Efficient Maintenance of Materialized Top-k Views

Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen

Abstract

We tackle the problem of maintaining materialized top-k views in this paper. Top-k queries, including `MIN` and `MAX` as important special cases, occur frequently in common database workloads. A top-k view can be materialized to improve query performance, but in general it is not self-maintainable unless it contains all tuples in the base table. Deletions and updates on the base table may cause tuples to leave the top-k view, resulting in expensive queries over the base table to "refill" the view. In this paper, we propose an algorithm that reduces the frequency of refills by maintaining a top-k' view instead of a top-k view, where k' changes at runtime between k and some $k_{\max} \geq k$. We show that in most practical cases, our algorithm can reduce the expected amortized cost of refill queries to $O(1)$ while still keeping the view small. The optimal value of $k_{\max}$ depends on the update pattern and the costs of querying the base table and updating the view. Compared with the simple approach of maintaining either the top-k view itself or a copy of the base table, our algorithm can provide orders-of-magnitude improvements in performance with appropriate $k_{\max}$ values. We show how to choose $k_{\max}$ dynamically to adapt to the actual system workload and performance at runtime, without requiring accurate prior knowledge.

Index Terms

Materialized view maintenance, top-k queries, query processing, data warehouse and repository.

A preliminary version of this paper appears in ICDE'03 [25].

Ke Yi, Hai Yu, and Jun Yang are with the Department of Computer Science, Duke University.

Gangqiang Xia and Yuguo Chen are with the Institute of Statistics and Decision Sciences, Duke University.

Efficient Maintenance of Materialized Top-k Views

I. Introduction

Top-k queries have received much attention from the database community in recent years [5], [9], [10], [3], [6]. An effective way of improving the performance of expensive queries is to maintain their results as materialized views [13]. However, incremental maintenance of materialized top-k views has been a relatively unexplored problem in the view maintenance literature. The main difficulty of this problem is that a top-k view is not *self-maintainable* [12] with respect to deletions and updates on the base table. That is, sometimes we must query the base table in order to maintain the top-k view properly; the view itself does not contain enough information required for maintenance.

For example, consider a materialized view containing 10 stocks with the highest price/earning ratios currently on the market. Suppose one of these stocks plummets, and its price/earning ratio drops below the current top 10. After this update, the view still contains the top 9 stocks, but in order to find the stock with the 10-th ranked price/earning ratio, we need to query the base table of all stocks. This query, which we call a *refill query*, can be expensive in general for a number of reasons, *e.g.*, the base table may be large, it may reside in a remote database, and the ranking criterion may involve expensive user-defined functions.

To avoid expensive refill queries over the base table, we can make a top-k view self-maintainable by augmenting it with auxiliary data, a technique well studied in data warehousing [22], [1]. For example, we may keep the $(k + 1)$-th ranked tuple as auxiliary data to help maintain a top-k

view, in the event that a tuple drops out of the top k. However, since auxiliary data must be maintained as well, we need the $(k+2)$-th ranked tuple in order to maintain the $(k+1)$-th, the $(k+3)$-th to maintain the $(k+2)$-th, *etc*. In general, to make a top-k view completely self-maintainable, we must essentially keep a copy of the entire base table, or at least an ordered index on the base table column used for ranking.

Now we are faced with a dilemma. One option is to maintain the original top-k view, which may require frequent costly refill queries. The other option is to maintain an ordered index on the entire base table, which has high storage and maintenance overhead but avoids refill queries altogether. Neither option seems completely satisfactory. Previous work on making views self-maintainable has often side-stepped the problem by not considering deletions and updates for SQL aggregates `MIN` and `MAX`, which are special cases of top-k views with $k = 1$.

Fortunately, we have a middle-ground to explore between the two extremes, without ruling out deletions and updates. This approach is based on two key observations. First, instead of requiring complete self-maintenance, we try to achieve *runtime self-maintenance* with high probability. That is, rather than devoting lots of additional resources to ensure that we never query the base table for view maintenance, we can devote much fewer additional resources and ensure that we only query the base table extremely rarely. The second observation is that a materialized view can have a dynamic definition. Instead of maintaining a top-k view, we maintain a top-k' view, where k' can change dynamically between k and some $k_{\max} \geq k$. We start with $k' = k_{\max}$, *i.e.*, a top-$k_{\max}$ view with more than the required number of tuples. We increase k' by one when an insertion or an update causes a tuple to enter the current top k' (unless k' already equals $k_{\max}$), and we decrease k' by one when a deletion or an update causes a tuple to leave the current top k'. We only query the base table when k' drops below k. By starting with $k_{\max}$ instead of k, we hope to lower the refill frequency and hence the amortized cost of view maintenance.

Beyond this conceptually simple idea, several interesting and non-trivial questions remain to be answered. Intuitively, as we increase $k_{\max}$, refill frequency decreases; on the other hand, the view takes more space, updating the view becomes more expensive, and more updates need to be applied to the view. Given these trade-offs, how do we choose right values of $k_{\max}$? What are the factors affecting the optimal $k_{\max}$ value? Under what conditions can we expect to achieve low amortized view maintenance cost with reasonably small values of $k_{\max}$? How do we choose $k_{\max}$ without accurate prior knowledge of the workload?

This paper explores in detail the issues mentioned above. Section II surveys related work. Section III describes our algorithm and cost model. Section IV explores the relationship between $k_{\max}$ and the refill frequency using the random walk model as a tool. Most importantly, we show that in most practical cases, we can reduce the expected amortized cost of refill queries to $O(1)$ with reasonably small $k_{\max}$ values. Section V considers several statistical models of base table updates and shows how to apply our analytical results in Section IV to these cases. Section VI experimentally obtains the parameters of our cost model, and demonstrates the effectiveness of our algorithm in realistic scenarios. Section VII proposes a procedure for choosing $k_{\max}$ which adapts to the actual system workload and performance at runtime.

II. Related Work

There is a large body of work on top-k queries [4], [5], [9], [10], [3], [7], [6], most of which focuses on how to evaluate these queries efficiently in various contexts. Most related to this paper is the work by Hristidis *et al*. [15], wherein they propose using materialized top-k views to speed up more complex preference queries. Their work focuses on selecting top-k views to materialize and using them to answer queries; incremental maintenance of these views is not considered. Therefore, their work is complementary to ours, and provides a good motivation for studying efficient maintenance of top-k views.

Materialized view maintenance is a well-known and well-studied problem, surveyed in [13]. The concept of self-maintenance is introduced in [2], [12], and the concept of runtime self-maintenance is introduced in [16]. The technique of using auxiliary data to make views self-maintainable is pioneered by [22], and has been successfully applied in many settings [1], [23], [17]. To the best of our knowledge, all prior work uses auxiliary data to achieve complete self-maintenance; none has considered using auxiliary data to increase the probability of runtime self-maintenance, which is the one of the key observations in this paper.

Until recently, most papers that deal with SQL `MIN` and `MAX` views (which are special cases of top-k views), *e.g.*, [11], [21], [1], [17], [24], cannot efficiently handle deletions or updates to the base table. Recent work by Palpanas *et al*. [18] proposes using *work areas* to maintain `MIN` and `MAX` views. Their approach has the same underlying idea as our algorithm in Section III, which we have developed independently. Besides this basic idea, they do not consider how to choose the size of the work area, while we make the following additional contributions: (1) we develop a probabilistic model for rigorous analysis of the algorithm; (2) we prove high-probability results that establish the effectiveness of the algorithm; and (3) we provide a procedure for choosing $k_{\max}$ (or size of the work area in their terminology) which adapts to the actual system workload and performance at runtime, without requiring accurate prior knowledge.

III. The Algorithm

Suppose we are interested in the top k tuples from a base table R of size N. We assume k is a constant much smaller than N, since typical users are interested only in a small subset of R that is most "important," *e.g.*, the ten most popular songs or the 100 most frequently accessed web sites. Suppose that tuples in R are identified by a column `id` and ranked according to the value of a column `val`. Tuples with larger values are ranked higher. For simplicity, we assume all values are distinct; in practice, ties can be broken arbitrarily using `id` values. The `val` column can be either stored explicitly in R or computed on the fly by some user-defined function. We assume that there is no index on R.`val`.

Our algorithm is conceptually very simple. We keep the top k' tuples (with `id` and `val` columns) in a materialized V, where k' can vary between k and some $k_{\max} \geq k$. Since $k \leq k'$, we can answer top-k queries using the contents of V.

We need to maintain V given the changes to the base table R. To keep our analysis clean, we only consider updates to R; that is, we assume that the identities of the N tuples in R remain fixed while their values change over time. It is straightforward to generalize our algorithm and analysis to handle insertions and deletions on R as well.

Let $v_{k'}$ be the value of the lowest ranked tuple currently in V. We assume that an update to R has the form $\langle id, val \rangle$, where val is the new value of the tuple identified by id. For each update to R, we perform an *update operation* on V. There are four cases to consider:

- The tuple identified by id is not in V, and $val < v_{k'}$. This update has no effect on V. We call this update an *ignorable update*.
- The tuple identified by id is in V, and $val > v_{k'}$. We update the value of this tuple in V to val. We call this update a *neutral update* ("neutral" in the sense that it does not change the value of k').
- The tuple identified by id is not in V, and $val > v_{k'}$. We insert $\langle id, val \rangle$ into V. We call this update a *good update* ("good" in the sense that it increases k' by one). If k' exceeds $k_{\max}$, we delete the lowest ranked tuple in V.

