Validation of SQL Queries over Streaming Warehouses

by

Ritika Jain

B.Tech Computer Science, Vellore Institute of Technology, Vellore, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia
(Vancouver)

August 2017

Abstract

There is often a need to recover the "missing" query that produced a particular output from a data stream. As an example, since a data stream is constantly evolving, the analyst may be curious about using the query from the past to evaluate it on the current state of the data stream, for further analysis. Previous research has studied the problem of reverse engineering a query that would produce a given result at a particular database state.

We study the following problem. Given a streaming database $D = \langle D_0, D_1, D_2 .. \rangle$, a result R_{out}, and a set of candidate queries $\mathscr{Q}$, efficiently find all queries $Q_i \in \mathscr{Q}$ such that for some state D_{j_i} of the stream, $Q_i(D_{j_i}) = R_{out}$, and report the pair (Q, wit_Q) where wit_Q is the *witness* of (in)validity. A witness for a valid query Q_{val} is a state D_i s.t. $Q_{val}(D_i) = R_{out}$. For an invalid query Q_{inval}, a witness is a pair of consecutive states (D_i, D_{i+1}) such that $R_{out} \setminus Q_{inval}(D_i) \neq \emptyset \neq Q_{inval}(D_{i+1}) \setminus R_{out}$.

We allow any PTIME computable monotone query to be included in $\mathscr{Q}$. While techniques developed in previous research can be used to generate the candidate query set $\mathscr{Q}$, we focus on developing a scalable strategy for quickly determining the witness. We establish theoretical worst-case performance guarantees for our proposed approach and show that it is no more than a factor of $O(\log |D_{\mathtt{RDS}}|)$ of the optimal "Lucky guess" strategy, where $Q(D_{RDS}) = R_{out}$. We empirically evaluate our technique and compare with natural baselines inspired from previous research. We show that the baselines either fail to scale or incur an inordinate amount of overhead by failing to take advantage of natural properties of a data stream. By contrast, our strategy scales effortlessly for very large data streams. Moreover, it never performs more than a small constant times the optimal amount of work, regardless of the state of the data stream that may have led to R_{out}.

Lay Summary

Data is constantly evolving and new data records are added every day, for example in a weather monitoring system. Data enthusiasts often like to ask the same question repeatedly over time to find evolving trends. Now, consider the scenario where a data analyst has queried this data at some time in the past and stored the answer that she obtained. However, due to negligence, the question that she asked is no longer available but she has access to the stored answer. In our work, we wish to help her find the "missing" question that she had asked on the data at a previous point in time. We do this using the stored answer and a set of possible questions that she could have guessed.

Preface

This thesis is submitted in partial fulfillment of the requirements for a Master of Science Degree in Computer Science. All the work presented in this dissertation are original work of the author, created in collaboration with Divesh Srivastava, Dmitri Kalashnikov, AT&T Labs Research and performed under the supervision of Prof. Laks.V.S.Lakshmanan.

Table of Contents

List of Tables

List of Figures

Glossary

BS Binary Search

DBLP Digital Bibliography & Library Project

FK foreign key

GC Geometric Chunking

LI Log Index

LS Linear Search

PK primary key

RDS Right Database State

RIC referential integrity constraint

SPJU selection-projection-join-union

SQL standardized query language

Acknowledgments

I would like to thank my supervisor Dr. Laks.V.S.Lakshmanan, whose selfless time and care were sometimes all that kept me going. I would also like to thank our collaborator, Dr. Divesh Srivastava, whose attention to detail drove me to produce nothing short of perfection.

I am grateful to all the professors I got an opportunity to interact and discuss my ideas with. In particular, I would like to thank Ivan Beschastnikh for his constant moral support and providing me with Azure credits towards running my experiments. I am also very grateful to Hu Fu for providing feedback and suggestions on my thesis.

I am extremely grateful to my collaborators and lab-mates for their useful discussions and morale boosts. I would like to thank my friends for being with me through all the ups and downs during this exciting journey at UBC.

Last but not the least, I am thankful to my family for always being there for me.

Chapter 1

Introduction

There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take lead in the introduction of a new order of things — Niccolo Machiavelli

Database management systems excel in recording dynamic data and in efficiently answering complex ad hoc queries over this data. The need for these core capabilities has only grown in recent years, with increasing availability of a large variety of transactional data feeds about different facets of the real world, and the concomitant desire by data enthusiasts to continually analyze this data to make data-driven decisions. As an example, the NYC Citi Bike data records the location, time and user of every checkout and checkin of a Citi Bike in New York city, and makes a version of this growing data set available for general use.[1] City planners may want to query this data to find the trips taken by Citi Bikers during peak commute hours on different weekdays, Citi Bike riders may want to know which bike stations are popular for checkouts and checkins in the vicinity of Grand Central station, and so on.

As new data becomes available, data enthusiasts have a tendency to issue the same queries repeatedly over time to understand the evolving nature of the data. For example, a city planner querying the NYC Citi Bike data may issue the same query every few days to understand the impact of construction in the NYC Penn station area on the trips taken during peak commute hours on different weekdays. Un-

[1] https://www.citibikenyc.com/system-data (visited on 24/08/2017).

less the data enthusiast is highly disciplined and diligently records detailed metadata with each query result, a common problem that arises is that it is easy to lose track of which query result was generated for which version of the data, making it difficult to do meaningful longitudinal analysis on this data. This is the core problem of *Query Validation* that we address in this work: *Given a database log $\mathbb{L}$ of records that are incrementally added to the database over time, a monotonic query Q and a result table R_{out}, efficiently determine if there exists a database state D_r containing all and only records in log $\mathbb{L}$ up to time r, such that $Q(D_r) = R_{out}$.*

Our core problem is relevant to many applications, including scientific reproducibility, query reverse engineering and so on. For example, Digital Bibliography & Library Project (DBLP) dataset is a popular dataset for benchmarking and reporting experiments in the social networks and database systems community. It is always evolving with time in an append-only fashion. Scientific reproducibility problem can be stated as follows: given the experimental results which were run on the DBLP data in the past, reproduce the results by identifying the relevant database snapshot e.g., Baid et al. run a known query on an unknown state of the DBLP database. In query reverse engineering, given a database log and a result table, Tran et al. try to identify a concise query and a recent database state that could have generated the result table.

1.1 Challenges

While our core problem is simply stated, and has a straightforward solution (run the query Q on every database state D_j generated by the log $\mathbb{L}$ up until time j for all $j \geq 0$, and check if $Q(D_j) = R_{out}$), it is easy to see that this solution can be extremely inefficient for big logs and complex queries. An optimal solution here would obviously need to make a "lucky guess" of the correct time r in the log $\mathbb{L}$, load all the log records up until time r in a database to get the correct database state D_r and validate that $Q(D_r) = R_{out}$. Our key result in this work is that it is possible to devise an algorithm that, without making any lucky guesses, is *close-to-optimal* in that its efficiency is within a logarithmic factor of the optimal algorithm.

What would such an algorithm look like? The "logarithmic factor" might lead the astute reader into thinking that binary search plays a role here, possibly by

first loading the entire log $\mathbb{L}$ into a database, then doing a binary search on the possible database states w.r.t. the log to identify the correct one. Note, however, that any solution that needs to first load the entire log $\mathbb{L}$ (with n records) into a database cannot be close-to-optimal if the correct database state has, say, fewer than $\log(n)$ records; in this case, just loading the entire log takes exponentially longer than the straightforward solution described above (for queries with polynomial data complexity).

1.2 Geometric Chunking

Our proposed algorithm, which we call *Geometric Chunking* , works in two phases.

1. In the first phase, it starts by loading a fixed-sized chunk of the log (in terms of the number of log records, say 1) into the database, then iteratively loading a geometrically increasing chunk size (say, doubling) of the log into the database so long as the correct database state can still be encountered subsequently: this can be checked efficiently by comparing $Q(D_r)$ and R_{out} on database states D_r at the boundaries of the loaded chunks.

2. If the database state D_s at the boundary of the most recently loaded chunk can be used to infer that the correct database state may have been crossed, the second phase is initiated. In the second phase, Geometric Chunking does a binary search on the database states *within* the most recently loaded chunk.

We prove that the efficiency of Geometric Chunking (GC) is within a logarithmic factor of the optimal algorithm (which would need to make lucky guesses).

Geometric Chunking is not only theoretically elegant, it is also practically efficient. This is achieved by making effective use of the capabilities of modern database management systems, including materialized views and view maintenance, dynamic indexes on derived tables, and stored procedures.

1.3 Contributions

In this work, our contributions range from the conceptual to the algorithmic, from the analytical to the experimental.

Our first contribution is conceptual: we formulate the technical problem of Query Validation on a database log, and argue that it is a core problem that arises in many applications.

Our second contribution is algorithmic: we devise the Geometric Chunking algorithm (described above) to elegantly and efficiently solve our problem.

Our third contribution is analytical: we build cost models for a family of candidate algorithms to solve the Query Validation problem, and prove that Geometric Chunking is within a logarithmic factor of the "lucky guess" optimal algorithm. We also show that using indexing on the log cannot improve by more than a logarithmic factor over Geometric Chunking.

Our fourth and final contribution is experimental: we carry out a large variety of experiments on benchmark TPC-H data and suitable queries to demonstrate the considerable advantages of Geometric Chunking over its competitors.

