the gap change. As expected the number of the Buffer Reset queries decreases with the increase of the Buffer size. The GA executes fewer Buffer Reset queries compared to EA. This happens because in GA the results of the Buffer Reset queries are forwarded to all rankings lying in lower levels in the lattice. So, these rankings have the chance to remove instances from their Buffer before it becomes full. This difference is more prominent for the smaller Buffer size. For gap 200% (i.e. two times the maximum possible addition in an instances score) the Buffer Reset queries increase. This happens because as the gap increases fewer estimated scores qualify for the top-k instances, thus more instances with estimated scores are added to the Buffer and so the Buffer becomes full more often. In the right plot of Figure 4 the Verification queries are shown. In this plot we can verify that the estimated scores of the instances qualify for the top-k results more often when the gap is smaller. As for the Buffer Reset queries GA executes also fewer Verification queries compared to EA.

In the plot shown in Figure 3 we can see the average time needed to process each update. For gap 200% the fact that our rankings have this special relation does not seem to be important since both EA and GA have the (almost) same performance. For gap set to 100% GA is faster for small

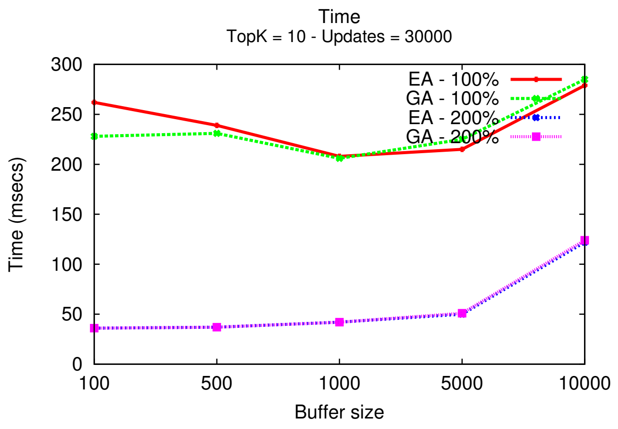


Figure 3: Runtime per update (80-20 Updates)

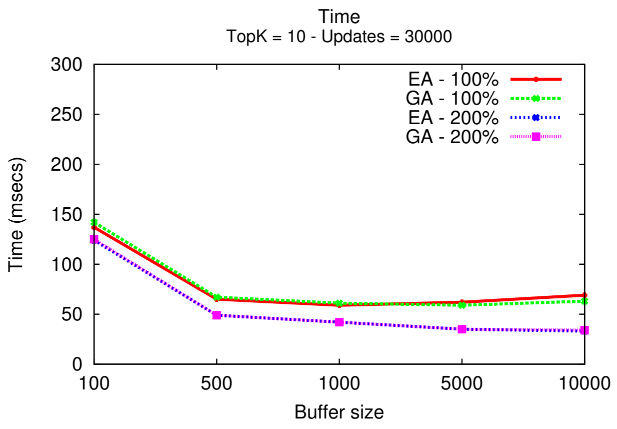


Figure 5: Runtime per update (Random Updates)

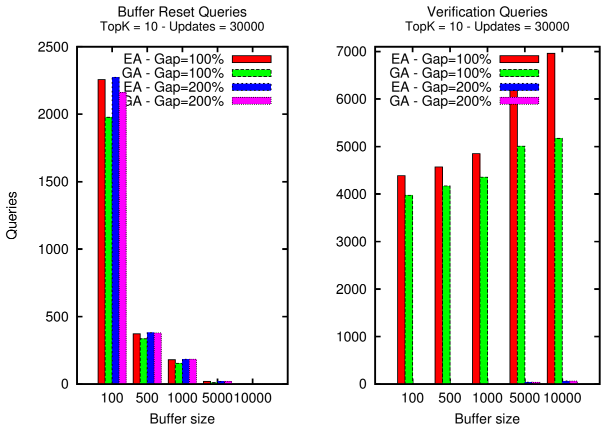


Figure 4: Buffer Reset and Verification Queries Queries (80-20 Updates)

Buffer size but gets slower as the Buffer size increases. This happens because an increase on the Buffer size results in a decrease in the number of Buffer Reset queries so, GA cannot benefit much from re-using the results of these queries to remove instances from the Buffers of other rankings. Additionally, the Verification queries take so little time to execute that the effort made to exchange the results between the rankings and compute the scores processing the received tuples overtakes the benefit of avoiding the execution of Verification queries. However, in a setting where the execution of Verifcation Queries is very slow the GA algorithm is expected to achieve better performance.

For the Naive Approach 239985 Verification queries are executed and the average time needed to process each update is more than 4 secs. These results are not included in the plots because they shadow the differences between our algorithms.

4.2 Random Updates

In the plots shown in Figure 6 we can see the number of queries executed when the updates are random. The less Verification queries observed, compared to the 80-20 case, are because when the updates are random the instances are less likely to be affected multiple times before the Buffer is reset. Thus, it is less likely an estimated score to grow enough to need the execution of a Verification query.

