Distrib Parallel Databases (2010) 27: 95–137
DOI 10.1007/s10619-009-7057-4

Maintenance of top-k materialized views

Eftychia Baikousi · Panos Vassiliadis

Published online: 15 October 2009

Abstract In this paper we present results on the problem of maintaining materialized top-k views and provide results in two directions. The first problem we tackle concerns the maintenance of top-k views in the presence of high deletion rates. We provide a principled method that complements the inefficiency of the state of the art independently of the statistical properties of the data and the characteristics of the update streams. The second problem we have been concerned with has to do with the efficient maintenance of multiple top-k views in the presence of updates to their base relation. To this end, we provide theoretical guarantees for the nucleation (practically, inclusion) of a view with respect to another view and the reflection of this property to the management of updates. We also provide algorithmic results towards the maintenance of a large number of views, via their appropriate structuring in hierarchies of views.

Keywords Top-k views · View refreshment

1 Introduction

The *top-k querying problem* concerns the retrieval of the top-k results of a ranked query over a database. Specifically, given a relation $R(tid, A_1, A_2, \ldots, A_m)$ and a query Q over R retrieves the top-k tuples from R having the k highest values according to a scoring function f that accompanies Q. Typically, f is a monotone ranking function of the form: $f : \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_m) \rightarrow \Re$.

Communicated by Ihab F. Ilyas.

E. Baikousi (✉) · P. Vassiliadis
Dept. of Computer Science, Univ. of Ioannina, 45110 Ioannina, Greece
e-mail: ebaikou@cs.uoi.gr

P. Vassiliadis
e-mail: pvassil@cs.uoi.gr

Related work has extensively dealt with the problem of efficiently computing the top-k results of a query. The first algorithms that occurred in this context are FA [4, 5] and TA [6], with various extensions that followed them for specific contexts (e.g., parallel or distributed computation, etc.). In recent years, in an attempt to achieve improved performance, researchers solve the problem of answering top-k queries via materialized views [2, 9, 10]. In this setting, results of previous top-k queries are stored in the form of materialized views. Then, a new top-k query may be answered through materialized views resulting in better performance than making use only of the base relation from the database. As typically happens with materialized views, though, when the source relation is updated, we need to refresh the contents of all the materialized views in order to reflect the most recent data.

The two main problems that pertain to the maintenance of materialized views are (a) the correct and efficient maintenance of a single view when updates occur to the base relation, and (b) the generalization of the maintenance problem for a large number of materialized views.

Maintaining a single top-k materialized view Concerning the problem of maintaining a single view, the first—and only—attempt that we are aware of is [19]. To sustain the update rate at the source relation without having to fully re-compute the materialized views, [19] maintain k_{max} tuples (instead of the necessary k) and perform *refill* queries whenever the contents of the materialized views fall below the threshold of k tuples. Yet, the approach of [19] suffers from the following problems: (a) the method is theoretically guaranteed to work well only when insertions and deletions are of the same probability (in fact, the authors deal with updates in their experiments), (b) there is no quality-of-service guarantee when deletions are more probable than insertions. In this paper, we compensate for these shortcomings by providing a method that is able to provide quality guarantees when the deletion rate is higher than the insertion rate. The case is not so rare if one considers that the number of persons logged in a web server or a portal presents anticipated high peaks and valleys at specific time points or dates. The first contribution of our work in this paper is to deal with these phenomena efficiently. The solution to the problem is not obvious for the following reasons. First, even if the value distributions of the attributes that participate in the computation of the score are known individually, it is not possible to compute the distribution of their linear combination (i.e., the score) unless they are stable probabilities—e.g., Normal, Cauchy. Second, even if we extend k with extra tuples to sustain the incoming stream of updates that eventually affects the top-k materialized view, the extra tuples increase the possibility that an incoming source update might affect the view, thus resulting in the need to recursively compute this extension. Finally, we need to accommodate statistical fluctuations from the expected values. To resolve all the above, we provide a principled method that operates independently of the statistical properties of the data and the characteristics of the update streams. The method comprises the following steps: (a) a computation of the rate that actually affects the materialized view, (b) a computation of the necessary extension to k in order to handle the augmented number of deletions that occur and (c) a fine tuning part that adjusts this value to take the fluctuation of the statistical properties of this value into consideration.

Maintaining a set of top-k materialized views The problem of maintaining multiple materialized views is quite important. Its most prominent occurrence has to do with the situation where incoming queries are cached and treated as materialized views to efficiently support the answering of subsequent queries. The application of such a technique can be large, for example one can refer to [20] with an example over a real-estate agency and to [12] for an example of a web server that helps users discover restaurants. The problem is hard if we require all the materialized views to be refreshed every time the source relation undergoes a change. A first workaround concerns the typical warehouse solution of collecting individual updates to larger batches that can be processed much more efficiently than treating each update one tuple at a time. Still, even in this setting, we would like to avoid visiting every view for every tuple. Two extra problems that occur are (a) it is not sufficient to simply include the appropriate tuples in the extent of a materialized view, but we need to compute their score and position them appropriately in this extent (so, the sharing of tuples between views does not relieve us a lot from the overheads) and (b) we cannot solve the problem by sorting the tuples by their value over a single attribute, since the scoring function takes several attributes into consideration. Still, it is possible to prune data from the batch when we can infer that they need not be checked against a certain view. Therefore, in this paper, we develop mathematical guarantees that can decide whenever the current contents of a view need to be updated from a certain batch of modifications, when we know that another view has been affected by this same batch. We assume that the tuples in the extent of our views include (a) the tuple identifier of the tuple in the base relation, (b) the scoring attributes (needed for the management of updates), and, (c) its score in the view. In our method, we introduce the idea of *nucleation* between views, which is quite similar to inclusion: a view V_2 nucleates another view V_1, whenever all tuples of the former belong to the extent of the latter, with the exception of their scores. The decision for this kind of inclusion is not straightforward; to avoid checking all the extents of two views we employ a geometric representation of the score function and the tuples of the two views and decide on the nucleation on the basis of this representation. Then, we structure views in a set of hierarchies, where each ancestor view nucleates its descendants. Updates can be pruned from a hierarchy, or a part of it, when a certain view in the hierarchy is unaffected from a modification; in this case, all its ancestor views avoid the test, too. At the same time, nucleation hierarchies come with a price: they are instance dependent and thus they need to be rechecked after the modifications of the view extents take place.

Roadmap The structure of this paper is as follows. In Sect. 2, we review related work. We present our method for the fine-tuning of the actual size of a top-k materialized view in Sect. 3. In Sect. 4 we provide a generalization of the results of Sect. 3 with respect to (a) the treatment of views with more than two attributes and (b) the usage of a monotone scoring function (instead of simply a linear function). In Sect. 5, we deal with the problem of maintaining more than one view; to this end, we introduce the notion of nucleation along with a set of algorithms for the management of multiple views structured in a hierarchy. In Sect. 6 we present experimental results and in Sect. 7 we conclude our results and present topics for future research. A first version of this paper has appeared in [1].

2 Related work and background

2.1 Efficient maintenance of materialized top-k views [19]

[19] deal with the following problem: Given a base table $R(id, val)$ containing N tuples, where val is the score of the tuple according to a scoring function and a materialized view $V(id, val)$ containing the top-k tuples from R according to their values, compute a k_{max} that is adjusted at runtime such that a refill query, that re-computes the view V from scratch for the missing part, is rarely needed. Assume an update of the form $\langle id, val\rangle$ occurs and let $val_{k'}$ the tuple with the lowest value in V. Then the update can be classified as *ignorable*, *neutral*, *good* or *bad*. *Ignorable* is an update when its id is not in V and $val < val_{k'}$ and thus there is no effect in V. A *neutral* update occurs when its id is in V and $val > val_{k'}$. Then the tuple id is updated with value val. An update is categorized as *good* update when its id is not in V and $val > val_{k'}$. Then this tuple is inserted in V and k' (where k' denotes the current size of the view) is increased by one. If k' exceeds k_{max} then the lowest tuple in V is deleted. A *bad* update describes an update whose id is in V and $val < val_{k'}$. The tuple id is then deleted from V and k' is decreased by one. If k' drops below k, a *refill operation* is performed. A refill operation queries the base table R and returns all tuples ranked between k and k_{max}. [19] formulated the problem through a random walk model. The values of k' between two refill operations are represented through a 1-dimensional random walk model. The points are represented as $\{0, \ldots, n\}$ where 0 denotes the starting point (k_{max}) and n ($k_{max} - k + 1$) the absorbing point at which a refill operation is needed. Assume that the random walk is currently in position i and a bad update moves the random walk to position $i + 1$ with probability p_i, whereas a good update moves the random walk to position $i - 1$ with probability q_i. In any other case the update is ignorable or neutral with probability $1 - p_i - q_i$. The problem is focused on analyzing the number of steps needed for the random walk model to go from 0 to n. In other words the analysis is conducted in order to find the probabilistic properties of the refill interval Z.

According to the assumptions that each step is independent of all previous choices and the probabilities of bad and good updates remain constant as updates occur in the view ($p_0 = p_1 = \cdots = p_{n-1} = p$ and $q_0 = q_1 = \cdots = q_{n-1} = q$) the following occur. When $p = q$ then if $n = N^{\frac{1}{2}+\varepsilon}$ the refill integral Z is greater than N with high probability being $\Pr[Z > N] \geq 1 - 4e^{-N^{2\varepsilon}/2}$, for any positive constant ε. When $p < q$, if $n = c \ln N$ the refill integral Z is greater than N with high probability being $\Pr[Z > N] > 1 - o(1)$, for constant c big enough depending only on p and q. When $p > q$, then, if $n = N$ the refill integral Z is on the order of n. An adaptive algorithm chooses k_{max} at runtime without using the probabilities of good and bad updates. The algorithm is trying to keep the refill interval Z around the value $Z_0 = C_{refill}/C_{update}$ (where C_{refill} is the observed cost of a refill query and C_{update} is the observed cost of a base table update). The algorithm counts the number of base table updates occurred from the last refill operation. If the updates are less than Z_0/a then k_{max} is increased whereas if the number of updates is greater than aZ_0 then k_{max} is decreased, where a is a constant parameter.

2.2 Algorithms for answering top-k queries over databases

In this section, we give a brief overview of the basic algorithms that answer a top-k query over a relation R. Firstly we describe the algorithms that provide an answer to a top-k query. Secondly, we describe the algorithms that make use of materialized views in order to answer a top-k query. Thirdly, we describe frameworks that make use of previously posed queries in order to answer new ones, mainly by caching previous results.

Fagin's algorithm (FA) [4, 5] Given a relation $R(tid, A_1, A_2, \ldots, A_m)$, from which a set of sorted lists $L = \{(tid, A_i)|tid, A_i \in R\} \forall A_i \in R$ is formed and a query scoring function $g(X)$ such that $g(X)$ is a monotone aggregation function, Fagin's algorithm *FA* retrieves the top-k tuples of R. This is achieved by a three-step process. First do sorted access to each of the m sorted lists, until there are at least k tuples seen in each of the m lists. Secondly, for each tuple X seen, do random accesses to each of the lists to find the ith attribute of that tuple, which is x_i. Thirdly for each X seen, compute its score $g(X) = g(x_1, x_2, \ldots, x_m)$. The output is the ordered set $\{(X, g(X)|X \in Y\}$ where Y contains the k tuples with the highest scores.

FA is correct when g is a monotone aggregation function. This is important in the sense that it assures that all tuples not seen under sorted access do not participate in the top-k tuples.

Threshold algorithm (TA) [6, 8, 14] *FA* is optimal in high probability sense whereas, the threshold algorithm is instance optimal. Similar to *FA*, *TA* can be applied over a database having m attributes. *TA* is expressed through a three-step process: First do sorted access in parallel to each of the m sorted lists. For each tuple X seen under a list, do random accesses to all the other lists to find the scores x_i of X. Compute the score $g(X) = g(x_1, x_2, \ldots, x_m)$ of the tuple X and remember X and its score if it is one of the k highest. Secondly, define the threshold value τ as $g(\underline{x_1}, \underline{x_2} \ldots \underline{x_m})$ where $\underline{x_i}$ is the score of the last tuple seen under sorted access to each of the lists. Halt when at least k tuples have been seen with score at least equal to τ. The output is then the ordered set $\{(X, g(X)|X \in Y\}$ where Y contains the k tuples that have been seen with the highest grades. *TA* is correct when g is a monotone aggregation function.

[6] have been proved that *TA* is instance optimal. An algorithm B is *instance optimal* over a class of algorithms $\boldsymbol{A}$ and a class of legal inputs $\boldsymbol{D}$ to the algorithms when $B \in \boldsymbol{A}$ and if for every $A \in \boldsymbol{A}$ and for every $D \in \boldsymbol{D}$, we have $cost(B, D) = O(cost(A, D))$, where $cost(B, D)$ is the middleware cost incurring by running algorithm B over database D.

Prefer [9, 10] PREFER is a system with a core algorithm that answers top-k queries using materialized views in a pipelined way. PREFER consists of two algorithms called (i) *ViewSelection* algorithm and (ii) *PipelineResults* algorithm. The *ViewSelection* algorithm decides which views should be materialized according to the system's performance requirements and a given relation. The goal of the *PipelineResults* algorithm is to rank the tuples of a relation $R(A_1, \ldots, A_n)$ of n attributes, according to a query q. The query q is characterized by a preference vector. A preference vector is

of the form $(w_1, w_2, \ldots, w_n)$ where each coordinate w_i denotes the preferred weight of the i-th attribute. Therefore, the scoring function of q becomes $\sum_{i=1}^{n} w_i \cdot A_i$. Algorithm *PipelineResults* employs a view $R_v(tid, score_v)$ that contains the tuples of R, ranked by another preference vector v. The algorithm computes a prefix R_v^1 from R_v that ensures that the first tuple t_q^1 of the sequence R_q is in R_v^1. Then, it computes the second prefix R_v^2 in order to retrieve the second tuple t_q^2 and so on until the first k tuples of the query are retrieved. The key concept of this algorithm is the computation of a watermark value, which works as a stopping condition in each iteration of the *PipelineResults* algorithm. The watermark value is a score with respect to the ranking function of the materialized view rather than the query that actually determines how deep in the ranked materialized view we should go in order to output the top result tuple of the query. The watermark value of the first iteration is the maximum value $T_{v,q}^1$ with the property $\forall t \in R$, $f_v(t) < T_{v,q}^1 \Rightarrow f_q(t) < f_q(t_v^1)$.