- The tuple identified by id is in V, and $val < v_{k'}$. We delete the updated tuple from V. We call this update a *bad update* (“bad” in the sense that it decreases k' by one). If k' drops below k, we perform a *refill operation* as described below.

The *refill operation* queries the base table and restores the size of the view to $k_{\max}$. This operation consists of the following two steps:

- Evaluate the *refill query* over R, which returns all tuples ranked between k and $k_{\max}$. Note that at the time of refill, if $k > 1$, V still contains the $(k-1)$-th ranked tuple. We can use the value of this tuple, v_{k-1}, to refine the refill query as: “return the top $k_{\max} - k + 1$ tuples among those whose values are less than v_{k-1}.”
- Insert the result of the refill query into V.

Further optimization is possible. For example, instead of waiting until k' drops below k, we could refill the view more “eagerly,” *i.e.*, when k' is close to but still larger than k. This approach would allow us to keep serving top-k queries from the view while waiting for the refill query on the base table to complete. However, our analysis will be based on the basic version of the algorithm.

A. Cost Model

The amortized cost of our algorithm per base table update is given by

$$C = C_{update} \times (1 - f_{ignore}) + C_{refill} \times f_{refill}. \qquad (1)$$

The cost of updating the view in an update operation, C_{update}, is $O(\log |V|)$, or $O(\log k_{\max})$, since we can implement V using any data structure that functions as a priority queue, say, a heap or a balanced search tree, which has an $O(\log |V|)$ lookup/insert/delete time. However, not every base table update causes a view update. Suppose f_{ignore} is the fraction of base table updates that are ignorable. The amortized cost of an update operation is $C_{update} \times (1 - f_{ignore})$.

The cost of a refill operation, C_{refill}, includes the following three components:

- The cost of processing the refill query over R. If $k_{\max} - k + 1$ is small enough, we can evaluate the refill query by making one pass over R while keeping in memory the top tuples (among those whose values are less than v_{k-1}) seen so far. In the worst case where $k_{\max} - k + 1$ is too large for memory, we can perform an external-memory sort of all R tuples with values less than v_{k-1}, and return the top $k_{\max} - k + 1$ tuples. In either case, we expect this cost to be $O(N)$ for practical memory sizes.
- The cost of retrieving the $k_{\max} - k + 1$ result tuples of the refill query. Depending on the actual database and application setup, this cost may involve the cost of binding result tuples out from the database to the application, or the cost of transmitting them to a remote application over the network. We expect this cost to be $O(k_{\max})$.
- The cost of inserting the $k_{\max} - k + 1$ result tuples into V. These tuples are already sorted and will be appended to V. For the data structures used to implement V (as discussed for the case of C_{update}), the total cost of appending $k_{\max} - k + 1$ tuples is $O(k_{\max})$.

It is reasonable to assume that $C_{update} \ll C_{refill}$. Therefore, to minimize C, we focus on reducing f_{refill}, the frequency of refill operations. Since the cost of a refill query is $O(N)$, if we can reduce f_{refill} to $1/N$, we will have reduced the amortized cost of refill queries to $O(1)$, an attractive goal. Intuitively, we can decrease f_{refill} by increasing $k_{\max}$. However, a larger $k_{\max}$ also increases C_{update} and C_{refill} and decreases f_{ignore}, so the trade-off must be considered carefully. In Section IV, we develop a theoretical model to study the effect of $k_{\max}$ on f_{refill}, and in Sections V and VI, we conduct simulations and experiments to see how $k_{\max}$ affects C_{update},

f_{ignore}, C_{refill}, and f_{refill} in practical scenarios.

IV. ANALYSIS

A. The Random Walk Model

From the description of the algorithm in the last section, we notice that the values of k' between two refill operations can be modeled as a random walk on the one-dimensional points $\{k-1, k, \ldots, k_{\max}\}$, where $k_{\max}$ is the starting point and $k-1$ is an absorbing point at which the random walk ends and a refill is needed. In order to use the standard notation of Markov chains, we map the one-dimensional points to $\{0, 1, \ldots, n\}$, making 0 as the starting point and n the absorbing point, where $n = k_{\max} - k + 1$. We need to analyze the probabilistic properties of the *refill interval* Z, or the number of steps it takes for the random walk to go from 0 to n. The expected refill frequency used in (1) is given by $f_{refill} = \mathbf{E}[1/Z]$.

For the purpose of our analysis, we are mostly interested in good and bad updates since they are the only updates that change the size of the view. Suppose that the random walk is currently at position i. With a bad update, the random walk moves to $i+1$; we assume that this move happens with probability p_i. With a good update, the random walk moves to $i-1$; we assume that this move happens with probability q_i. Otherwise, the update is either ignorable or neutral, and the random walk stays at i with probability $1 - p_i - q_i$.

In our random walk model, we assume that the choice at each step is independent of all previous choices. This assumption may not hold for arbitrary update workloads. In Section IV-F, we show how to generalize our analytical results when this assumption is dropped, and in Section V, we show how to apply our generalized results to update workloads where the independence assumption does not hold.

B. Expected Refill Interval

The first step is to find the expectation of Z, $\mathbf{E}[Z]$, which is called the *hitting time* from 0 to n in the terminology of Markov chains. Denote the hitting time from i to n as h_i. By the properties of Markov chains, we have

$$h_i = 1 + \sum_{j=0}^{n-1} p_{ij} h_j,$$

where p_{ij} is the *transition probability* from i to j. The above equation also can be written in a matrix form: $\mathbf{h} = \mathbf{b} + \mathbf{P}\mathbf{h}$, or

$$(\mathbf{I} - \mathbf{P})\mathbf{h} = \mathbf{b}, \tag{2}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix, $\mathbf{P} = (p_{ij})$, $\mathbf{h} = (h_0, \ldots, h_{n-1})^T$, and $\mathbf{b} = (1, \ldots, 1)^T$. Using the probabilities of good and bad updates defined in Section IV-A, we can write $\mathbf{P}$ as follows:

$$\mathbf{P} = \begin{bmatrix} 1-p_0 & p_0 & & & & \\ q_1 & 1-p_1-q_1 & p_1 & & & \\ & q_2 & 1-p_2-q_2 & p_2 & & \\ & & & \ddots & & \\ & & & q_{n-2} & 1-p_{n-2}-q_{n-2} & p_{n-2} \\ & & & & q_{n-1} & 1-p_{n-1}-q_{n-1} \end{bmatrix}, \tag{3}$$

where all unspecified entries are zeros.

To simplify calculation, we consider the special case of $p_0 = p_1 = \cdots = p_{n-1} = p$ and $q_1 = q_2 = \cdots = q_{n-1} = q$, *i.e.*, the probabilities of bad and good updates (p and q respectively) remain constant as $|V|$ changes. However, this assumption is not true in general and will be dropped later. Solving (2), we obtain the following result.

Lemma 1 (Expected refill interval): When $p_0 = \cdots = p_{n-1} = p$ and $q_1 = \cdots = q_{n-1} = q$, the expected hitting time of the random walk from i to n is given by

$$h_i = \begin{cases} \frac{(n+i+1)(n-i)}{2p}, & p = q; \\ \frac{(n-i)(1-t)-t^{i+1}+t^n}{p(1-t)^2}, & p \neq q \text{ and } t = q/p, \end{cases}$$

for $i = 0, \ldots, n-1$.

Please note that $\mathbf{E}[Z] = h_0$. This lemma confirms our intuition that the hitting time increases with bigger views. In particular, when $p = q$, it seems sufficient to choose $n = \Theta(\sqrt{N})$ so that $\mathbf{E}[Z] = N$, which means that we perform a refill operation every N updates on average. In general, however, we cannot guarantee that $\mathbf{E}[1/Z]$, the expected refill frequency, is $1/N$. It is conceivable that the actual distribution of Z is not centered at its mean value; there may be a significant probability for Z to be much smaller than its mean, causing $\mathbf{E}[1/Z]$ to be much bigger than $1/\mathbf{E}[Z]$. Unfortunately, $\mathbf{E}[1/Z]$ has no closed-form formula. On the other hand, given the transition matrix in (2), we can compute $\mathbf{E}[1/Z]$ numerically. However, a numerical solver alone cannot provide any bound on $\mathbf{E}[1/Z]$ in general. Next, we develop a series of high-probability results which enable us to bound $\mathbf{E}[1/Z]$.

Before proceeding to the high-probability bounds, we first introduce the following lemma, which helps us drop the assumption that $p_0 = \cdots = p_{n-1} = p$ and $q_1 = \cdots = q_{n-1} = q$.

Lemma 2: Let W_1 be a random walk with transition probabilities p_j and q_j, $0 \leq j \leq n-1$. Let W_2 be another random walk, whose transition probabilities are identical to those of W_1 except at position i: W_2 moves to $i+1$ with probability p'_i and moves to $i-1$ with probability q'_i, where $p'_i + q'_i \geq p_i + q_i$ and $p'_i/q'_i \geq p_i/q_i$. Then the hitting time of W_1 stochastically dominates that of W_2.