Chapter 2

Background and Problem Statement

No problem can withstand the assault of sustained thinking
— Voltaire

We consider a transaction log $\mathbb{L}$ corresponding to a streaming warehouse, where transactions consist only of append operations. One practical scenario of this setting is discussed in update scheduling in streaming data warehouses [11] where external sources push append-only data streams into the warehouse at a range of interarrival times. In general, temporal data involving tables with transaction-time support tend to be append-only as well.

Furthermore, we assume that transactions leave the database state consistent, w.r.t. applicable integrity constraints. Specifically, if there is a referential integrity constraint (RIC) from relation R to S on key attributes K, then whenever a transaction inserts a tuple t into R, either S already contains a tuple s with $s[K] = t[K]$ or such a tuple is added as part of the same transaction.[1] The transaction log associates a timestamp with every inserted tuple, reflecting the time at which the tuple was added to the warehouse. All tuples added in a transaction are assigned the same timestamp. Positions on a log $\mathbb{L}$ can be associated with corresponding

[1] In this case, K is a foreign key (FK) of R and the primary key (PK) of S.

database states: we denote the state associated with position r as D_r. We refer to states associated with consecutive integers as consecutive states.

Given a SQL query Q, a transaction log $\mathbb{L}$, and an output relation R_{out}, we say that Q is *valid* w.r.t. $\mathbb{L}$, provided there exists a position r on $\mathbb{L}$ such that $Q(D_r) = R_{out}$. If there is no such position, we say Q is *invalid*. In practice, we may encounter a set of candidate queries $\mathscr{Q}$ along with a log $\mathbb{L}$ and an output relation R_{out}, and may want to identify which of the queries in $\mathscr{Q}$ are valid (resp., invalid). In this work, we consider monotone standardized query language (SQL) queries, focusing on queries involving no aggregation. In database theory, a monotone query is defined as a query that does not lose any tuples that it previously produced [1]. Formally, *a query Q over a schema $\mathscr{D}$ is monotonic iff for every pair of instance I, J of $\mathscr{D}$, $I \subseteq J \implies Q(I) \subseteq Q(J)$.*

Furthermore, we make the following assumptions: *(i) the first state of the log is empty; (ii) the queries considered return an empty result when evaluated on an empty database; (iii) they return a superset of R_{out} when evaluated on the last state of the log.* The assumptions are natural and not restrictive. In particular, Assumption (iii) can be verified by evaluating the query on the last state of the log: notice that any query not satisfying this property cannot be valid.[2] We further assume, w.l.o.g., that the queries we consider have an output schema that is compatible with the given output relation R_{out}.

A key computational question we are interested in is determining the (in)validity of a query w.r.t. a given output relation and a transaction log. We make use of the notion of a witness as a certificate of (in)validity of a given query. If a query is valid, any state at which it exactly returns the given output relation can be regarded as a witness. If it is invalid, the witness we produce must offer a concise certificate that there is no state in the log at which the query will exactly produce the output relation. This is formalized below.

Definition 1 (Witness)**.** *Given a transaction log $\mathbb{L}$, an output relation R_{out}, and a valid query Q, a* witness *of validity of Q is a state D_r with the property that $Q(D_r) \setminus R_{out}$ and $R_{out} \setminus Q(D_r)$ are both empty. For an invalid query Q, the* witness

[2] In practice, we may of course face a workload not satisfying this assumption. In Section 5, we evaluate our algorithms on such workloads.

```
    SELECT A, B                    SELECT A, B
Q1: FROM   R               Q2: FROM   R
    WHERE  C=1 and B>=2            WHERE  C=1 and B>=3
```

R

RID	A	B	C
t_1	1	2	1
t_2	1	3	1
t_3	2	3	2
t_4	2	3	1

R_{out}

A	B
1	3
2	3

Figure 2.1: Example database state and queries.

of invalidity consists of a pair of consecutive states, say D_r and D_{r+1}, such that $R_{out} \setminus Q(D_r)$ is non-empty and $Q(D_{r+1}) \setminus R_{out}$ is non-empty.

Notice that the second condition occurs when we have two adjacent states, r and $r+1$, such that R_{out} is a subset of $Q(D_r)$ and $Q(D_{r+1})$ has additional tuples not in R_{out} (in particular, $Q(D_{r+1})$ is a superset of R_{out}).

The definition of witness for a valid query is equivalent to the condition $Q(D_r) = R_{out}$. In this case, the state D_r is referred to as a Right Database State (RDS). Notice that for a valid query, there may be multiple right database states. From the monotonicity of Q and the append-only nature of the warehouse, it follows that whenever D_r and D_s are RDS's for a query Q, every state in between D_r and D_s must be an RDS for Q too. In other words, the set of RDS's for a query form an interval over timestamps.

For an invalid query, by definition, there is no state at which Q returns exactly the output relation R_{out}. Since the warehouse is append-only, and Q is monotone, it follows that there must be a pair of consecutive states D_r and D_{r+1} such that $Q(D_r)$ misses some tuples in R_{out} and $Q(D_{r+1})$ contains some spurious tuples not present in R_{out}. This captures the following "check-mate" intuition: no state to the past of D_r could lead to the result R_{out}, since on those states, Q will produce a subset of $Q(D_r)$; and no state to the future of D_{r+1} helps either, since Q will produce a superset of $Q(D_{r+1})$ on those states. In view of our assumption above, the witness of an invalid query is well defined.

The next example illustrates witness states.

Example 1 (Witness States)**.** Consider the database consisting of one relation R and the queries `Q1` and `Q2` shown in Figure 2.1. Consider two different transaction logs. In the first log, $\mathbb{L}_1$, tuples are added into R in the order shown, one at a time: $R_0 = \emptyset, R_1 = \{t_1\}, ..., R_4 = \{t_1, ..., t_4\}$. In the second log, $\mathbb{L}_2$, $R_0 = \emptyset$, and tuples t_2 and t_3 are added in timestamp 1, and tuples t_1 and t_4 are added in timestamp 2, i.e., $R_1 = \{t_2, t_3\}$, and $R_2 = \{t_1, t_4\}$.[3]

First, consider the log $\mathbb{L}_1$. On this log, $\texttt{Q2}(R_4) = R_{out}$ and R_4 is the only state at which this is true. Clearly, it is a witness to the validity of `Q2`. Now, consider `Q1`. Notice that $\texttt{Q1}(R_0) = \emptyset$ and $\texttt{Q1}(R_1) = \{(1,2)\}$ and thus, $R_{out} \setminus \texttt{Q1}(R_0) \neq \emptyset \neq \texttt{Q1}(R_1) \setminus R_{out}$. Therefore, (R_0, R_1) provides the witness for invalidity of $Q1$ for this log.

Next, consider the log $\mathbb{L}_2$. On this log, $\texttt{Q2}(R_2) = R_{out}$ and R_2 is the only state at which this is true. Clearly, it is a witness to the validity of `Q2`. Now, consider `Q1`. Notice that $\texttt{Q1}(R_1) = \{(1,3)\}$ and $\texttt{Q1}(R_2) = \{(1,2),(1,3),(2,3)\}$. Since $R_{out} - \texttt{Q1}(R_1) \neq \emptyset \neq \texttt{Q1}(R_2) - R_{out}$, the pair of states (R_1, R_2) from the log $\mathbb{L}_2$ forms the witness for the invalidity of `Q1`. □

The formal statement of the problem studied is as follows.

Problem 1 (Problem Studied)**.** *Given a transaction log $\mathbb{L}$, an output relation, and a workload consisting of a set of queries $\mathcal{Q}$, find a witness for every query in $\mathcal{Q}$, i.e., for every valid query $Q \in \mathcal{Q}$, find a state D_r such that $Q(D_r) = R_{out}$ and for every invalid query $Q \in \mathcal{Q}$, find a pair of states D_r, D_{r+1}, such that $R_{out} \setminus Q(D_r) \neq \emptyset \neq Q(D_{r+1}) \setminus R_{out}$.*

[3]For simplicity of notation, we use R_i to denote the database state at position i in either log $\mathbb{L}_1$ or $\mathbb{L}_2$. The intended log should be clear from the context.

Chapter 3

Solution Strategies

The solution often turns out more beautiful than the problem
— Richard Dawkins

In this section, we present strategies to solve **Problem 1**. We will first consider and explain some baseline strategies inspired from previous work and then suggest improvements in Section 3.2 where we discuss Static Chunking Strategy. Next, we discuss our proposed approach Geometric Chunking in Section 3.3. In Section 3.4, we consider the use of a log index for further speeding up geometric chunking.

The intuition behind our approach is the following. If a query Q evaluated on the database state D_r at timestamp r produces a proper subset of R_{out} (i.e. $R_{out} \setminus Q(D_r) \neq \emptyset$ and $Q(D_r) \setminus R_{out} = \emptyset$), we know that we should move forward in the log[1]. Similarly, if R_{out} is a proper subset of $Q(D_r)$ (i.e. $Q(D_r) \setminus R_{out} \neq \emptyset$ and $R_{out} \setminus Q(D_r) = \emptyset$), we need to evaluate Q on D for timestamps $< r$. However, if both conditions are satisfied i.e, $R_{out} \setminus Q(D_r) \neq \emptyset$ and $Q(D_{r+1}) \setminus R_{out} \neq \emptyset$, then we can provide (D_r, D_{r+1}) as the witness for invalidity. [2] In fact, $Q(D_r) \setminus R_{out} \neq \emptyset \neq R_{out} \setminus Q(D_r)$ is sufficient to provide the witness (D_r, D_{r+1}). This is because the monotonicity argument proves that Q will not be able to produce exactly R_{out} even for timestamps $\tau > r$ since $Q(D_\tau) \setminus R_{out}$ will never become non-empty.