Linear programming adaptation of the threshold algorithm LPTA [2] LPTA is based on the *TA* algorithm. LPTA is applied on a set of materialized views in order to answer top-k queries. LPTA is implemented through a two-step procedure. Assume a set of materialized views $V = (V_1, \ldots, V_r)$ that contain the base views. For a relation R containing an attribute A_i, a base view V_i is a materialized view of the form (id, A_i) ordered over all the tuples of relation R. The first procedure of LPTA is the *SelectViews* algorithm. Algorithm *SelectViews*(V, Q) determines the most efficient subset $U \subseteq V$ over a set of materialized views V, in order to execute a given query Q. The set U is the most efficient subset of V in the sense that it produces the answer to the top-k query most efficiently among all possible subsets of V. The *SelectViews* algorithm is based on a simple greedy heuristic procedure that selects the subset U that has the cheapest cost. Secondly, the LPTA algorithm obtains an answer to Q combining all the information conveyed by the views in U. Each view $V(tid, score_v)$ is a set of pairs of the form (tuple identifier, score of that tuple) using the view's scoring function. LPTA starts with an empty top-k buffer and proceeds in the following four steps. Firstly, it does sorted accesses in parallel to each of the views. Secondly, for each tuple X read from a view, random accesses are done on relation R in order to find the scores x_i of X. Thirdly, the score $t(X) = t(x_1, x_2, \ldots, x_m)$ of the tuple X in regards to the query Q is computed and the top-k buffer is updated. Fourthly, the stopping condition is checked. In order to check the stopping condition, a linear program is solved. Assume that the last tuple read from each view V_i has score $score_i$ in regards to its scoring function SF_i. The objective function of the linear program is the query's score function. The constraints for the linear program are the inequalities $SF_i \leq score_i$. The stopping condition holds when the solution of the linear program is at least equal to the minimum value of the top-k buffer. In case the set of views U is equal to the set of base views then LPTA becomes the TA algorithm.

K-Skyband [13, 15, 18] Top-k queries are closely related to the notions of skyline and K-skyband. The notion of K-skyband was first introduced by Papadias et al. in [15]. A K-skyband query returns the set of points which are dominated by at most $K - 1$ other ones. Therefore, when $K = 1$, the K-skyband corresponds to the conventional skyline. Since the K-skyband contains the set of points that are dominated by at most $K - 1$ other ones, any top-k query can accurately be answered

from the points belonging in a K-skyband, as long as $K > k$. In [13], the authors study continuous monitoring top-k queries over sliding windows, by making use of K-skyband. Moreover, Vlachou et al. in [18], have studied the problem of answering top-k queries over a distributed environment. Vlachou et al. in [18] propose a system called SPEERTO that supports top-k query processing over horizontally partitioned data stored on peers organized in a super-peer network. Each super-peer is responsible for collecting and caching the K-skyband of its peers and producing its own K-skyband. Super-peers exchange a summary of their K-skybands. Therefore, when a top-k query is posed over a super-peer, the latter transmits the query to the appropriate super-peers and collects the results in order to accurately answer the query.

K-skybands are powerful enough to answer any top-k query with $k < K$. However, the applicability of K-skybands is not universal as the benefits of this good property are balanced by severe disadvantages. The two main problems with K-skybands are their size and the inability to regulate this size. The size of a K-skyband depends upon the distribution of the dataset [15] (something that our proposed method of maintaining top-k views is independent of) as well as the dimensionality [18] (i.e., number of attributes in the dataset). The size of the K-skybands is a dominating factor: (a) the system must have the space to store it in main-memory (if there is no room for the *whole* skyband then this caching is useless) and (b) the volume of updates that must be served is proportional to this size. These two factors make it quite hard to control the size of the stored tuples and therefore obviously difficult to monitor the updates that will occur and affect the K-skyband. Materialized views are independent from these two aspects and this makes the management of materialized views worth. Our proposed method stores for a top-k materialized view a slighter increased size of k tuples instead of k, in order to maintain the view (see experimental method in Sect. 6). We believe that the idea of maintaining a K-skyband instead of multiple cviews seems more ineffective (considerably greater overhead) when taking into consideration the small overhead of points materialized in each view and the advantages of constructing the hierarchy path among views.

View caching [7, 11, 18, 20] The exploitation of the result set of a previous query for the answering of a subsequent query is frequently encountered in the research literature (see for example [11] and [7]) under the name of query or view caching. Once a query is maintained in main memory for this purpose, it practically becomes a materialized view (and as such, (a) it can be used for query answering, and (b) it needs to be maintained in the presence of changes). Concerning the case of top-k queries, in [20], the authors describe a system called BRANCA that answers top-k queries over an acyclic network of servers. The main idea of this system is based on the rationale of caching the results and information from previously posed top-k queries in order to make use of them for future ones. This technique results in less communication cost over the network when a new top-k query arrives. As already mentioned, [VDNV08] propose a system called SPEERTO that supports top-k query processing in a distributed environment making use of caching techniques through K-skybands.

3 Fine-tuning of views to sustain high update rates

In this section we present our method for the fine tuning of materialized views defined over a relation that goes through updates in high rates. First, we formally define the problem. Second, we sketch our method and then, we move on to further detail the individual steps of the method.

Before proceeding, we present a couple of motivating examples to contextualize our discussion. Assume an online news web site where people can create profiles, post announcements, recommend URLs or other on-line material, and get informed on postings made by other users. There are several such sites like digg (www.digg.com), reddit (www.reddit.com) or Google groups (groups.google.com). In all these sites, subscribers have a profile that ranks postings according to certain characteristics of each posting. Assume a base table *Postings*(*P_ID*, *contents*, *topic*, *publisher*, *date*) that contains all the postings. The profile of subscriber is practically a formula that assigns a score to every posting according to matching of the profile to the person who published the posting, the topics and the age of the posting; all these matchings can be computed by appropriate distance functions. Therefore, with respect to the settings of this paper, (a) there is a base table containing all the postings where additions are made by individuals by inserting new material and (b) a very large set of views, with each user profile acting as a ranking view over the base table. The consistency of the views with respect to their content and the efficiency of the maintenance process are the two main desiderata by this system. Moreover, a third desideratum is the smooth scale up of the efficiency of the process when the number of views (i.e., online subscribers) rises. On a typical situation, additions are quite more frequent than deletions or updates since users upload new postings about interesting sites or on-line resources. Still, there are situations where heavy deletion rates come up. Take for example cases of spam attack, or jokes (like for example April fool's day) that completely destroy the coherence and contents of a topic (and consequently, of all the contents of all the user views that get informed on the contents of one or more topics). In case the site is moderated, the moderator can intervene and massively delete the offending postings. This can be either an ad-hoc intervention due to a heavy load of unwanted activity to a specific topic, or a scheduled, regular cleanup activity that removes noise from the lists of topics. In this kind of situation, it is quite helpful to adjust the views extents by taking the increased deletion rate into consideration,

The above situation is one of possible situations taking place in the Internet today. Lately, administrators seem to observe quite a large number of irregularities in the behavior of on-line users. Peaks and valleys in the load of servers are too far from the 40% rule-of-thumb for the excessive load of a server—sometimes in the area of 100% to 1000% [16]. Dealing with this kind of excessive loads via hardware is unrealistic (even if cloud computing is employed); thus the need for algorithmic solutions is evident. Assuming that a site monitors the hits made by readers and creates views to expose the top-viewed material to the administrator, we have a situation where system catalogues the characteristics of clients accessing the material of a site. Then, a relation *CurrentlyLoggedUsers*(*IP*, *ip-geo*, *start-of-session*, *estimated-end-of-session*, *pages-accessed-so-far*) can be derived over these statistics. Advertisers might create a view for each product and assess the top-k candidates to see the advertisement

on the basis of their geographical characteristics and navigation history—again these views employ a set of distance functions to obtain the matching of a session to the profile of a product. Again the effectiveness and the efficiency of the maintenance process are the two important factors. A subtle but interesting point, however, is the fact that whereas peaks in the load are the main problem of the administrator, valleys present a problem for the advertisers: Since users leave the site massively, it is important to retain in each view the ones that actually stay in the site, without having to recompute the view from scratch (especially if the number of views is large).

3.1 Formal definition of the problem

Given a base relation $R(ID, X, Y)$ that originally contains N tuples, a materialized view V that contains top-k tuples of the form (id, val) where val is the score according to a function $f(x, y) = ax + by$ and a, b are constant parameters, the update ratios Λ_{ins}, Λ_{del} and Λ_{upd} for insertions, deletions and updates respectively over the base relation R,

Compute k_{comp} that is of the form $k_{comp} = k + \Delta k$.

Such that the view will contain at least k tuples, $k \leq k_{comp}$, with probability p, after a period T.

Assume a base relation $R(ID, X, Y)$, that contains N tuples a materialized view V that contains top-k tuples of the form (id, val) where val is the score according to a function $f(x, y) = a \cdot x + b \cdot y$ and a, b are constant parameters. Assume that the last tuple in the view has value val_k. Given the aforementioned update rates, insertions, deletions and updates occur in the base relation R with probabilities P_{INS}, P_{DEL} and P_{UPD} respectively. These probabilities are expressed as:

$$P_{INS} = \frac{\Lambda_{INS}}{\Lambda_{INS} + \Lambda_{DEL} + \Lambda_{UPD}}, \qquad P_{DEL} = \frac{\Lambda_{DEL}}{\Lambda_{INS} + \Lambda_{DEL} + \Lambda_{UPD}} \quad \text{and}$$

$$P_{UPD} = \frac{\Lambda_{UPD}}{\Lambda_{INS} + \Lambda_{DEL} + \Lambda_{UPD}}.$$

In the rest of our deliberations, updates are treated as combinations of deletions and insertions. This is a quite reasonable treatment, since we are mainly interested in the statistical properties of the rates of these actions and not in their hidden semantics. A simple method for the conversion of the involved rates is given in Sect. 3.3.

Our problem is to find a k_{comp} that will guarantee that the view will be maintained when insertions and deletions will occur in R. In order to do so, we must estimate the number of insertions and deletions that might affect the view. In other words, we need to compute the probability of the view being affected by a tuple inserted in R or deleted from R.

Assume that a new tuple $z(id, x, y)$ is inserted in R. The probability of this tuple affecting the view is $p(z > val_k)$. Hence, the probability of a new tuple to be inserted in R affecting the view V is p_{ins}^{aff} which is expressed as: $p_{ins}^{aff} = p(z > val_k) \cdot p_{ins}$. The probability of a tuple to be deleted from R affecting the view V is p_{del}^{aff} which occurs as $p_{del}^{aff} = p(z > val_k) \cdot p_{del}$.

A problem that occurs with the maintenance of k_{comp} tuples at the view side is that k_{comp} incurs extra maintenance overheads, since the tuples of Δk can suffer updates too. Thus, we need to compute p_{ins}^{aff} and p_{del}^{aff} for the case where k_{comp} tuples are maintained. Therefore, the view V will contain k_{comp} tuples instead of k. Assume that the last tuple of the view containing k_{comp} tuples is $val_{k_{comp}}$. Consequently, the probability of a new tuple z to affect the view V is $p(z > val_{k_{comp}})$ whereas the probability of a new tuple to be inserted in R affecting the view occurs as: $p_{ins}^{aff} = p(z > val_{k_{comp}}) \cdot p_{ins}$. Respectively the probability of a tuple z to be deleted from R affecting the view V can be expressed as: $p_{del}^{aff} = p(z > val_{k_{comp}}) \cdot p_{del}$.

3.2 Sketch of the method

The proposed method is focused around three main steps: first, we compute the percentage of the incoming source updates that affect a top-k materialized view; second, we compute a safe value for the additional view tuples that we need in order to sustain high deletion rates; third, we fine tune this value with a safety range of values. Specifically, the three main steps are:

1. Given Λ_{INS}, Λ_{DEL} and Λ_{UPD}, we can compute λ_{ins} and λ_{del}, p_{ins} and p_{del}, and finally, p_{ins}^{aff} and p_{del}^{aff} as well as λ_{ins}^{aff} and λ_{del}^{aff}.
   p_{ins} and p_{del} denote the probabilities of an insertion and deletion occurring on the base table R respectively. p_{ins}^{aff} and p_{del}^{aff} denote the probabilities of insertions and deletions that affect the view V respectively. These probabilities are expressed as a function of k_{comp}. λ_{ins}^{aff} and λ_{del}^{aff} denote the ratios of insertions and deletions occurring in the view V in the period of operations T. Updates are treated as a combination of deletions and insertions thus λ_{ins} and λ_{del} denote the ratios of insertions and deletions including those occurring from updates.
2. Compute k_{comp} as a function of λ_{ins}^{aff}, λ_{del}^{aff}.
   k_{comp} denotes the number of tuples that the view V should initially contain, such that after a period of operations T, V will contain at least k tuples.
3. Fine-tune k_{comp} by using the variance of the probability that a deletion and insertion action affects the materialized view.

3.3 Handling of updates

Assume the insertion, deletion and update rates Λ_{INS}, Λ_{DEL} and Λ_{UPD}. We can compute the respective rates λ_{ins} and λ_{del} where updates are treated as combinations of deletions and insertions, through the following set of equations, where T is an arbitrary operation period.

λ_{ins} = number of insertions including those from updates / T
λ_{del} = number of deletions including those from updates / T
Λ_{INS} = number of insertions / T
Λ_{DEL} = number of deletions /T
Λ_{UPD} = number of updates /T

Therefore, $\lambda_{ins} = \Lambda_{INS} + \Lambda_{UPD}$ and $\lambda_{del} = \Lambda_{DEL} + \Lambda_{UPD}$. In addition, p_{ins} and p_{del} can be expressed through the usage of ratios as $p_{ins} = \frac{\lambda_{ins}}{\lambda_{ins}+\lambda_{del}}$ and $p_{del} = \frac{\lambda_{del}}{\lambda_{ins}+\lambda_{del}}$ respectively.

3.4 Computation of the actual rates that affect V

The problem now is to compute the probabilities p_{ins}^{aff} and p_{del}^{aff} that affect the view V. These probabilities can be computed as $p_{ins}^{aff} = p_{ins} \cdot p(z > val_{k_{comp}})$ and $p_{del}^{aff} = p_{del} \cdot p(z > val_{k_{comp}})$ respectively. Actually, p_{ins}^{aff} is the number of insertions affecting the view V divided by the number of insertions and deletions occurring on the base table R and p_{del}^{aff} is the number of deletions affecting the view V divided by the number of insertions and deletions occurring on the base table R. Now the problem is focused upon finding the probability $p(z > val_k)$.

