ERO Project - Self-tuning Database Operations by Assessing the Importance of Data

Additional Details

Boris Glavic

IIT DB Group Technical Report
IIT/CS-DB-2023-1

2023-08

http://www.cs.iit.edu/~dbgroup/

ERO Project Details

Additional Details

Boris Glavic

2023-08-09

In this technical report, we present our research results on relevance-based data managements. This project was partially sponsored by Oracle through a three year ERO project titled *Self-tuning Database Operations by Assessing the Importance of Data.*

1 Overview

The goal of this three-year project was the development of techniques for **self-tuning database operations based on data relevance**, i.e., assessing what data is needed for which query / workload and utilizing this information to improve various aspects of data management. Specifically, we have investigated **provenance sketches** which are coarse-grained approximation of what data is relevant for answering a given query. Provenance sketches over-estimate what data is relevant using a horizontal partitioning of input tables (this partitioning does not have to necessarily correspond to the physical data layout of the tables) - the sketch for a query Q contains all fragments that contain at least one row of relevant data. In the first two years of the project we have developed a suite of techniques for using provenance sketches, compact representations of what data is relevant for a query or workload, and to speed-up the evaluation of queries. Query answering with provenance sketches enables database systems to exploit storage indexes, zone maps, in-memory caches and other physical design artifacts for important types of queries like top-k and `HAVING` queries that were previously not supported. Specifically, we have developed techniques for creating sketches by instrumenting queries to output sketches instead of query results, reusing existing sketches for new queries (we pay the overhead of creating a sketch for a query Q1 once and then amortize this cost by using the sketch to answer future queries), automatically determining when a sketch is safe to use (as sketches over-approximate what data is needed to answer a query, the data covered by the sketch may not be sufficient for answering a query when the query is non-monotone, e.g., some queries including top-k operators or HAVING clauses are non-monotone), and have developed a cost model for sketches (this enables us to decide what horizontal partitioning to use to create a sketch as the size of sketch depends on what horizontal partitioning it is based on).

The final outcome of the third year of this project is the development of algorithm and their implementation for automated selection of provenance sketches based on their estimated size and for incremental maintenance of sketches under updates. The techniques for maintaining sketches have been implemented in an in-memory incremental maintenance engine for provenance sketches. Furthermore, we have explored the use of sketches in converged databases, demonstrating their potential benefits for semi-structured data management (JSON). The techniques developed in this project have been evaluated through extensive experiments demonstrating that significant performance improvements can be achieved by using provenance sketches including for standard benchmark queries which database systems are heavily optimized for. In summary, this project has laid the foundation for the integration of relevance-based data management techniques into commercial database engines.

1.1 Outline

The remainder of this document is structured as follows. In Section 2.3, we give a brief introduction to provenance-based data skipping. In Section 3, we discuss our techniques for maintaining provenance sketches under updates using an incremental in-memory engine specialized for maintaining sketches and present experimental results comparing this approach against our previous work that has to compute sketches from scratch when the data changes. In Section 4, we introduce techniques for estimating the size of provenance sketches statically and discuss how to utilize these techniques to make cost-based decisions for what sketches to create. In Section 5, we discuss preliminary insights into the use of provenance sketches for semi-structured data management. Finally, in Section 6 we summarize the achievements of this project. We discuss future work in 7.

2 Provenance-based Data Skipping

In this section, we give an introduction to **provenance-based data skipping** using detailed examples to motivate the advantages of using provenance as dynamic information (this information is data dependent and, thus, has to be captured at query runtime) for what data is relevant for answering a query over approaches that rely on static information (query structure and metadata such as constraints) to decide what subset of the data is relevant for answering a query.

2.1 Provenance-based Data Relevance versus Static Approaches

Consider the following example that motivates that the dynamic view on relevance provided by provenance enables new types of optimizations that are not possible with a static view that ignores the data. Assume a simple data skipping approach that excludes fragments of a partitioned table from query processing if they are guaranteed to not include data that is needed to answer the query. For instance, consider a sales table (`depName, item, numItemsSold`) partitioned on `depName`. For sake of the example assume that the each fragment stores a single `depName` value. Any reasonable database optimizer will be able to exclude fragments for a query such as the one shown below since only the fragment containing the *toys* department can contain any relevant data.

```
SELECT sum(itemsSold)
FROM sales
WHERE depName = ‘toys’
```

depName	item	numItemsSold
toys	doll	367
toys	teddy	500
toys	shovel + bucket	600
electronic	x-box	544
electronic	iphone	140
furniture	chair	800
furniture	table	230

While this approach can be effective, it is not always applicable. Namely, if it is not possible to determine from the query alone which fragments are needed to answer the query. Consider the query shown below. Note that only two departments (`toys` and `furniture`) fulfill the `HAVING` clause and, thus, contribute to the final result (`numHigh = 2`).

```
SELECT count (*) AS numHigh
FROM (SELECT depName, sum(itemsSold)
      FROM sales
      GROUP BY depName
      HAVING sum(sales) > 1000)
```