[1] Since there are some additional tuples in R_{out} which are missing from $Q(D_r)$. Monotonicity argument dictates that we move forward in the log.

[2] This provides a witness of invalidity since for states prior to D_r, we will only get subsets of R_{out} and for states D_τ beyond D_{r+1}, $Q(D_\tau)$ will always contain spurious tuples not present in R_{out} (because $Q(D_\tau) \setminus R_{out}$ will never be non-empty).

From here on, evaluating $R_{out} \setminus Q(D_r)$ and $Q(D_r) \setminus R_{out}$ will be collectively referred to as a **probe** on database state D_r.

3.1 Baseline Strategies

To the best of our knowledge, there is only one approach previously discussed in the literature by Tran et al. that is relevant to our problem. We refer to it as the *Backward Naïve strategy*. Originally, this has been implemented to solve a more general problem where updates involving both additions and deletions to the log are considered. Hence, this is unable to take advantage of monotonicity and cannot "skip" ahead. It is forced to progress through each database state sequentially, probing on each time stamp until the Right Database State (RDS) is found.

3.1.1 Backward Naïve strategy

Given a sequence of database states $D_1, D_2, ..., D_{last-1}, D_{last}$, we start with the most recent state i.e. D_{last} and do a backward linear scan, probing on each database state until an RDS is found or the query is found to be invalid. We load the entire data up to the most recent timestamp *last* and probe on D_{last}. If $Q(D_{last}) \subset R_{out}$, we see that assumption (iii) in Section 2 is violated, making Q invalid. Suppose $Q(D_{last}) \supseteq R_{out}$. If we encounter a state D_r such that $Q(D_r) = R_{out}$, we return D_r as a witness of validity. Otherwise, let D_r be the first state (going backward) at which $R_{out} \setminus Q(D_r)$ is non-empty. Then we can return (D_r, D_{r+1}) as a witness, since by construction, $Q(D_{r+1}) \setminus R_{out}$ is non-empty.

Notice we can optimize this by maintaining a view on Q. This reuses the computation done for query evaluation on the previous database state instead of computing the query from scratch each time.

3.1.2 Forward Naïve strategy

Here, we do a *forward* linear scan one database state at a time, starting from the beginning of the log, i.e., state D_1. Let D_r be the current state. If $R_{out} \setminus Q(D_r)$ is non-empty, we keep probing the next state until one of the following happens:

(i) $Q(D_r) = R_{out}$ — in this case, we return D_r as a witness of validity of Q;

(ii) $R_{out} \setminus Q(D_r)$ *as well as* $Q(D_r) \setminus R_{out}$ *is non-empty* — in this case, as explained before, we return (D_r, D_{r+1}) as a witness of invalidity of Q;

(iii) $R_{out} \setminus Q(D_r)$ *is non-empty and* $Q(D_r) \setminus R_{out}$ *is empty but* $Q(D_{r+1}) \setminus R_{out}$ *is non-empty* — again, we return (D_r, D_{r+1}) as a witness of invalidity of Q.

Notice, that in forward naïve in general, we do not have to load the entire database log. Rather, we load one state at a time as we progress forward. Here, loading a database state means loading the additional tuple corresponding to the transaction occurring after the previous state was loaded.

3.2 Static Chunking Strategy

Probing on every database state is wasteful and computationally expensive. We can reduce the number of probes we perform by carefully choosing the next point to probe in the log. The idea is to pick the next point in the log based on the previous comparison between R_{out} and $Q(D_r)$. Suppose at timestamp r, R_{out} is a proper superset of $Q(D_r)$, then from our assumption of monotonicity (discussed in Section 2) we know that the possibility of finding an `RDS` only occurs at timestamps $> r$. Instead of sequentially probing every timestamp $> r$, we can "skip" ahead in the log.

For this purpose, we will partition the log $\mathbb{L}$ into chunks. We create chunks on timestamp, such that the new tuples being added to the data warehouse only touch the new chunks. Let the size of our chunk be B and D_r be the most recently probed state. Next, we load a chunk of size B and probe at the end of the chunk i.e., at D_{r+B}. Now, if R_{out} is a proper subset of $Q(D_{r+B})$, we know that the possibility of finding the `RDS` is in between r and $r+B$, and we can perform Binary Search on this chunk (as outlined in Procedure BinarySearch). However, if at timestamp, $r+B$, R_{out} is a superset of $Q(D_{r+B})$, we know that we should keep progressing forward in the log, and we choose our next probe at $r+2*B$ and so on. The advantage here is that we save on the expensive probes at every timestamp between r and $r+B$. We further optimize this strategy by materializing a view for Q as discussed in [7]. Instead of computing $Q(D_{r+B})$ from scratch, we incrementally maintain it from the previous computation $Q(D_r)$.

Procedure Probe defines performing a probe on the state D_r. We evaluate two conditions in the probe:

1. $Q(D_r) \setminus R_{out}$
2. $R_{out} \setminus Q(D_r)$

This leads to the following four possibilities-

- Both 1 and 2 are empty: it is a valid query since $Q(D_r) = R_{out}$.
- Both 1 and 2 are non-empty: it is an invalid query as explained previously.
- 1 is empty and 2 is non-empty: proceed forward in the log since R_{out} has extra tuples not present in $Q(D_r)$.
- 1 is non-empty and 2 is empty: perform a binary search in the current partition since R_{out} is a subset of $Q(D_r)$.

We provide the pseudo-code for our approach in Algorithm 1. Given a workload of queries $\mathcal{Q}$, we maintain a view for each query $Q \in \mathcal{Q}$. We load a chunk once, probe at the end of the chunk (D_{end}) for all the candidate queries in $\mathcal{Q}$. If the Probe function returns "VALID", we return the query Q and witness wit_Q which is a single state, say D_r. If the Probe function returns "INVALID", we return $(Q, \langle D_r, D_{r+1} \rangle)$ where Q is the invalid query and the witness wit_Q is a pair of states. If the probe returns "FORWARD", we know that possible `RDS` lies in the future, and we load another chunk. Similarly, if probe return "BACKWARD", we know that the possible `RDS` lies somewhere in the current chunk and we perform a binary search on the current chunk defined by (start,end). The views for queries that receive a witness of (in)validity are dropped and they are removed from $\mathcal{Q}$. The remaining queries proceed to the next chunk.

Please note that the pseudo-code is common to both Static Chunking and Geometric Chunking strategy. The difference lies in the way the chunking is performed which is captured by the set of intervals *CHUNKS* defined in Equation 3.1 for Static Chunking and Equation 3.2 for *GC*.

For static chunking strategy, each chunk is of the same size. Let B be the size of the chunk, then $CHUNKS$ will be the set of intervals:

$$\begin{aligned} CHUNKS[\,][2] = \{\ & [1,B]\ ,\ [B+1,2*B]\ ,\ [2*B+1,3*B]\ ,\ [3*B+1,4*B]\ ,\ \ldots \\ & \ldots\ [(k-1)*B+1,k*B]\ \} \end{aligned} \tag{3.1}$$

where $k*B$ is greater than or equal to the last time stamp of the log. Clearly, static chunking requires $\lceil \texttt{RDS}/B \rceil$ number of chunk loads to search for the RDS and $\log_2(B)$ number of binary probes in the worst case.

Loading a chunk is captured in the Procedure LoadNextChunk. Each chunk is a row in the matrix $CHUNKS[\,][2]$ (defined in Equation 3.1). The first timestamp of the chunk is the first element of the row. Similarly, end of the chunk is the second element of the row. These are called *start* and *end* respectively in the pseudo-code.

Procedure BinarySearch explains the binary search that takes place within a chunk. *low* is the beginning of the chunk and *high* is the last time stamp of the chunk. Recall that we already probed at *high* as part of the end of the chunk probe. We now probe on *mid* of the chunk. If $Q(D_{mid})$ is a proper subset of R_{out}, then we perform the next binary search in the right half, i.e. between $mid+1$ and *high*. Else if $Q(D_{mid})$ is a proper superset of R_{out}, then we perform the next binary search in the left half, i.e. between *low* and $mid-1$. If both $Q(D_{mid}) \setminus R_{out}$ and $R_{out} \setminus Q(D_{mid})$ become non-empty, we can provide a witness of invalidity for query Q i.e., $wit_Q = (\langle D_{mid}, D_{mid+1} \rangle)$. Note, if we reach two adjacent states D_{low}, D_{high} (where $low+1 = high$) such that $Q(D_{low})$ is a proper subset of R_{out} and $Q(D_{high})$ contains a tuple not in R_{out}; then the witness of invalidity is the pair of states (D_{low}, D_{high}).

