Towards Optimal Snapshot Materialization To Support Large Query Workload For Append-only Temporal Databases

Amin Beirami
Faculty of Science
UOIT
Oshawa, Canada
beirami.m.a@gmail.com

Ken Pu
Faculty of Science
UOIT
Oshawa, Canada
ken.pu@uoit.ca

Ying Zhu
Faculty of Business IT
UOIT
Oshawa, Canada
ying.zhu@uoit.ca

***Abstract*—We present several results on optimal snapshot materialization for append-only temporal databases in order to support very large scale query workload. Our data model is temporal relational data stored in an append-only database. When the temporal database receives multiple queries querying at different timestamps along the timeline, it would be prohibitively expensive to recompute the snapshots at each of the timestamps.**

In this paper, we present a practical solution to support large query load by materializing m snapshots at optimal timestamps. We show that optimal snapshot timestamps can be computed efficiently in *linear time*. We further show that with varying query load, we can dynamically adjust snapshots to adjust to the changin query load.

***Keywords*-temporal databases; materialized views; query workload; optimization**

I. Introduction

Recently, as part of the Big Data movement, information systems have been designed to archive every bit of data over time [1]. In contrast to traditional data warehouse databases, modern databases are aimed to store *every* update, insert and deletion as a timeline. This is known as *multi-version databases* [2].

Example 1: Consider the following table `Products` which stores the product information.

id	part-name	Manufacture	Price
0010	Game Console	Nintendo	$399
0011	Keyboard	Microsoft	$89
⋮	⋮	⋮	⋮

Suppose that there will be frequent updates to the products such as addition of new products, renaming of existing products, and price adjustments. The `product` table can only store the eventual state of the database *after* each updates, but not the updates themselves. The update transactions are quite valuable themselves.

We would like to augment the database schema to make `product` into a *temporal* table which we will call $\texttt{product}^T$. The temporal table is to store *all* modifications to the `product` table. Hence, it has the following schema:

:updates:	:deleted:	id	part-name	Manufacture	Price
00000001	false	0010	Game Console	Nintendo	$359
00000002	false	0020	Monitor	Samsung	$159
00000003	false	0010	Game Console	Nintendo	$399
00000004	true	0011	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮

It shows four specific updates done to the `product` table:

- The *game console* is discounted to a new price $359.
- A new product *Monitor* is added to the table.
- The *game console* is no longer on sale, so its price is back to $399.
- The keyboard (product ID 0011) is discontinued, so it's deleted from the table.

II. Problem Definition

Definition 1 (Temporal database): Let $r_1, r_2, \ldots r_n$ be the tables in a database D. Denote the attributes of each table as $\mathrm{attr}(r_i)$.

A temporal table, denoted r_i^T, is a table with the attributes:

$$\mathrm{attr}(r_i^T) = \{\mathrm{updates}, \mathrm{deleted}\} \cup \mathrm{attr}(r_i)$$

By a temporal database, we mean the database obtained by augmenting D by the temporal tables:

$$D^T = D \cup \{r_i^T : r_i \in D\}$$

There is a lot of value in maintaining D^T because it stores the entire update history of D. Traditionally, this may be seen as an overly burdening task, but that is no longer the case with increasingly affordable cloud storage solutions. Furthermore, the update history provides many valuable insights into the dynamics of data. We are motivated to support queries for the database at a specific time.

Definition 2 (Snapshots and queries): Given a temporal table r^T, and a timestamp t, we denote $r(t)$ to be the table instance obtained by applying the updates in r^T where $r^T[\mathrm{updates}] \leq t$.

A snapshot is a materialized version of $D(t) = \{r_1(t), r_2(t), \ldots, r_n(t)\}$. A snapshot query is an arbitrary relational query on $D(t)$.

We can construct the snapshots using simple windowing functions (as in supported by PostgreSQL [3]).

```
snapshot(r, t) =
 WITH T AS (
  SELECT id, {last_value(x) as x : x ∈ attr(r)} OVER W
  FROM r^T
  WHERE updates ≤ t
  WINDOW W AS PARTITION BY id ORDER BY updates
 )
 SELECT id, {x : x ∈ attr(r)} FROM T
 WHERE NOT T.deleted
```

The query $\text{snapshot}(r, t)$ computes the snapshot of r at timestamp t by applying the latest update of each tuple up to timestamp t, while removing tuples that have been deleted.