In order to compute the above probability we will use the Empirical Cumulative Distribution Function $F_n(x)$ (ECDF). Instead of using a particular parametric cumulative distribution function, we will use ECDF which is a non parametric cumulative distribution function that adapts itself to the data. ECDF returns the values of a function $F(x)$ such that $F_n(x)$ represents the proportion of observations in a sample less than or equal to x. $F_n(x)$ assigns the probability $1/n$ to each of n observations in the sample. In other words $F_n(x)$ estimates the true population proportion $F(x)$. ECDF is formally defined as follows [17]:

Let $X_1, X_2, \ldots, X_n$ be independent, identically distributed random variables and let $x_1 < x_2 < \cdots < x_n$ denote the values of the order statistics of the sample. Then the empirical distribution function $F_n(x)$ is defined by the following formula:

$$F_n(x) = \begin{cases} 0, & x < x_1, \\ \frac{i}{n}, & x_i \le x < x_{i+1}, \\ 1, & x_n \le x. \end{cases}$$

The alternative definition of $F_n(x)$ is:

$$F_n(x) = \frac{number\_of\_values\_in\_the\_sample\_that\_are\_ \le x}{n}.$$

Assume that the base relation R contains N tuples and the view V should contain k_{comp} tuples. If we order these tuples according to their values then there are $N - k_{comp}$ tuples in R with value less than the value of k_{comp}. The following theorem implies that when the sample size n is large, $F_n(x)$ is quite likely to be close to $F(x)$ over the entire real line.

Glivenko-Cantelli Theorem [3] *Let $F(x)$ denote the density function of the distribution from which the random sample $X_1, X_2, \ldots, X_n$ was drawn. For each given number $x(-\infty < x < \infty)$ the probability that any particular observation X_i will be less than or equal to x is $F(x)$. Therefore, it follows from the law of large numbers that as $n \to \infty$, the proportion $F_n(x)$ of observations in the sample that are less than or equal to x will converge to $F(x)$ uniformly over all values of x. Let D_n be the*

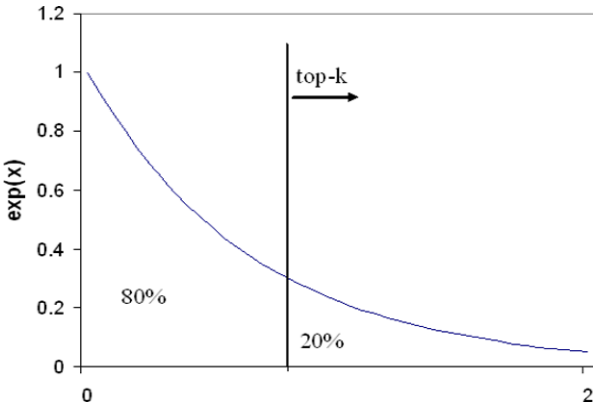


Fig. 1 Exponential probability distribution

upper bound of the difference of $F_n(x)$ *by* $F(x)$, $D_n = \sup_{-\infty<x<\infty} |F_n(x) - F(x)|$. *Then, the Glivenko-Cantelli theorem states that* $D_n \xrightarrow{P} 0$.

Therefore, the probability of a tuple z affecting the view V can be expressed as:

$$\begin{aligned} p(z > val_{k_{comp}}) &= 1 - p(z \leq val_{k_{comp}}) = 1 - F_N(k_{comp}) \\ p(z > val_{k_{comp}}) &= 1 - \frac{N - k_{comp}}{N} = \frac{k_{comp}}{N}. \end{aligned} \tag{1}$$

As a more general example, consider a base relation R where the score of its tuples according to a function follow an exponential distribution in the interval [0, 2] and that a view V requires the top-k tuples of R according to their score value. In Fig. 1 the probability distribution function of an exponential distribution is illustrated. In addition, assume that the top-k tuples are the 20% of the relation R and thus the vertical line top-k shown in Fig. 1 denotes the values of the tuples that participate in the top-k view. Thus, the values in the view are greater or equal to 0.3. Assume a new tuple t following the same exponential distribution being inserted in R. For the new tuple t the probability of its value participating in the top-k ones is again 20%.

Again, consider a similar situation where a view contains the top-k tuples from a base relation R according to a scoring function. Assume that the score values of R this time follow a beta distribution in the interval [0, 1] with parameters given as 5 and 2. Figure 2 illustrates the probability distribution function of such a distribution. Similar to the previous example, the vertical line illustrated as top-k in Fig. 2 denotes that the view contains 20% of R's tuples where the values participating in the view are greater or equal to 1.7. Assume a new tuple denoted as t being inserted in R. The new tuple t will again follow the same beta distribution and the probability of t having a value greater than 0.8 is 20%.

Therefore, λ_{ins}^{aff} and λ_{del}^{aff} are computed through the following equations:

$$\lambda_{ins}^{aff} = p_{ins}^{aff} \cdot (\lambda_{ins} + \lambda_{del}) \quad \text{and} \quad \lambda_{del}^{aff} = p_{del}^{aff} \cdot (\lambda_{ins} + \lambda_{del}).$$

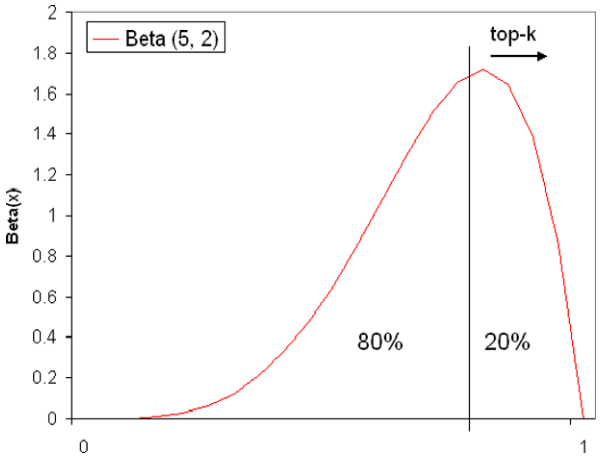


Fig. 2 Beta probability distribution

According to (1), λ_{ins}^{aff} and λ_{del}^{aff} can be expressed as:

$$\begin{aligned} \lambda_{ins}^{aff} &= p_{ins} \cdot p(z > val_{k_{comp}}) \cdot (\lambda_{ins} + \lambda_{del}) \\ \lambda_{ins}^{aff} &= p_{ins} \cdot \frac{k_{comp}}{N} \cdot (\lambda_{ins} + \lambda_{del}) \quad \text{and} \end{aligned} \tag{2}$$

$$\begin{aligned} \lambda_{del}^{aff} &= p_{del} \cdot p(z > val_{k_{comp}}) \cdot (\lambda_{ins} + \lambda_{del}) \\ \lambda_{del}^{aff} &= p_{del} \cdot \frac{k_{comp}}{N} \cdot (\lambda_{ins} + \lambda_{del}). \end{aligned} \tag{3}$$

3.5 Computation of k_{comp}

The last step of the method is to compute k_{comp}, in such a way that it will guarantee that the view will contain at least k tuples, $k \leq k_{comp}$, with probability p, after a period of operation T. In other words compute a k_{comp} that is of the form $k_{comp} = k + \Delta k$. In general, when the ratio of insertions λ_{ins} is greater than that of deletions λ_{del} it is clear that V will be maintained. The problem arises when the opposite occurs and the ratio of deletions is greater than that of insertions. In such a case it is vital to compute a value for k_{comp} that can guarantee that V will contain at least k tuples after a period of operations.

Let us denote the frequency of deletions that affect the view V as λ_{del}^{aff}. In a period of time T, in order to keep the view maintained the following inequality should hold: $k_{comp}^{T} - \lambda_{del}^{aff} \cdot T \geq k$.

Thus, in case both insertions and deletions occur in a period of time T, in order to keep the view maintained for k_{comp} the following inequality should hold $k_{comp} \geq k + (\lambda_{del}^{aff} - \lambda_{ins}^{aff}) \cdot T$. Clearly, to minimize memory consumption, we need to take the minimum possible k_{comp} and thus treat the above inequality as the equation $k_{comp} = k + (\lambda_{del}^{aff} - \lambda_{ins}^{aff}) \cdot T$.

Therefore, by replacing λ_{ins}^{aff} and λ_{del}^{aff} from (2) and (3) the following equality occurs:

$$k_{comp} = k + (p_{del} - p_{ins}) \cdot (\lambda_{ins} + \lambda_{del}) \cdot \frac{k_{comp}}{N} \cdot T$$

$$\Rightarrow k_{comp} = k + (\lambda_{del} - \lambda_{ins}) \cdot \frac{k_{comp}}{N} \cdot T. \tag{4}$$

Thus, by solving the above equation according to k_{comp} we obtain:

$$k_{comp} = k \cdot \frac{N}{N + (\lambda_{ins} - \lambda_{del}) * T}. \tag{5}$$

Equation (5) has a meaning when $N + (\lambda_{ins} - \lambda_{del}) \cdot T > 0$. This states that the size of the base relation R will not fall below 0, after updates occur in a period of operations T. At the same time, when $\lambda_{ins} - \lambda_{del} < 0$ (i.e., the case we are particularly interested in), then the fraction is greater than 1 and thus, $k_{comp} > k$.

3.6 Fine-tuning of k_{comp}

Although we now have a formula to compute the value of k_{comp}, we have expressed the probability of a new tuple $z(id, x, y)$ affecting the top-k_{comp} tuples of the view as $p(z > val_{k_{comp}})$. Assume that a new tuple z is inserted in R. The probability of this tuple to affect the view is $p(z > val_{k_{comp}})$ whereas, the probability of this tuple not affecting the view is $1 - p(z > val_{k_{comp}})$. The above can be expressed as a Bernoulli experiment with two possible events. These are (a) the new tuple being inserted in V with probability of success $p(z > val_{k_{comp}})$ and (b) the exact opposite where the new tuple is not inserted in V with probability $1 - p(z > val_{k_{comp}})$. When the ratio of insertions occurring in the base relation R are λ_{ins}, a Bernoulli experiment is occurring λ_{ins} times where the probability of success is $p(z > val_{k_{comp}})$ and the number of successes follows a Binomial distribution. The probability of having Y_{ins} affected insertions in the view follows a Binomial distribution of the form $Binomial(\lambda_{ins}, p(z > val_{k_{comp}}))$. The variance of the above distribution can be expressed as:

$$\mathrm{Var}(Y_{ins}) = \lambda_{ins} \cdot p(z > val_{k_{comp}}) \cdot (1 - p(z > val_{k_{comp}})).$$

The above formula indicates that insertions expected to affect the view may vary by $\mathrm{Var}(Y_{ins})$. Correspondingly, if there are λ_{del} deletions occurring in the base relation R, then the variance of these deletions expected to affect the view is $\mathrm{Var}(Y_{del}) = \lambda_{del} \cdot p(z > val_{k_{comp}}) \cdot (1 - p(z > val_{k_{comp}}))$. This occurs as the variance of the Binomial distribution $B(\lambda_{del}, p(z > val_{k_{comp}}))$, which is similar to the one used for insertions.

Therefore in the worst case, in order to guarantee that the view will contain at least k tuples with confidence 95%, where $k \leq k_{comp}$, (4) becomes as stated below:

$$k_{comp} = k + (\lambda_{del} - \lambda_{ins}) \cdot \frac{k_{comp}}{N} \cdot T + 2 \cdot \mathrm{Var}(Y_{del}) + 2\,\mathrm{Var}(Y_{ins}). \tag{6}$$

The confidence rate of 95% occurs from statistical properties concerning the variance factor appearing in formula (6). In case another confidence percentage is needed, formula (6) can be adjusted according to typical statistical methods [3].

3.7 Discussion

The problem of maintaining a view when updates occur in a base relation R, mainly lies in the problem of estimating the number of updates that will affect the view. Statisticians have contributed in this by providing formulas that compute the value of a probability of the form $p(z > val_{k_{comp}})$. However, the formula of such a probability depends on the distribution that the variable z follows. In our context, the variable z is a linear combination of the form $a \cdot x + b \cdot y$ where x and y are values from the attributes X and Y of the base relation. Even if the distributions that X and Y follow are known, the distribution of the score Z can not be computed unless X and Y follow a *stable* distribution. A stable distribution (e.g., Normal, Cauchy) has the property of stability. This property states that if a number of independent identically distributed (iid) random variables have a stable distribution, then a linear combination of these variables will have the same distribution. Therefore, the distribution of the variable Z can only be known in few cases. However, even if the distribution of the score was known, the probability $p(z > val_{k_{comp}})$ could be computed only with respect to the val_k instead of the value $val_{k_{comp}}$. This is because the $val_{k_{comp}}$ could not been know in advance. Therefore, an iterative procedure would be needed. This occurs from the fact that we could compute the effect top-k tuples could have but not the effect the extra tuples would arise. Thus, a recursive procedure would be required.

3.8 Example

As an example, consider the base relation $R(ID, X, Y)$ initially containing N tuples with $N = 20$ where attributes X and Y follow a uniform distribution over the interval $[0, 100]$. In addition, consider a materialized view V that contains the top-3 tuples ($k = 3$) of the form (id, val) where $val = 3 \cdot x + 7 \cdot y$ is the score according to a function $f(x, y) = a \cdot x + b \cdot y$ and $a = 3$, $b = 7$. The base relation R and the initial state of V are shown in Fig. 3. Finally, the update ratios are $\Lambda_{ins} = 5$, $\Lambda_{del} = 15$ and $\Lambda_{upd} = 0$. We will compute k_{comp} such that the view would contain k_{comp} tuples instead of k in order to be kept maintained when insertions, deletions and updates will occur in the base relation R. Moreover, let the period of operations occurring set as $T = 1$.