Algorithm 1: Chunking Strategy

Input: Log $\mathbb{L}$, result table R_{out}, workload $\mathscr{Q}$

Output: (Q, wit_Q)

Initialization: Init()

```
1  if LoadChunk = true then
2      LoadNextChunk(ch)
3      LoadChunk ← false
4  else
5      foreach Q ∈ 𝒬 do
6          if Probe(Q, R_out, D_end) = "VALID" then
7              drop view for Q and remove Q from 𝒬
8              return (Q, ⟨D_end, D_end+1⟩)
9          else if Probe(Q, R_out, D_end) = "INVALID" then
10             drop view for Q and remove Q from 𝒬
11             return (Q, ⟨D_end, D_end+1⟩)
12         else if Probe(Q, R_out, D_end) = "FORWARD" then
13             LoadChunk ← true
14             ch ← ch + 1                          ▷ Loads next chunk
15         else
16             BinarySearch(start,end)      ▷ Perform Binary Search
                                  ▷ within the current partition
```

Procedure Init()

```
1  Create empty tables
2  Initialize CHUNKS                  ▷ as per Equation 3.1 or 3.2
3  LoadChunk ← false
4  ch ← 0
5  LoadNextChunk(ch)
```

Procedure LoadNextChunk(ch)

start ← CHUNKS[ch][0] ▷ start is first timestamp ▷ of the chunk
end ← CHUNKS[ch][1] ▷ end is the last timestamp ▷ of the chunk
Load tuples between start and end for all relations

Procedure Probe(Q, R_{out}, p)

if $Q(p) \setminus R_{out} = \emptyset$ *AND* $R_{out} \setminus Q(p) = \emptyset$ **then**
    return "VALID"
else if $Q(p) \setminus R_{out} \neq \emptyset$ *AND* $R_{out} \setminus Q(p) \neq \emptyset$ **then**
    return "INVALID"
else if $Q(p) \setminus R_{out} = \emptyset$ *AND* $R_{out} \setminus Q(p) \neq \emptyset$ **then**
    return "FORWARD"
else
    return "BACKWARD"

Procedure BinarySearch(low,high)

while *low* $\leq$ *high* **do**
    if *high* = *low+1* **then**
        drop view for Q and remove Q from $\mathcal{Q}$
        return (Q,$\langle D_{low}, D_{high} \rangle$)
        break
    else
        mid ← (low+high)/2
        if *Probe*(Q, R_{out}, D_{mid}) = *"VALID"* **then**
            drop view for Q and remove Q from $\mathcal{Q}$
            return (Q,$\langle D_{mid}, D_{mid+1} \rangle$)
            break
        if *Probe*(Q, R_{out}, D_{mid}) = *"INVALID"* **then**
            drop view for Q and remove Q from $\mathcal{Q}$
            return (Q,$\langle D_{mid}, D_{mid+1} \rangle$)
            break
        else if *Probe*(Q, R_{out}, D_{mid}) = *"FORWARD"* **then**
            low ← mid+1
        else
            high ← mid-1

3.3 Geometric Chunking Strategy

In Geometric Chunking strategy, we geometrically increase each chunk size by a constant factor c.

Let B be the size for the first chunk (referred to as the base chunk size). We assume the default value of $c = 2$. [3] Then $CHUNKS$ is the set of intervals:

$$\begin{aligned} CHUNKS\,[\,][2] = \{\ & [1,B]\ ,\ [B+1, B+2*B]\ ,\ [B+2*B+1, B+2*B+4*B]\ ,.. \\ & ...\ [(2^{k-1}-1)*B+1, (2^k-1)*B]\ \} \end{aligned} \tag{3.2}$$

where $(2^k-1)*B$ is greater than or equal to the last time stamp of the log.

Please note that the procedure to provide witness of (in)validity is the same as before. The only difference is that the in *GC*, the chunk size grows geometrically allowing us to quickly reach within the proximity of the `RDS`. Let $\lceil \texttt{RDS}/B \rceil$ be m and $[2^{k-1}*B-B+1, 2^k*B-B]$ be the chunk we perform binary search over. Then *GC* requires $\log_2(m)$ number of chunk loads to search for the `RDS`. Once we load the chunk containing the possible `RDS`, a binary search (described in Procedure BinarySearch) is performed as before. Here, the chunk size over which we perform Binary search is $2^{k-1}*B$ and therefore, *GC* will perform $(k-1)+log_2(B)$ number of binary probes[4] in the worst case.

3.4 Log Index

To speed up geometric chunking further, we could construct a *log index*. The idea is that for value v of each attribute A of each relation R in the database, we store the earliest timestamp τ_v^A at which any tuple with $A = v$ was inserted into R according to the log. Suppose query Q involves relation R and $R.A$ is one of the projected

[3] We have experiments exploring other values of c as well.

[4] $log_2(2^{k-1}*B) = (k-1)+log_2(B)$

attributes of Q. If R_{out} contains a tuple t with $t.A = v$, then obviously $Q(D_r) \neq R_{out}$ for all $r < \tau_v^A$. We can generalize this intuition for a conjunction of attributes: e.g., for a set of attributes $X \subseteq \text{schema}(R)$, it is easy to see that

$$Q(D_r) \neq R_{out}, \forall r < \max_{A \in X} \tau_v^A.$$

Finally, we can also show that

$$Q(D_r) \neq R_{out}, \forall r < \max_{t \in R_{out}} \max_{A \in X} \tau_{t[A]}^A,$$

where $\tau_{t[A]}^A$ denotes the earliest timestamp at which any tuple with $A = t[A]$ was added into relation R according to the transaction log. Notice that another max can be applied over all relations in Q. More precisely, for each tuple in R_{out} we can find the latest timestamp when a contributing tuple in any relation R participating in Q was added to the database. We can then take max of this quantity over all tuples in R_{out}.

$$Q(D_r) \neq R_{out}, \forall r < \max_{t \in R_{out}} \max_{R \in Q} \max_{A \in X} \tau_{t[A]}^A,$$

where $\tau_{t[A]}^A$ denotes the earliest timestamp at which any tuple with $A = t[A]$ was added into relation R according to the transaction log.

Of course, one could build multidimensional indices on sets of attributes and exploit them to prune probes on early states of the transaction log.

Chapter 4

Technical Results

I've always believed that if you put in the work, the results will come
— Michael Jordan

In this section, we establish our main technical results. Let $\mathbb{L}$ be a transaction log. Recall that for a valid query Q, an RDS corresponds to a log position r such that $Q(D_r) = R_{out}$. As seen in Section 2, a valid query may have multiple Right Database States. Without ambiguity, we denote any RDS as $D_{\texttt{RDS}}$.

4.1 Assumptions and Cost Model

In this section, we assume that the available memory is adequate for probing any one database state. This means not only that the database state D being probed and R_{out} fit in available memory, but also any intermediate results of evaluating $Q(D)$ fit in memory. Thus, the cost model we employ is in terms of the main memory model. In our experiments (Section 5), we relax this assumption and compare the different strategies under limited memory. In this section, we consider polynomial time computable selection-projection-join-union (SPJU) queries.

4.2 Basic Operations

We establish some terminology to help compare different strategies for query validation. A strategy needs to load some database state, either in whole, or a timestamp at a time, or a chunk at a time, and needs to evaluate the query being val-

idated, on the loaded state. Once the query is evaluated, the result needs to be compared with R_{out}. Here, we refer to query evaluation together with this comparison as a *probe*. Both static and geometric chunking strategies consist of a Linear Search (LS) phase and a Binary Search (BS) phase, each potentially consisting of several probes. During LS, whenever a probe is found to result in a proper subset of R_{out}, the next chunk is loaded and probed. If a probe on a state D_r results in a proper superset of R_{out}, the strategy proceeds to the BS phase, to search if there is a previous state D_s, $s < r$, at which $Q(D_s) = R_{out}$. Clearly, the binary search involves a logarithmic number of probes in the size of D_r. Notice that the cost of each probe during BS is upper bounded by that of D_r. Finally, an orthogonal point is that any of the strategies may employ materialized views to speed up the probes.

4.3 The Oracle

We propose a reference oracle strategy for the sake of comparison and calibration of various algorithms. Given a valid query Q, the oracle knows the position RDS of a right database state in the log, loads the data for $D_{\texttt{RDS}}$ corresponding to this RDS, probes it with the query Q, compares the result with R_{out}, and produces the certificate that $Q(D_{\texttt{RDS}}) - R_{out}$ and $R_{out} - Q(D_{\texttt{RDS}})$ are empty, thus validating Q at $D_{\texttt{RDS}}$. Notice that the oracle simply loads one chunk of data and then evaluates the query. It does not involve any binary search. It thus acts as a bar for the most efficient possible strategy among those considered.

4.4 Overview of Results

We consider two different settings for query validation: (i) when no materialized views are used and (ii) when materialized views are used and are maintained (incrementally) by the underlying DBMS. For both settings, under certain assumptions about the cost model which we will make precise shortly, we establish our results on the cost of the static and geometric chunking strategies. Specifically, we show that the geometric chunking strategy proposed incurs a cost that is no more than a factor $O(\log |D_{\texttt{RDS}}|)$ times that of the oracle. By contrast, we show that the static chunking strategy can incur linearly more cost than the oracle.

Finally, we consider a possible additional optimization for query validation on top of geometric chunking. We can create a *log index* which keeps track of the earliest timestamp when a tuple with certain attribute-value is added into each relation. Such an index can be used to avoid wasteful probes. We dicsuss the utility of such a log index and show that the cost saving achieved by a log index over a geometric chunking strategy without such an index is bounded by a similar logarithmic factor.

Unless otherwise mentioned, all logarithms are base 2. While GC permits expanding chunk size using any factor $c \geq 1$, w.l.o.g., we assume the default value of $c = 2$. Other values of c are explored in our experiments. Notice *GC* with $c = 1$ is equivalent to static chunking strategy.

4.5 No Views

In this section, we consider a query evaluation model that does not make use of materialized views. This means that whenever a query needs to be evaluated on a different database state in the log, the query will be evaluated on that state from scratch. We start with a simple observation.