Intuitively, compared with W_1, W_2 is less likely to stay at i and more likely to move to $i+1$ than to $i-1$. Therefore, the hitting time of W_2 should be shorter than that of W_1. A rigorous proof is given in Appendix I. With the help of Lemma 2, we can transform a random walk with different p_i and q_i's into one with $p_0 = \cdots = p_{n-1} = p$ and $q_1 = \cdots = q_{n-1} = q$, by repeatedly applying the lemma. Then we can instead bound the hitting time of the second random walk. The results automatically hold for the original random walk since it can only be "better".

C. High-Probability Results When $p_i = q_i$

We first concentrate on the most interesting case where $p_i = q_i$, *i.e.*, the view shrinks and grows with equal probability. We expect this case to be common: If the distribution of the tuple values used for ranking is stationary, the rate at which tuples enter the top-k' view must be the same as that at which tuples leave the top-k' view. We would like to provide a high-probability guarantee, *i.e.*, $Z = \Omega(N)$ holds with high probability. If so, the expected amortized cost of refill queries, $O(N)/Z$, will be $O(1)$. Our main result is the following theorem. In fact, the condition $p_i = q_i$ can be relaxed to $p_i \le q_i$.

Theorem 1: When $p_i \le q_i$, for $1 \le i \le n-1$, if $n = N^{\frac{1}{2}+\epsilon}$ for any positive constant ϵ, the refill interval Z is greater than N with high probability:

$$\mathbf{Pr}[Z > N] \ge 1 - 4e^{-N^{2\epsilon}/2}.$$

To prove Theorem 1, we need the following two lemmas.

Lemma 3 (Hoeffding, 1963 [14]): If $X_1, X_2, \ldots$ are independent and bounded as $a_i \le X_i \le b_i$, then for any $t > 0$, the partial sums $S_n = \sum_{i=1}^{n} X_i$ have the following probability inequality:

$$\mathbf{Pr}[S_n - n\mu \ge nt] \le \exp\left(-\frac{2n^2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right),$$

where $\mu = \mathbf{E}[X_i]$.

Lemma 4 (Petrov, 1975 [19]): If random variables $X_1, X_2, \ldots$ are symmetrically distributed and independent, then

$$\mathbf{Pr}\left[\max_{1\leq k\leq n} |S_k| \geq x\right] \leq 2\mathbf{Pr}[|S_n| \geq x].$$

Now we are ready to proceed with the proof of Theorem 1.

Proof: We first create another random walk W' with $p'_0 = 2p$, and $p'_i = q'_i = p$ for $1 \leq i \leq n-1$, where $p = \max_{0\leq i\leq n-1}(p_i + q_i)/2$. By Lemma 2, it is easy to see that the hitting time of the original random walk stochastically dominates that of W'. We further extend W' to a random walk W'' on $\{\ldots, -n, \ldots, -1, 0, 1, \ldots, n, \ldots\}$, where 0 is the starting point, and all transition probabilities are p. It is easy to see that W'' simply "mirrors" W', so the hitting times of W' and W'' should be identically distributed. We will bound the probability that Z'', the hitting time from 0 to either $-n$ or n of W'', is greater than N.

Let $X_1, X_2, \ldots$ be the steps taken by W'', which can be -1, 0 or 1. These random variables are independent. Then

$$\begin{aligned}
\mathbf{Pr}[Z'' \leq N] &= \mathbf{Pr}[\max_{1\leq k\leq N} |S_k| \geq n] \;\leq\; 2\mathbf{Pr}[|S_N| \geq n] \qquad \text{(Lemma 4)}\\
&= 4\mathbf{Pr}[S_N \geq n] \;\leq\; 4\exp(-\frac{2n^2}{N\cdot 2^2}) \qquad \text{(Lemma 3)}\\
&= 4e^{-N^{2\epsilon}/2}.
\end{aligned}$$

Since $\mathbf{Pr}[Z'' \leq N] \geq \mathbf{Pr}[Z \leq N]$, the theorem follows. ■

With this theorem, the following corollary comes naturally.

Corollary 1: When $p_i = q_i$, the expected amortized cost of refill queries, $O(N) \times \mathbf{E}[1/Z]$, is $O(1)$, if $n = N^{\frac{1}{2}+\epsilon}$, for any positive constant ϵ.

TABLE I
THEORETICAL BOUNDS ON $\mathbf{Pr}[Z > N]$ AND $\mathbf{E}[N/Z]$ FOR PRACTICAL VALUES OF N AND n.

N	n	n/N	lower bound on $\mathbf{Pr}[Z > N]$	upper bound on $\mathbf{E}[N/Z]$
100	30	30%	0.9556	1.1037
1000	100	10%	0.9730	1.2426
10^4	400	4%	0.9987	1.0322
10^5	1300	1.3%	0.9991	1.0650
10^6	4500	0.45%	0.9998	1.0355

Proof: Because $N/Z \leq N/n = N^{\frac{1}{2}-\epsilon}$, we have $\mathbf{E}[N/Z] \leq 1 \cdot \left(1 - 4e^{-N^{2\epsilon}/2}\right) + N^{\frac{1}{2}-\epsilon} \cdot 4e^{-N^{2\epsilon}/2} = O(1) + o(1) = O(1)$. ■

Although the requirement of $n = N^{\frac{1}{2}+\epsilon}$ is not as good as our first impression that $n = \Theta(\sqrt{N})$, it is still good enough to generate satisfying performance of our algorithm in practice. Table I lists some practical values of N and n. For each pair of N and n, we show the lower bound on $\mathbf{Pr}[Z > N]$ according to Theorem 1 and the upper bound on $\mathbf{E}[N/Z]$ according to Corollary 1. We see that our algorithm performs exponentially better as N goes up. For example, for a base table with a million tuples, a view containing only the top 0.45% of all tuples is enough to provide a refill interval longer than one million updates with probability 99.98%. Please note that the values of $\mathbf{Pr}[Z > N]$ and $\mathbf{E}[N/Z]$ shown in Table I are theoretical bounds; actual performance should be even better.

D. High-Probability Result When $p_i < q_i$

When $p_i < q_i$, a base table update is more likely to grow the view than to shrink it. Intuitively, we should expect a long refill interval even for small views. Indeed, according to Lemma 1, h_0 is large because of the exponential term t^n in the numerator, where $t > 1$. As the following theorem shows, we can use a logarithmic-size view to reduce the amortized cost of refill queries to $O(1)$ with high probability. To avoid the situation where the gap between p_i and q_i depends on i and may get arbitrarily small, we require that q_i is larger than p_i by at least a constant factor.

Theorem 2: When $\delta \cdot p_i < q_i$, for $1 \leq i \leq n-1$ and some constant $\delta > 1$, if $n = c \ln N$, the refill interval Z is greater than N with high probability, *i.e.*, $\mathbf{Pr}[Z > N] > 1 - o(1)$, for constant c big enough, depending only on δ.

Proof: We will only consider the case $p_0 = \cdots = p_{n-1} = p$ and $q_1 = \cdots = q_{n-1} = q = \delta p$ where $p = 1/(1+\delta)$. By Lemma 2, the hitting time of this random walk is stochastically dominated by any other one with $\delta \cdot p_i < q_i$. Note that this normalized random walk never stays at the same place for two consecutive steps except at position 0.

We now bound the probability of $Z \leq N$. For any instance of this random walk, the last phase of the walk must be one that moves from 0 to n without touching 0. Let X be the length of this phase. Clearly, $X \geq n$. We have $\mathbf{Pr}[Z \leq N] \leq \mathbf{Pr}[X \leq N] = \sum_{i=n}^{N} \mathbf{Pr}[X = i]$.

If the last phase of the random walk takes i steps to move from 0 to n with no stays, there must be $(i+n)/2$ steps moving right ("$+1$") and $(i-n)/2$ steps moving left ("-1"). (If $(i+n)/2$ is not an integer, $\mathbf{Pr}[X = i] = 0$.) This condition is necessary for $X = i$. By Chernoff's bound, we have

$$
\begin{aligned}
\mathbf{Pr}[X = i] \;&\leq\; \mathbf{Pr}[\text{number of ``}-1\text{'' steps} = \frac{i-n}{2}] \;\leq\; \mathbf{Pr}[\text{number of ``}-1\text{'' steps} \leq \frac{i-n}{2}] \\
&\leq\; \exp\left(-\frac{qi}{2}\left(\frac{qi - \frac{i-n}{2}}{qi}\right)^2\right) \;=\; \exp\left(-\frac{qi}{2}\left(1 - \frac{1}{2q} + \frac{n}{2qi}\right)^2\right) \\
&<\; \exp\left(-\frac{q}{2}\left(1 - \frac{1}{2q}\right)^2 i\right).
\end{aligned}
$$

Let $c_1 = \frac{q}{2}(1 - \frac{1}{2q})^2$. Since $q > \frac{1}{2}$, c_1 is a positive constant. Thus $\mathbf{Pr}[X = i] < e^{-c_1 n} = e^{-c_1 c \ln N} = N^{-c_1 c}$, and $\sum_{i=n}^{N} \mathbf{Pr}[X = i] < N \cdot N^{-c_1 c} = N^{1-c_1 c}$. Thus, as long as we choose c such that $c > \frac{1}{c_1} = \frac{8\delta(\delta+1)}{(\delta-1)^2}$, the refill interval is greater than N with probability $1 - o(1)$. ■

The next corollary follows naturally.