According to the method of Sect. 3.2, the ratios λ_{ins} and λ_{del} are 5 and 15 respectively. Therefore, $p_{ins} = 0.25$ and $p_{del} = 0.75$. The probability $p(z \geq val_{k_{comp}})$ can be calculated according to the following:

$$p(z \leq val_{k_{comp}}) = F_N(val_{k_{comp}})$$
$$p(z \leq val_{k_{comp}}) = (\text{number of elements in sample with score} \leq val_{k_{comp}})/N$$
$$p(z > val_{k_{comp}}) = k_{comp}/20.$$

Fig. 3 Base relation R

R		
id	**X**	**Y**
1	56	41
2	58	62
3	15	97
4	78	86
5	69	10
6	96	60
7	12	43
8	74	76
9	26	71
10	95	92
11	34	51
12	27	36
13	19	25
14	68	81
15	91	82
16	84	65
17	41	59
18	37	37
19	23	17
20	47	27

V	
id	**Z**
10	929
15	847
4	836

In consequence, the probabilities p_{ins}^{aff} and p_{del}^{aff} can be calculated as:

$$p_{ins}^{aff} = p_{ins} \cdot p(z \geq val_{k_{comp}}) = 0.25 \cdot \frac{k_{comp}}{20} \quad \text{and}$$

$$p_{del}^{aff} = p_{del} \cdot p(z \geq val_{k_{comp}}) = 0.75 \cdot \frac{k_{comp}}{20}.$$

Given the previous probabilities, the effective update ratios for the view V are then:

$$\lambda_{ins}^{aff} = p_{ins}^{aff} \cdot (\lambda_{ins} + \lambda_{del}) = 0.25 \cdot \frac{k_{comp}}{20} \cdot (5 + 15)$$

$$\lambda_{del}^{aff} = p_{del}^{aff} \cdot (\lambda_{ins} + \lambda_{del}) = 0.75 \cdot \frac{k_{comp}}{20} \cdot (5 + 15).$$

The above formulas state that if 5 insertions will occur in the base relation R, λ_{ins}^{aff} will affect the view and if 15 deletions occur then λ_{del}^{aff} will affect the view respectively. To be more specific the ceiling function is applied on λ_{ins}^{aff} and λ_{del}^{aff}. Therefore, for k_{comp} the following inequality holds:

$$k_{comp} \geq k + (\lambda_{del}^{aff} - \lambda_{ins}^{aff}) \cdot T \Rightarrow k_{comp} \geq 6$$

where actually $k_{comp} = 6$. Thus, k_{comp} should be 6 in order to keep the view maintained after insertions, deletions and updates will occur in the base relation R. Suppose that insertions and deletions, shown in Fig. 4, occur in the base relation R. The view V contains initially top-6 tuples and after updates the view will contain top-3 tuples. These are shown in Fig. 5 where the dark shading denotes the initial top-3

Fig. 4 Insertions and deletions occurring in base relation R

insertions			deletions		
id	**X**	**Y**	**id**	**X**	**Y**
21	25	33	1	56	41
22	18	64	2	58	62
23	97	83	3	15	97
24	31	50	4	78	86
25	53	82	5	69	10
			7	12	43
			8	74	76
			10	95	92
			11	34	51
			12	27	36
			13	19	25
			15	91	82
			16	84	65
			17	41	59
			20	47	27

Fig. 5 The view V prior and subsequent to updates

V		
id	**Z**	
10	929	Deleted
23	872	Inserted
15	847	Deleted
4	836	Deleted
14	771	
8	754	Deleted
25	733	Inserted
3	724	Deleted

V	
id	**Z**
23	872
14	771
25	733

tuples of V whereas the light shading denotes the extra top-3 tuples in order to have top-k_{comp} tuples.

4 Generalization of the problem

The above problem can be generalized for a relation R containing more than 2 attributes. Assume that the relation is of the form $R(ID, X_1, X_2, \ldots, X_n)$ and the scoring function of the view includes all the attributes X_i or a number of them. The problem then can be generalized as:

4.1 Formal definition of the problem generalized for more than two attributes

Given a base relation $R(ID, X_1, X_2, \ldots, X_n)$ that originally contains N tuples, a materialized view V that contains top-k tuples of the form (id, val) where val is the score according to a function $f(x_1, x_2, \ldots, x_n) = a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_n \cdot x_n$ and $a_1, a_2, \ldots, a_n$ are constant parameters, the update ratios Λ_{ins}, Λ_{del} and Λ_{upd} for insertions, deletions and updates respectively over the base relation R,

Compute k_{comp} that is of the form $k_{comp} = k + \Delta k$.

Such that the view will contain at least k tuples, $k \leq k_{comp}$, with probability p, after a period T.

The solution to the problem is similar to the previous three-step method which leads to the computation of (5). This is because the computation of k_{comp} from (5) is independent of the attributes that participate in the scoring function of V.

4.2 Formal definition of the problem generalized for non-linear monotonic functions

Given a base relation $R(ID, X_1, X_2, \ldots, X_n)$ that originally contains N tuples, a materialized view V that contains top-k tuples of the form (id, val) where val is the score according to a monotone function $f(x_1, x_2, \ldots, x_n)$, the update ratios Λ_{ins}, Λ_{del} and Λ_{upd} for insertions, deletions and updates respectively over the base relation R,

Compute k_{comp} that is of the form $k_{comp} = k + \Delta k$.

Such that the view will contain at least k tuples, $k \leq k_{comp}$, with probability p, after a period T.

In general, the scoring function of the view can be any monotonic function and not obligatory a linear function. The monotonic property is important in order to make use of the ECDF distribution function. Remember that ECDF returns the values of a function $F(x)$ such that $F_n(x)$ represents the proportion of observations in a sample less than or equal to x. Therefore, it is necessary that the values among a sample have an order. In other words, for the setting of our problem, the values of the sample are the tuples and their score according to the scoring function of V.

5 Multiple view updates

So far, our deliberations have been focused on the fine tuning of the size of a materialized view in order to sustain high update rates. The next step in our investigation of the field of top-k materialized view refreshment is to consider the case where more than one views need to be materialized. We will split the overall problem in two parts:

The first problem that we consider concerns the *dominance* of a view over another and how this reflects to the view refreshment problem. In other words, we investigate whether we can efficiently infer when the updates over a view directly affect the materialized contents of another view. Formally, assume a relation $R(ID, X, Y, \ldots)$ and two materialized views $V_1(ID, X, Y, s_1)$ and $V_2(ID, X, Y, s_2)$ that contain k_1 and k_2 tuples respectively. The score s_1 of V_1 is defined as $s_1 = a_1 \cdot x + b_1 \cdot y$ and the score s_2 of V_2 is defined as $s_2 = a_2 \cdot x + b_2 \cdot y (a_1, a_2, b_1, b_2$ are positive parameters). Assume that updates occur at the relation R, and one of the views is affected by them (i.e., its extent has to be updated). Then, the question that arises is whether it is possible to know a-priori if the impact of these updates deterministically results in the necessity to update the other view too. We provide guarantees for this case via a geometrical representation of the views and their scoring equations and we can safely determine the effect of an update on a view on the basis of its effect on another view.

The second problem that we consider involves the design of an efficient structure for a large set of top-k materialized views in order to speed up their maintenance. The constructed structure is based on the abovementioned dominance relationship among the views. We introduce hierarchies for the views and test batches of updates over

the bottom of the hierarchies. If the updates affect the bottom view, its immediate ancestors are candidates for being affected by the updates; otherwise, we can surely alleviate them from the burden of being tested against the update under examination. Obviously, the same pattern recursively propagates throughout all the hierarchy as long as a member of the hierarchy is affected.

The structure of this section is as follows. First, we start with preliminary ideas coming from the related literature and subsequently, we expand these results to discuss the case of view dominance. The third part of the section involves the discussion of view maintenance for large sets of views.

5.1 Preliminaries & background

In this subsection, we provide some background from the related literature on the problem. Our results build upon the findings of [2] and the LPTA method.

The key intuition of the LPTA algorithm can be visualized through a geometric representation. Assume a relation $R(id, X, Y)$ where the domains of X and Y are normalized over the interval [0, 1]. Apart form the base views V_x and V_y, assume two materialized views $V_u(id, Score_1)$ and $V_d(id, Score_2)$. Scores $Score_1$ and $Score_2$ are defined as linear functions over the attributes of the relation R. In addition, assume a query Q with a linear scoring function as well. The scoring functions of the views and the query can be depicted as lines. In particular, the line of a linear scoring function of the form $w(a \cdot x + y) = score$ is depicted as: $y = a^{-1} \cdot x$. Since the line is perpendicular to the scoring function, the product of their slopes should be equal to -1. The linear scoring function is depicted as its perpendicular line for the reason that the score of a tuple $t(id, x, y)$ in regards to the scoring function can be found by projecting that point over the corresponding line. In Fig. 6a we depict a view V_u and a query Q via the corresponding lines. Assume that the tuple with the k-th largest score according to Q is denoted as M. In addition, AB denotes the line that passes through M and is perpendicular to the line Q. Then, the top-k tuples according to Q belong in the region of the triangle ABR. This is due to the fact that top-k tuples will have a score higher than the score of the k-th tuple. The only possible points that can have a higher score than the point M are contained in the triangle ABR.

Assume now we want to answer the query Q by using the tuples stored in a materialized view V. The way LPTA proceeds, is by performing sorted accesses over the tuples of V. In the geometric representation, this can be visualized as sweeping a line perpendicular to the line of the view towards the point $O(0, 0)$. The order of tuples read by LPTA through sorted accesses over V is identical to the order of the points met by sweeping the line towards O.

In case only V_u is available, the stopping condition for the algorithm is reached when the sweeping line crosses position A_1B. This occurs because, the view should encounter all tuples whose score in respect to Q are at least equal to the score of the point B. Remember that points M and B have the same score in regards to Q and therefore, the region below the line A_1B does not contain any tuples with score greater than the score of M. Similarly, in case only view V_d is available, the stopping condition is reached when the sweeping line crosses position AB_2. In case both views V_u and V_d are available, the stopping condition is reached when the sweeping lines intersect in a point that lies on the line AB where in Fig. 6c is denoted as S.

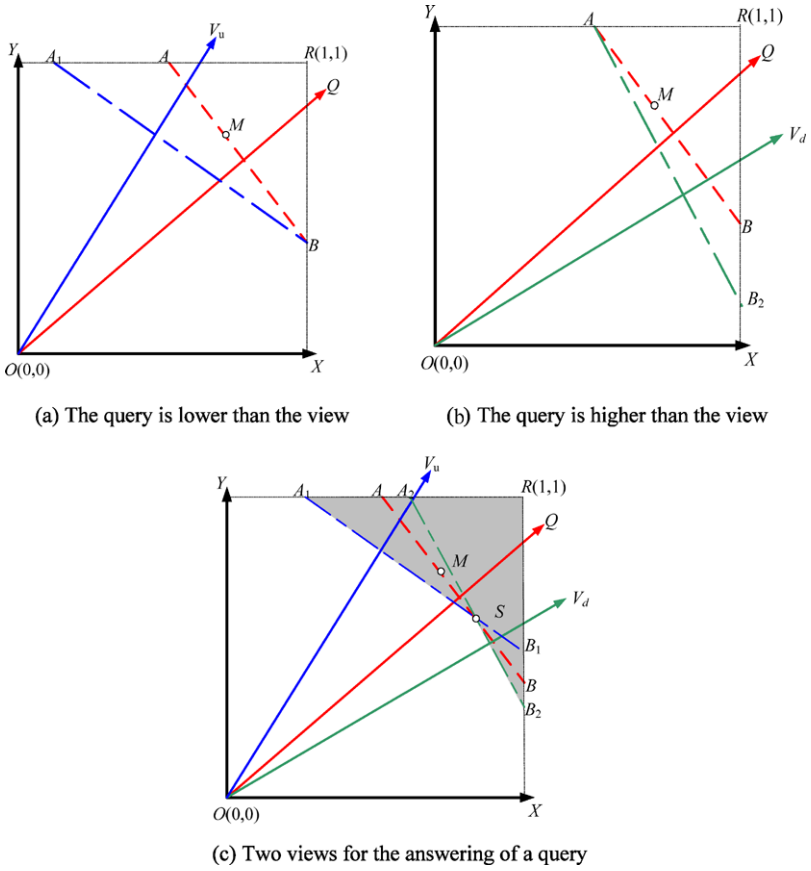


Fig. 6 Visual demonstration of the LPTA technique for query answering top-k via views

In the first case, where only V_u is used for answering Q, the number of sorted accesses performed through LPTA is the number of points that belong in the region of the triangle A_1BR. Correspondingly, if only V_d is used, the number of points that belong in the region of the triangle AB_2R is the number of sorted accesses LPTA will perform.

The best choice of the set of views that will answer Q depends upon the number of points that will be accessed, since the points accessed is identical to the number of sorted accesses LPTA will perform. Assume that the number of tuples visited when only V_u is used (i.e., the number of points that belong in the triangle A_1BR) is T_1. The number of tuples visited when only V_d is used (i.e., the number of points that belong in the triangle AB_2R) is denoted as T_2. The number of tuples visited when both views V_u and V_d are used (i.e., the number of points in the region A_1SB_2R which is the grayed area in Fig. 6c) is denoted as T_3. Then, V_u will be preferred in case T_1 is less than T_2 and less than T_3. Respectively, view V_d will be preferred when T_2 is less than T_1 and less than T_3. Finally, both views would be preferred in case T_3 is less than T_1 and T_2.

5.2 View nucleation

Assume a relation $R(ID, X, Y, \ldots)$ and a materialized view $V(ID, X, Y, s)$ that contains k tuples, scored via s which is defined as $s = w_x \cdot x + w_y \cdot y = w \cdot (a \cdot x + y)$. The parameters w_x, w_y, w, a are positive numbers. To simplify notation, we will often denote the view as $V(a, k)$. Assume now a relation $R(ID, X, Y, \ldots)$ and two materialized views $V_1(ID, X, Y, s_1)$ and $V_2(ID, X, Y, s_2)$ that contain k_1 and k_2 tuples respectively, with the score s_1 of V_1 defined as $s_1 = a_1 \cdot x + b_1 \cdot y$ and the score s_2 of V_2 defined as $s_2 = a_2 \cdot x + b_2 \cdot y$. All a_1, a_2, b_1, b_2 are positive numbers. Assume now that updates occur to the base relation and they must be propagated to the views. In a typical relational situation with SPJ queries, we would say that a view V_1 is contained within view V_2, if the extent (i.e., the materialized tuples) of view V_1 is always a subset of the extent of view V_2. In our case, due to the fact that the scores of the materialized tuples are different, we slightly tweak the terminology and instead of the 'containment' terms we employ a terminology around the notion of 'nucleus'.

Definition Assume a relation $R(ID, X, Y, \ldots)$ and two materialized views $V_1(a_1, k_1)$ and $V_2(a_2, k_2)$. A view V_2 *nucleates* a view V_1 if for each tuple $t(t.id, t.x, t.y, \ldots) \in R$ that belongs to the extent of V_2 as a tuple $t_2(t.id, t.x, t.y, s_2(t)) \in V_2$ (i.e., with a score $s_2(t)$), a respective tuple $t_1(t.id, t.x, t.y, s_1(t))$ obligatorily belongs to the extent of V_1. We denote this nucleation as $V_2 \subseteq V_1$.

Definition Two views $V_1(a_1, k_1)$ and $V_2(a_2, k_2)$ are *nucleus equivalent* if both V_2 nucleates V_1 and V_1 nucleates V_2.