Observation 1. *Let Q be a monotone SQL query and let D_r, D_s be any two database states from the log such that $r \leq s$. Then under any plan p chosen by the query optimizer, the cost of evaluating Q on D_r under p is no more than that of evaluating Q on D_s under p.*

Proof. Since we consider a streaming warehouse setting, our logs are append-only and thus $D_r \subseteq D_s$. Since Q is monotone, we have $Q(D_r) \subseteq Q(D_s)$. It follows that under any given plan for Q, obtaining the result $Q(D_s)$ would involve at least as much work as obtaining the result $Q(D_r)$. □

4.5.1 Geometric Chunking

One of the assumptions we make in this section is that the query optimizer uses a consistent plan for evaluating queries regardless of the size of the database state. For a database state D, we let $|D|$ denote its size in number of tuples.

Theorem 1. *Let Q be any valid monotone SQL query that is computable in PTIME and let $D_{\texttt{RDS}}$ be the database state corresponding to a RDS w.r.t. query Q. Then the cost of the geometric chunking strategy for validating Q using the log is at most $O(\log|D_{\texttt{RDS}}|)$ times the cost of the oracle validating Q at the database state $D_{\texttt{RDS}}$.*

Proof. Suppose on any state D_r, $Q(D_r)$ can be evaluated in time $O(|D_r|^k)$, where k is some constant. This is the cost incurred by the oracle for validating Q w.r.t. D_r. It is obvious that the geometric chunking strategy performs a logarithmic number of probes, i.e., $\lceil\log|D_r|\rceil$ probes during the LS phase. In the worst case, the $(\lceil\log|D_r|\rceil-1)^{th}$ probe could have just "missed" the correct position of the RDS $D_{\texttt{RDS}}$, with the result that the $\lceil\log|D_r|\rceil^{th}$ probe could correspond to a database state D_s such that $|D_s|\leq 2|D_{\texttt{RDS}}|$. Clearly, the cost of the last probe dominates the cost of all preceding probes, by virtue of Observation 1. Next, BS may involve a number of additional probes equal to $\log|D_s| = O(\log|D_r|)$. The cost of every probe is upper bounded by $O(|D_s|^k) = O(2^k|D_r|^k) = O(|D_r|^k)$, since k is a constant. Putting these together, it follows that the cost of the geometric chunking strategy validating Q using the transaction log is at most $O(\lceil\log|D_r|\rceil|D_{\texttt{RDS}}|^k)$. The theorem follows. □

4.5.2 Log Index

We next establish an upper bound on the savings that can be achieved using *any* log index, no matter how complex or sophisticated.

Let Q be a valid query. Recall our previous discussion on Log Index in Section 3.4. For any tuple in R_{out}, we take the maximum timestamp when a contributing tuple from a relation R participating in Q was added to the log. We can then take a maximum over all the tuples in R_{out}. This quantity is denoted as $LI(R_{out},Q)$. This is also equivalent to the position in the transaction log for which we can prune probes on all states prior to $LI(R_{out},Q)$. We now have the following:

Theorem 2. *The cost of geometic chunking strategy without log index is at most $O(\log|D_{LI(R_{out},Q)}|)$ times that of geometric chunking with log index.*

Proof. The argument is analogous to that of Theorem 1. The key idea is to realize that GC with log index will start at the state $D_{LI(R_{out},Q)}$ while GC without log index

will start from the initial state. Thus, the saving of the former compared to the latter comes from avoiding the $\log\lceil D_{LI(R_{out},Q)}\rceil$ probes during the LS phase of GC without log index. After this, both GC without log index and GC with log index behave similarly. It is clear that GC without log index has a cost that is $O(\log|D_{LI(R_{out},Q)}|)$ times that of GC with log index. □

Notice that in practice, depending on the selectivity of the log index, $LI(R_{out},Q)$ may be much smaller than RDS. Its construction and maintenance involve additional overhead.

4.5.3 Static Chunking

Next, we consider the cost of the static chunking strategy over the oracle. Let Q be any valid monotone SQL query, let $D_{\texttt{RDS}}$ be the database state corresponding to a RDS w.r.t. query Q. Then the cost of the static chunking strategy for validating Q using the transaction log can be $O(|D_{\texttt{RDS}}|)$ times the cost of the oracle validating Q at the database state $D_{\texttt{RDS}}$, as we show below.

Suppose $D_{\texttt{RDS}}$ contains n tuples and that evaluating $Q(D_{\texttt{RDS}})$ takes $O(n^k)$ time. Let B be the size of a (static) chunk. Suppose B is a constant and let $m = \lceil n/B\rceil$. Let D_s be the last (and largest) database state probed during the LS phase. Since B is a constant, $|D_s| = O(|D_{\texttt{RDS}}|)$. During the BS phase, an additional number of $O(\log|D_{\texttt{RDS}}|)$ probes are performed. The total cost incurred during the LS phase of static chunking is $O(|B|^k + (2|B|)^k + \cdots + (m|B|)^k) = O(|B|^k \sum_{i=1}^{m} i^k) = O(|B|^k m^{k+1}) = O(n^{k+1})$, since B is a constant. The second equality follows from Faulhaber's formula [16]. Ignoring the additional cost of BS, the overall cost of static chunking is $O(n)$ times that of oracle. Consider an instance of the problem such that evaluating Q on $D_{\texttt{RDS}}$ takes n^k time. The analysis above is tight in the sense that on this instance, the cost of static chunking can take up to $O(n^{k+1}/B)$ times that of the oracle.

4.6 With Views

In this subsection, we consider the setting where query evaluation at different database states leverages incremental view maintenance strategies [7]. More pre-

cisely, suppose a query Q is evaluated at a state D_r and the result $Q(D_r)$ is cached. Suppose it is now required to be evaluated at state D_s, where $r \leq s$. Then $Q(D_s)$ can be obtained from $Q(D_r)$ by incrementally evaluating $Q(D_s)$, given $Q(D_r)$ and the "delta" or change between D_s and D_r [17]. Notice that since the log is append-only, the incremental query evaluation only consists of additions and no deletions. Recall that the log associates a timestamp with every tuple, viz., the time at which the tuple was inserted into the warehouse. The algorithms we develop in this work take advantage of certain stylized views, which keep track of the timestamp of every tuple that was used in deriving an output tuple of the view. We illustrate this with an example.

Example 2 (Time aware views)**.** Consider the following query.

```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE,
       O_TOTALPRICE
FROM   CUSTOMER, ORDERS, LINEITEM
WHERE  O_ORDERKEY = L_ORDERKEY AND
       C_CUSTKEY = O_CUSTKEY AND C_NATIONKEY = 3
```

Suppose that for each relation R, R.TS denotes the timestamp attribute, that indicates the time at which a particular tuple was added to relation R in the warehouse. Then the following modified query can be used to keep track of the times of the participating tuples used in deriving each answer tuple in the query (view).

```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE,
       O_TOTALPRICE, C.TS, O.TS, L.TS
FROM   CUSTOMER C, ORDERS O, LINEITEM L
WHERE  O_ORDERKEY = L_ORDERKEY AND
       C_CUSTKEY = O_CUSTKEY AND C_NATIONKEY = 3
```

For a query Q, we denote by Q_{TS} the rewritten version of Q that keeps track of the times of participating tuples, as illustrated above. We refer to Q_{TS} as the time-aware version of Q. In this work, by views, we mean the time-aware views illustrated above, unless otherwise specified. We make the following assumptions.

Consider a database state D_r and a state D_s obtained by appending a set of tuples to D_r, i.e., $D_s = D_r \cup \Delta D$. The naive approach to query evaluation is to

evaluate each of the queries $Q(D_r)$ and $Q(D_s)$ independently from scratch, without any cached results. The incremental evaluation approach proceeds by first evaluating $Q_{TS}(D_r)$ and caching the result. Then it uses incremental view maintenance techniques [7] to directly evaluate $Q_{TS}(D_s)$ from the cached result $Q_{TS}(D_r)$ and the "delta" tuples ΔD. Suppose that w denotes the number of relation instances appearing in Q. We assume that the cost of incremental evaluation of Q_{TS} on D_r and D_s is no more than w times that of naive evaluation of Q on D_r and D_s. This is a reasonable, if conservative, assumption.

Chapter 5

Experiments

Good judgement come from experience, and experience comes from bad judgement — Rita Mae Brown

This section presents the experimental study to evaluate the performance of our proposed approach, Geometric Chunking Strategy against the TPC-H benchmark [18]. Below, we outline the key goals of the experiments.

- Demonstrate the performance gains of our proposed approach, Geometric chunking strategy, over baselines (Section 5.1.3) and over static chunking (Section 5.3).
- Compare the performance of Geometric chunking against the Oracle strategy (Section 5.4).
- Test the scalability of our approach under limited main memory, which may prevent the database and/or intermediate results from fitting in memory (Section 5.6).
- Vary the various parameters outlined in Table 5.5 and analyze their impact on our approach (Section 5.5,5.7).

5.1 Environment and Setting

All the experiments were conducted on a machine running 64-bit Windows 7 OS with Intel Core i7-6600U CPU @ 2.60GHz with 20GB RAM. Our code base on the client side is developed in Java JDK 1.5, and on the server side we use Microsoft SQL Server 2014 - Enterprise Edition. In this thesis, we present results for experiments on TPC-H 10 GB database. Additionally, we also ran experiments on TPC-H 1GB and 0.1GB and inferred same trends. We use 23 selection-projection-join queries (with their corresponding workloads), 22 of which are TPC-H queries and 1 query is created by us which demonstrates the cycle schema graph involving *all* the relations as shown in Figure 5.3c. To meet the requirements of our approach, we use the modified TPC-H queries taken from Zhang et al.'s paper which were created by dropping the group by aggregates and arithmetic expressions. Additionally, we append some projection conditions derived from the original TPC-H queries. For running our experiments, we first create a database log and a query workload, elaborated in Section 5.1.1 and 5.1.2 respectively. All running times reported in this thesis are taken as a median over 5 runs. We perform an additional first run which is used to warm up the cache and timing for this run is discarded.