Corollary 2: When $\delta \cdot p_i < q_i$, for $1 \leq i \leq n-1$ and some constant $\delta > 1$, the expected amortized cost of refill queries is $O(1)$, if $n = c \ln N$, for some constant c big enough.

Proof: Following Theorem 2, we have $\mathbf{E}[N/Z] < 1 \cdot (1 - N^{1-c_1 c}) + N \cdot N^{1-c_1 c}$. Thus, when $c > \frac{2}{c_1}$, we have $\mathbf{E}[N/Z] = O(1)$. ∎

E. When $p_i > q_i$

When $p_i > q_i$, a base tuple update is more likely to shrink the view than to grow it. According to Lemma 1, $\mathbf{E}[Z]$ is on the order of n, meaning that we would need $n = N$ to bring the expected refill interval up to the order of N. Here, increasing the size of the view still decreases the expected refill frequency, but at a much slower rate than the case $p_i \leq q_i$.

Nevertheless, we feel that the case of $p_i > q_i$ is unusual in practice, because when $p_i > q_i$, tuples are trying to "escape" from the top-k list. Typically, people are more interested in scenarios where tuples are "competing" with each other to enter the top-k list. In such scenarios, we would have $p_i \leq q_i$, where our algorithm is most effective.

F. Dropping the Memorylessness Assumption

Throughout our analysis, we have assumed a memoryless random walk model in which the choice at each step is independent of all previous choices. Dropping this assumption requires replacing the conditions on p_i and q_i in Theorems 1 and 2 and Corollary 1 and 2 with more general ones.

Definition 1: Consider a random walk W with memory on $\{0, 1, \ldots, n\}$. We say that W is origin-tending *if, regardless of the previous steps taken, the probability of W moving from i to $i-1$ is always no less than that of moving from i to $i+1$, where i is the current position of*

W and $0 < i < n$. We say that W is strictly origin-tending *if, regardless of the previous steps taken, the probability of W moving from i to $i-1$ is always no less than δ times that of moving from i to $i+1$, where $\delta > 1$ is a constant.*

Theorem 1 and 2 (and Corollary 1 and 2) can be shown to be applicable under the new conditions "origin-tending" and "strictly origin-tending."

Theorem 3: If the random walk is origin-tending (resp. strictly origin-tending), the refill interval Z is greater than N with probability $1 - o(1)$, by choosing $n = N^{\frac{1}{2}+\epsilon}$ for any positive constant ϵ (resp. $n = c \ln N$ for some constant c big enough).

We provide the proof of this generalized theorem in Appendix II. In Section V, we will see some examples that require the application of these generalized theorems and corollaries.

V. Case Studies of Update Workloads

In Section IV, we have concluded that our algorithm is most effective when the random walk is origin-tending. In this section, we study several statistical models of update workloads. For most of the workloads we consider, the random walk is origin-tending. We also perform simulations to measure the transition probabilities as well as the fraction of ignorable updates, which will be used in Section VI-D in evaluating the effectiveness of our algorithm.

Case 1: Cumulative Total Sales

Suppose we are interested in the top k all-time best-selling books in a bookstore. The vast majority of the transactions are purchases that increase the cumulative total sales figures of the books purchased. Transactions that decrease the sales figures, *e.g.*, returns or cancelled orders, are very rare. Under this workload, the probability of a bad update is almost nil since it is extremely unlikely for a book to drop out of the top-k list because of a return or a cancelled order. Interestingly, for this seemingly simple workload, the random walk that models how k' changes is not memoryless, because the probability of a book entering or leaving the top-k' list depends not only on the current transaction, but also on the history of earlier transactions. Nevertheless, the random walk is obviously strictly origin-tending, so the algorithm should be effective even for small values of $k_{\max}$. In fact, in the special case where the total sales figures never decrease, $k_{\max} = k$ would suffice because the top-k view would be self-maintainable.

Case 2: Random Up-and-Downs

Next, we consider a case where the values in the base table increase and decrease equally likely. Suppose each item starts with some initial value drawn from a symmetric unimodal distribution (*e.g.*, normal distribution) with mean μ. In each time step t, an item is chosen uniformly at random to be modified by X_t, where X_t follows some symmetric unimodal distribution with mean 0. We assume the choice of X_t is independent of the choices made in earlier time steps. Let S_t be the value of the chosen item at the end of time step t and S_{t-1} be the value of this item at the end of the previous time step; $S_t = S_{t-1} + X_t$. It is easy to see that S_{t-1} and S_t also have symmetric unimodal distributions with mean μ. This model can be used reasonably to describe many up-and-down processes, *e.g.*, stock prices, fortunes of gamblers, *etc*.

Like in Case 1, the random walk that models how k' changes is not memoryless, because the probabilities for an item to enter and leave the top-k' list in time step t depend on the actual values of the items before t, which in turn depend on the history of previous updates. Fortunately, as discussed in Section IV-F, we still know that our algorithm is effective as long as we can show that the random walk is origin-tending.

Suppose $v_{k'}$ is the value of the lowest ranked item in the top-k' view at the beginning of time step t. Let q_t denote the probability of a good update at time t, and p_t denote the probability of a bad update at time t. Suppose the probability density functions of S_{t-1} and X_t are $f_S(s)$ and $f_X(x)$, respectively. We have

$$
\begin{aligned}
q_t &= \mathbf{Pr}[S_{t-1} < v_{k'},\, S_{t-1} + X_t \geq v_{k'}] = \int_0^{\infty} \int_{v_{k'}-x}^{v_{k'}} f_S(s)\, f_X(x)\, ds\, dx, \\
p_t &= \mathbf{Pr}[S_{t-1} \geq v_{k'}, S_{t-1} + X_t < v_{k'}] = \int_{-\infty}^{0} \int_{v_{k'}}^{v_{k'}-x} f_S(s)\, f_X(x)\, ds\, dx \\
&= \int_0^{\infty} \int_{v_{k'}}^{v_{k'}+x} f_S(s)\, f_X(x)\, ds\, dx \qquad \text{(because } X_t \text{ is symmetrically distributed around } 0\text{).}
\end{aligned}
$$

Because the distribution of S_{t-1} is symmetric about μ, $v_{k'} > \mu$ when $k_{\max}$ is small. Furthermore, the distribution must be decreasing after μ, because it is also unimodal. Therefore, for any $x > 0$,

$$
\int_{v_{k'}-x}^{v_{k'}} f_S(s)\, ds > \int_{v_{k'}}^{v_{k'}+x} f_S(s)\, ds,
$$

which leads to the conclusion that $p_t < q_t$. Therefore, the random walk is origin-tending and we may choose $k_{\max} = N^{\frac{1}{2}+\epsilon}$.

Case 3: Total Sales in a Moving Window

Finally, we consider a more complicated model involving a moving time window. Suppose we are interested in ranking books by their total sales during the last w time steps. For each book b, let X_t^b be the number of copies of b sold during time step t. At the end of the time step t, we update the total sales to be $(X_{t-w+1}^b + \cdots + X_{t-1}^b + X_t^b)$. Suppose that for each b, all X_i^b's are independently and identically distributed (*i.i.d.*). Let $v_{k'}$ be the total sales of the lowest ranked book in the top-k' view at the beginning of time step t. We have

$$
\begin{aligned}
q_t^b &= \mathbf{Pr}[X_{t-w}^b + \cdots + X_{t-2}^b + X_{t-1}^b < v_{k'},\; X_{t-w+1}^b + \cdots + X_{t-1}^b + X_t^b \geq v_{k'}], \\
p_t^b &= \mathbf{Pr}[X_{t-w}^b + \cdots + X_{t-2}^b + X_{t-1}^b \geq v_{k'},\; X_{t-w+1}^b + \cdots + X_{t-1}^b + X_t^b < v_{k'}],
\end{aligned}
$$

where q_t^b (p_t^b) denotes the probability that the update on b at the end of time step t is good (bad). Let $S = \sum_{i=t-w+1}^{t-1} X_i^b$. Since X_i^b's are *i.i.d*, we have

$$\begin{aligned} q_t^b &= \mathbf{Pr}[X_{t-w}^b + S < v_{k'},\ S + X_t^b \geq v_{k'}] &= \mathbf{Pr}[X_t^b + S < v_{k'},\ S + X_{t-w}^b \geq v_{k'}] \\ &= \mathbf{Pr}[X_{t-w}^b + S \geq v_{k'},\ S + X_t^b < v_{k'}] &= p_t^b. \end{aligned}$$

Therefore, the random walk is origin-tending, and we may choose $k_{\max} = N^{\frac{1}{2}+\epsilon}$.

In addition to the theoretical analysis above, we conduct two simulations of this update workload in order to measure the transition probabilities of the random walk model and the fraction of ignorable updates. The actual values of these parameters will be used in Section VI-D to evaluate the effectiveness of our algorithm.