Clearly, the main idea behind nucleation is that despite the difference in scores, the 'nucleus' of a tuple (i.e., the tuple identifier and the scoring attributes) are the same in the respective materialized tuples.

5.3 Updates for nucleated views

Can we efficiently decide when a view V_1 is nucleated by another view V_2? In this subsection, we will deal with this problem based on an analysis conducted via a geometric representation. Remember that the treatment of Das et al. [2] results in characterizing each view via a line that is laid in the space X, Y and starts from the beginning of the two axes. Specifically, assume the views V_1 and V_2 defined as $V_1(ID, X, Y, s_1)$ and $V_2(ID, X, Y, s_2)$ that contain k_1 and k_2 tuples respectively, with the score s_1 of V_1 defined as $s_1 = a_1 \cdot x + b_1 \cdot y$ and the score s_2 of V_2 defined as $s_2 = a_2 \cdot x + b_2 \cdot y$. These two views are characterized by the lines $y = b_1 \cdot a_1^{-1} \cdot x$ and $y = b_2 \cdot a_2^{-1} \cdot x$ respectively. There are two cases depending on the scoring functions of V_1 and V_2 and, consequently, on the slopes of their characteristic lines. The first case is trivial in the sense that the two views are practically characterized by the same line. The second case concerns the typical situation when the lines of the two views are different. In the sequel, we discuss these cases in more detail.

Case 1: $\frac{a_1}{b_1} = \frac{a_2}{b_2}$

In this situation, the equation of V_1 is proportional to the equation of V_2. Without loss of generality assume that the equation of V_1 is $s_1 = a_1 \cdot x + b_1 \cdot y$ and the equation

of V_2 is $s_2 = \lambda(a_1 \cdot x + b_1 \cdot y)$ where $\lambda \in \Re^+$. Then, the line that characterizes both views is $y = b_1 \cdot a_1^{-1} \cdot x$. There are two sub-cases in this situation.

Case 1.1: $k_1 = k_2$. In addition, assume that both views contain the same number of tuples, i.e., $k_1 = k_2$. In this case, any update affecting V_1 will definitely affect V_2 and vice versa. The only difference between the results of the two views will be the score of their tuples. Obviously, if V_1 contains a tuple t with score $s_1(t)$ then the same tuple will belong in V_2 but with score $s_2(t) = \lambda \cdot s_1(t)$.

Lemma *If the equation of a view V_1 is proportional to the equation of a view V_2 with the same extent size k of materialized tuples, then they both contain the exact same tuples (i.e., they are nucleus equivalent) with the same ordering.*

Proof Assume that the equation of V_1 is $s_1 = a_1 \cdot x + b_1 \cdot y$ and the equation of V_2 is $s_2 = \lambda(a_1 \cdot x + b_1 \cdot y)$ where $\lambda \in \Re^+$. In addition, assume $t_k(x_k, y_k)$ is the last tuple in V_1. Then for any tuple $t(x_t, y_t)$ from V_1, obviously by definition $s_1(t) \geq s_1(t_k)$. In other words, $a_1 \cdot x_t + b_1 \cdot y_t \geq a_1 \cdot x_{tk} + b_1 \cdot y_{tk}$. Multiplying this inequality with the proportion λ, we get $\lambda(a_1 \cdot x_t + b_1 \cdot y_t) \geq \lambda(a_1 \cdot x_{tk} + b_1 \cdot y_{tk})$. This states that $s_2(t) \geq s_2(t_k)$ for every tuple t from V_1. However, the last inequality is the definition of the top-k tuples of V_2. Therefore, any tuple in V_1 will be in V_2 as well. In addition, if for two tuples t_1 and t_2 from V_1 we know that $s_1(t_1) \geq s_1(t_2)$ then by multiplying the inequality with the parameter λ we get $s_2(t_1) \geq s_2(t_2)$. This proves that tuples t_1 and t_2 appear with the same ordering in V_2 as well. □

Corollary *If the equation of a view V_1 is proportional to the equation of a view V_2 with the same extent size k of materialized tuples, whenever V_1 is affected by an update, V_2 will be affected as well and vice versa.*

Proof Assume a tuple $t(x_t, y_t)$ being updated (inserted or deleted) in R and t affects V_1 with score $s_1(t)$. This means that $s_1(t) = a_1 \cdot x_t + b_1 \cdot y_t$ and $s_1(t) \geq s_1(t_k)$, where t_k is the last tuple materialized in V_1. Multiplying the above inequality by the parameter λ we get $\lambda \cdot s_1(t) \geq \lambda \cdot s_1(t_k)$ which can be written as $s_2(t) \geq s_2(t_k)$. From the above lemma t_k is also the last materialized tuple in V_2. Therefore, tuple t has a higher score than t_k for V_2 as well. Therefore, tuple will also affect V_2. □

Case 1.2: $k_1 < k_2$. Consider now the case where the equations of the two views V_1 and V_2 are still proportional, but $k_1 < k_2$ (which means that V_1 contains less tuples than V_2). In this case, V_1 nucleates V_2 and any update affecting V_1 will definitely affect V_2 as well.

Corollary *If the equation of a view $V_1(a, k_1)$ is proportional to the equation of a view $V_2(a, k_2)$ and $k_1 < k_2$, V_1 nucleates V_2.*

Proof According to the above lemma, the top-k_1 tuples are exactly the same for both views. The inverse however, does not always hold. This is because an update occurring in V_2 might be affecting the tuples that are ranked below k_1 and thus, the k_1 tuples of V_1 will not suffer any change. Obviously, if an update occurring in V_2 affects the top-k_1 tuples then it will affect V_1 as well. □

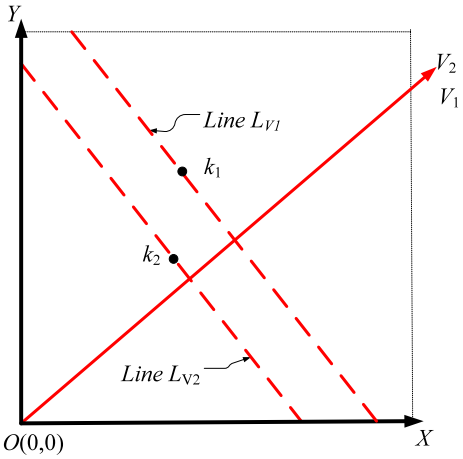


Fig. 7 Both views are of proportional equations

Case 2: $\frac{a_1}{b_1} \neq \frac{a_2}{b_2}$

In this situation, the equations of the two views are completely different. In this case, since the equations of the two views are not proportional, the only piece of information that can be used in order to conduct a conclusion with respect to the nucleation of the two views is the position of the last tuple of each view. Again, assume two views $V_1(ID, X, Y, s_1)$ and $V_2(ID, X, Y, s_2)$ with k_1 and k_2 tuples respectively where score s_1 is defined as $s_1 = a_1 \cdot x + b_1 \cdot y$ and s_2 is defined as $s_2 = a_2 \cdot x + b_2 \cdot y$. The lines that characterize the two views are V_1: $y = b_1 \cdot a_1^{-1} \cdot x$ and V_2: $y = b_2 \cdot a_2^{-1} \cdot x$ respectively (see Fig. 8 or Fig. 9). Let t_{k_1} be the last tuple materialized in V_1 with score $s_1(t_{k_1})$ and L_1 be the line which is vertical to the line of V_1 and passes from point t_{k_1}. The area above the line L_1 contains the top-k_1 tuples with respect to V_1. Now, take the line L_2, which is vertical to V_2 and passes through the point t_{k_2}, where t_{k2} is the last tuple materialized in V_2. The area above line L_2 contains points that belong to V_2. In addition, let I denote the point where L_1 and L_2 intersect.

The position of the intersection point I is critical in regards to the knowledge of whether updates affecting one view will affect the other view or not. Assume that the active domains of attributes X and Y are $X \in [x_{min}, x_{max}]$ and $Y \in [y_{min}, y_{max}]$. We will employ the term *active area* to refer to the region in which any tuple from relation R belongs. This is constrained within a rectangle defined by the points (x_{min}, y_{min}) and (x_{max}, y_{max}). Checking whether point I lies inside the active area or not can be easily done when the last tuple of each view is known. Line L_1 is expressed as: $a_1 \cdot x + b_1 \cdot y = s_1(t_{k_1})$ and line L_2 is expressed as: $a_2 \cdot x + b_2 \cdot y = s_2(t_{k_2})$. Therefore, the coordinates of point $I(x_I, y_I)$ can be found by solving the linear system of L_1 and L_2. Specifically,

$$x_I = (a_1 \cdot b_2 - a_2 \cdot b_1)^{-1} \cdot (b_2 \cdot s_1(t_{k_1}) - b_1 \cdot s_2(t_{k_2})) \quad \text{and}$$

$$y_I = (a_1 \cdot b_2 - a_2 \cdot b_1)^{-1} \cdot (a_1 \cdot s_2(t_{k_2}) - a_2 \cdot s_1(t_{k_1})).$$

Depending on the position of where point I lies we have the following cases:

Case 2.1: *Point I lies outside of the active area*. Point I lies outside of the active area if at least one of its coordinates x_I, y_I does not belong in the active domains of X

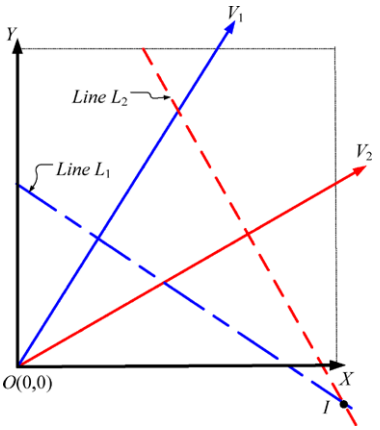


Fig. 8 Intersection of two views outside the active area

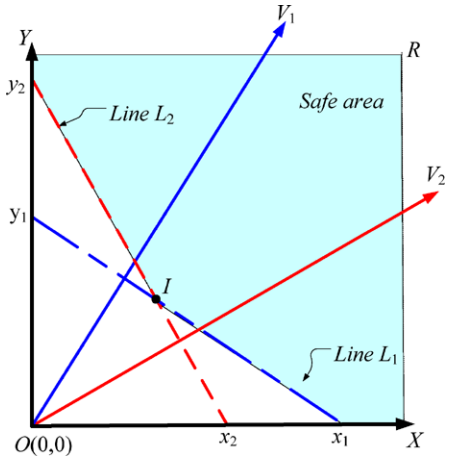


Fig. 9 Intersection of two views inside the active area

and Y respectively. In fact, in case point I lies outside the active area (see Fig. 8), then all tuples materialized in one view are also materialized in the other view as well. This situation indicates that whenever an update occurs in V_2, this will definitely affect V_1 as well. The inverse however is not always true.

In Fig. 8, tuples of V_2 also belong in V_1 and V_2 nucleates V_1. In other words, V_2 is a subset of V_1 in the sense that any tuple in V_2 will be part of V_1 but with a different ranking and score.

Case 2.2: *Point I lies inside the active area*. Point I lies inside the active area if both of its coordinates x_I, y_I belong in the active domains of X and Y respectively. In case point I lies within the active area, there is no clear guarantee of the way the views are affected when updates occur. However, there is a sub-area which we refer to as *safe area*, where both views will be affected in the same way. Observe Fig. 9, where the safe area is the convex defined by the points y_2, I, x_1, R. This area contains points that both belong in V_1 and V_2. If an update occurs within this safe area then if one view is affected then obviously the other view will be affected.

On the other hand, there are two critical areas where an update might occur and affect one view but not the other. These two critical areas are the two triangles tr_1: $y_1 y_2 I$ and tr_2: $x_1 x_2 I$. Assume the relation R is updated with a tuple t that falls within the triangle tr_1. This means that either t is inserted in R and its representation lies within tr_1, or t belonging in tr_1 is deleted from R. Then, t will affect V_1, but will leave V_2 unaffected. Similarly, if tuple t falls within the triangle tr_2, then V_2 will suffer changes whereas V_1 will remain unchanged.

Case 2.3: *Special case*. Assume two views $V_1(a_1, k_1)$ and $V_2(a_2, k_2)$ as the ones depicted in Fig. 9, where point I is within the active area. The safe area of these two views is the convex defined by the points: y_2, I, x_1, R. The main observation that can be made is that *the tuples in the safe area are common* and therefore, the two views share the same set of top-k tuples, $k \leq k_1, k_2$ (although, possibly with different ordering for each view, since each point in the safe area has a different score for each of the two views). The areas outside the safe area contain $k_1 - k$ and $k_2 - k$ tuples for each view, respectively.

In addition, assume now that both (i) $k_1 = k_2$ and (ii) the two critical regions tr_1: $y_1 y_2 I$ and tr_2: $x_1 x_2 I$ are void of tuples. In such a case when an update occurs, a conclusion can be conducted depending on the type of the update (i.e., insertion or deletion):

- If the update is a deletion and affects one of the views, then it will definitely affect the other view.
- However, if an insertion occurs and affects one of the views, then depending on the position of the insertion the other view might be or not affected. This depends on whether the insertion lies within the safe area or in one of the non-common triangles.

5.3.1 Discussion & summary

It is important to stress that *the nucleation relationship of the two views is typically dependent on the specific instances (expect for special cases) and has to be reevaluated each time that updates occur*.

Whenever an update occurs that affects at least one of the views, the position of its respective line (L_1 and/or L_2) is altered. In fact, when an insertion occurs in at least one of the views, the position of its respective line is moved towards the upper right part of the active area (or, infinity, if one chooses to think without active areas). Similarly, when a deletion occurs in a view, its respective line is moved towards the beginning of the axes. Therefore, lines L_1 and/or L_2 should be recomputed after every update that affects at least one of the views. Consequently, point I should be recomputed, too. This might also cause the change from the situation where I is outside the active area to the situation where I is inside the active area and vice versa.

Combining the above cases the following theorem occurs (the proof is obvious by referring to the lemmas and discussions of this section).