There is no way to determine statically (without taking the data into account) for which departments the `HAVING` clause will evaluate to true. However, provenance unearths this information. If we capture provenance at the granularity of fragments for this query (the provenance is a set of fragments, `{toys, furniture}` in our example), then a subsequent execution of the query can be sped up by excluding fragments that do not contain any provenance, e.g., by adding a selection condition:

```
depName = 'toys' OR depName = 'furniture'
```

resulting in the following query:

```
SELECT count (*) AS numHigh
FROM (SELECT depName, sum(itemsSold)
      FROM sales
      WHERE depName = 'toys' OR depName = 'furniture'
      GROUP BY depName
      HAVING sum(sales) > 1000)
```

Given these extra conditions any decent query optimizer will use these conditions to exclude other fragments of the partitioning on `depName`. That is, the execution of this query will not touch any data in the `furniture` fragment (the last two rows of the table). Similar issues arise if the selection conditions of a query are not aligned with the data layout, e.g.,

```
SELECT sum(itemsSold)
FROM sales
WHERE item IN (‘teddy’, ‘doll’);
```

For this example, there is no selection condition on the column `depName` on which the table is partitioned on. However, the items listed here all belong to the *toys* department. Provenance captured at the granularity of fragments would expose this. Again, we can force the database optimizer to skip partitions by adding a selection condition.

```
depName = 'toys'
```

2.2 Access-based versus Provenance-based Relevance

Many techniques for optimizing query performance such as caching and data lifecycle management require data usage to be tracked to inform lifecycle management decisions. For instance, a popular approach for data caching is to cache data that has been accessed frequently (is *"hot"*) in main memory with the hope that this will enable future queries to largely be answered from the cache. For lack of better alternatives, current approaches for tracking data usage define data usage as *"has been accessed by a query"*. Obviously, data that was not accessed to answer a query, is irrelevant for computing the query's result. This has to be true, unless the query execution engine is buggy and does not correctly implement the semantics of the query language that was used. However, by far not all data that is accessed is typically relevant for answering a query. As in the example above, we did propose to use provenance to determine which subset of the accessed data is actually relevant to answer a query.

Example 2.1. Consider the simple SQL query shown below which returns the details of all sales made by the *"electronics"* department.

```
SELECT *
FROM sales
WHERE depName = 'electronics';
```

If no index on `depName` exists or the selectivity of the query is not sufficiently low, then the database system's optimizer may decide to execute the query using a full table scan, i.e., by accessing the full sales table. However, obviously only sales from the "electronics" department have any affect on the result. Provenance information captures precisely what data is relevant for computing a query's result. For instance, applying any database provenance approach, e.g., our GProM system (`https://github.com/IITDBGroup/gprom`), we would be able to correctly identify the relevant part of the input, the set of rows with department equal to "electronics".

One may argue that the example above does not motivate the need for provenance tracking since the set of relevant data can be determined upfront by a static analysis of the query. However, as the following example demonstrates, static analysis is often not enough.

Example 2.2. Consider the SQL query shown below which exhibits non-trivial relevance relationships. This query computes the number of departments with more than 1000 sales:

```
SELECT count(*) AS numHigh
FROM (SELECT depName, SUM(itemsSold)
      FROM sales
      GROUP BY depName
      HAVING SUM(sales) > 1000)
```

The data provenance for this query only contains sales of departments with more than 1000 total sales. For such a query it is not possible to statically determine what parts of the input table will be relevant for answering the query.

As explained in the example above, data provenance captures dynamically (at query runtime and data-dependent) what data is relevant for answering a query. The critical observation here is that once we know what data is relevant for a query (is sufficient for producing the query's result), we can exploit this knowledge to filter out irrelevant data early on during query processing. However, for this to be feasible we had to overcome the following fundamental challenges:

- **Efficient Provenance Capture**: Capturing provenance for a query has to be efficient since we spend time to capture provenance for a query which has to be amortized later to improve the performance of subsequent queries. Thus, to maximize performance, the runtime overhead incurred by capturing provenance has to be low.

- **Low Storage Requirements**: We will have to materialize provenance information for a large number of queries. To be able to keep this information available in memory for, e.g., instrumenting a query to skip data based on provenance, the storage requirements of provenance have to be low.

- **Effective Use of Provenance**: The provenance representation has to be light-weight enough to enable its use for the optimizations we propose. For example, it has to be possible to determine which rows belong to the provenance to for effectively utilizing this information to skip data or determine whether a query can be answered from data that has been cached in main memory. Specifically, it should be possible for the database to utilize existing physical design to efficiently prune data that is not relevant for a query.

- **Pay once, use many times**: When capturing provenance for one query, we would like to use the provenance for as many subsequent queries as possible. Again, the goal is to maximize the benefits of captured provenance.

2.3 Provenance-based Data Skipping

In provenance-based data skipping we seek to exploit relevance information to improve data management. The fundamental idea behind provenance-based data skipping (**PBDS**) is to generate compact over-approximations of what data is relevant for a query (a so-called **provenance sketch** for the query) based on horizontal range-based partitions of an input table (that do not have to correspond to a physical partitioning of the table). A sketch created for a query Q_1 can then be used to speed up the execution of a similar query Q_2 by injecting additional `WHERE` clause conditions into the query that filter out irrelevant data (data that does not belong to the sketch).