Proposition 1: Assume that the tables are updated at a constant rate over time, then the complexity of $\text{snapshot}(r, t)$ is

$$\mathcal{O}(|\{x : x \in r^T \text{ and } x.\text{updates} \leq t\}|) \simeq \mathcal{O}(t)$$

A. Query answering using snapshots

Using precomputed materialized view has been shown to be highly effective [4], [5]. Thus, in order to answer a query $Q(t)$ on $D(t)$, we propose to first precompute snapshots $\text{snapshot}(r, t)$ and then evaluate Q. If some snapshots are precomputed and materialized, then we can save on the computational cost in answering the query $Q(t)$.

Proposition 2: Let r be a temporal relation, with s and t being timestamps. Suppose we have materialized $\text{snapshot}(r, s)$. Then $\text{snapshot}(r, t)$ can be computed with complexity:

$$\mathcal{O}(|\{x : x \in r^T \text{ and } x.\text{updates} \in [s, t]\}|) \simeq \mathcal{O}(|s - t|)$$

B. Optimal materialization of snapshots

We define the central problems of this paper, namely computing the timestamps at which we can materialize the snapshots to best answer a given query workload. Let $T_Q = \{q_1, q_2, \ldots, q_n\}$ be the timestamps of n queries, each querying the database at $D(q_i)$. We define the cost functions as the total query answering cost given some precomputed and materialized snapshots.

Definition 3 (Query answering cost): Suppose that a single snapshot at time s is materialized, then

$$\text{cost}(T_Q|s) = \sum_{q \in T_Q} |q - s|$$

If we have multiple snapshots at times $S = \{s_1, s_2, \ldots, s_m\}$ materialized, then

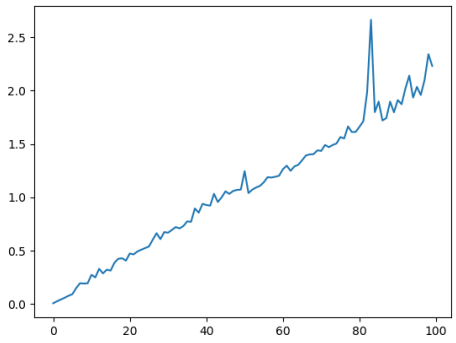

Figure 1. Query answering for 100 queries without snapshots.

$$\text{cost}(T_Q|S) = \sum_{q \in T_Q} \min\{|q - s| : s \in S\}$$

Definition 4 (Optimal snapshot placement): The *single snapshot placement* problem is to compute the timestamp s^* such that $\text{cost}(T_Q|s)$ is minimized.

The *m-snapshots placement* problem is to compute a set of timestamps $S^* = \{s_1, s_2, \ldots s_m\}$ such that $\text{cost}(T_Q|S)$ is minimized.

In the remainder of the paper, we present the algorithms and experimental evaluation to demonstrate that the optimal snapshot placement problems can be efficiently solved, and that their solutions yield a practical solution to support queries of temporal databases.

III. Algorithm

We first solve the simpler single snapshot placement problem.

Proposition 3: The single snapshot placement problem can be solved in $\mathcal{O}(|T_Q|)$, with the optimal snapshot placement at $s^*(T_Q) = \text{median}(T_Q)$, which can be computed in $\mathcal{O}(|T_Q|)$.

So by Proposition 3, it's straightforward to compute the optimal placement of a single snapshot with respect to a given query workload. However, in practice, we want to allocate a number, m, of snapshots to be placed, where m is a determined by the available resource. In this section, we extend Proposition 3 to handle arbitrary number of snapshots.

First we will present an recursive algorithm that solves the optimal m-snapshot placement problem.

Without loss of generality, let's assume that the query timestamps are sorted in $Q = \{q_1, q_2, \ldots, q_n\}$. Here Q is the sorted list of query timestamps. We denote $Q[i, j] = \{q_i, q_{i+1}, \ldots, q_{j-1}, q_j\}$.

The objective is to compute $S = \{s_1, s_2, \ldots, s_m\}$ such that the query answering time given by $\text{cost}(Q|S)$ is minimized.

Proposition 4 (Segmentation of queries): Given any snapshot timestamps in a sorted order $S = \{s_1, s_2, \ldots, s_m\}$ such that $s_i \leq s_{i+1}$, the snapshots partition the queries Q into m non-overlapping segments $Q[1, i_1], Q[i_1 + 1, i_2], \ldots Q[i_{m-1}, i_m]$ such that queries in $Q[i_j, i_{j+1}]$ uses s_j in the optimal query answer strategy.

Proposition 4 allows us to construct a dynamic programming algorithm that computes the *exact* optimal snapshot placements.