Theorem *Assume two views* $V_1(ID, X, Y, s_1)$ *and* $V_2(ID, X, Y, s_2)$ *that contain* k_1 *and* k_2 *tuples and have their scores defined as* $s_1 = a_1 \cdot x + b_1 \cdot y$ *and* $s_2 = a_2 \cdot x +$

$b_2 \cdot y$, *respectively. In addition, without loss of generality, assume for the slopes of the lines* L_1 *and* L_2 *that* $\frac{a_1}{b_1} \leq \frac{a_2}{b_2}$. *When updates occur in the relation* R *and the view* V_1 *is affected, then, the view* V_2 *will be affected if one of the following holds*:

(i) *The scoring function of* V_1 *is proportional to the scoring function of* V_2 *and* $k_1 \leq k_2$
(ii) *The intersection point* I *of* L_1 *and* L_2 *lies outside the active area, and* L_2 *is above* L_1
(iii) *The intersection point* I *lies inside the active area, critical areas* tr_1 *and* tr_2 *are void of tuples and updates are only deletions.*
(iv) *The intersection point* I *lies inside the active area, critical areas* tr_1 *and* tr_2 *are void of tuples and insertions occur only within the safe area.*

5.4 Updating multiple nucleated views

Assume a relation $R(ID, X, Y, \ldots)$ containing initially n tuples. In addition, assume that our user requirements allow us to structure the updates that occur in R in a batch way, with ΔR^+, ΔR^- denoting the insertions and deletions of a batch respectively. Assume a set of m materialized views $\mathbf{V} = \{V_i(ID, X, Y, s_i) | 1 \leq i \leq m\}$ where each view V_i contains k_i tuples with score s_i defined as $s_i = a_i \cdot x + b_i \cdot y$. When updates occur in R, the set of views $\mathbf{V}$ should be maintained appropriately. In a naïve manner, ΔR^+ and ΔR^- would be checked over each view of the set $\mathbf{V}$. However, if there are nucleation relationships among them, the update process can be done more efficiently. In this section we describe an algorithm that updates a set of views by taking advantage of the nucleation relationships among them.

Representation of nucleation relationships as hierarchy paths Assume that there exist several nucleation relationships among the set of views $\mathbf{V}$. Taking into consideration the nucleation between views, we can construct a number of hierarchy paths among them. Each hierarchy path will contain the views that are related by nucleation relationships. As a simple example, assume that V_1 nucleates V_2 and V_2 nucleates V_3. This can be depicted as a hierarchy shown in Fig. 10 where the nucleation relationship is represented as an ancestor-descendant relationship (i.e., the fact that V_1 nucleates V_2 is depicted as V_1 being the ancestor of V_2). In other words, *when a view* V_i *is an ancestor of a view* V_j *in a hierarchy path, all tuple ids of* V_i *are also contained in the materialized tuples of* V_j *at this specific point in time*. Following the same example, the hierarchy path H_1 from Fig. 10 indicates that all the tuples materialized in V_1 are also materialized in V_2 and all tuples materialized in V_2 are materialized in V_3. Since, tuples materialized in V_1 are also in V_2 and all tuples from V_2 are materialized in V_3, by induction, all tuples in V_1 are also part of the materialized tuples in V_3 as well. Therefore, when an update affects a view that is part of a hierarchy path, then all its descendants will be affected by this update. On the other hand, if an update is not affecting the lowest view from a hierarchy path, then it will definitely not affect any of its ancestors. According to this, we propose a procedure for updating a number of views based on their nucleation. *We need to stress that the relationships are instance-dependent, i.e., they depend on the contents of the views at a specific time point and they need to be reevaluated after each update occurs*. Also, this explains

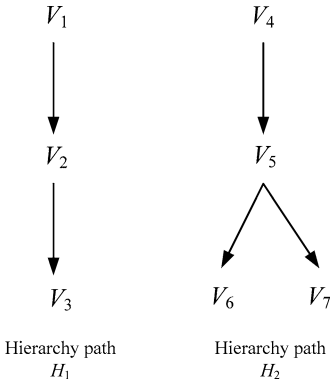


Fig. 10 Hierarchies for efficient view updates

why we structure our discussion around *batches* of updates (as opposed to individual modifications). From the theoretical point of view, individual modifications are a special case of batch updates; at the same, tuple-at-a-time updates can be an overkill when compared to the processing of batches.

Before proceeding to the algorithms that update the views of a set $\mathbf{V}$ we need to construct the algorithm that creates the hierarchy paths. Firstly, we describe the algorithm that constructs the hierarchy paths among the views from set $\mathbf{V}$.

Algorithms How can we create a number of hierarchy paths according to the nucleation relationships for a set of m views $\mathbf{V}$?

First of all, in order to create the hierarchy paths we need to find out whether two views are connected through a nucleation relationship. Thus, we need to find whether the intersection point I between two views lies inside or outside of the active area. The algorithm *Check Intersection Point* has as input: (a) the characteristics of two materialized views $V_1(ID, X, Y, s_1)^{k_1}$, with $s_1 = w_1(a_1 \cdot x + y)$ and $V_2(ID, X, Y, s_2)^{k_2}$, with $s_2 = w_2(a_2 \cdot x + y)$ where k_1 and k_2 denote the number of materialized tuples in V_1 and V_2 respectively, (b) t_{k_1} and t_{k_2} denoting the last tuple in V_1 with score $s_1(t_{k_1})$ and the last tuple in V_2 with score $s_2(t_{k_2})$ respectively, and, (c) the maximum and minimum values of attributes X and Y in R. The output of the algorithm is a distinctive value according to one of the following cases: (a) there is not a nucleation relationship, (b) V_2 nucleates V_1, (c) V_1 nucleates V_2 and (d) V_1 and V_2 are nucleus equivalent.

The algorithm first checks whether the intersection point I lies within the active area or not. In case the point I lies outside of the active area (i.e., there exists a nucleation relationship between V_1 and V_2), it calculates which view nucleates which. In order to decide upon this, we make use of the relative position of the lines L_1: $w_1(a_1 \cdot x + y) = s_1(t_{k_1})$ and L_2: $w_2(a_2 \cdot x + y) = s_2(t_{k_2})$ by comparing their slopes. In case intersection point I lies outside of the active area, the output of the algorithm is one of the following:

(a) Slope of L_1 is smaller than slope of L_2
   a. Intersection point I lies above and/or left outside of the active area, then V_2 nucleates V_1
   b. Intersection point I lies below and/or right outside of the active area, then V_1 nucleates V_2

Input: Two materialized views $V_1(ID, X, Y, s_1)^{k1}$, with $s_1 = w_1\ (a_1 \cdot x + y)$ and $V_2(ID, X, Y, s_2)^{k2}$, with $s_2 = w_2\ (a_2 \cdot x + y)$ and maximum and minimum values of attributes *X* and *Y* in *R*,

Output: the position of the intersection point I of V_1 and V_2

Begin

```
1.  x_I = (a1·b2-a2·b1)^-1·(b2·s1(tk1)- b1·s2(tk2))
2.  y_I = (a1·b2-a2·b1)^-1·(a1·s2(tk2)- a2·s1(tk1))   //compute coordinates for point I
3.  if (X_min ≤ x_I ≤ X_max and Y_min ≤ Y_I ≤ Y_max){
4.          return(no nucleation);
5.  }
6.  else {
7.          slope1 = -a1;
8.          slope2 = -a2;
9.          if ( slope1 < slope2){
10.                 if(I ∈ above and/or left of active area){
11.                         return( V2 nucleates V1);
12.                 }
13.                 else if (I ∈ below and/or right of active area){
14.                         return( V1 nucleates V2);
15.                 }
16.         }
17.         else if (slope1 > slope2){
18.                 if(I ∈ above and/or left of active area){
19.                         return( V1 nucleates V2);
20.                 }
21.                 else if (I ∈ below and/or right of active area){
22.                         return( V2 nucleates V1);
23.                 }
24.         }
25.         else return(nucleus equivalent);
26. }
```

End.

Fig. 11 Algorithm Check Intersection Point

(b) Slope of L_1 is greater than slope of L_2
   a. Intersection point I lies above and/or left outside of the active area, then V_1 nucleates V_2
   b. Intersection point I lies below and/or right outside of the active area, then V_2 nucleates V_1

(c) Slopes of L_1 and L_2 are equal, then V_1 and V_2 are nucleus equivalent.

Input: A set of views **V** = {V_i(ID, X, Y, s_i) | 1 ≤ i ≤ m},

Output: a set of hierarchy paths **H** = {H_j | 1 ≤ j ≤ l}

Begin

```
1.  H = {Hj | Hj=Vi}  //every view forms a hierarchy path
2.  For every Hj of H {
3.        Begin from root Vj of Hj {
4.              For every Hl ≠ Hj of H {
5.                    Begin from root Vl of Hl {
6.                          CI=CheckInerstectionPoint(Vj,Vl)
7.                           if (CI ≠ no nucleation){
8.                                 Remove Hj, Hl from H
9.                                 Hj = Merge { Hj, Hl }
10.                                Add Hj in H
11.                          }
12.                          Vl = Vl-1 // move a level down the path Hl
13.                   } until CI ≠ no nucleation
14.             }
15.             Vj = Vj-1  // move a level down the path Hj
16.       } until CI ≠ no nucleation
17. }
18. Return(H)
```

End.

Fig. 12 Algorithm Create Hierarchy paths

Having described the algorithm *Check Intersection Point*, we can now proceed with the algorithm that constructs the hierarchy paths among the views. Let the set of hierarchy paths be denoted as $\mathbf{H} = \{H_j | 1 \leq j \leq l\}$ where $l \leq m$. Each hierarchy path H_j is a partial order (denoted as $\prec$) among the views. Consider the hierarchy path H_1 depicted in Fig. 10. Then, for views V_1, V_2, and V_3, their partial orders are defined as: $V_1 \prec V_2 \prec V_3$. The algorithm *Create Hierarchy paths* initially treats each view of the set **V** as a hierarchy path of its own. Then, in an iterative manner, it checks if nucleation relationships exist among views of hierarchy paths starting from the root and proceeding top-down. In case there is a partial order between a view of a hierarchy path and a view of another hierarchy path (i.e., a view from one hierarchy path nucleates a view from another hierarchy path), the two hierarchy paths are merged into a new hierarchy path via the addition of an edge. The algorithm proceeds until all nucleation relationships are considered.

Now, once the hierarchy paths have been constructed, we can update the views by taking into consideration the fact that any update not affecting a lower view in a hierarchy path will not affect any of its ancestors. In fact, the algorithm works in a bottom up way for every hierarchy path constructed. Initially, we check if the updated tuples ΔR^+ and ΔR^- of R, affect the lowest views from each hierarchy path. Then, the set of ΔR^+ tuples are split into two sets: (a) an *Ignorable* set that contains all the

Input: Hierarchy paths **H** ,ΔR^+ tuples inserted in R and ΔR^- tuples deleted from R,

Output: all views in all paths of **H** are maintained

Begin

```
1.  Let V_l be the lowest view in a hierarchy path
2.  For all hierarchy paths H_j {
3.    For all V_l in H_j{
4.      V = V_l
5.      Aff+ = ΔR+
6.      Aff- = ΔR-
7.      Ign+ = {}
8.      Ign- = {}
9.      do{
10.         For all tuples t+ in Aff+ {
11.               if (t+ not affects V){
12.                     Ign+ = Ign+ ∪ {t+}
13.               }
14.         }
15.         For all tuples t- in Aff- {
16.               if (t- not affects V){
17.                     Ign- = Ign- ∪ {t-}
18.               }
19.         }
20.         Aff+ = Aff+ - Ign+
21.         Aff- = Aff- - Ign-
22.         Update V with tuples in Aff+ and Aff-
23.               V = parent(V)
                  //set the view V to be its immediate ancestor from hierarchy path
24.     } until the root of the hierarchy path is reached
25.   }
26. }
```

End.

Fig. 13 Algorithm Maintain View Updates

tuples from ΔR^+ that do not affect the view, and, (b) an *Affecting* set that contains all the rest. In the next step, the algorithm proceeds by checking which updates affect the immediate ancestor of the previous view. However, there is no need to check every update from the set ΔR^+. Instead, only updates contained in the *Affecting* set are checked. Similarly to the previous step, the *Affecting* set is now recursively split into two new sets (a) *Ignorable* and (b) *Affecting*. The same procedure is conducted for the set ΔR^-, where in each step the set of possible updates are split into (a) *Ignorable* set and (b) *Affecting* set. This procedure is repeated for every hierarchy path, until the

root of the path is reached. In addition, every time a view V is checked and the sets of *Ignorable* and *Affecting* tuples are created, V is updated in regards to the *Affecting* set of tuples.

Notice that in the *Create Hierarchy Paths* algorithm, if a view does not participate in any hierarchy path, then it creates a hierarchy path of its own. Therefore, there all views will be eventually refreshed. In other words, in the worst case where no view nucleates another one, the algorithm is simplified to the naïve algorithm where every view is checked and maintained.

After each batch of updates has been checked and performed over the views, the hierarchy paths must be reconstructed. This is due to the fact that when updates occur in views then their relative positions and therefore, their nucleation relationships, are altered. In other words, before a new batch of updates is processed, the hierarchy paths should be appropriately reconstructed. To this end, we execute algorithm *Create Hierarchy Paths*.

6 Experiments

In this section, we report on the experimental assessment of (a) the estimation of the essential view size in order to sustain a high rate of updates and (b) updating multiple views by making use of the nucleation relationship among them. We start with presenting the experimental methodology and discuss our findings over the first set of experiments and then continue by describing the experimental methodology and results over the second set of experiments.

6.1 Experimental study of sustaining high rate of deletions

Throughout this section we describe the experimental methodology and conclusions over the proposed method of sustaining a materialized view in the presence of high deletion rates. Our experimental study has been conducted towards assuring the following two goals:

1. *Effectiveness*. The first desideratum of the experimental study has been the verification of the fact that the proposed method can accurately sustain intervals with high deletion activity in the workload. In other words, the experimental goal was to verify that a top-k materialized view contains at least k items, in at least 95% of the cases.
2. *Efficiency*. The second desideratum of the experimental study has been the establishment of the fact that the computation of the exact number of auxiliary view tuples is faster than the computation of refill queries as proposed in the related literature. As well as the number of auxiliary view tuples is less than the number proposed in [19].

To achieve the first goal we have estimated k_{comp} via two methods: (a) without the fine tuning that uses the rates' variances (i.e., through formula (5)) and (b) with this fine tuning (i.e., through formula (6)). For both methods, we have computed the number of tuples that were deleted from the view, below the threshold of k.

Table 1 Experimental parameters

Size of source table R (tuples)	$\|R\|$	1×10^5, 5×10^5, 1×10^6, 2×10^6
Size of mat. view (tuples)	k	5, 10, 100, 1000
Size of update stream (pct over $\|R\|$)	λ	1/1000, 1/100
Deletion rate over insertion rate (ratio)	D/I	1.0, 1.5, 2.0

In the context of the second goal, we have measured three metrics: (a) the memory overhead for k_{comp} and k_{comp} with tuning, measured as the number of extra tuples that we need to keep in the view, (b) the time overhead for computing k_{comp} and k_{comp} with tuning as compared to the necessary time to compute the refill queries of [19] and (c) the time needed to compute the formula for k_{comp}. Again, we have evaluated these metrics using both the aforementioned methods.