In the first simulation, the number of copies sold for each book in each time step follows the same Poisson distribution with mean 50. In the second simulation, for each book, we use a Poisson distribution with a different mean; furthermore, these mean values form a Zipf distribution. In both simulations, we use a base table of 1000 books and vary the size of the view from 1 to 1000.

Results from the two simulations are shown in Figures 1 and 2 respectively. Both figures plot the probabilities of good, bad, and ignorable updates observed under different view sizes. We make the following observations from the simulation results: (1) the probabilities of good and bad updates are equal and typically small; (2) they increase with the size of the view initially, but once the view becomes large enough, they begin to decrease; (3) the probability of ignorable updates decreases from about 1 to 0 as the size of the view increases from 1 to the size of the base table. The first observation confirms the fact that the random walk is origin-tending. The last observation implies that increasing the view size has the negative effect of increasing the

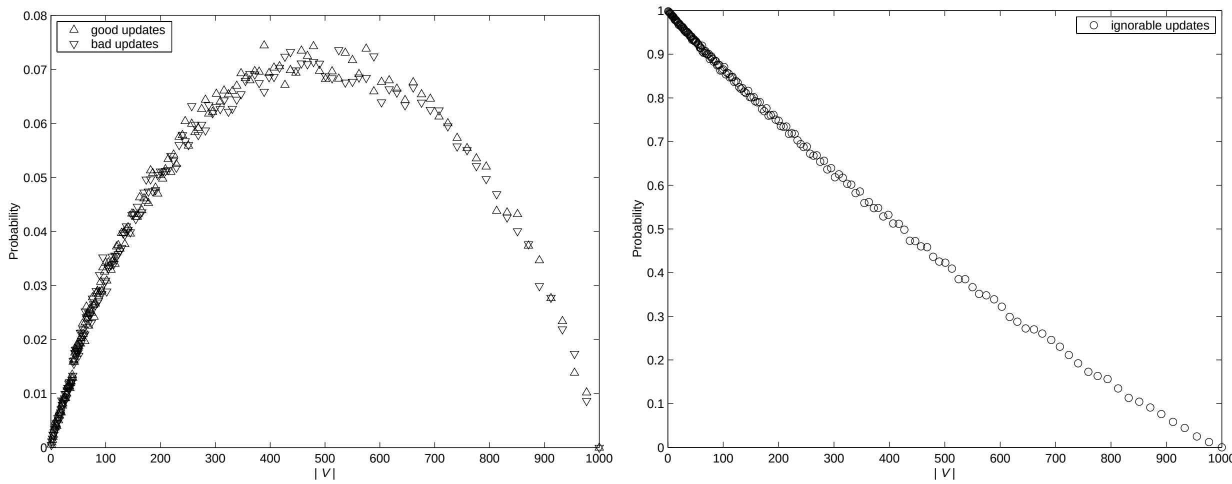

Fig. 1. Results from the first simulation.

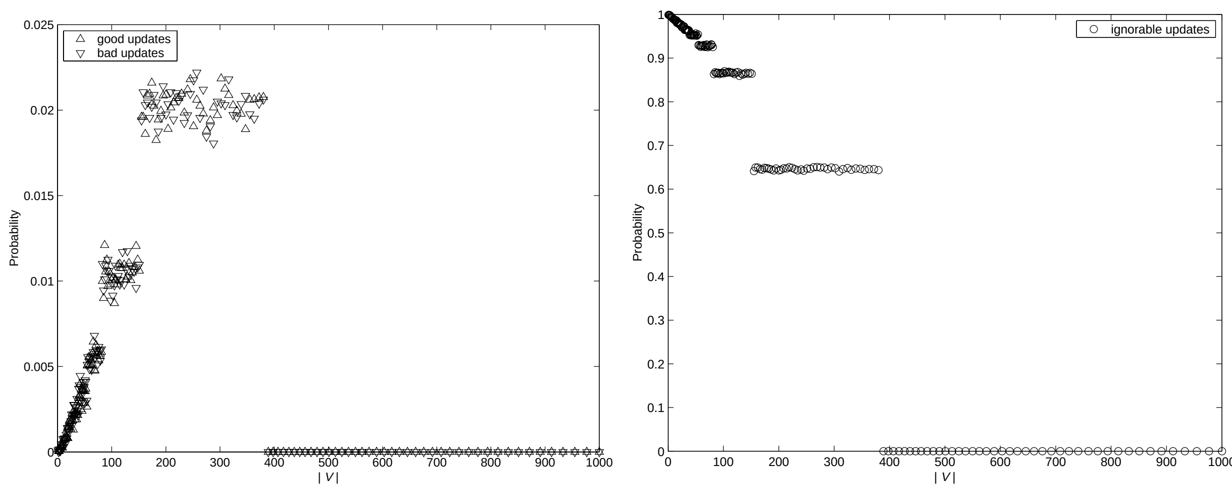

Fig. 2. Results from the second simulation.

number of base table updates to be processed.

Summary

For the cases considered in this section, the random walk is always origin-tending, although it may or may not be memoryless. Admittedly, the real world situations are far more complicated to model accurately. However, from these simple case studies, we have reasonable confidence that an origin-tending random walk approximates many practical update workloads well, for which our algorithm provides good performance.

VI. EXPERIMENTS

We have conducted three sets of experiments in order to validate our discussion on the cost model in Section III and to obtain realistic values of the model parameters. The first set of experiments measures the performance of refill queries in a commercial database system; the second set of experiments measures the performance of updating top-k views managed by a commercial database system; the last set of experiments measures the performance of updating top-k views managed directly by an application. At the end of this section, we evaluate the effectiveness of our algorithm using realistic values of the model parameters.

A. Refill Queries

We conduct our experiments on a Windows 2000 server with a 1.4GHz Pentium 4 processor and 1GB of RAM, running the latest version of a commercial database system from a major vendor. We set the size of database buffer pool at 500MB, and the size of the sort heap at 200MB.

We create a base table R with integer `id` and `val` columns, together with other columns of mixed data types, for a total size of roughly 160 bytes per tuple. To populate R, we generate `id` values sequentially in increment of 1, and `val` values randomly from the interval $[1, 2^{30}]$. Our experiments do not cover situations where `val` is computed on the fly; we expect the costs of refill queries to be higher in such cases. There is a primary B^+-tree index on R.`id` and no index on R.`val`.

The refill query is evaluated over R and returns the `id` and `val` values for tuples ranked between k and $k_{\max}$. Suppose that the $(k-1)$-th (lowest) ranked tuple in the view at the time of refill has value v_{k-1}. The refill query is shown below in extended SQL syntax:

```
SELECT id, val FROM R WHERE val < v_{k-1} ORDER BY val DESC
FETCH FIRST k_max - k + 1 ROWS ONLY OPTIMIZED FOR k_max - k + 1 ROWS;
```

For simplicity, the above query does not consider ties, although we do handle them in our experiments using a slightly more complicated `WHERE` condition.

We vary the following parameters in our experiments: (1) N, or $|R|$, the size of the base table, from 10^5 to 3×10^6; (2) k, which indirectly determines v_{k-1}, from 10 to 10^3; and (3) $k_{\max}$, from 10 to 10^4. The choice of parameter values are constrained by $k \leq k_{\max} \leq N$. We measure the total elapsed time of running the refill query, including the time to write the $k_{\max} - k + 1$ output rows to a log file.

A total of 600 result data points are shown in Figure 3. For each value of $|R|$, we plot all running times collected for different k and $k_{\max}$ values, as well as the average, minimum, and maximum running times. We find that the cost of refill query is roughly linear in the size of the base table, confirming the $O(N)$ bound in Section III-A. For this particular experimental setup, this cost is approximately $29.5 \times |R|$ μsec.

The cost of processing the refill query may depend on $k_{\max} - k + 1$, the size of the output. Indeed, from the output of the database optimizer, we find that the optimized execution plan includes a special sort operator that produces only the top $k_{\max} - k + 1$ tuples. However, from Figure 3, we see that the effects of k and $k_{\max}$ are negligible compared with that of N.

We also have considered the case of a secondary B$^+$-tree index on R.`val`, which is applicable so long as `val` is not computed on the fly. The downside of this index is the additional overhead in processing base table updates. However, since this index effectively orders all R tuples, we would expect the refill queries to run significantly faster, at least for small values of $k_{\max} - k + 1$ (large values may result in excessive random disk I/O's if `id` values are not stored directly in

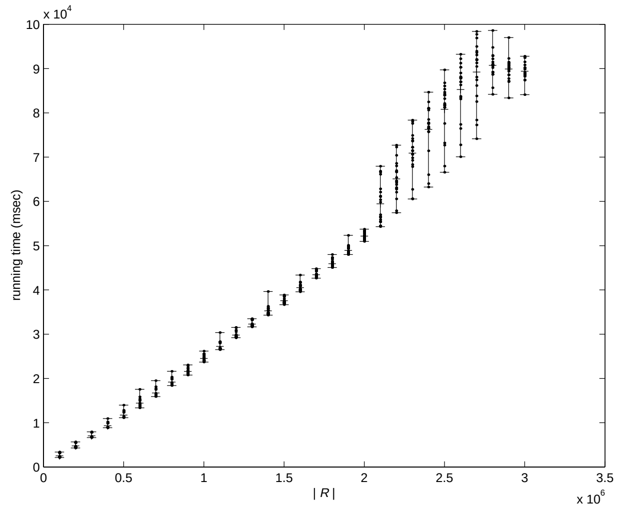

Fig. 3. Costs of refill queries.

the index). Unfortunately, we are unsuccessful at "hinting" our database optimizer to pick the index plan (even when $k = k_{\max}$). On the other hand, a secondary B$^+$-tree index on R.val is in essence a self-maintainable top-k view with $k = N$ managed by the same database. Hence, the update performance results for a top-k view with $k = N$ (Section VI-B) still provide us with some information to evaluate the trade-offs of using a secondary index; we shall come back to this discussion in Section VI-D.

