Efficient Maintenance of Materialized Top-*k* Views

Ke Yi, Hai Yu, Jun Yang
Dept. of Computer Science, Duke University
Gangqiang Xia, Yuguo Chen
Inst. of Statistics and Decision Sciences, Duke University

Materialized top-*k* views

- Base table: *T*(*id*, *val*)
- A top-*k* query:

```
SELECT id, val FROM T
ORDER BY val FETCH FIRST k ROWS ONLY;
```

  - Special cases: `MIN` and `MAX`
  - Need at least one scan of *T* (assuming there is no ordered index on *T*.*val*)
- Want better query response time?

☞ Standard trick—make it a materialized view

Maintaining a top-*k* view

- Self-maintainable (i.e., no need to query base table) in many cases
  - Insertion
  - Deletion of a tuple outside the top *k*
  - Update of a tuple that does not cause it to drop out of the top *k*
- Not self-maintainable in other cases
  - Deletion of a tuple from the top *k*
  - Update of a tuple causing it to drop out of the top *k*
  - ☞Need an expensive refill query over the base table to find the new *k*-th ranked tuple

Traditional warehousing solution

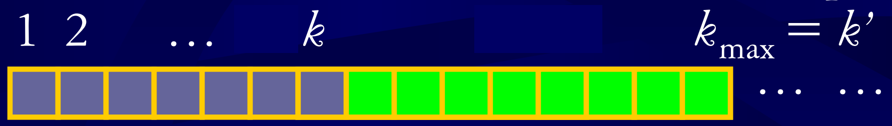
- ❖ Make views completely self-maintainable by storing additional auxiliary views
  - Example: to make $\sigma_{p1} R \bowtie_{p} \sigma_{p2} S$ self-maintainable, store $\sigma_{p1} R$ and $\sigma_{p2} S$
- ❖ To make a top-*k* view completely self-maintainable, we need to store a copy of the entire base table!
  - ☞Cost is too high: not just storage, but also the overhead of maintaining the copy
- ❖ Why pay such a high cost to catch some rare cases?

Two observations

- Instead of complete compile-time self-maintenance, aim at achieving runtime self-maintenance with high probability
  at much lower cost
  ☞ "Optimize for the common case"
- Instead of static auxiliary view definitions determined at compile-time, allow dynamic auxiliary view definitions
  which change according to the update workload
  ☞ Like a "semantic cache" of auxiliary data

A simple algorithm

- Idea: maintain a top-k' view, where k' changes at run-time but stays between k and some $k_{\max}$
  - The extra tuples serve as a “buffer” to deter refill queries



V: a top-k' view

$v_{k'}$: value of the lowest ranked tuple currently in V

Update: tuple t has its value updated to val

- **Ignorable**: t not in V, $val < v_{k'}$ Do nothing
- **Neutral**: t in V, $val > v_{k'}$ Update V; no change to k'
- **Good**: t not in V, $val > v_{k'}$ Insert t into V; increment k'
  - If k' exceeds $k_{\max}$, discard the lowest ranked tuple in V
- **Bad**: t in V, $val < v_{k'}$ Delete t from V; decrement k'
  - If k' drops below k, issue a refill query to restore k' to $k_{\max}$

Remaining questions

- How do we choose a right value for k_{max}?
- What factors affect the optimal k_{max} value?
  - Trade-off: increasing k_{max} reduces refill frequency, but
    - V takes more space
    - Updating V takes longer
    - More updates need to be applied to V
- How effective is the algorithm with small k_{max}?
- How do we choose k_{max} without accurate prior knowledge about the update workload?