Example 2.3. For example, evaluating the query shown below over the example table shown below returns a single result `(CA,10000)`. The provenance of the query's result are the rows highlighted in bold. Evaluating the query over the provenance $Prov(Q, D)$ is guaranteed to yield the same result as running the query over the whole table (the provenance of a query is guaranteed to be **sufficient** for answering the query). To over-approximate what subset of the table is relevant (belongs to the provenance of the query), we can (virtually) horizontally range-partition the table, e.g., on state as shown below, and record which fragments contain (some) relevant data. For the example, this is the fragment $\{f_1\}$. We refer to such an over-approximate as a **provenance sketch**. To utilize a provenance sketch to filter irrelevant data, a sketch can be translated into `WHERE` clause conditions that filter out data that does not belong to the sketch, `state = 'CA'` for this example.

```
SELECT state, sum(sales) AS rev
FROM sal
GROUP BY state
HAVING sum(sales) > 5000
```

state	city	sales	fragment
IL	Chicago	3000	
IL	Schaumburg	500	f_1
IL	Springfield	10	
CA	**Sacramento**	**2000**	
CA	**San Francisco**	**8000**	f_2
CA	Santa Cruz	1000	

We formally define provenance sketches a below.

Definition 2.1. A provenance sketch $\mathcal{P}$ for a query Q and database D contains for each table R in D accessed by Q a pair:

- a horizontal partitioning $\mathcal{F}_R$ for R
- a set of fragments: $\mathcal{P}(R) \subseteq \mathcal{F}_R$

Data contained in a sketch:

$$D_{\mathcal{P}} = \bigcup_{f \in \mathcal{P}} f$$

We require provenance sketches to over-approximate provenance:

$$Prov(Q, D) \subseteq D_{\mathcal{P}}$$

Note that the fact that provenance sketches have to cover all tables accessed by a query is not a practical requirement, but just assumed here to simplify the definition. A provenance sketch which covers a subset of the tables can be modeled by using a partitioning with a single fragment (that contains all data of the table).

2.3.1 Creating Sketches

Given a partitioning $\mathcal{F}$, of a table R and a query Q, we can generate a sketch for Q wrt. $\mathcal{F}$ through query instrumentation. That is we generate another query Q_{sketch} which returns the sketch. This is achieved by using a set of a query instrumentation rules for individual relational algebra operators that generate initial sketches as "seeds" (for table access operators) or propagate and combine input sketches according the provenance semantics for the operator (other algebra operators). The rules we have developed ensure that the sketch generated as the output over-approximates all data that is relevant for the query.

Example 2.4. For example consider the following query:

```
SELECT dept, avg(salary) AS avgsal
FROM emp
GROUP BY dept
HAVING count(*) > 10
```

To instrument this query to generate a sketch, according to a range-partitioning on attribute `salary` using ranges $\{[0, 500], [501, 800], [801, 5000], [5001, 20000]\}$, we first determine for each input row which fragment it belongs to using a `CASE` statement and create singleton sets with these fragments (encoded as bitvectors). We merge these individual sketches using bit-wise or (set union) to produce a sketch for each group. Finally, the sketches for all groups fulfilling the `HAVING` clause are merged into the final sketch for the query as shown below.

```
SELECT bit_or_agg(provsketch) AS provsketch
  FROM
    (SELECT dept, avg(salary) AS avgsal, bit_or_agg(provsketch) AS provsketch
     FROM
       (SELECT salary, dept,
               CASE WHEN salary BETWEEN 0 AND 500 THEN 1 << 0
               WHEN salary BETWEEN 501 AND 800 THEN 1 << 1
                        ...
               END AS provsketch
          FROM empl) init_sketch
     GROUP BY dept
     HAVING count(*) > 10) agg
```

2.3.2 Using Sketches to Speed-up Queries

If we know that a sketch $\mathcal{P}$ contains sufficient information for answering a query Q, then we an instrument Q to filter out data not belong to $\mathcal{P}$ early on by injecting additional `WHERE` clause conditions above a table access to filter data not belonging to the sketch early on. Given the right physical design, the DBMS may be able to utilize existing indexes, zone maps, partitioning, or other physical design artifacts to evaluate these conditions efficiently.

Example 2.5. Assume the sketch $\mathcal{P}$ produced by the query from the previous example, contains fragments $\{[0, 500], [801, 5000]\}$ and let us assume that we determined already that $\mathcal{P}$ can be used to answer the query shown below.

```
SELECT dept, avg(salary) AS avgsal
FROM emp
GROUP BY dept
HAVING count(*) > 100
```

To instrument the query to use the sketch we create `WHERE` clause conditions that filter all rows whose salary is not in one of the fragments of the sketch as shown below.

```
SELECT dept, avg(salary) AS avgsal
FROM emp
WHERE (salary BETWEEN 0 AND 500) OR (salary BETWEEN 801 AND 5000)
GROUP BY dept
HAVING count(*) > 100
```

2.3.3 Reusing Sketches

In the examples shown above we have assumed that we know whether a sketch for a query Q_1 can be used to answer a query Q_2. That is the case if the data contained in the sketch is sufficient for Q_1. In general, this not trivial to determine. As part of our work we have developed techniques for determining statically whether a sketch for query Q_1 can be used for a query Q_2 where both Q_1 and Q_2 are instances of the same query with bind parameters and have demonstrated that this technique is effective through end-to-end experiments where we evaluate a workload consisting of a large number of instances of a set of query templates and compare the runtime of executing the workload without sketches and executing the workload using sketches (no sketches are created before execution of the workload, i.e., the cost of creating sketches where needed is included in the execution time).