In the plot shown in Figure 5 we can see the average time needed to process each update. What is interesting to observe is that the time needed to process each update when the updates are random either decreases constantly as the Buffer size increases or has a slight increase for very big size of the Buffer. On the contrary, in the 80-20 case, the increase in time is more acute. Again the reason is that, when the updates are random, the instances have lower probability to be affected twice before re-setting the Buffer. Hence, keeping an instance in memory longer (bigger sizes of the Buffer) does not cause its estimated score to become big enough to qualify for the top-k results and to cause a Verification query. So, as long as the number of instances in the Buffer does not cause the Buffer Reset query to become very slow, increasing its size can only be beneficial, with respect to time. On the other hand, in the 80-20 updates, keeping instances longer in memory results in many Verification queries which deteriorate the runtime.

For the Naive Approach 239977 Verification queries are executed and the average time needed to process each update is more than 4 secs. Again, these results are not included in the plots so that the differences between our algorithms can be shown clearer.

4.3 Additional Instances

Of course the befit achieved is not for free. Both EA and GA store some additional instances. The number of these instances depends on the selection of the gap and the size of the Buffer. The plot in Figure 7 shows the change in the number of additional instances for the various sizes of the Buffer when the gap is set to 100% and 200%. In the plot it is obvious that setting the gap to 200% needs a lot more space. Since in both cases the number of additional instances stored to the Buffer is the same the difference is solely due to the increase in the gap. Especially in the very specific rankings (those binding all three secondary attributes to some instance) doubling the gap may result in having even five times more instances because of the small scores assigned to them. The line *ALL* shows the maximum number of instances (grouped using the filtering attributes) existing in the database. We use it as a baseline to give a notion of the extra storage cost.

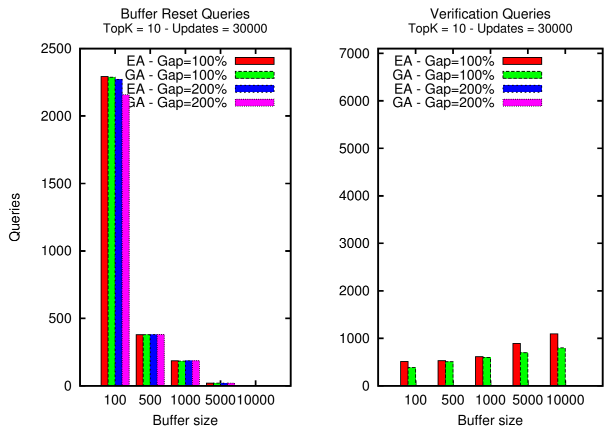


Figure 6: Buffer Reset and Verification Queries (Random Updates)

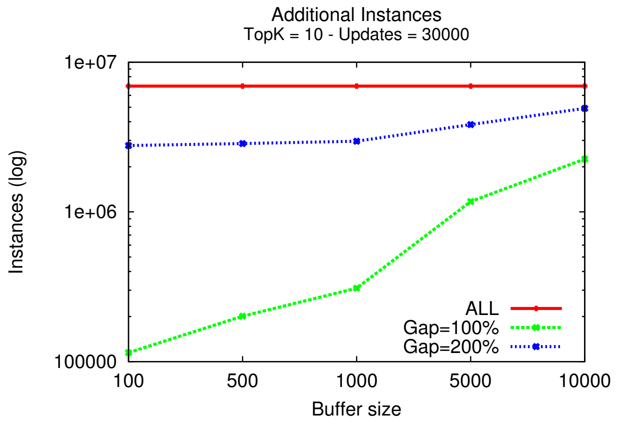


Figure 7: Extra instances stored (log scale)

5. CONCLUSIONS AND ONGOING WORK

We addressed the problem of maintaining top-k rankings in the presence of fast updates arriving in an underlying database. Such a setup calls for methods that provide accurate (exact) top-k rankings while limiting the communication with the database itself. We presented two algorithms to solve that maintenance problem, which are centred around computing score estimates for previously unseen instances and leveraging containment relations for results recycling. Although our results are preliminary, they provide useful insights on the impact of the various parameters in the effectiveness of our methods.

In ongoing work we aim at tailoring the parameters of the algorithms to the observed distribution of scores in each ranking. In particular, we work on trading-off performance and quality of the maintained results. One way of approaching this is to model the score distribution in the tail of rankings and to use it to compute more realistic estimated scores, at the risk of introducing errors to the top-k ranking.

6. REFERENCES

[1] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. Masm: efficient online updates in data warehouses. In *SIGMOD Conference*, pages 865–876, 2011.

[2] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[3] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[4] Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.

[5] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[6] Parisa Haghani, Sebastian Michel, and Karl Aberer. The gist of everything new: personalized top-k processing over web 2.0 streams. In *CIKM*, pages 489–498, 2010.

[7] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 205–216. ACM Press, 1996.

[8] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[9] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD Conference*, pages 61–72, 2006.

[10] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Monetdb/datacell: Online analytics in a streaming column-store. *PVLDB*, 5(12):1910–1913, 2012.

[11] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.

[12] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[13] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[14] Kanthi Nagaraj, K. V. M. Naidu, Rajeev Rastogi, and Scott Satkin. Efficient aggregate computation over data streams. In *ICDE*, pages 1382–1384, 2008.

[15] TPC-H: Transaction processing performance council. tpc-h, an ad-hoc, decision support benchmark. `http://www.tpc.org/tpch/`.

[16] Man Lung Yiu, Nikos Mamoulis, and Vagelis Hristidis. Extracting k most important groups from data efficiently. *Data Knowl. Eng.*, 66(2):289–310, 2008.

[17] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD Conference*, pages 299–310, 2005.