A closer look at the maintenance cost

Amortized cost of processing one update =

$$C_{\text{update}} \times (\, 1 - f_{\text{ignore}} \,) + C_{\text{refill}} \times f_{\text{refill}}$$

- C_{update}: cost of updating V; $O(\log k_{\max})$
- f_{ignore}: fraction of updates that are ignorable (decreases as $k_{\max}$ increases)
- C_{refill}: cost of a refill operation; $O(N)$, where N is the size of the base table
- f_{refill}: frequency of refill operations

☞ Since $C_{\text{refill}} \gg C_{\text{update}}$, a reasonable goal is to reduce f_{refill} to $1/N$, so the second product becomes $O(1)$

Random walk model

- ❖ Between two refills, the value of k' follows a random walk on points $\{ k - 1, k, \ldots, k_{\max} \}$
  - Begins with $k_{\max}$ (right after a refill)
  - Moves left on a bad update
  - Moves right on a good update
  - Stays put on an ignorable or neutral update
  - Ends with $k - 1$ (when another refill is needed)
- ☞ Refill interval Z = hitting time from $k_{\max}$ to $(k - 1)$

- ❖ Assume probabilities of bad and good updates are fixed at p and q for now; will drop this assumption later

First try: expected hitting time

h_i: expected time to hit $(k - 1)$ starting from i

- $h_{k_{\max}} = 1 + p \times h_{k_{\max}-1} + (1-p) \times h_{k_{\max}}$
- $h_i = 1 + p \times h_{i-1} + q \times h_{i+1} + (1-p-q) \times h_i$
- $h_{k-1} = 0$

❖ Can solve for $h_{k_{\max}}$ $(= \mathbb{E}[Z])$ directly

- E.g., if $p = q$ then $h_{k_{\max}} = (k_{\max} - k + 1)\,(k_{\max} - k + 2) / (2p)$
  - That is, we can choose $k_{\max} = (k-1) + N^{0.5}$ so that $\mathbb{E}[Z] \approx N$

❖ But we want $\mathbb{E}[f_{\text{refill}}] = \mathbb{E}[1/Z]$, which is not equal to $1 / \mathbb{E}[Z]$ in general!

☞ Change strategy: make sure that $\mathbb{P}[Z > N]$ is high

High-probability result when $p = q$

- Theorem: When $p = q$, if $k_{\max} = (k-1) + N^{0.5+\varepsilon}$ then $\mathbb{P}[Z > N] \geq 1 - 4 \cdot \exp(-N^{2\varepsilon} / 2)$

☞ In English

When bad and good updates are equally likely,
we can pick $k_{\max}$ to be a just a bit more than sqrt(N)
in order to ensure that, with high probability,
refill only occurs after at N updates

- We think $p = q$ is a common case
  - If the value distribution is stationary, the rate at which tuples enter top k should be the same as the rate at which they leave top k

High-probability result when $p < q$

- ❖ Theorem: When $p < q$, if $k_{\max} = (k-1) + c \ln N$, then $\mathbb{P}[Z > N] \geq 1 - o(1)$
  - For a large enough constant c depending only p and q
- ☞ In English

  When bad updates are less likely than good updates,

  we can pick $k_{\max}$ to be $O(\ln N)$

  in order to ensure that, with high probability,

  refill only occurs after at N updates
- ❖ Intuitively, this case is better because the view is more likely to grow than to shrink

What if $p > q$?

- The view is more likely to shrink than to grow
- Need $k_{\max} = O(N)$ to bring $\mathbb{E}[Z]$ up to N
  - Might as well keep a copy of the base table!
  - We conjecture no good solution exists
- We also hope $p > q$ is a rare case
  - Typically, people enjoy watching tuples "compete" with each other to enter top k
  - It is less interesting to watch tuples trying to "escape" from top k

Generalization

- No need to assume that p and q are fixed
- No need to assume that random walk is memoryless
- Theorem for $p = q$ still holds if "$p = q$" is replaced by "random walk W is origin-tending"
  - That is, regardless of the previous steps taken, the probability of W moving towards $k_{\max}$ is always no less than that of moving towards k
- Theorem for $p < q$ still holds if "$p < q$" is replaced by "random walk W is strictly origin-tending"
  - That is, regardless of the previous steps taken, the probability of W moving towards $k_{\max}$ is always no less than δ times that of moving towards k, where $\delta > 1$