3 Maintenance of Sketches Under Updates

A provenance sketch is created over a particular version of a database and can be used to answer queries efficiently as long as the database does not change. Similar to materialized views, when the database is updated, then sketches may become stale and no longer correctly reflect what data is needed to answer a query. To avoid having to invalidate sketches after every update to the database, we need methods for **maintaining sketches under updates**. While *incremental view maintenance* techniques could be directly applied to this problem, because sketches are generated by queries, the approximate nature of sketches enables trade-offs between size of the sketch (and, thus, the benefits of using a sketch) and the cost of maintaining the sketch. Furthermore, having specialized techniques for data associated with sketches rather than incrementalizing the query that generates the sketch can be beneficial.

Over the course of this project we have developed such techniques and have have investigated two implementation strategies:

- **(i) SQL-based strategy**: the state required for incremental maintenance of operators which produce data annotated with sketches is persisted as database tables. Incremental maintenance is modeled as queries and updates over these tables. This approach has the advantage that data-heavy operations are executed inside the database and that there is no need to transfer large amounts of data between the client and DBMS. However, incremental operations can not always be efficiently expressed in SQL and materializing state in tables results in overhead for updating and accessing the state compared to an in-memory data structure.
- **(ii) Native in-memory strategy**: the state for incremental maintenance is kept in-memory and an in-memory execution engine is used that implements the incremental semantics of operators. This approach has the advantage that we can exploit specialized data structures that are suited best for incremental maintenance operations for data annotated with sketches. However, it requires replicating parts of the database execution engine in a middleware.

Independent of the choice of strategy also have studied how to separate incremental maintenance into two steps:

- (i) generating a table of deltas from updates that serves as input to incremental maintenance
- (ii) updating a sketch based on such a delta using incremental implementations of operators

This has the advantage that we can develop new strategies for each of these two steps without having to change the implementation of the other step. We discuss our methods for these two steps in more detail in the following.

3.1 Incremental Maintenance of Sketches

We have developed generic rules for maintaining sketches under updates, one for each type of relational algebra operator. These rules expect as input a delta that consists of a set of delta tuples which either correspond to the insertion or deletion of a tuple from an input of an operator associated with a set of fragments (a partial sketch). Intuitively, the set of fragments associated with the tuple contains exactly the fragments that belong to the provenance of this tuple. This model is powerful enough to encode changes to an (intermediate) query result as well as to encode changes to the provenance (sketch) of an (intermediate) result tuple. For instance, consider a tuple t associated with a set of fragments $\{f_1, f_2, f_3\}$. If f_3 has to be deleted from the provenance of the tuple, then this would be modeled as two delta tuples: \$-t $\rightarrow \{f_1, f_2, f_3\}$ and $+t \rightarrow \{f_1, f_2\}$, i.e., tuple t with provenance $\{f_1, f_2, f_3\}$ is deleted and tuple t with provenance $\{f_1, f_2\}$ is inserted. This can then be further optimized by allowing a delta to update the sketch for a result tuple that already is in the output of the operator before the delta was applied, e.g., in the example we could use $t \rightarrow +\{f_3\}$ to denote that f_3 is added to the sketch for t. In our incremental maintenance approach, each operator takes as input a set of such deltas and has to output a set of deltas representing the changes to its output (and the provenance of the output). Furthermore, incremental maintenance for some operators may require us to maintain additional state, e.g., for group-by aggregation we have to store the aggregation result for each group and which fragments are in the provenance of each group. The advantage of this operator-centric approach is that it allows incremental maintenance of any query that is constructed from the operators for which we have defined such rules.

3.2 In-Database Generation of Deltas

As mentioned above, the first step in incremental maintenance is to determine the delta between the database version for which a sketch was originally produced and the current database version (or the database version seen by a query for which was want to use a sketch). We have investigated different techniques for how our incremental maintenance approach can be informed about updates to compute such deltas. For the first prototype we assumed that all updates are provided as SQL statements (e.g., as provided by Oracle's audit logging functionality) each associated with a SCN. A provenance sketch is also associated with an SCN that indicates which database version is reflected in the sketch. We can maintain multiple outdated versions of a sketch for long running transactions that need an older version of a sketch. This is sensible as sketches are typically small (100s to 1000s of bytes). When we decide to use a sketch created for a query that did run at SCN x to answer a query that will see the version of the database at SCN $y > x$, then we need to make sure that the sketch reflects all updates executed between x and committed before y. Thus, we have to retrieve all relevant updates between x and y (the delta between x and y) and apply incremental maintenance to create the sketch version for SCN y. If storage is a concern, then we can garbage-collect sketches for which we have newer versions that are sufficient for serving active and future transactions. For instance, assume we have two versions of a sketch at SCN x and y with $x < y$ and the oldest currently active transaction is running under SCN $z \geq y$. In this case, we can delete the sketch for SCN x as it not visible to the current or any future transactions and if we have to bring the sketch up to date for any future transaction, then we can do this more efficiently using the version for SCN y then the one for SCN x as the one for SCN y already reflects all updates between SCN x and y.