A. Recursive formulation

Let $\mathrm{opt}(Q, m)$ be the optimal m-snapshot placements for the query workload Q.

Proposition 5 (Optimality of sub-problems): Let $S^* = \mathrm{opt}(Q, m)$. Let $\mathcal{Q}$ be the partition of segments created by S^*. Then, the prefix of S^* is also an optimal $m-1$ snapshot placement of the prefix of $\mathcal{Q}$. Formally,

$$\mathrm{prefix}(S^*) = \mathrm{opt}(\cup \mathrm{prefix}(\mathcal{Q}), m-1)$$

We can formulate a recursive definition of $\mathrm{opt}(Q, m)$ using Proposition 5. The intuition is that we try out all possible *last* segment of Q, and pick the one with the lowest cost.

The recursive definition of $\mathrm{opt}(Q, m)$ is given as:

- Base case $\mathrm{opt}(Q, 1) = \{\mathrm{median}(Q)\}$.
- Induction on m:

$$i^* = \mathrm{argmin}\{\mathrm{cost}(\mathrm{opt}(Q[1, i], m-1)) : i \in [1, n]\}$$

$$\mathrm{opt}(Q, m) = \mathrm{opt}(Q[1, i^*]) \cup \{\mathrm{median}(Q[i^* + 1, n])$$

Proposition 6: The recursive formulation of $\mathrm{opt}(Q, m)$ requires $\mathcal{O}(2^m)$.

Fortunately, we are able implement $\mathrm{opt}(Q, m)$ in polynomial time as a dynamic programming problem.

B. Dynamic programming formulation

We can build a table **OPT** as a two dimensional array indexed by (i, k) where $i \in [1, n]$ and $k \in [1, m]$. Each entry in the table $\mathbf{OPT}[i, k] = \mathrm{opt}(Q[1, i], k)$. We can compute $\mathbf{OPT}[i, k]$ in a bottom up fashion [6].

```
computeOPT(Q, m) =
 n = |Q|
 OPT[i, 0] = ∞
 for k = 1 → m
   for i = 1 → n
     j* = argmin_{j∈[1,i]} (cost(OPT[j, k−1]) + cost(Q[j+1, n]))
     OPT[i, k] = OPT[j*, k−1] ∪ {median(Q[j+1], n)}
   end for
 end for
```

Proposition 7: The complexity of computing all the entries of **OPT** is $\mathcal{O}(mn^2)$.

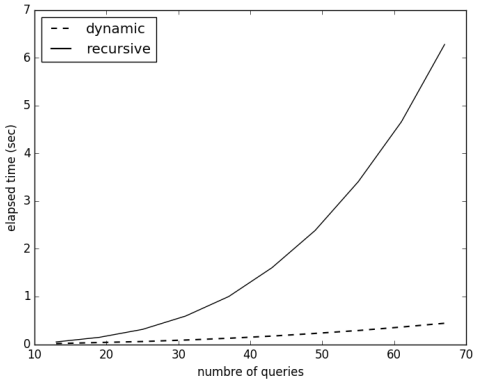

Figure 2. Optimization time with respect to the number of queries

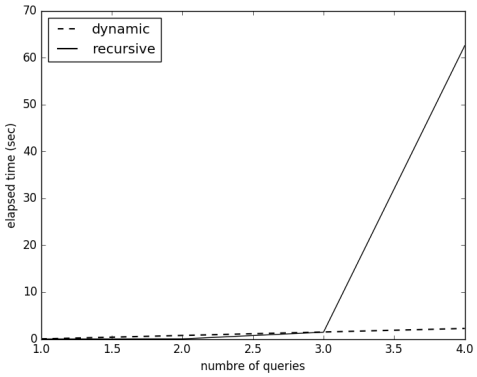

Figure 3. Optimization time with respect to the number of snapshots

IV. Experimental Evaluation

In order to evaluate our algorithms, we have generated synthetic temporal databases. We have also generated database using the TPCH schema with 1,000,000 tuples in the main table. We also created a set of random database modify, insert and delete updates at 1000 distinct timestamps. This creates a temporal database with 1000 timestamps.

To confirm that, without querying at a specific timestamp q, the cost is $\mathcal{O}(q)$ as given by Proposition 1, we sampled 100 query timestamps and measured the query answering performance, shown in Figure II-B. It confirms that the linear-time cost function is indeed valid in practice.

To evaluate the performance of our *optimal* snapshot computation, we evaluated the recursive formulation given by Section III-A and the dynamic programming formulation Section III-B. Figure 2 and Figure 3 show the snapshot placement computational time is significantly reduced by dynamic programming.