Case study: random up-and-downs

- ❖ Initial values: symmetric unimodal distribution with mean μ
- ❖ In each time step, choose an item at random and modify it by a value drawn from a symmetric unimodal distribution with mean 0
- ❖ What are the odds of this update being good/bad?
- ❖ Can show: $p < q$ as long as top-k values $> \mu$
  - ☞ Random walk is origin-tending
  - ☞ $k_{\max} = N^{0.5+\varepsilon}$ is enough

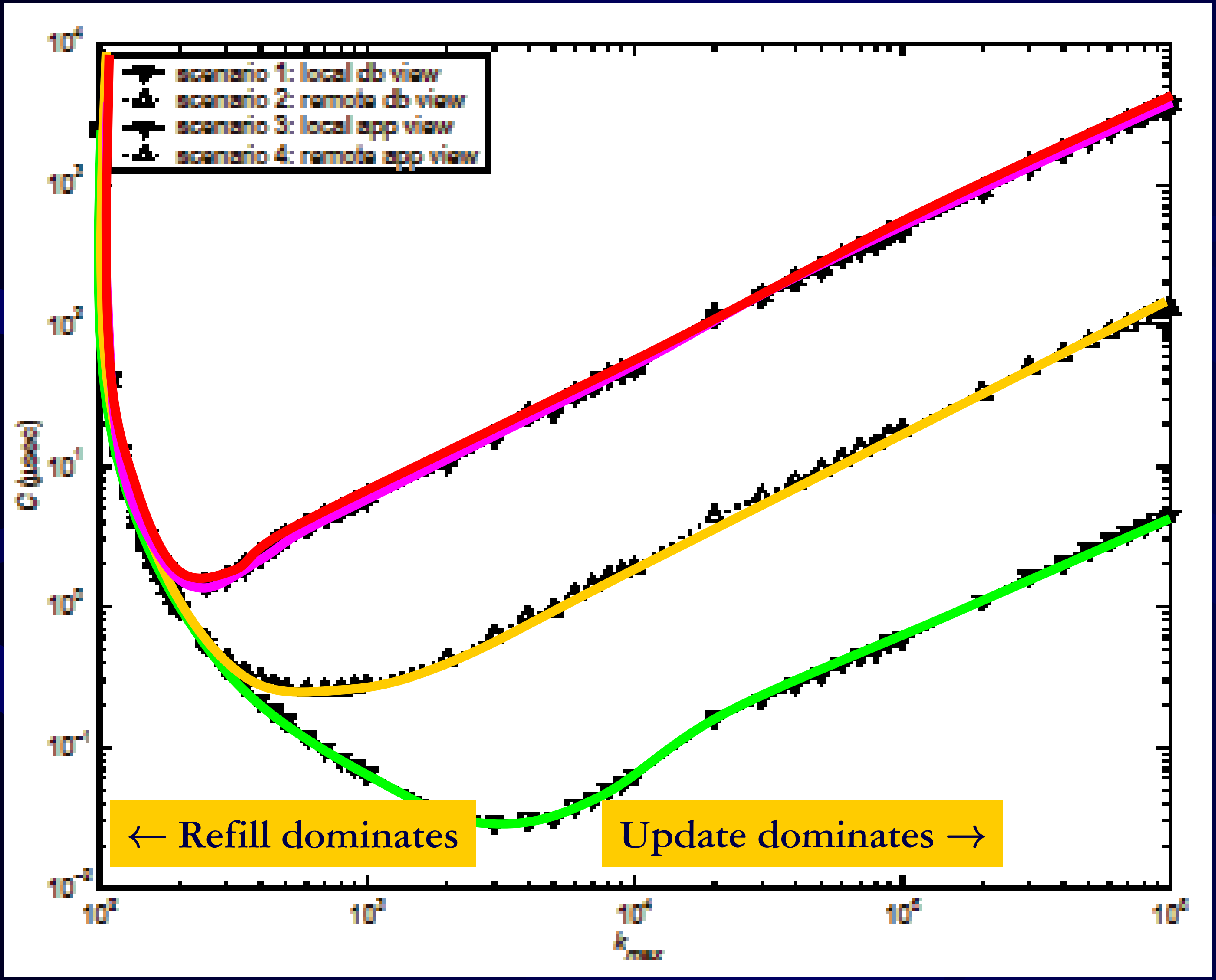
Case study: total sales in a moving window