3.2.1 Strategies for retrieving deltas

We have considered the following strategies (some of which require changes to Oracle internals and, thus, could not be implemented, but only simulated):

- **Audit log**: we use an audit log (as supported by Oracle) to retrieve all DML statements run by transactions that committed between x and y. Then we use reenactment, a technique developed in a prior ERO project for retroactively computing the provenance of transactions to generate a temporal query simulating these statements. With minor modifications to the reenactment technique, it is possible to generate reenactment queries that return the delta between x and y rather then the version of a table at y. The advantage of this approach is that it piggybacks onto existing audit logging facilities. That is, for customers that already use audit logging, this method does not incur any additional runtime overhead for query and transaction processing. However, the disadvantage is that it requires SQL code to be shipped to our system and our system to parse these SQL statements.

- **Time travel**: using time travel (e.g., Oracle's flashback data archive functionality) we can determine which row versions were created between SCN x and y and generate deltas based on this information. Alternatively, we retrieve the version of a table x as well as the version of this table at y and then compute their symmetric difference in SQL. This approach is advantageous for customers that do not use audit logging and for workloads with many SQL statements that each change only a few rows. Of course, time travel has to be available for the table to retrieve the table at SCN x. For instance, for users that do not activate flash-back archive (FBA) only versions that have not been garbage-collected yet (are still available in the Undo segment) can be retrieved.

- **In-memory columnar notification mechanism**: We have started to explore different ways of retrieving updates such as computing a diff between two database versions or using the notification mechanisms implemented in Oracle for in-memory columnar. For such an approach we can extract the differences between pages that have changed and use this to generate the delta.

3.2.2 Optimizations

We have studied several techniques for improving the performance of delta generation:

- **Merge updates**: for tables with a primary key, we only have to retain the first version of the tuple valid at y (this is the version that has to be deleted) and the latest version (valid at x, this version will be inserted). This is particularly effective if the distribution of updates is skewed towards a relatively small number of *hot* rows. This technique is theoretically also applicable for tables without PKs, but we have to decide what columns are used to identify tuples and have to deal with multiple tuples with the same identity that have different values in the same database version.

- **Focus on relevant tables**: given a provenance sketch for a query Q that accessed tables R_1, ..., R_n, any update of a table S not in this list can be ignored

- **Pushing selections into delta computation**: For sketches of queries including selections, we can use the selection condition to directly filter delta rows that do not fulfill the selection condition as these rows are guaranteed to not result in any changes to the sketch we want to maintain.

3.3 SQL-based implementation

Our first implementation of our incremental maintenance rules for sketches uses the database to store and update the state of incremental operators. This has the advantage that no data has to be transferred between GProM (our middleware) and the database. However, we are limited to algorithms for maintenance that can be efficiently implemented in SQL and cannot use specialized data structures to store operator state. This makes this approach significantly less efficient than the incremental query engine we describe next.

3.4 A Incremental In-Memory Maintenance Engine

In addition to the SQL based implementation we have implemented a prototype in-memory engine for incremental maintenance of sketches inside GProM. An operator of this engine takes as input a delta table (encoded in columnar form), a set of tuples which where deleted from (inserted into) the input table(s) of the operator, and computes a delta table for the operator's output. In contrast to traditional incremental view maintenance, these operators work on annotated relations where each tuple is associated with a (partial) provenance sketch. The result of evaluating an execution plan that consists of such operators over a delta table representing changes to the database is an updated provenance sketch. These operators maintain state that can be reused across maintenance runs and can be persisted in the database.

3.4.1 Optimizations

1. Joins The standard approach for incrementally maintaining a join requires access to the tables before the delta was applied. If ΔR denotes the delta for table R and R denotes the version before the update, then the delta of the join result $\Delta(R \bowtie S)$ can be computed as $\Delta R \bowtie S \cup R \bowtie \Delta S \cup \Delta R \bowtie \Delta S$. Note that this requires access to both R and S. We could either store both R and S in-memory, but this only works if both tables fit into memory. Alternatively, we can compute $\Delta R \bowtie S$ and $R \bowtie \Delta S$ in the database by sending ΔR and ΔS to the database and send $\Delta(R \bowtie S)$ back to the incremental engine. However, this requires the execution of additional queries in the database and results in more communication between the database and the incremental engine. To optimize this process further, we can considered to only store bloom filters for the join attributes of R and S in the incremental engine. These bloom filters can then be used for classical semi-join reducers to filter ΔR and ΔS using the bloom filters, implementing the semi-join $\Delta R \ltimes B(S)$ where $B(S)$ is the bloom filter for the join attribute of S. In the ideal case the result of such a semi-join is empty and we can avoid the round trip to the database. If the result of the semijoin is not empty then may have still reduced the size of the deltas to be send to the database.

3.5 Experimental Results

We have experimentally evaluated our in-memory engine and compared it against regenerating a sketch (recapture). We show a few selected results here.

3.5.1 Data, Setup & Workload

We synthetically generated data for the experimental evaluation. We use a table with

Boris says: TODO

rows and the following schema. The data is randomly generated using the following distribution:

Boris says: TODO

. All experiments were run on

Boris says: DESCRIBE MACHINE

. We describe the queries used in the experiments in the following:

Query `Agg1` is **TODO**

```
SELECT
```

3.5.2 Simple Aggregation Queries with Having

In the first experiment we evaluated incremental maintenance for a simple aggregation query `Agg1` varying the relative size of the input delta (fraction of the input table changed by the delta). The results are shown in Figure 1. For deltas of a size up to 6% of the data (a relatively large delta), incremental maintenance outperforms the approach we recaptures the sketch.