To illustrate that the optimal snapshot placement indeed produces the best query answering performance, we compared the query answering cost of three approaches:

- Pick m random timestamps to place the snapshots.

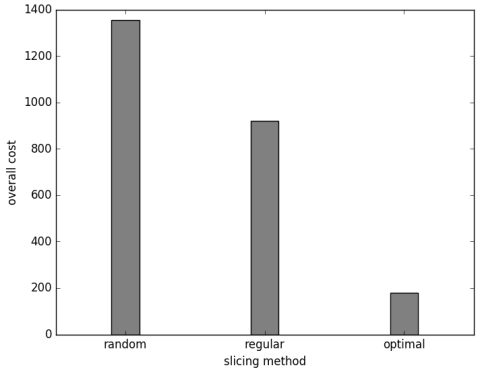

Figure 4. Relative query answering cost

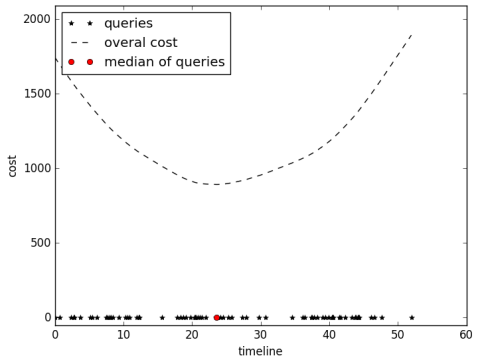

Figure 5. Cost of query answering using a single snapshot over different snaptshot timestamps

- Pick m evenly intervaled timestamps to place the snapshots.
- Pick m timestamps computed by dynamic programming.

Figure 4 shows that the placements obtained by dynamic programming clearly beats the other two approaches.

We verify the optimality of placing a single snapshot given in Proposition 3. We set $m = 1$, and measured the query answering cost over different choices of snapshot placement. Figure 5 shows that the optimal placement is at the median of the query times, as predicated by Proposition 3.

Finally, we demonstrate the query answering performance with increasingly many snapshots in Figure 6. It shows that more snapshots will improve the query performance. We argue that this is a crucial verification that our approach will enable efficient temporal query processing for databases that store the timeline of its tables.

V. Conclusion and Future Work

In this paper, we have presented some results obtained toward optimally supporting temporal relational queries of databases that store the timelines of its relational tables. In order to avoid recomputation of the database states while

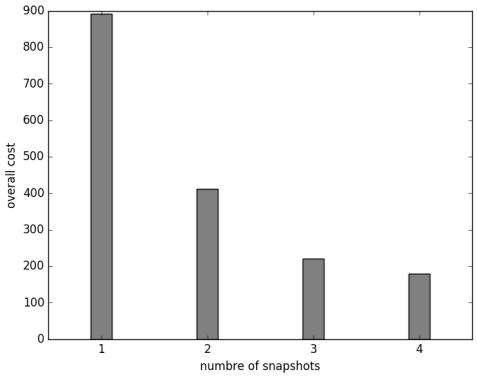

Figure 6. Query answering cost with increasing number of snapshots

making use of the storage space efficiently, our solution is to materialize m snapshots at well-chosen timestamps.

We have constructed a model to describe the query answering cost, and this cost model allows us to formulate the m-snapshot placement problem as an optimization problem. We showed that dynamic programming can be used to solve the problem *exactly*. Our experimental evaluation demonstrates that our cost model agrees with relational database systems, and that our snapshot placements improve query processing significantly.

This is on-going research. As future work, we will be investigating approximation methods to further speed up the snapshot placement calculuation. We also wish to investigate maintaining snapshot placement dynamically to cope with a dynamic query workload.

References

[1] Y. Tian, Y. Ji, and J. Scholer, "A prototype spatio-temporal database built on top of relational database," in *Information Technology-New Generations (ITNG), 2015 12th International Conference on*. IEEE, 2015, pp. 14–19.

[2] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross, "Reducing database locking contention through multi-version concurrency," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1331–1342, 2014.

[3] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[4] M. K. Sohrabi and V. Ghods, "Materialized view selection for a data warehouse using frequent itemset mining." *Jcp*, vol. 11, no. 2, pp. 140–148, 2016.

[5] J. Du, R. J. Miller, B. Glavic, and W. Tan, "Deepsea: Progressive workload-aware partitioning of materialized views in scalable data analytics." in *EDBT*, 2017, pp. 198–209.

[6] D. Kossmann and K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 1, pp. 43–82, 2000.