5.1.1 Database Log Generation

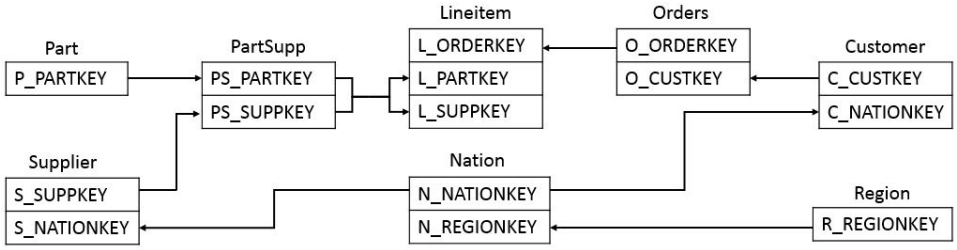


Figure 5.1: TPC-H Schema

The TPC-H database has 8 relations: Lineitem, Orders, Customer, PartSupp, Part, Supplier, Nation and Region with the schema shown in Figure 5.1. To create a database log, we start with an empty log and keep adding tuples one at a time as per the following approach:

1. Randomly select a table R (out of the 8 relations) from which a tuple needs to be added to the database log, say at timestamp τ.

2. Randomly select a tuple $t \in R$ which has not been added to the log yet.

3. In order to satisfy the integrity constraints, check for any tuples that are referenced by the tuple $R(t)$ to be added, via FK – PK references. Add all such referenced tuples, at the timestamp τ and then add the tuple $R(t)$ at the same timestamp τ.

Thus, more than one tuple may be added at the same timestamp in order to satisfy the integrity constraints. Notice that there are no foreign key – primary key references from tuples in Part and Region. Thus, if R is one of these relations, only one tuple will be added to the log at a given timestamp.

We start from $\tau = 1$ and go on till all the tuples have been added to the log. At the end of this process, each tuple in the database log receives a timestamp corresponding to the time it was inserted into the log.

The following example illustrates the database log generation process.

Example 3 (Database Log Generation)**.** Let us assume that we want to insert a tuple from the Lineitem table at Timestamp=t as shown in 5.2a. From Figure 5.1, we know that the Lineitem table has dependencies in Orders and PartSupp tables. Therefore the corresponding tuple from the Orders table, 5.2e and PartSupp table, 5.2b are inserted in the log with the same timestamp t. The Orders table further has PK-FK reference in the Customer table on CUSTKEY. Additionally, Customer relation which has dependencies in the Nation and Nation in the Region. Similarly, PartSupp table has dependencies in the Part and Supplier relations. Supplier has dependencies in the Nation table, which further has dependencies in the Region table. Therefore, the corresponding tuples from these relations are inserted into the log with the same timestamp as shown in Figure 5.2.

Notice that two tuples from the Nation table are inserted with the same timestamp. This is because the tuple with N_NATIONKEY=10 is inserted due to the S_NATIONKEY dependency (from the Supplier relation) and N_NATIONKEY=17 comes from the C_NATIONKEY=17 dependency (from the Customer relation).

L_ORDERKEY	L_PARTKEY	L_SUPPKEY	TS
158246	173493	6011	t

(a) LINEITEM

PS_PARTKEY	PS_SUPPKEY	TS
173493	6011	t

(b) PARTSUPP

P_PARTKEY	TS
173493	t

(c) PART

S_SUPPKEY	S_NATIONKEY	TS
6011	10	t

(d) SUPPLIER

O_ORDERKEY	O_CUSTKEY	TS
158246	85445	t

(e) ORDERS

C_CUSTKEY	C_NATIONKEY	TS
85445	17	t

(f) CUSTOMER

N_NATIONKEY	N_REGIONKEY	TS
10	4	t
17	1	t

(g) NATION

R_REGIONKEY	TS
4	t
1	t

(h) REGION

Figure 5.2: Example showing partial tables to illustrate database log generation

To create the database log for TPC-H SF=0.1, we first generate the database with SF=1 using the dbgen program (include citation) which gives us a database of

size 1 GB. We randomly select 10% tuples from the largest relation i.e. Lineitem and then generate the log satisfying integrity constraints as outlined above. For 1 GB and 10 GB scale experiments we use the entire tables created by running the dbgen tool [1] with SF=1 and SF=10 respectively.

5.1.2 Workload Generation

As an input to our problem, we take a set of candidate SQL queries $\mathscr{Q}$ which we generate manually for the purpose of these experiments. For each query, we generate seemingly similar queries with differing constraints in the where clause, joining constraints or the relations participating in the joins. In our workload, we consider one superset query, one subset query and the right query.

Superset query

A superset query is a query which produces a superset of the result set at $D_{\texttt{RDS}}$. Superset queries are typically constructed by either eliminating some conditions in the where clause; or by relaxing some constraint, for example consider the following valid query:

```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE,
       O_TOTALPRICE
FROM   CUSTOMER, ORDERS, LINEITEM
WHERE  O_ORDERKEY = L_ORDERKEY AND
       C_CUSTKEY = O_CUSTKEY AND C_NATIONKEY = 3
```

One obvious method to obtain a superset query for the above query is to lose the C_NATIONKEY=3 constraint in the where clause. Another method of creating a superset query is to widen the selection condition as follows:

```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE,
       O_TOTALPRICE
FROM   CUSTOMER, ORDERS, LINEITEM, NATION, REGION
WHERE  O_ORDERKEY = L_ORDERKEY AND
       C_CUSTKEY = O_CUSTKEY AND
```

[1] Available for download at http://www.tpc.org/tpch/default.asp [18] (visited on 24-08-2017).

```
        C_NATIONKEY = N_NATIONKEY AND
        N_REGIONKEY=R_REGIONKEY AND R_NAME = 'AMERICA'
```

In the above query, we are selecting customer names with their order details for the region America. The original valid query was selecting customer names with their order details for the nation Canada (corresponding to C_NATIONKEY=3) which is a subset of the wider region America. Therefore, even though we seem to be adding an extra joining relation and a joining condition, we are actually widening our selection criteria.

Typically, superset queries fail fast on early database states as they have spurious tuples which are not present in the result set R_{out}.

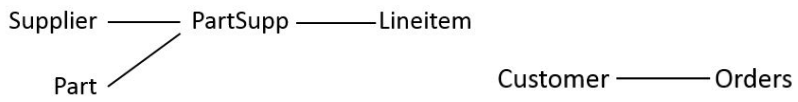
Subset query

A subset query is a query which produces a subset of the result set at D_{rds}. The attentive reader would observe that such a query negates our third assumption in Chapter 2, i.e. *the candidate query returns a superset of R_{out} when evaluated on the last state of the log*. However, in practice we relax this assumption and we will see that our approach is able to weed out these queries as well.

To generate subset queries, we add additional constraints in the where clause, for example:

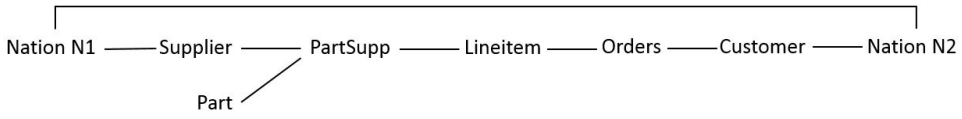
```
SELECT C_NAME, C_CUSTKEY, O_ORDERKEY, O_ORDERDATE,
       O_TOTALPRICE
FROM   CUSTOMER, ORDERS, LINEITEM, NATION
WHERE  O_ORDERKEY = L_ORDERKEY AND
       C_CUSTKEY = O_CUSTKEY AND
       C_NATIONKEY = N_NATIONKEY AND N_NATIONKEY = 3
       AND L_RECEIPTDATE < L_COMMITDATE
```

The subset queries take longer to fail since the stopping condition of $R_{out} \setminus Q(D_r)$ and $Q(D_r) \setminus R_{out}$ both being non-empty are achieved for values of $i >$ RDS. For $i <=$ RDS, $R_{out} \setminus Q(D_r)$ is not null but $Q(D_r) \setminus R_{out}$ is null, so we keep loading tuples until $Q(D_r) \setminus R_{out}$ becomes non empty as well. This typically involves loading an extra chunk beyond the chunk containing the RDS.



(a) Schema graph for Q15

(b) Schema graph for Q22



(c) Schema graph for Q23

Figure 5.3: Schema graphs for default queries.

5.1.3 Default Parameter Configuration

The various parameters that we study and their values are represented in Table 5.5. Default values (shown in bold) are used, unless stated otherwise. In Sections 5.5 and 5.7, we vary the parameters and analyze their impact.

Figure 5.4 shows the running time for all the queries at RDS-10 million. We plot time complexity (Total running time) on the Y-axis against a measure of expression complexity (the number of relations present in the query) on the X-axis. We select three queries: the longest running query, the shortest running query and the most complex query in terms of the structure and number of relations (Q15,Q22,Q23 resp.). We use these three queries to present the results in this thesis. The schema graph for them is provided in Figures 5.3a, 5.3b and 5.3c.