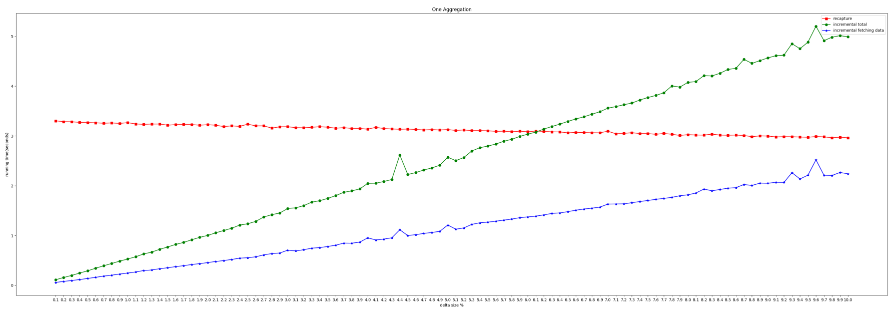

Figure 1: Incremental maintance of a sketch vs. recapture for `Agg1` varying relative size of the delta

3.5.3 Varying number of groups

We now compare the performance for two variants of `Agg1` - one returning 1k groups and the other one returning 5k groups. The results are shown in Figures 2 and 3. Performance decreases slightly in the number of groups: incremental maintenance outperforms recapture for deltas of a size up to 2.6% of the data for 1k groups and 2.2% for 5k groups.

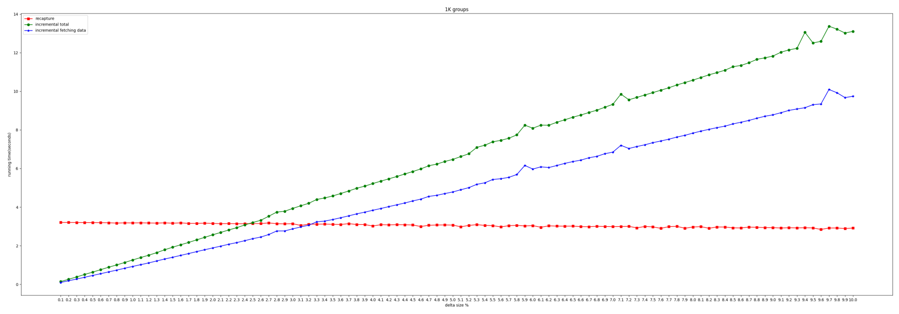

Figure 2: Incremental maintance of a sketch vs. recapture for `Agg1` varying relative size of the delta (1k groups).

3.5.4 Pushing Selections into Delta Retrieval

In this experiment we evaluate the effectiveness of pushing the selection conditions of a query into the generation of the delta. As expected, the runtime of retrieving the delta when pushing the filter decreases when the selectivity of the query decreases.

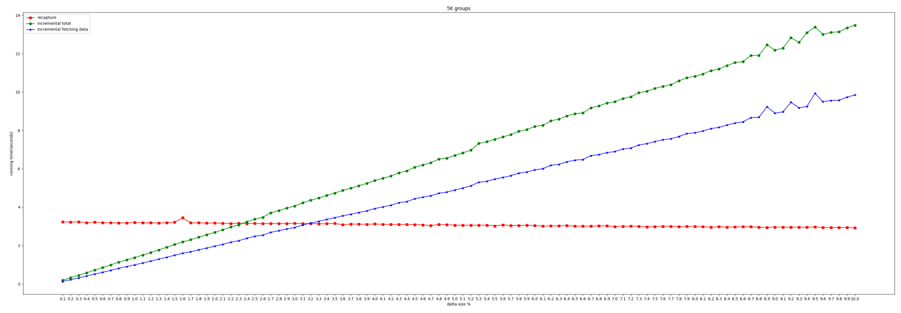

Figure 3: Incremental maintance of a sketch vs. recapture for `Agg1` varying relative size of the delta (5k groups).

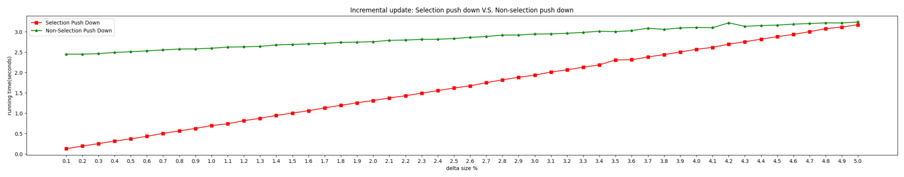

Figure 4: Pushing filters into the delta computation varying query selectivity.