- ❖ Sales for a book b over time: $X^b{}_1, X^b{}_2, \ldots, X^b{}_t, \ldots$ (assume all independently & identically distributed)
- ❖ Interested in total sales of b in a moving window: $\sum_{t-w+1 \leq t' \leq t} X^b{}_{t'}$
- ❖ As t moves forward, what are the odds that b moves in/out of top-k?
- ❖ Can show: $p = q$
  - ☞ Random walk is origin-tending
  - ☞ $k_{\text{max}} = N^{0.5+\varepsilon}$ is enough

Experiments

- Scenarios
  - Base table in DBMS
  - Top-*k* view can be maintained by application (in-memory heap) or by DBMS (B^+-tree)
    - Different update cost
  - Top-*k* view can be maintained locally or remotely
    - Different refill cost
  - ☞4 possible combinations
- Costs are real ☺ (measured for different view/query sizes)
- Data/updates are synthetic ☹, but not over-simplistic
  - Simulation of total sales in a moving window, with daily sales following a Poisson distribution

Maintenance cost vs. k_{max}



← Refill dominates

Update dominates →

Remote db view

Local db view

Remote app view

Local app view

Choosing $k_{\max}$ in practice

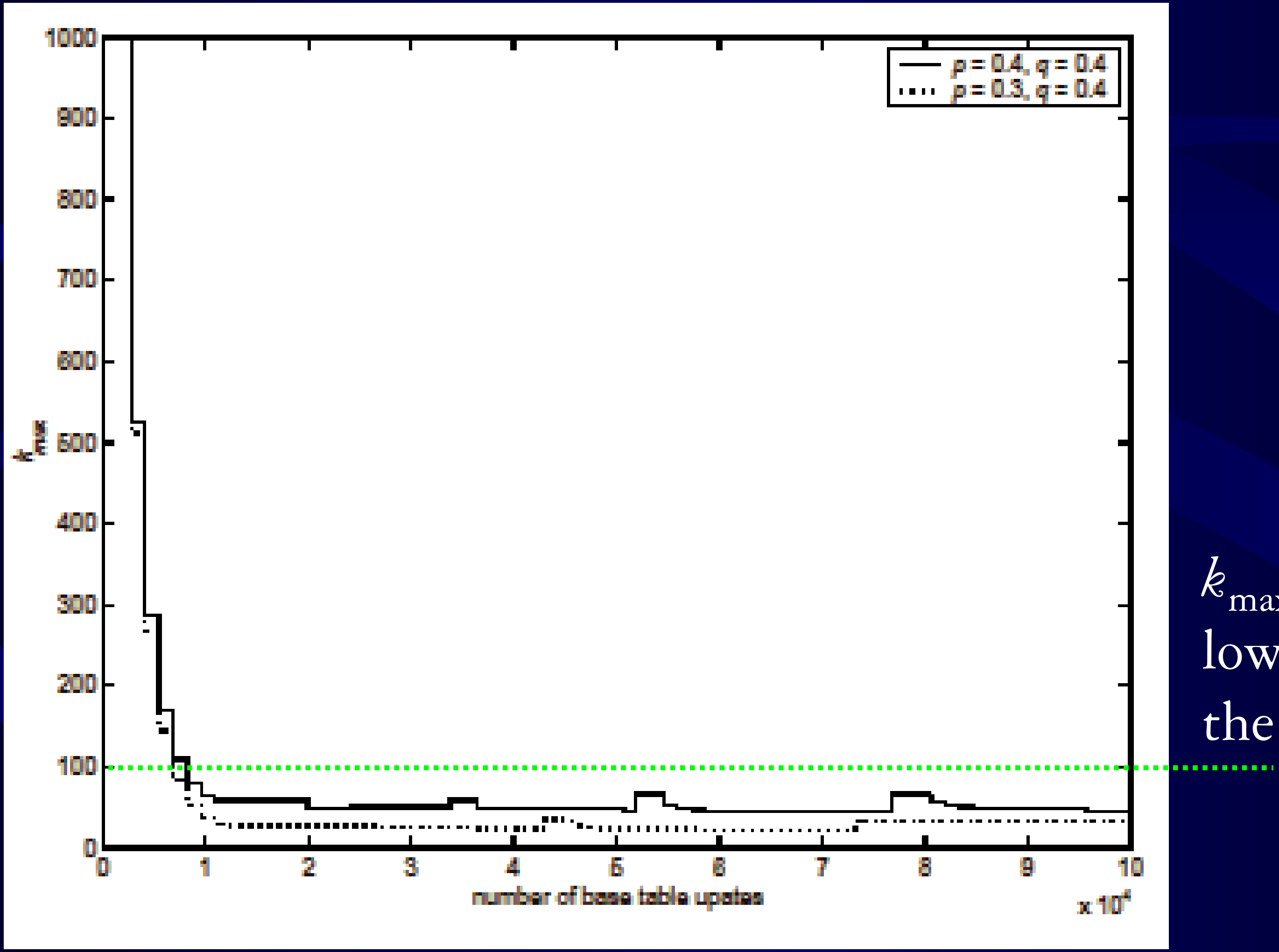
- Theoretical bounds may not be tight/accurate enough
- p and q are difficult to measure
- p, q, and costs may vary at runtime