Q23 (created by us is shown below) selects supplier attributes and the region for suppliers that supply orders to customers within the same nation.

```
SELECT S_NAME, S_ADDRESS, S_NATIONKEY, S_ACCTBAL,
       R_REGIONKEY
FROM   PART, SUPPLIER, PARTSUPP, CUSTOMER, ORDERS,
       LINEITEM, NATION N1, NATION N2, REGION
WHERE  P_PARTKEY=PS_PARTKEY AND S_SUPPKEY=PS_SUPPKEY
       AND S_NATIONKEY=N1.N_NATIONKEY AND
       N1.N_REGIONKEY=R_REGIONKEY AND PS_PARTKEY=L_PARTKEY
       AND PS_SUPPKEY=L_SUPPKEY AND L_ORDERKEY=O_ORDERKEY
       AND O_CUSTKEY=C_CUSTKEY AND C_NATIONKEY=N2.N_NATIONKEY
       AND N1.N_NATIONKEY=N2.N_NATIONKEY
```

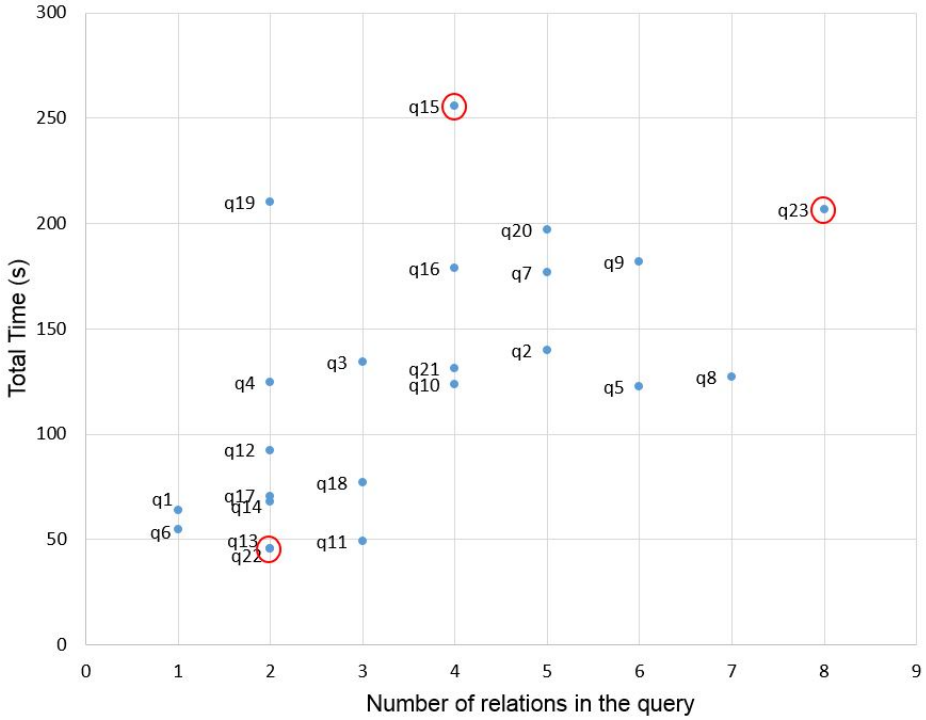


Figure 5.4: Expression complexity (on X-axis) and Time complexity (on Y-axis) for all queries for Geometric approach at RDS-10 million

Parameter	Values
Database	TPC-H SF:0.1GB, SF:1GB, **SF:10GB**
Queries	Q1-**Q15**, Q16-**Q22**, **Q23**
Strategies	Forward Naïve, Backward Naïve, Static Chunking, **Geometric Chunking**
RDS	10k, 100k, 1 million, **10 million**, 30 million, 50 million, 70 million
Base chunk size	1k , **10k**, 100k, 1 million
c	1.5, **2**, 3, 4
Materialized Views	**With** or w/o materialized views
Memory provided to SQL Server	5GB, 10GB, 15GB **20 GB**
Query workload	**Right Query**, Full Query Workload
Log Index	**Without** or with Log Index

Figure 5.5: Parameter configuration (default in bold); SF = scale factor.

Query	RDS	Forward Naïve				Backward Naïve				Geometric Chunking			
		#LS	#BS	Cut-off Time (in s)	RDS found?	#LS	#BS	Cut-off Time (in s)	RDS found?	#LS	#BS	Total Time (in s)	RDS found?
q15	1 mil	66	n/a	173.8	No	2	n/a	173.8	No	7	6	34.7	Yes
	10 mil	202	n/a	1,277.8	No	**10**	n/a	1,277.8	No	10	9	255.6	Yes
q22	1 mil	**94**	n/a	49.1	No	2	n/a	49.1	No	7	6	9.8	Yes
	10 mil	339	n/a	228.3	No	7	n/a	228.3	No	10	9	45.6	Yes
q23	1 mil	35	n/a	208.7	No	2	n/a	208.7	No	7	6	41.7	Yes
	10 mil	49	n/a	1,032.6	No	9	n/a	1,032.6	No	10	9	206.5	Yes

Table 5.1: Comparison of Geometric with Naïve approaches at RDS 1, 10 million. Last TS: 11,000,000.

5.2 Comparison with baselines

Table 5.1 shows the comparison between Forward Naïve, Backward Naïve and Geometric Chunking approach. To keep the running time tractable, we run this experiment on a log with timestamps upto 11 million instead of the entire database log ($\approx$ 82 million). We also cut-off the running time for naïve approaches at 5 * *time taken by Geometric Chunking* for similar configuration. The experiment is conducted on the Q15, Q22, Q23 for two RDS values, one at the beginning on the log (1 million) and the other towards the end of log (10 million). We observe that forward naïve performs more number of linear scans (#LS) compared to backward naïve. This is because backward naïve loads the entire data upto the last timestamp of the log whereas forward naïve loads as it goes. Clearly, for RDS at 1 million, loading the entire data upto 11 million is wasteful.

Forward naïve has the best shot to find the RDS for $\texttt{RDS} = 1\ million$, similarly backward naïve for $\texttt{RDS} = 10\ million$. Therefore, we compare these two scenarios. The forward naïve for RDS at 1 *million*, in the best case is able to traverse 94 database states and reach $r = 94$. The backward naïve for RDS at 10 *million*, in the best case is able to traverse 10 database states and reach $r = 999,991$. It is obvious that both forward and backward naïve approaches are never able to find the RDS and fail to scale.

5.3 Comparison with Static Chunking

In this experiment, we evaluate the performance of static chunking for three chunk sizes - 10k, 1.28 million and 81.92 million against the geometric chunking approach with base chunk size=10k and c=2. The static chunk sizes represent a good mix of a small chunk (10k), medium sized chunk (1.28 million) and very large chunk ($\approx$82 million). We show the comparison across a range of RDS values taken at {10,000; 100,000; 1,000,000; 10,000,000; 30,000,000; 50,000,000; 70,000,000} for the default queries. The static chunking approach is allowed twice the maximum running time taken by *GC* across all RDS values. We refer to this as capping time.

Figures 5.9,5.10 and 5.11 depict the ratio of the running time of the approach to the minimum time taken by any approach for that RDS. Broken lines indicate

that the approach was unable to finish within the capping time. We observe that *GC* approach lies within 3 times of any static chunk size.

Figures 5.6,5.7 and 5.8 depict the absolute running times (in ms) which is a combination of loading time, linear time and binary time for Q15, 22 and 23 respectively. Loading time is the time it takes to load a chunk into the SQL Servers memory. Linear time involves the time taken for evaluating $R_{out} \setminus Q$ and $Q \setminus R_{out}$ at the end of each chunk. Binary time refers to the time taken for performing binary search within a chunk to find the `RDS`.

To understand the trends in these plots, we look at the chunk boundaries for each of these strategies below.

Geometric chunking

[0, 10000, 30000, 70000, 150000, 310000, 630000, 1270000, 2550000, 5110000, 10230000, 20470000, 40950000, 81910000, 163830000]

Static Chunking (10k)

[0, 10000, 20000, 30000, 40000, 50000, 60000 ... 70000000]

Static Chunk (1.28 million)

[1280000, 2560000, 3840000, 5120000, 6400000 ... 83200000]

Static Chunk (81.92 million)

[0, 81920000, 163840000]

From Figures 5.6 and 5.9, we observe that for RDS value of 10k, both static chunking strategy with chunk size 10k and *GC* perform well. This is because both require one chunk load (to load data upto 10k), 2 evaluations at the end of the chunk and no binary search. For static chunk sizes of 1.28 million and 81.92 million, we do extra work in loading the data upto 1.28 million and 81.92 million respectively and then performing binary search. The largest chunk sized static strategy of 81.92 million performs up to 400 times worse than *GC* for smallest `RDS` values of 10k. This is synonymous to loading everything in one chunk and then performing a binary search over the entire data. For smaller RDS values this has

two disadvantages: firstly, the time taken to load tuples upto 81.92 million is more; secondly, performing binary search on a large chunk takes more time.

For the next RDS value of 100k, *GC* requires 4 chunk loads upto 150,000 linear evaluations at the end of each chunk and 3 binary searches. Static chunking with chunk size 10k requires to load 10 horizontal chunks but 0 binary searches and therefore the ratio is still comparable. Static chunking 1.28 million requires one linear load (upto 1.28 million) and 6 binary searches. For the largest chunk size of 81.92 million, we do an enormous amount of extra work in loading the data upto 81.92 million and then performing 12 binary searches to get to 100k. This is visible in the very high ratio.

For RDS value of 1 million, *GC* requires 7 chunk loads and 6 binary searches whereas static chunking 1.28 million requires 1 chunk load and 5 binary searches. As expected, static chunking of 1.28 million performs well at RDS of 1 million. Static chunking with 10k and 81.92 million compare worse because of the high loading and linear time (for 10k) and high loading and binary search time (for 81.92 million). Beyond 1 million, static chunking 10k is unable to complete within the capping time and therefore, we see a flat line at 8,500,000 ms (which is the capping time).