4 Cost-based Selection of Provenance Sketches

Based on a preliminary experimental evaluation of our cost estimation techniques for sketches, we have identified several areas of improvement. Recall that for cost estimation we combine histograms with sampling. Specifically, for aggregations with large number of groups our approach was significantly overestimating aggregation results, because we did not estimate the total number of groups and scaled estimates based on that. Furthermore, for such aggregations our estimates of the aggregation result for a particular group will dependent on a small number (in worst case) one sample we have for the group. This leads to high variance in the estimates for aggregation values which in turn leads to poor estimation of the selectivity of HAVING conditions evaluated over such aggregation results. To address this problem we have investigated different techniques for estimating the number of tuples per group and the correlation of this metric with the group-by attribute values. Furthermore, we are currently investigating how to selectively gather more information about a sample of groups when the number of group-by values is very large (which implies that we are unlikely to have sufficient information about each group to accurately estimate aggregation function results).

We have designed several strategies for determining what sketches to consider for creation starting from simplistic strategies such as randomly selecting an attribute and always using primary key attributes over strategies that heuristically select attributes by static analyzing the query (e.g., group-by and selection attributes are often decent candidates for creating sketches) to strategies that use our cost estimation approach to select the attribute(s) which are estimated to yield the smallest sketches (possibly combined with prefiltering based on statically analyzing the query).

4.1 Estimating Sketch Size

We treat the estimation of the size of a sketch as an approximate-query-processing (AQP) problem. We have to estimate the count of fragments in the sketch.

4.2 Experimental Results

We now evaluate the effectiveness of multiple strategies for selecting sketches, some of which utilize our cost model.

4.2.1 Compared strategies

Based on our experiment experience, for the most cases, the performance of the attribute which has small distinct value number is much worse than the performance of the attribute which has large distinct value number. Thus, for our experiments, we firstly have a pre-filtering of the candidates of attribute, filtering the attribute which has fewer distinct value. Many strategies can be applied to choose the optimal attribute for provenance sketch. Different strategies include random picking from all attributes candidates after pre-filtering which is notated as **RAN-ALL**, random picking from query-relatively attributes which is notated as **RAN-REL-ALL**, random picking from group-by attributes which is notated as **RAN-GB**, random picking from primary key attributes which is notated as **RAN-PK**. random picking from aggregation attributes which is notated as **RAN-AGG**. The random picking strategy is that we choose the attributes for provenance sketches from the candidates using the uniform assumption. For example, for random picking from primary key, we uniformly random choose one attribute from all the primary key attributes for provenance sketch. The **CB-OPT-REL** strategy involves selecting the relative attributes based on the cost-based optimal approach. The **CB-OPT-GB** strategy involves selecting the group by attribute based on the cost-based optimal approach. The **CB-OPT** strategy involves selecting the best one based on the cost-based optimal approach.

4.2.2 Self-tuning with Sketches

In this set of experiments we evaluated the end-to-end performance of running a workload, generating sketches where needed and using re-using existing sketches where possible.

4.2.3 Summary

We have demonstrated that our techniques can estimate the size of for sketches using histogram, sampling, and approximate query processing technique with the accuracy needed to make informed choices about what attributes to partition on to generate more effective sketches. Furthermore, we explored whether taking static information about what attributes are "relevant" for a query (e.g., attributes used in group-by expressions) into account can improve the performance of costing by avoiding to generate cost estimates for sketches on attributes that are unlikely to yield good sketches. In end-to-end experiments we compared our cost-based approach against strategies that select sketch candidates based on heuristics and have demonstrated that the overhead of costing sketch candidates is amortized easily by selecting more effective sketches, leading to significant improvement over these baselines.

5 Hierarchical Sketches for Semi-structured Data

Master student Anjali Veer did investigate different ideas for how to create hierarchical sketches for semi-structured data and did evaluate the feasibility of using existing physical design techniques to skip data based on such sketches. The preliminary results produced by this project demonstrate that several existing techniques such as full text path indexes, expression indexes and zone maps (with modifications to skip data based on path expressions) can be used to efficiently skip data based on