In all our experiments we have used one relation $R(RID, X, Y)$ and one view $V(RID, score)$ with a formula $score = 3X + 7Y$. The parameters that we have tested for their effect over the aforementioned measures are: (a) the number of relation tuples, (b) the number of materialized top-k results, (c) the fraction of the delete rate, over the insertion rate and (d) the percentage of the update stream over the relation size. We have not altered the time window T in our experiments; nevertheless, this is equivalent to varying the last parameter (denoted as λ), which measures the amount of modifications that take place as a percentage of the size of R. In other words, it is equivalent to increase the modifications number instead of reducing the window size.

We have tested the method over data whose attributes X and Y followed the Gaussian (with mean $\mu = 50$ and variance $\sigma = 10$ for both X, Y), negative exponential (with $a = 1.5$ for X and $a = 2.0$ for Y) and Zipf distributions (with $a = 2.1$ for both X, Y). The notation for the parameters and the specific values that we have used are listed in Table 1. All the experiments were conducted on a 2.8 GHz Pentium4 PC with 1 GB of memory.

6.1.1 Effectiveness of the method

The effectiveness of the method is demonstrated in Fig. 14 and Fig. 15. We present results organized by the data distribution of the attributes and compare two methods for computing k_{comp}, (a) the method including the fine-tuning part and (b) the method simply based on formula (5). We have conducted the full range of combinations of the values listed in Table 1.

In Fig. 14, we fix D/I to 1.5 and k to 1000 (the largest possible value) and vary the size of R (in the X-axis) and the update stream size (in different lines in the Figure). Each experiment has been conducted 5 times. We measure both the average and the maximum number of misses. In Fig. 15 we report only the maximum number of misses, as it appears to be in analogy with the average in almost all the cases, and we vary k and D/I, while keeping R fixed to 1M rows and λ to 1%. The findings are as follows:

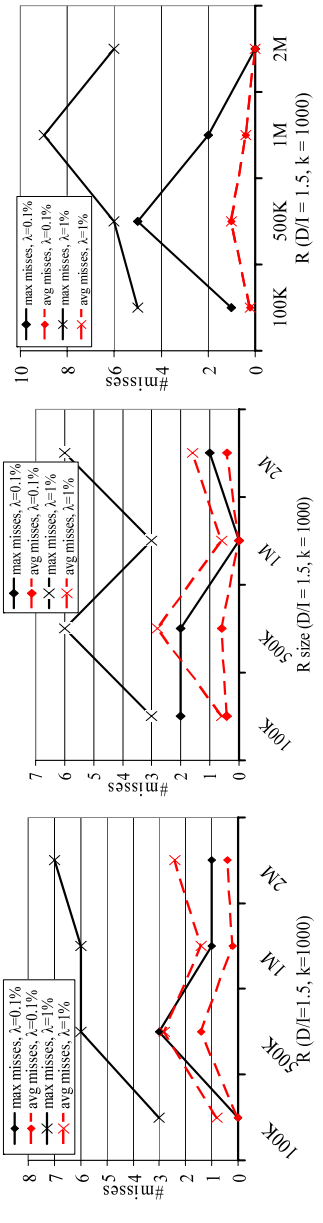


Fig. 14 Maximum and average misses as a function of $|R|$ and λ

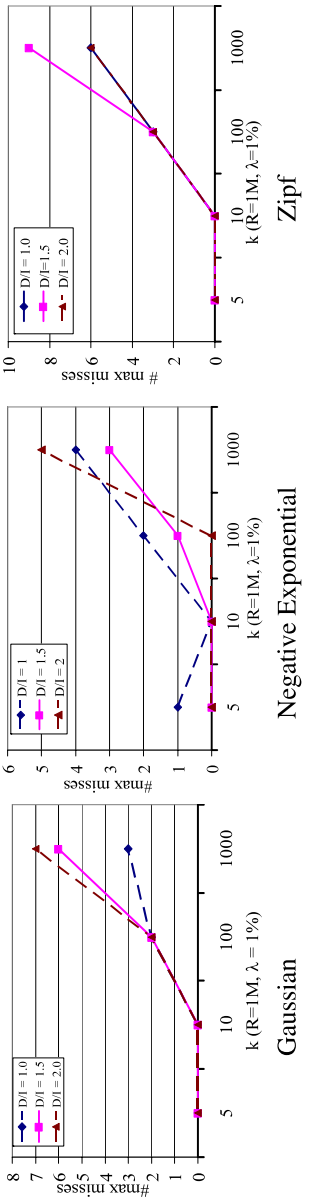
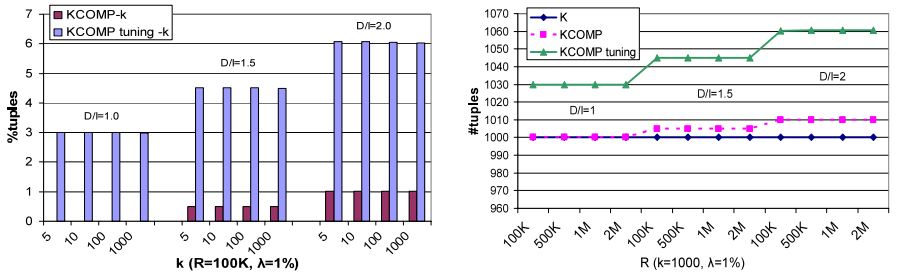


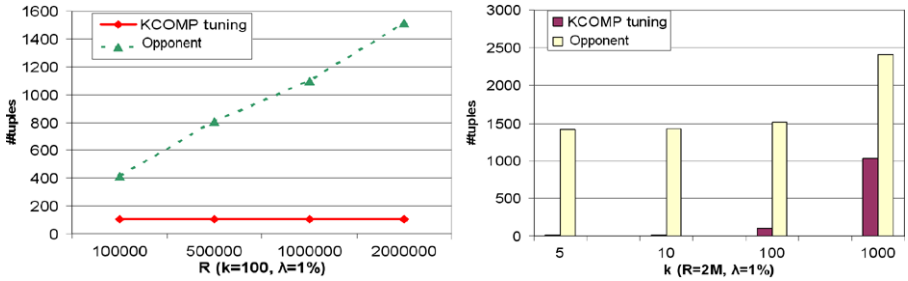
Fig. 15 Maximum misses as a function of k and D/l



(a) Percentage of extra tuples as a function of k and D/I

(b) Number of extra tuples as a function of R and D/I

Fig. 16 Comparison of k, k_{comp}, and k_{comp} with tuning



(a) Number of extra tuples as a function of R

(b) Number of extra tuples as a function of k

Fig. 17 Comparison of k_{comp} with tuning and [19]

- The fine tuning method gives 0 losses, and thus describes the bold line lying on top of the X-axis in Figs. 14 and 15.
- If the fine tuning was not included, misses would have been encountered. In cases where insertions are close to deletions, the underestimation of the value of k_{comp} would lead to potentially important errors (in the Zipf case, errors have come up to 9 misses which is almost 1% of the top-k view size). The 5% misses that could be expected after the fine tuning due to the 95% confidence can be attributed to the size of the update streams; had the update stream been larger, misses would have occurred.
- It is also interesting how the distribution of data affects the stability of the error (Gaussian seems to converge, as expected, whereas the Zipf drops when the percentage of k is small over R, as the hot values are rather fixed).

Our experimental study has also explored the case of larger workloads of updates that may occur in the base relation. Specifically, the experiments were conducted by making use of three different scenarios of possible update workloads. All the scenarios were applied over a database of 1 million records with attributes x and y following the Gaussian distribution (in any case, the distribution of data does not have an effect to the effectiveness of the method as our aforementioned experiments have demonstrated). Every experiment was conducted 100 times in order to eliminate cases where the actual values of the tuples inserted or the tuples deleted contribute significantly to

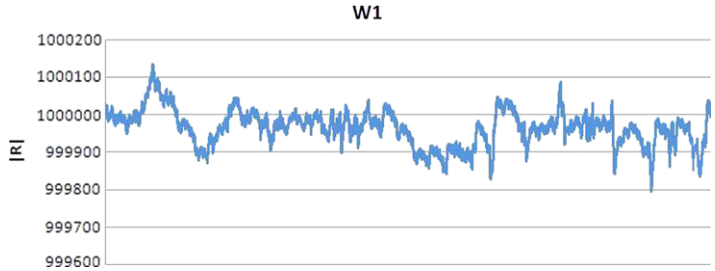


Fig. 18 Size of relation R ($|R|$) over time as insertions and deletions take place for workload W_1 having a ratio of deletion rate over insertion rate $D/I = 1.0$

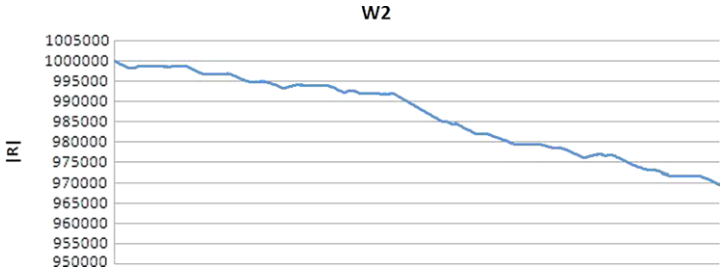


Fig. 19 Size of relation R ($|R|$) over time as insertions and deletions take place for workload W_2 having a ratio of deletion rate over insertion rate $D/I \approx 2.0$

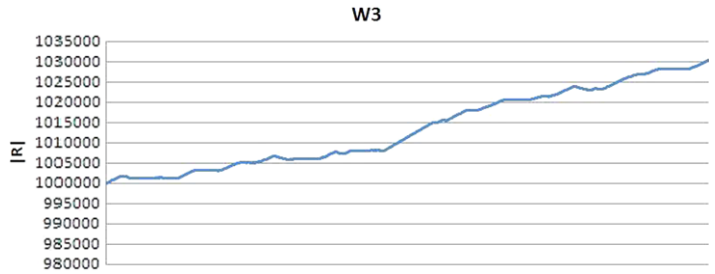


Fig. 20 Size of relation R ($|R|$) over time as insertions and deletions take place for workload W_3 having a ratio of deletion rate over insertion rate $D/I \approx 0.5$

the experimental results. All three workloads contain 91 thousand updates occurring in the base relation and in all three of them the insertions and deletions do not occur uniformly. There are peaks and valleys of high insertion and deletion rates throughout all three scenarios. The first workload (denoted as W_1), depicted in Fig. 18, contains updates where insertions and deletions are of the same size (specifically, 45500 insertions and 45500 deletions). The two other workloads are constructed in order to test the method to extreme cases. In the second workload (denoted as W_2), shown in Fig. 19, deletions are approximately twice as many as the insertions (specifically, 60700 deletions and 30300 insertions). The third workload (denoted as W_3), shown in Fig. 20, is the inverse of workload W_2. Specifically, W_3 occurred by replacing in

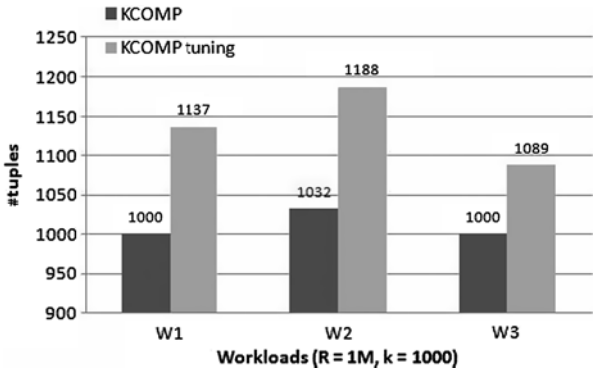


Fig. 21 Memory overhead expressed as the number of tuples stored in the view

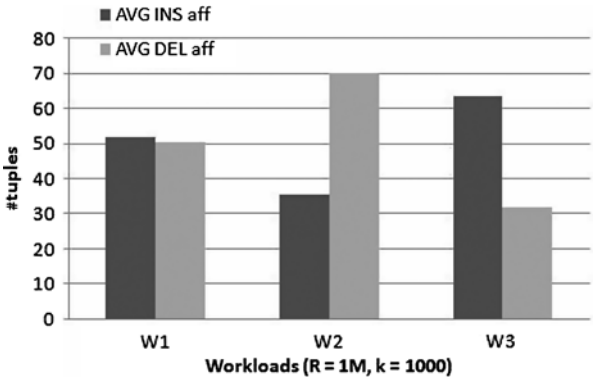


Fig. 22 Average number of Insertions and Deletions that affect the top-k tuples in the view

workload W_2 deletions with insertions and vice versa. Thus, W_3 constitutes of 60700 insertions and 30300 deletions, having a ratio of deletion rate over insertion rate approximately equal to 0.5.

In order to assure that a large number of updates will affect the top-k view results, we have set the parameter k to 1000 tuples. The resulting numbers of tuples that are either inserted or deleted in the extent of the top-k view are depicted in Fig. 22 for all the workloads.

For all these three workloads, we have counted the number of misses that occurred (as a measure of how often we would have to run refill queries) as well as the memory overhead for k_{comp} and k_{comp} with tuning, measured as the number of extra tuples that we need to keep in the view. Our findings are as follows:

- Concerning the number of misses, the number of missed tuples was exactly zero for all the three workloads and in each one of the 100 runs of every workload.
- Concerning the memory overheads, the extra tuples that we had to store for the top-1000 view of our experiments was quite low. The results for k_{comp} and k_{comp} with tuning are shown in Fig. 21 for all three workloads. Observe that in all three scenarios the number of extra tuples materialized over 1000 tuples, due to the extra tuning (i.e., the difference of k and k_{comp} with tuning) does not exceed 188 tuples. Specifically, the mixed workload W_1 requires 137 extra tuples (i.e., a 13.7% increase over k). Workload W_2 that is heavy on deletions (and therefore requires a

provision for a larger k_{comp}, in order to sustain the high deletion rate) requires an increase of 18.8% (although the deletion rate is twice as high as the insertion rate). Workload W_3 which is heavy on insertions only requires an increase of 0.89% over k. In particular, in workload W_3, formula (5) gives for k_{comp} the value of 971 tuples instead of 1000 tuples, due to the high insertion rate in regards to the deletion rate. However, in the experimental setup we have used as k_{comp} the maximum value between k and the computed value of k_{comp} from formula (5).