B. Database View Updates

For the following set of experiments, we assume that the materialized view V is managed by a commercial database system (possibly remote and not necessarily the same as the one managing the base table). We use the same experimental setup as in Section VI-A. We vary $|V|$, the size of the view, from 2 to 10^6. There are a primary B$^+$-tree index on V.id and a secondary B$^+$-tree index on V.val. The second index does increase the update cost, but we feel that it is more realistic to have this index for allowing fast accesses to the sorted top-k list.

For each V, we generate 40 random update streams. Each update stream includes a mix of 1000 deletions and 1000 insertions. Each deletion removes a random tuple from V by `id`; each insertion adds a tuple to V with randomly generated `id` and `val` values. Deletions and insertions alternate in the update stream, keeping $|V|$ constant during an experiment. For each update stream, we measure the average running time of a pair of deletion and insertion and take it to be the view update cost. The results are shown in Figure 4. For each value of $|V|$, we plot all view update costs measured from 40 random update streams, as well as the average, minimum, and maximum costs.

Because of the B^+-tree indexes on V, we expect the update cost to be logarithmic in $|V|$. Interestingly, the update cost turns out to be a step function according to Figure 4. Several factors may have contributed to this phenomenon, including poor locality in the randomly generated update streams and a large branching factor of database B^+-trees. Because of poor locality, lower-level pages of the B^+-tree tend not to stay in the database buffer pool; thus, the update cost roughly corresponds to the number of levels in the B^+-tree. Because of the large branching factor, the number of levels in the B^+-tree increases extremely slowly with $|V|$ and stays constant over wide ranges of $|V|$. Given that the range of $|V|$ is small in practice, we observe only two "steps" in Figure 4.

C. Application View Updates

For the following set of experiments, we assume that V is maintained in memory by an application program that specializes in serving requests for top-k tuples. We believe this scenario is common in practice because: (1) V is typically small enough to fit in application memory; (2) the operations on V are simple and frequent, so the application can implement them without the overhead of using a database system; and (3) we are not worried about losing the data in V in case of failures, since V always can be recomputed from R.

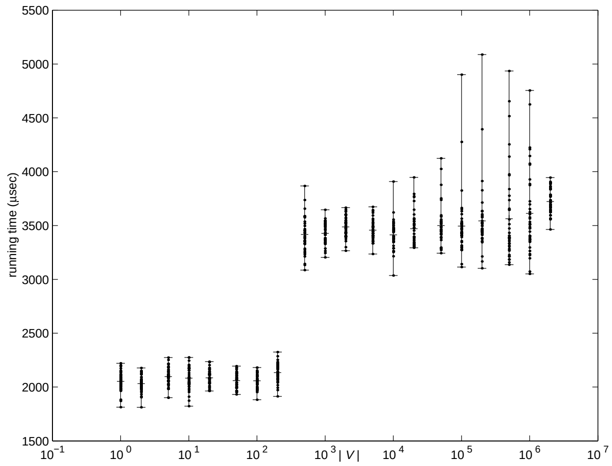

Fig. 4. Costs of database view updates.

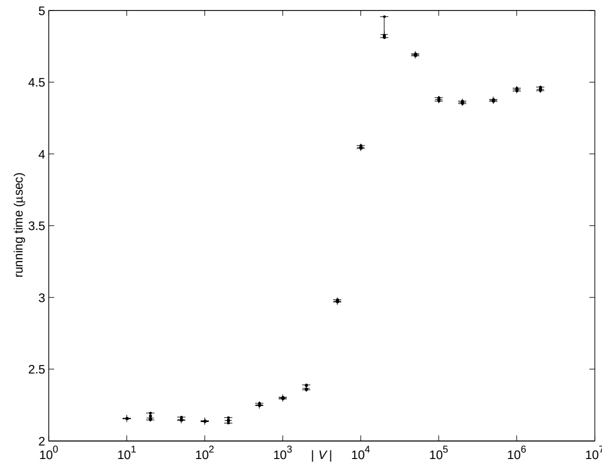

Fig. 5. Costs of application view updates.

We conduct our experiments on a Sun Blade 100 workstation with a 500MHz UltraSPARC-IIe processor, 256KB of level-2 cache, and 512MB of RAM. The application is written in C and compiled with `gcc` using `-O3` option. We implement V using two memory-resident data structures. An implicit binary heap (implemented as an array) stores the (`id`, `val`) pairs in V, with `val` being the search key. A hash table supports efficient lookup of a binary heap node by `id`. Both the binary heap and the hash table have size on the order of $|V|$. Alternatives to the binary heap would be balanced search trees such as the red-black tree, but they may be less efficient than the binary heap because there is no need to maintain a complete ordering of all $|V|$ tuples.

For each V, we generate 10 random update streams, each consisting of 10^7 deletions and 10^7 insertions mixing together. In Figure 5, we plot, for each value of $|V|$, all update costs measured from 10 random update streams, as well as the average, minimum, and maximum costs. Again, the update cost turns out to be step function. We attribute this phenomenon to the uniform random distribution of generated updates and the effect of caching. Because updates are generated uniformly, a large portion of them access relatively few heap nodes, bringing down the expected cost of a heap update to a constant [8], [20]. When $|V|$ is small enough, most accesses result in cache hits. Once $|V|$ grows beyond a certain point, most accesses result in cache misses, because of the lack of locality in randomly generated update streams.

D. Effectiveness of the Algorithm

In this subsection, we evaluate the effectiveness of our algorithm in several scenarios, using the values of model parameters measured in previous subsections. We assume that $|R| = 10^6$ and $k = 100$, *i.e.*, we are interested in maintaining a top-100 view from a base table of one million tuples. In Figure 6, we show the expected amortized maintenance cost as a function of $k_{\max}$, the size of the view that our algorithm starts with. Four curves are shown for the four scenarios we consider: (1) the view is maintained by the same database as the base table; (2) the view is maintained by a remote database; (3) the view is maintained by a local application on the database server with the base table; (4) the view is maintained by a remote application. For scenarios (2) and (4), we assume that the network bandwidth is 500K bits/sec and the latency is masked. Costs of refill queries and view updates are taken from Figures 3, 4, and 5. The update workload is the one used by the first simulation of Case 3 in Section V; probabilities of good, bad, and ignorable updates are extrapolated from Figure 1.

From Figure 6, we see that all four curves exhibit similar trends. When $k_{\max} = k$, the expected maintenance cost is very high. Intuitively, since we are simply maintaining the original top-100 view, every bad update results in an expensive refill query. Initially, as $k_{\max}$ increases, the expected maintenance cost drops rapidly, because a bigger $k_{\max}$ dramatically reduces the expected refill frequency. However, once the refill frequency becomes low enough, the cost of updating the view begins to dominate; increasing $k_{\max}$ at this point not only drives up the cost of an update operation, but also requires more updates to be propagated and applied because fewer

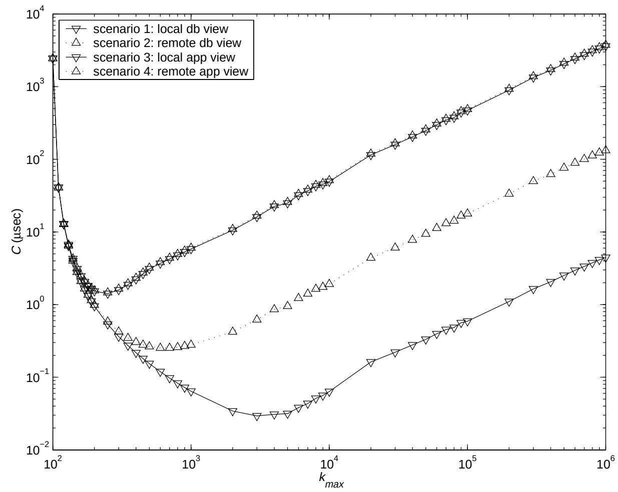

Fig. 6. Expected maintenance cost.

updates are ignorable. In the extreme case where $k_{\max} = |R|$, the view becomes a copy of the base table with `id` and `val` columns. In this case, the refill frequency is 0 because the copy is self-maintainable, but the overhead of maintaining the copy and the high memory requirement make this approach unattractive. Since a secondary index on R.`val` is essentially a view with all $|R|$ tuples, Figure 6 also shows that it might not be a good idea to create this index for the sole purpose of computing top-k queries or maintaining top-k views where k is small. In summary, Figure 6 clearly illustrates the importance of choosing appropriate $k_{\max}$. Proper choice of $k_{\max}$ (in this case, on the order of $\sqrt{N}$) can bring orders of magnitude of performance improvement over the simple approaches of choosing $k_{\max}$ to be k or N.