- Idea: dynamically adjust $k_{\max}$ so that
  amortized cost of refill $\approx$ that of view update
  - Start with some guess for $k_{\max}$ ($N^{0.6}$ is reasonable)
  - Target refill interval: C_{refill} / C_{update} (observed at runtime)
  - If actual refill interval $<$ target / α, increase $k_{\max}$ by a factor
  - If actual refill interval $>$ target $\cdot$ α, decrease $k_{\max}$ by a factor

☞ Allow some leeway (α) from the target interval

Experiments with adaptive algorithm

$N = 10{,}000;\ k = 10$



k_{max} can be lower than what the theory predicts

Conclusion and future work

- Top-*k* view maintenance: a little trick goes a (provably) long way!
- Main idea: auxiliary data for high-probability runtime self-maintenance
- Currently working on generalizing the idea to other types of views (e.g., joins)

☞ For detailed proofs and experiment results, see `http://www.cs.duke.edu/~junyang/papers/yyyxc-topk.ps`

Related work

- Lots of work on view self-maintenance
  - Blakeley et al., *TODS* 1989; Gupta et al., *EDBT* 1996
  - Huyn, *VLDB* 1997: runtime self-maintenance
  - Quass et al., *PDIS* 1996, etc.: auxiliary data for compile-time self-maintenance
    - ☞We propose auxiliary data for runtime self-maintenance with higher probability
- Lots of work on top-*k* queries
  - Most focuses on efficient query processing
  - Hristidis et al., *SIGMOD* 2001: select ordered/top-*k* views to materialize
    - ☞We support efficient maintenance algorithm
- Top-*k* view maintenance
  - Traditionally: deletes/updates to `MIN` and `MAX` are not handled
  - Palpanas et al., *VLDB* 2002: “work areas” for `MIN` and `MAX`
    - ☞We provide rigorous analysis and guidelines for choosing sizes of “work areas”
  - Babcock & Olston, upcoming *SIGMOD* 2003: approximate distributed top-*k* maintenance, focus on reducing communication