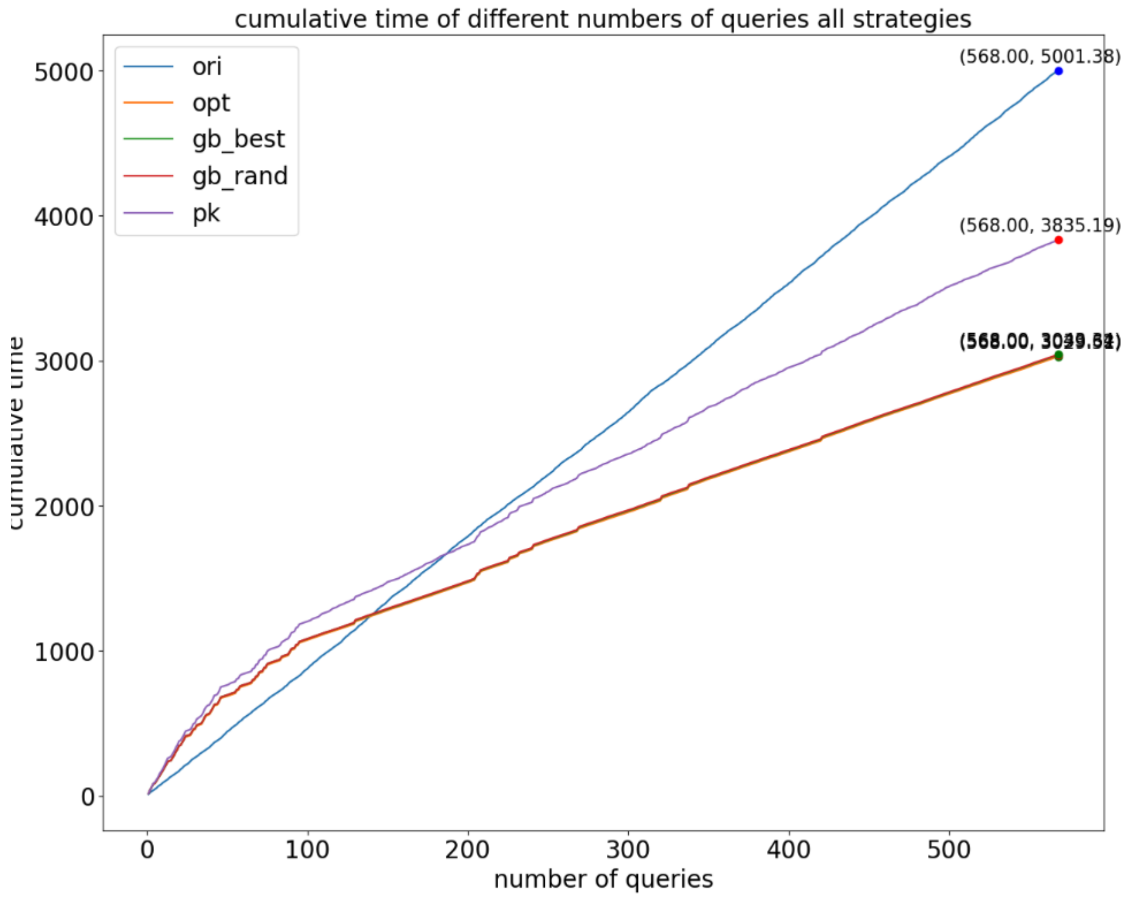

Figure 5: Cumulative runtime of existing a workload comparing all strategies for selection-aggregation queries with 2 group-by attributes on TPCH.

JSON path expressions. This demonstrate the potential of hierarchical sketches for semi-structured data.

6 Summary

In this project we have developed a suite of techniques for improving data management based on data relevance. The main contribution is the development of provenance sketches, light-weight over-approximations of what data is relevant for a query, and the use of sketches for improving the performance of queries. We have developed a full suite of techniques and have implemented them in our database middleware **GProM** (available as open-source at `https://github.com/IITDBGroup/gprom`) which is a query rewrite frontend to multiple DBMS including Oracle and PostgreSQL.

The developed techniques are ready for integration into DBMS for immediate impact on query performance. However as we will explain in 7, there are several short-term and long-term extensions to these techniques that would result in significant additional improvements of the utility of these techniques including for converged databases and vector databases.

6.1 Artifacts

- open source implementation of provenance-based data skipping including the new techniques developed in the last three years in GProM: `https://github.com/IITDBGroup/gprom`

6.2 Publications

- **Provenance-based Data Skipping** published at **PVLDB** in 2021 [NLL+21] and presented at **VLDB 2022** (`https://vldb.org/pvldb/vol15/p451-niu.pdf`)
- Xing Niu's Ph.D. thesis: **Integrating Provenance Management and Query Optimization** thesis as pdf
- **Oracle PBDS Experiments** ,Boris Glavic, Xing Niu, Pengyuan Li and Ziyu Liu, Technical Report #IIT/Cs-db-2022-01 (`http://cs.iit.edu/%7edbgroup/assets/pdfpubls/GN22.pdf`)

7 Future Work

As explained above, the developed techniques are mature enough for integration into database system. Nonetheless, there are several immediate and long-term extensions that would further increase the potential of relevance-based data management.

7.1 Converged databases

We see significant potential for the use of provenance sketches for converged databases to (i) speed up the execution of queries over semi-structured and property graph data and (ii) to make informed decisions of what data to index (only relevant data should be indexed). During this project, master student Anjali Veer has evaluated the potential of building sketches based on paths in JSON documents. However, additional research is needed to determine strategies for selecting what paths to build provenance sketches on and how to extend this idea for property graphs. This is an immediate extension of the results of this project.

7.2 Vector databases

Vector databases have seen significant adoption and interest in recent years. Using an AI model, data is mapped into an embedding space that preserves semantic similarity between data points. This enables similarity-based queries (perhaps expressed in natural language) by mapping queries into the embedding space and ranking data based on their distance to the query. As data can now be mapped into a vector space, this enables new types of provenance sketches that are based on clustering data based on their distance. This is an immediate extension of our work in this project. Furthermore, we envision that similar to, e.g., a model can learn the correspondence between images and textual descriptions, it will also be possible to learn embedding for queries and data based on the correspondence between data and queries (this data is relevant for that query).

7.3 Value of data

We envision an objective metric of the value of data based on relevance. Such a metric could be the basis for dealing with dark data (data that is not utilized) through controlled recommendations (dark data is recommended to users to learn over time which data is under-utilized and which data is not useful) and would have important applications in many aspects of data management (including automated physical design and data life-cycle management based on data value).

References

[NLL+21] Xing Niu, Ziyu Liu, Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. Provenance-based data skipping. *Proceedings of the VLDB Endowment*, 15(3):451 – 464, 2021.