Comparing the four curves in Figure 6, we see that managing the top-k' view in the application is faster than managing it in a database. Also, managing the view locally is faster than managing it remotely across the network (although the difference is minuscule on the logarithmic scale for a database view). In general, other conditions being equal, we should choose a bigger $k_{\max}$ if the costs of transmitting and applying updates are lower.

VII. Choosing $k_{\max}$ Adaptively

So far, much of our analysis requires knowing the relationship between the probabilities of good and bad updates. In many practical situations, however, the update pattern is not known in advance and may change at runtime; exact values of the transition probabilities for different view sizes are difficult to measure. In this section, we propose an adaptive algorithm that does not require any prior knowledge of the transition probabilities; instead, $k_{\max}$ is chosen at runtime and adjusted dynamically for changing workloads.

The basic idea behind this algorithm is to try to control the refill interval Z around some target value of $Z_0 = C^*_{\mathit{refill}}/C^*_{\mathit{update}}$, where C^*_{refill} is the observed cost of a refill query, and C^*_{update} is the observed cost of processing a base table update. Intuitively, with an expected refill frequency of $1/Z_0$, neither the refill operation nor the update operation is a bottleneck. Typically, Z_0 is on the order of $\Theta(N)$, which means that the amortized cost of refill queries is down to $O(1)$. The algorithm maintains statistics of the observed costs C^*_{refill} and C^*_{update}, and counts the number of base table updates since the last refill operation. If this number is less than Z_0/α, $k_{\max}$ is increased; if it is greater than αZ_0, $k_{\max}$ is decreased. Here, α is a constant parameter used to fine-tune the algorithm; we have chosen $\alpha = 2$, which works well in practice. The detailed algorithm is shown in Figure 7.

We have conducted some simulations for $N = 10000$ and $k = 10$, using the cost parameters measured in Section VI but assume that $p_0 = \cdots p_{n-1} = p$ and $q_1 = \cdots = q_{n-1} = q$. In order to keep the running time of our simulations manageable, we use relatively high values for p and q, the probabilities of bad and good updates. Figure 8 shows how the adaptive algorithm chooses $k_{\max}$ over time for two simulations. For the first simulation, the random walk is origin-tending ($p = q = 0.4$); for the second simulation, the random walk is strictly origin-tending ($p = 0.3$ and

- Tunable parameters:
  - $\alpha = 2$ specifies the acceptable distance from the "optimal" hitting time.
  - $\beta = 0.5$ limits how much $k_{\max}$ can increase at a time.
  - $\gamma = 0.5$ limits how much $k_{\max}$ can decrease at a time.
- At initialization time:
  - $k_{\max} \leftarrow N^{0.6}$, an initial guess based on Theorem 1.
  - $k_{\min} \leftarrow k_{\max}$; $k_{\min}$ tracks the smallest $|V|$ value since the last refill operation.
  - Initialize V with the top-$k_{\max}$ tuples; use the running time as an initial estimate of C^*_{refill}.
  - $T \leftarrow 0$; T records the number of base table updates since the last refill operation.
- At runtime, for each base table update:
  - Process the update; use the running time to update C^*_{update}.
  - $T \leftarrow T + 1$; $k_{\min} \leftarrow \min\{k_{\min}, |V|\}$.
  - $Z_0 \leftarrow C^*_{\mathit{refill}}/C^*_{\mathit{update}}$.
  - If refill is needed for this update, then:
    - If $T < Z_0/\alpha$, increase $k_{\max}$: $k_{\max} \leftarrow \min\{\frac{Z_0/\alpha}{T} \times k_{\max}, (1+\beta) \times k_{\max}\}$.
    - Refill V to $k_{\max}$ tuples; use the running time to update C^*_{refill}.
    - $T \leftarrow 0$; $k_{\min} \leftarrow k_{\max}$.
  - If $T > \alpha Z_0$, then:
    - Decrease $k_{\max}$: $k_{\max} \leftarrow k_{\max} - \gamma(k_{\min} - k)$.
    - Reduce V to $k_{\max}$ tuples, *i.e.*, delete tuples ranked $(k_{\max}+1)$-th or lower.
    - $k_{\min} \leftarrow k_{\min} - \gamma(k_{\min} - k)$.
    - $T \leftarrow (1-\gamma) \times \alpha Z_0$.

Fig. 7. An adaptive algorithm for choosing $k_{\max}$.

$q = 0.4$). The adaptive algorithm starts with the same $k_{\max}$ for both simulations, but over time, $k_{\max}$ takes on different values that are appropriate for respective workloads. From Figure 8, we see that $k_{\max}$ quickly converges to a fairly small range of values for each simulation. However, there are still small fluctuations in $k_{\max}$ over time. We attribute this phenomenon to the variance in hitting time. Occasionally, a very short (or long) run may cause $k_{\max}$ to go down (or up). Thus, in addition to α, our algorithm provides two tunable parameters β and γ, which guard against large increases and decreases in $k_{\max}$, respectively. In effect, $k_{\max}$ stays within a small range in which the expected performance of view maintenance is equally good, so small fluctuations in $k_{\max}$ do not matter in practice.

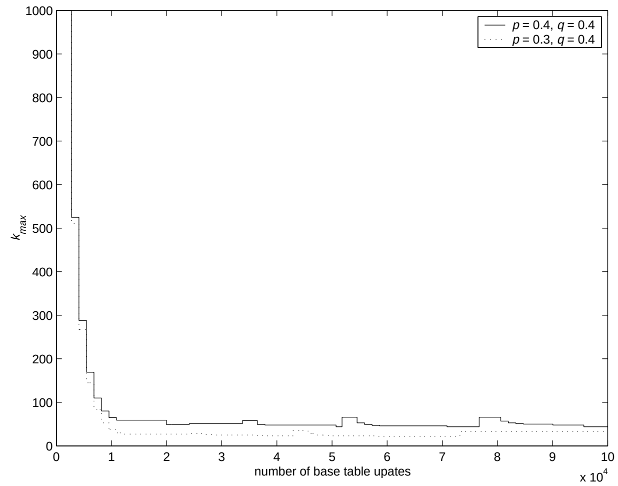

Fig. 8. Behavior of the adaptive algorithm.

VIII. Conclusion

In this paper, we propose a probabilistic approach to tackle the problem of maintaining materialized top-k views. Rather than trying to achieve complete self-maintenance, we try to achieve runtime self-maintenance with high probability by maintaining a dynamic top-k' view where $k' \geq k$. For cases where the random walk is origin-tending or strictly origin-tending, we show that even a little extra investment in k' can dramatically reduce the amortized maintenance cost per update with high probability.

Acknowledgment: We would like to thank Jeff Vitter, John Reif, and Zhihui Wang for their careful readings of our earlier drafts and helpful discussions.

References

[1] M. O. Akinde, O. G. Jensen, and M. H. Böhlen. Minimizing detail data in data warehouses. In *Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 293–307, 1998.

[2] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, September 1989.

[3] N. Bruno, L. Gravano, and S. Chaudhuri. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. on Database Systems*, 2002.

[4] M. J. Carey and D. Kossmann. On saying “enough already!” in SQL. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 219–230, Tucson, Arizona, 1997.

[5] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 158–169, New York City, New York, August 1998.

[6] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[7] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, 2002.

[8] E.-E. Doberkat. Inserting a new element into a heap. *BIT*, 21(3):255–269, 1981.

[9] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, pages 411–422, Edinburgh, Scotland, September 1999.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 2001 ACM Symp. on Principles of Database Systems*, 2001.

[11] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the 1995 Intl. Conf. on Very Large Data Bases*, pages 358–369, September 1995.

[12] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of the 1996 Intl. Conf. on Extending Database Technology*, pages 140–144, March 1996.

[13] Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, June 1999.

[14] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.

[15] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 2001.

[16] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 26–35, Athens, Greece, 1997.

[17] M. K. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. *Data Knowledge Engineering*, 32(1):87–109, 2000.

[18] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, August 2002.

[19] V. V. Petrov. *Sums of Independent Random Variables*. Springer-Verlag, 1975.

[20] T. Porter and I. Simon. Random insertion into a priority queue structure. *IEEE Trans. on Software Engineering*, SE-1,

3:292–298, 1975.

[21] D. Quass. Maintenance expressions for views with aggregation. In *Proc. of the 1996 ACM Workshop on Materialized Views: Techniques and Applications*, pages 110–118, June 1996.

[22] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[23] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. In *Proc. of the 2000 Intl. Conf. on Extending Database Technology*, pages 395–412, Konstanz, Germany, March 2000.

[24] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proc. of the 2001 Intl. Conf. on Data Engineering*, Heidelberg, Germany, April 2001.

[25] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, 2003.

APPENDIX I

PROOF OF LEMMA 2

Recall that X *stochastically dominates* Y if $\mathbf{Pr}[X > a] \geq \mathbf{Pr}[Y > a]$ (or, equivalently, $\mathbf{Pr}[X \leq a] \leq \mathbf{Pr}[Y \leq a]$) for all a.