6.1.2 *Efficiency of the method*

We compared the values of k_{comp} without the fine tuning (i.e., through formula (5)) and k_{comp} *tuning* with this fine tuning. The comparison of the above values was conducted for all three distributions as well as for all parameters listed in Table 1. Due to the fact that our formula is independent of the distribution the tuples follow we only present some indicative results. In Fig. 16, we compare k_{comp} and k_{comp} *tuning*, (a) as a function of k, where the size of R is 100000 tuples, and, (b) as a function of the size of R where we have fixed $k = 1000$. For both of them and for all possible values of D/I the size of the update stream λ is 1% and the distribution is the Negative exponential. In Fig. 16(a), the Y-axis denotes the percentage of extra tuples materialized in regards to the value of k. For instance, when the size of R is set to 100 thousand tuples, the deletion rate over the insertion rate D/I is 1, and k is 1000 tuples, the percentage of extra tuples materialized through k_{comp} *tuning* is 3% of k, i.e., only 30 extra tuples. From both graphs in Fig. 16 we observe that k_{comp} is slightly greater than k and k_{comp} *tuning* is slightly greater than k_{comp} in all cases. The number of the auxiliary tuples in the view (i.e., k_{comp} and k_{comp} *tuning*) in the maximum case is approximately 1% and 6% respectively. Thus, the number of the auxiliary tuples does not cause a great extra memory cost.

In Fig. 17, we compare the value of k_{comp} *tuning* with the one proposed by [19]. Again, we compare the above (a) as a function of k where the size of R is set to 2M (the largest possible value) and (b) as a function of R where k is fixed to 100. In both graphs the distribution is the negative exponential. The parameter $D/I = 1$, since it is the only value that can be compared with the proposed method in [19]. We notice that the number of tuples proposed by [19] is significantly larger than the one proposed in our method. Thus the memory cost in our method is considerably less.

The second part of our experimental results had to do with the comparison of the time needed to compute the value of k_{comp} as compared to the time needed to re-compute the view as part of a refill query. The method using k_{comp} results in a single overhead, the computation of k_{comp}, and no extra refill queries. We measured the time necessary to perform the computation of k_{comp} which has consistently been negligible (practically 0 in all occasions). On the contrary, refill queries for the view computation (i.e., without k_{comp}) incur some extra overhead, depicted in Fig. 23, that measures the computation time needed for the view computation for a value of k in microseconds. Based on these measurements we can argue that the benefits from the computation of k_{comp} are evident.

Fig. 23 Time to build the top-k view (microseconds)

N	K	Gauss	Negative exponential	Zipf
100K	5	328000	348500	242000
100K	10	333000	345667	239667
100K	100	335500	343000	239667
100K	1000	395333	406000	299500
500K	5	1650667	1715500	1216333
500K	10	1650667	1713000	1208333
500K	100	1653167	1710500	1205667
500K	1000	1736667	1796167	1291833
1M	5	3298667	3429000	2427167
1M	10	3301333	3426667	2429667
1M	100	3304000	3439500	2422167
1M	1000	3403167	3520500	2606667
2M	5	6650667	6900500	5406333
2M	10	6653167	6900833	4909000
2M	100	6747167	6906000	4906500
2M	1000	6895500	7082833	4992167

6.2 Experimental study for multiple views updates

In this section we describe the experimental study and findings of maintaining multiple views by making use of the nucleation relationship among them. The experimental study has focused on proving the correctness and efficiency of the proposed method. We have implemented the algorithms described in Sect. 5.4 and compared them with a base method which we refer to as *naïve* method. The naïve method checks a batch of updates over each view independently and applies them appropriately. In order to test the correctness of the proposed nucleation method we have compared the results of the updated views with the results of applying the updates over each view independently and the outcome has been absolute identical. Having secured the correctness of our algorithms' implementation the rest of the experimental study focused on proving the efficiency of the proposed method in terms of the time needed to apply updates over multiple views. Our experiments have demonstrated that, indeed when batches of updates are applied to a multitude of top-k views, using the nucleation relationships is faster than the naïve method. Under the context of proving the efficiency of the nucleation method, we have measured the time needed to maintain multiple views in the presence of updates over the base relation, for both the nucleation and naïve method.

In all our experiments we have used a relation $R(RID, X, Y)$ where the attribute values of X and Y were generated randomly from the interval $[0, 10000]$. All the views needed to be maintained are of the form $V(RID, X, Y, score)$ where *score* is a weighted sum over the attributes X and Y. Particularly, the scoring function of the views is of the form $score = w_x \cdot X + w_y \cdot Y$, with the parameters w_x and w_y being randomly generated from the interval $[0, 1]$. The parameters that we have tested for their effect on the efficiency of the view refreshment are: (a) the number of relation tuples, (b) the maximum number of materialized top-k results within a set of views expressed as a percentage over the relation size, (c) the number of materialized views

Table 2 Experimental parameters

Size of source table R (tuples)	$\|R\|$	2×10^5, 3×10^5, 4×10^5
Max size of mat. tuples (pct over $\|R\|$)	max_k	1/100, 1/1000
Number of views	M	100, 1000
Size of insertion stream (pct over $\|R\|$)	λ	1/10, 1/100, 1/1000

needed to be maintained and (d) the percentage of the insertion stream over the relation size. We have kept the fraction of the delete rate, over the insertion rate constant and equal to 0.5.

The notation for the parameters and the specific values that we have used are listed in Table 2. All of the experiments were conducted on a 2.53 GHz Core Duo PC with 3.12 GB of memory.

In all the experiments the measure for time is expressed as number of seconds. The comparison of the time needed for the two methods has been conducted for all possible combinations of the above parameters listed in Table 2. We run every experiment five times and the results presented here are the average time. In all charts of Fig. 24 the Y-axis indicates the time needed for the two methods to apply the updates. The X-axis shows (a) the size of the source table R and (b) the size of the insertion stream. Specifically, for each possible value of $|R|$ (i.e., 200, 300 and 400 thousand tuples) X-axis also indicates the stream of insertions for all three possible values (i.e., 1/10, 1/100 and 1/1000 percentage of $|R|$). Since, the fraction of the deletion rate over the insertion rate is set to be 0.5 the number of updates occurring can be calculated as 1.5 times the value of parameter λ, times the value of parameter $|R|$. The naïve method is denoted with the darker grey color, whereas the nucleation method is presented with the lighter grey color. In all charts we can notice that the nucleation method is faster than the naïve. The title of each chart also clarifies the fixed value of the parameters M and max_k.

Graphs (a) and (b) in Fig. 24 demonstrate the time needed for applying updates over a set of 100 views. In these two graphs the maximum number of tuples materialized in each view expressed as a rate over $|R|$ is 0.1% and 1% respectively. In graph (a) of Fig. 24 the ratio time between the two methods is not that significant but still the nucleation method is faster than the naïve method. In graph (b) of Fig. 24 we observe that time needed for nucleation method is approximately half the time needed for the naïve method. This is due to the fact that the number of views is 100 and in each view the maximum number of tuples materialized is only 200, 300 and 400 respectively for each size of R. In other words, the larger the extent of the views (due to the size of k), the larger the benefits from the nucleation method are.

In graphs (c) and (d) of Fig. 24 we see the time needed for the two methods over a set of 1000 views (as opposed to 100 views for the cases of (a) and (b)). The maximum value of tuples materialized in each view is set to 0.1% and 1% respectively. In graph (c) the ratio time between the two methods ranges approximately between 2 and 4. In graph (d), the time needed for the nucleation method is approximately 4 times faster than the naïve method. Again, nucleation scales up much better than the naïve method. Moreover, if one reads Fig. 24 vertically, one can observe that the

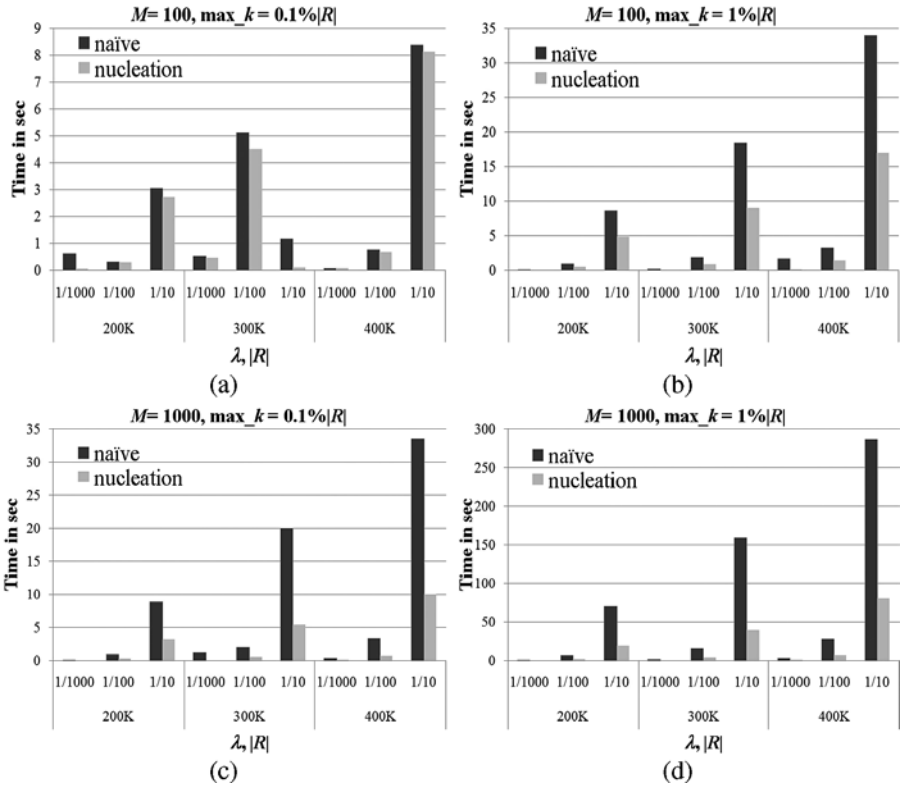


Fig. 24 Comparison between naive and nucleation method. All graphs show the time of applying updates as a function of insertion size and $|R|$

scaling capabilities involve both the extend of the view and the number of materialized views; in fact, the improvements in cases (c) and (d) where a larger number of views is maintained are significantly higher than the improvements of cases (a) and (b) where a smaller number of views is maintained.

In all the graphs of Fig. 24 we can observe that the time needed for the naïve method scales up linearly with respect to the number of updates occurring in the base relation. Considering the nucleation method the time scales up almost linearly as well.

7 Conclusions

In this paper we handle the problem of maintaining materialized top-k views and provide results in two directions. The first problem we have been concerned with has to do with the maintenance of top-k views in the presence of high deletion rates. We have provided a principled method that complements the inefficiency of the state of the art independently of the statistical properties of the data and the characteristics of the update streams. The method comprises the following steps: (a) a computation of

the rate that actually affects the materialized view, (b) a computation of the necessary extension to k in order to handle the augmented number of deletions that occur and (c) a fine tuning part that adjusts this value to take the fluctuation of the statistical properties of this value into consideration. The second problem we have been concerned with concerns the case of multiple top-k views and their efficient maintenance in the presence of updates to their base relation. We have provided theoretical guarantees for the establishment of the effect of updates to a certain view, whenever we know that another view has been updated. We have also provided algorithmic results towards the maintenance of a large number of views, via their appropriate structuring in a hierarchy of views. Our experimental results demonstrate that the proposed methods outperform the state-of-the-art and behave well when the problem parameters scale up.

Acknowledgements We would like to thank the anonymous reviewers of the paper for constructive comments concerning the intuition and the experimental validation of our approach.

References

1. Baikousi, E., Vassiliadis, P.: Tuning the top-k view update process. In: 3rd Multidisciplinary Workshop on Advances in Preference Handling (M-Pref 2007), held in conjunction with VLDB 2007, Vienna, Austria, 23 September 2007
2. Das, G., Gunopulos, D., Koudas, N., Tsirogiannis, D.: Answering top-k queries using views. In: Proc. of the 32nd VLDB Conference, pp. 451–462, Seoul, Korea, 2006
3. DeGroot, M.H., Schervish, M.J.: Probability and Statistics. Addison-Wesley, Reading (2002)
4. Fagin, R.: Combining fuzzy information from multiple systems. In: Proc. of the 15th ACM Symposium on Principles of Database Systems, pp. 216–226, Montreal, Canada, 1996
5. Fagin, R.: Fuzzy queries in multimedia database systems. In: Proc. of the 17th ACM Symposium on Principles of Database Systems, pp. 1–10, Seattle, USA, 1998
6. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. **66**(4), 614–656 (2003)
7. Graefe, G.: Dynamic query evalutation plans: Some course corrections. Bull. Tech. Comm. Data Eng. **23**(2), 3–6 (2000)
8. Güntzer, U., Balke, W.-T., Kießling, W.: Optimizing multi-feature queries for image databases. In: Proc. of the 26th VLDB Conference, pp. 419–428, Cairo, Egypt, 2000
9. Hristidis, V., Papakonstantinou, Y.: Algorithms and applications for answering ranked queries using ranked views. VLDB J. **13**(1), 49–70 (2004)
10. Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER a system for the efficient execution of multi-parametric ranked queries. In: Proc. of the ACM Special Interest Group on Management of Data Conference (SIGMOD), pp. 259–270, Santa Barbara, USA, 2001
11. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)
12. Marian, A., Bruno, N., Gravano, L.: Evaluating top-k queries over web-accessible databases. ACM Trans. Database Syst. (TODS) **29**(2), 319–362 (2004)
13. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. In: Proc. of the ACM Special Interest Group on Management of Data Conference (SIGMOD), pp. 635–646, Chicago, Illinois, USA, 2006
14. Nepal, S., Ramakrishna, M.V.: Query processing issues in image (multimedia) databases. In: Proc. of the 15th International Conference on Data Engineering (ICDE), pp. 22–29, Sydney, Australia, 1999
15. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Trans. Database Syst. (TODS) **30**(1), 41–82 (2005)
16. Schlossnagle, T.: Dissecting today's Internet traffic spikes. Posted on 15 January 2009 at http://omniti.com/seeds/dissecting-todays-internet-traffic-spikes
17. Trivedi, K.S.: Probability and Statistics with Reliability, Queuing and Computer Science Applications. Wiley, New York (2002)

18. Vlachou, A., Doulkeridis, C., Norvag, K., Vazirgiannis, M.: On efficient top-*k* query processing in highly distributed environments. In: Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 2008
19. Yi, K., Yu, H., Yang, J., Xia, G., Chen, Y.: Efficient maintenance of materialized top-*k* views. In: Proceedings of the 19th International Conference on Data Engineering (ICDE), pp. 189–200, Bangalore, India, 2003
20. Zhao, K., Tao, Y., Zhou, S.: Efficient top-*k* processing in large-scaled distributed environments. Data Knowl. Eng. **63**, 315–335 (2007)