Proof: Let $P(x, start, end)$ denote the probability of going from position $start$ to position end in exactly x steps, without hitting position i, n, or end except in the last step. Clearly, $P(x, start, end)$ does not depend on the transition probabilities at position i. Therefore, $P(x, start, end)$ is the same for both W_1 and W_2.

Let Z_j and Z'_j denote the hitting times of random walks W_1 and W_2 starting from position j, respectively. We will prove a stronger statement than Lemma 2: Z_j stochastically dominates Z'_j for all $j = 0, \ldots, n$. That is, $\mathbf{Pr}[Z_j > a] \geq \mathbf{Pr}[Z'_j > a]$ for all j and a. Since hitting times are non-negative integers, we only need to prove the statement for non-negative integer values of a.

The proof is induction on a. The base case of $a = 0$ is trivial. Suppose that for all $a < m$ and $j = 0, \ldots, n$, $\mathbf{Pr}[Z_j > a] \geq \mathbf{Pr}[Z'_j > a]$. Now consider the case when $a = m$. For any j

such that $i < j \leq n$, we have

$$\begin{aligned}
\mathbf{Pr}[Z_j > m] &= \mathbf{Pr}[Z_j > m,\ W_1 \text{ hits } n \text{ without ever hitting } i] \\
&\quad + \mathbf{Pr}[Z_j > m,\ W_1 \text{ hits } i \text{ before hitting } n] \\
&= \sum_{x>m} P(x,j,n) + \sum_{x>0} P(x,j,i)\,\mathbf{Pr}[Z_i > m-x].
\end{aligned}$$

Similarly,

$$\mathbf{Pr}[Z'_j > m] = \sum_{x>m} P(x,j,n) + \sum_{x>0} P(x,j,i)\,\mathbf{Pr}[Z'_i > m-x].$$

By induction hypothesis, $\mathbf{Pr}[Z_i > m-x] \geq \mathbf{Pr}[Z'_i > m-x]$, so $\mathbf{Pr}[Z_j > m] \geq \mathbf{Pr}[Z'_j > m]$. The case when $0 \leq j < i$ can be handled similarly.

In the case when $j = i$, let $u = \mathbf{Pr}[Z'_i > m-1] - \mathbf{Pr}[Z'_{i+1} > m-1]$ and $v = \mathbf{Pr}[Z'_{i-1} > m-1] - \mathbf{Pr}[Z'_i > m-1]$. We have

$$\begin{aligned}
\mathbf{Pr}[Z_i > m] &= p_i\,\mathbf{Pr}[Z_{i+1} > m-1] + q_i\,\mathbf{Pr}[Z_{i-1} > m-1] + (1-p_i-q_i)\,\mathbf{Pr}[Z_i > m-1] \\
&\geq p_i\,\mathbf{Pr}[Z'_{i+1} > m-1] + q_i\,\mathbf{Pr}[Z'_{i-1} > m-1] + (1-p_i-q_i)\,\mathbf{Pr}[Z'_i > m-1] \\
&= \mathbf{Pr}[Z'_i > m-1] - p_i u + q_i v; \\
\mathbf{Pr}[Z'_i > m] &= p'_i\,\mathbf{Pr}[Z'_{i+1} > m-1] + q'_i\,\mathbf{Pr}[Z'_{i-1} > m-1] + (1-p'_i-q'_i)\,\mathbf{Pr}[Z'_i > m-1] \\
&= \mathbf{Pr}[Z'_i > m-1] - p'_i u + q'_i v.
\end{aligned}$$

Note that $u \geq 0$ and $v \geq 0$, because for any z, j_1, j_2 where $0 \leq j_1 < j_2 \leq n$,

$$\begin{aligned}
\mathbf{Pr}[Z'_{j_1} > z] &= \sum_{x>0} \mathbf{Pr}[W_2 \text{ starts at } j_1 \text{ and first hits } j_2 \text{ in } x \text{ steps}]\,\mathbf{Pr}[Z'_{j_2} > z-x] \\
&\geq \mathbf{Pr}[Z'_{j_2} > z] \sum_{x>0} \mathbf{Pr}[W_2 \text{ starts at } j_1 \text{ and first hits } j_2 \text{ in } x \text{ steps}] \\
&= \mathbf{Pr}[Z'_{j_2} > z].
\end{aligned}$$

To prove that $\mathbf{Pr}[Z_i > m] \geq \mathbf{Pr}[Z'_i > m]$, we only need to show that $-p_i u + q_i v \geq -p'_i u + q'_i v$, or, equivalently, $(p'_i - p_i)u - (q'_i - q_i)v \geq 0$. Since $p'_i + q'_i \geq p_i + q_i$ and $p'_i/q'_i \geq p_i/q_i$, there exist $r, \delta \geq 0$ such that $p_i = (p'_i - \delta)(1-r), q_i = (q'_i + \delta)(1-r)$. We have

$$\begin{aligned}(p'_i - p_i)u - (q'_i - q_i)v &= (p'_i r + \delta(1-r))u - (q'_i r - \delta(1-r))v \\ &= (p'_i u - q'_i v)r + \delta(1-r)(u+v) \\ &= (\mathbf{Pr}[Z'_i > m-1] - \mathbf{Pr}[Z'_i > m])r + \delta(1-r)(u+v) \quad \geq \quad 0,\end{aligned}$$

and Z_i stochastically dominates Z'_i. ■

Appendix II

Proof of Theorem 3

Proof: Our approach to dropping the independence assumption basically follows the same line of reasoning in the proof of Lemma 2.

Consider a random walk W with memory on $\{0, 1, \ldots, n\}$. Let H be the set of all possible histories (steps taken) starting at 0. Each step in the history may be "L" (moving left), "R" (moving right) or "S" (staying), and we use $\overline{hs}$ to denote the concatenation of a history h with an additional step s (L, R, or S). For any history $h \in H$, let $e(h)$ be the ending position of the random walk that follows h, and let $p(h)$ and $q(h)$ be the transition probabilities of moving from $e(h)$ to $e(h)+1$ and $e(h)-1$, respectively. Since W is origin-tending (strictly origin-tending), we have $p(h) \leq q(h)$ ($\delta \cdot p(h) < q(h), \delta > 1$) for all $h \in H$. We define another random walk W' with fixed transition probabilities $p' = \max_{h \in H}(p(h) + q(h))/(1+t)$ and $q' = \max_{h \in H}(p(h) + q(h))/(1 + \frac{1}{t})$ where $t = \min_{h \in H}(q(h)/p(h))$. Clearly we have $p' + q' \geq p(h) + q(h)$ and $p'/q' \geq p(h)/q(h)$ for all $h \in H$. The results of Theorem 1 and 2 hold for the random walk W', so all we need to show is that the hitting time of W stochastically dominates that of W'.

Let $Z(h)$ and $Z'(h)$ denote the hitting times of random walks W and W' starting at $e(h)$ and following the history h, respectively. We will prove that $\mathbf{Pr}[Z(h) > a] \geq \mathbf{Pr}[Z'(h) > a]$ for all $h \in H$. The proof is induction on a. The base case of $a = 0$ is trivial. Suppose that for all $a < m$ and $h \in H$, $\mathbf{Pr}[Z(h) > a] \geq \mathbf{Pr}[Z'(h) > a]$. Now consider the case $a = m$.

For any $h \in H$, let $u = \mathbf{Pr}[Z'(\overline{hS}) > m-1] - \mathbf{Pr}[Z'(\overline{hR}) > m-1]$ and $v = \mathbf{Pr}[Z'(\overline{hL}) > m-1] - \mathbf{Pr}[Z'(\overline{hS}) > m-1]$. We have

$$
\begin{aligned}
\mathbf{Pr}[Z(h) > m] &= p(h)\,\mathbf{Pr}[Z(\overline{hR}) > m-1] + q(h)\,\mathbf{Pr}[Z(\overline{hL}) > m-1] \\
&\quad +(1-p(h)-q(h))\,\mathbf{Pr}[Z(\overline{hS}) > m-1] \\
&\geq p(h)\,\mathbf{Pr}[Z'(\overline{hR}) > m-1] + q(h)\,\mathbf{Pr}[Z'(\overline{hL}) > m-1] \\
&\quad +(1-p(h)-q(h))\,\mathbf{Pr}[Z'(\overline{hS}) > m-1] \qquad \text{(induction hypothesis)} \\
&= \mathbf{Pr}[Z'(\overline{hS}) > m-1] - p(h)u + q(h)v; \\
\mathbf{Pr}[Z'(h) > m] &= p'\,\mathbf{Pr}[Z'(\overline{hR}) > m-1] + q'\,\mathbf{Pr}[Z'(\overline{hL}) > m-1] \\
&\quad +(1-p'-q')\,\mathbf{Pr}[Z'(\overline{hS}) > m-1] \\
&= \mathbf{Pr}[Z'(\overline{hS}) > m-1] - p'u + q'v.
\end{aligned}
$$

Note that W' is memoryless, $Z'(\overline{hS}) = Z'(h)$. We can show that $u, v \geq 0$ and $-p(h)u + q(h)v \geq -p'u + q'v$ using the same technique as in Appendix I, which completes the proof. ■