For the larger RDS values of 50 and 70 million, we observe that static chunking 81.92 million performs well since only one chunk load is required and therefore only 2 linear comparisons are performed; whereas for *GC* we require extra linear comparisons as we need to load 13 chunks upto 81,910,000. Additionally, we require 12 binary searches for *GC* compared to 10 binary searches for static chunking 81.92 million. The other two static methods perform worse because of the high number of chunk loads and linear comparisons. To give some perspective, at an RDS of 50 million, a static chunk size of 10k would load 5000 chunks and would also perform 5000*2 ($R_{out} \setminus Q$) and ($Q \setminus R_{out}$) comparisons. *GC* strategy would load tuples upto 81,910,000 in 13 loads, require 13*2 linear comparisons and perform 12 binary searches on a larger chunk of [40,950,001-81,910,000]. However, the extra binary searches performed by the *GC* approach on a larger chunk significantly outweigh the loading and linear time for static approach.

In general, we observe that *GC* strategy is able to catch up to any static chunking strategy, irrespective of the location of the RDS because of the geometric multiplication.

In Figure 5.7, we observe a drop in running time for *GC* strategy at RDS value of 50 million. Here we load 12 chunks up to 40,950,000, perform 2*12 linear comparisons and 0 binary searches. This is because for Q22, no new tuple gets added to R_{out} beyond time stamp 32,386,674. [2]. Hence the R_{out} at 40,950,000 is the same as R_{out}s at 50 million and 70 million. In fact, R_{out}s beyond 32,386,674 are all same. The algorithm returns witness of validity at time stamp $40,950,000$ for both RDS values of 50,70 million and therefore, requires 0 binary searches. This is observed by the drop at 50 million and then a flat curve from 50 to 70 million for *GC*. We observe a similar phenomenon for Q23 in Figure 5.8, where there is a drop at 70 million for *GC* strategy. No new tuple gets added to R_{out} beyond 60,950,000. Hence, there is no binary search required and a witness of validity is achieved through the linear comparisons at the end of the 13th chunk, i.e at 81,910,000.

In general, static chunking performs well for RDS values closer to their chunk sizes. However, no single static chunk size performs better than *GC* across the range RDS values. Since we have no way of knowing beforehand where the RDS lies and secondly, we cannot chose a single static chunk size that would perform well.

[2] Q22 is a join between Customer and Orders relation. The way the query is structured, all the tuples get added early on by timestamp=32,386,674.

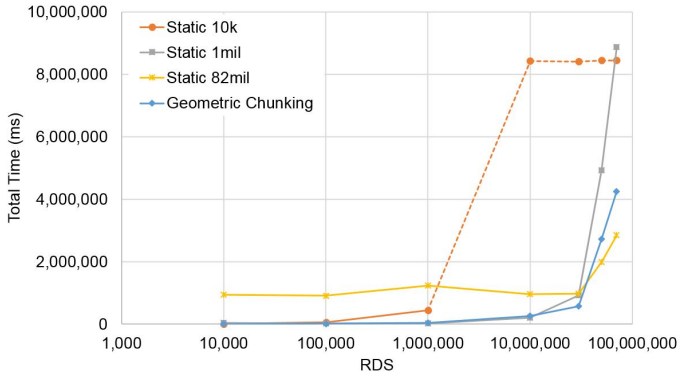


Figure 5.6: Static versus Geometric Chunking (RDS on x-axis is shown in log scale) for Q15

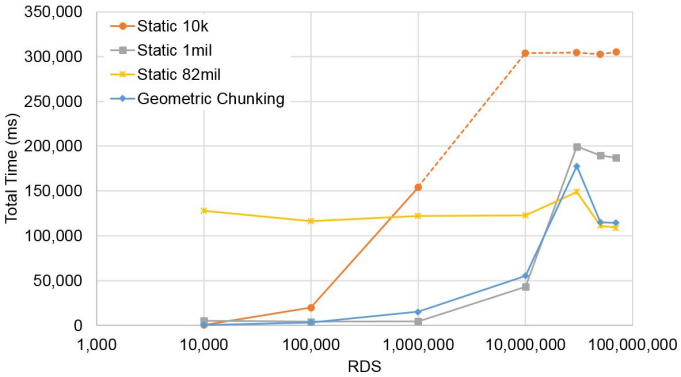


Figure 5.7: Static versus Geometric Chunking (RDS on x-axis is shown in log scale) for Q22

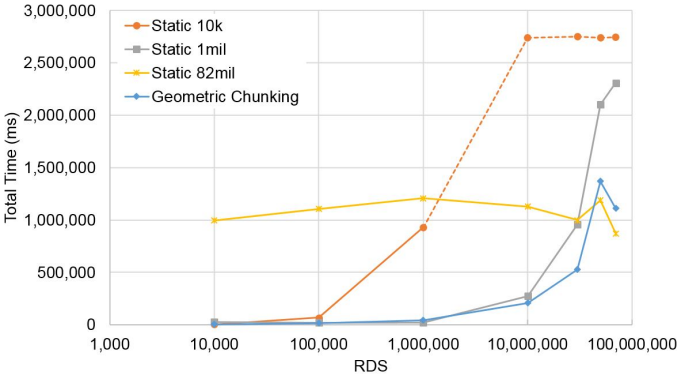


Figure 5.8: Static versus Geometric Chunking (RDS on x-axis is shown in log scale) for Q23

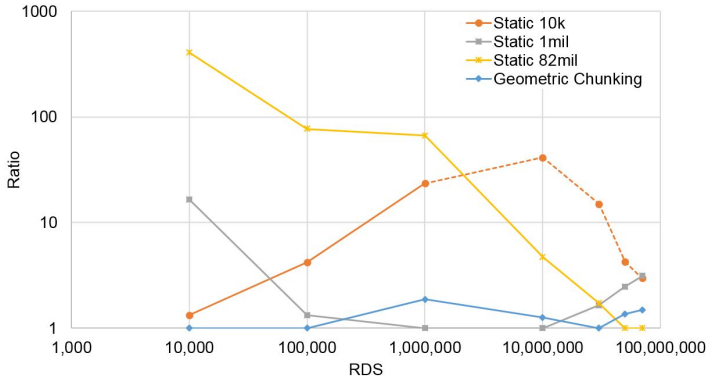


Figure 5.9: Static versus Geometric Chunking for Q15. Ratio=time of approach/MIN(.) Both axis are shown on log-scale.

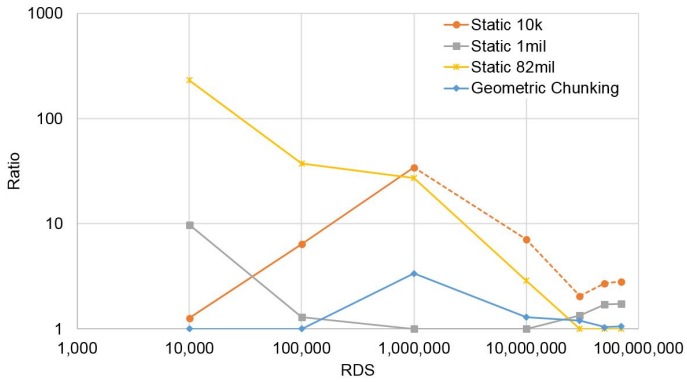


Figure 5.10: Static versus Geometric Chunking for Q22. Ratio=time of approach/MIN(.) Both axis are shown on log-scale.

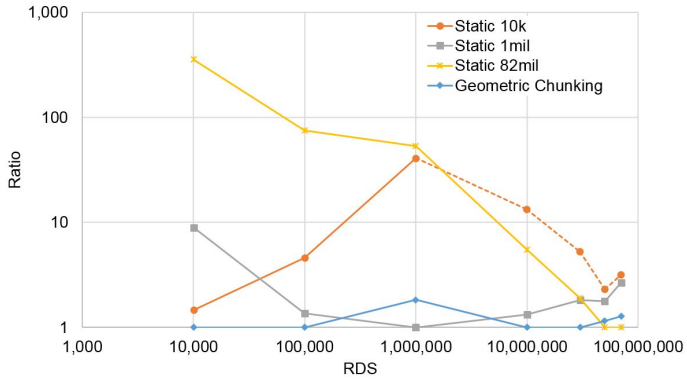


Figure 5.11: Static versus Geometric Chunking for Q23. Ratio=time of approach/MIN(.) Both axis are shown on log-scale.

5.4 Comparison with the oracle

We define Oracle approach as the scenario where an Oracle provides us with the exact location of the RDS, almost like a lucky guess. The chunk size in this case is exactly the RDS value, such that the data up to the RDS is loaded in one single chunk, all at once. We run this experiment at RDS = 1 million, 5 million, 10 million, 30 million, 70 million for the three default queries = Q15, Q22, Q23. The geometric chunking strategy is run with the default parameters, i.e. c=2 and base chunk size=10k. Figure 5.12 shows the distribution of the ratio, where ratio is defined as the time taken by geometric chunking approach to the time taken by the "Oracle" approach. The histogram shows the percentage of cases where the ratio falls within each bin. We observe than for more than 50% of the times, *GC* performs within 3 times of the "Oracle". Each bin in the histogram is further divided according to the contribution from each RDS value (represented by a colored pattern). The cases where the ratio is greater than 3 is largely for smaller RDS values where the absolute loss is not too much anyway.

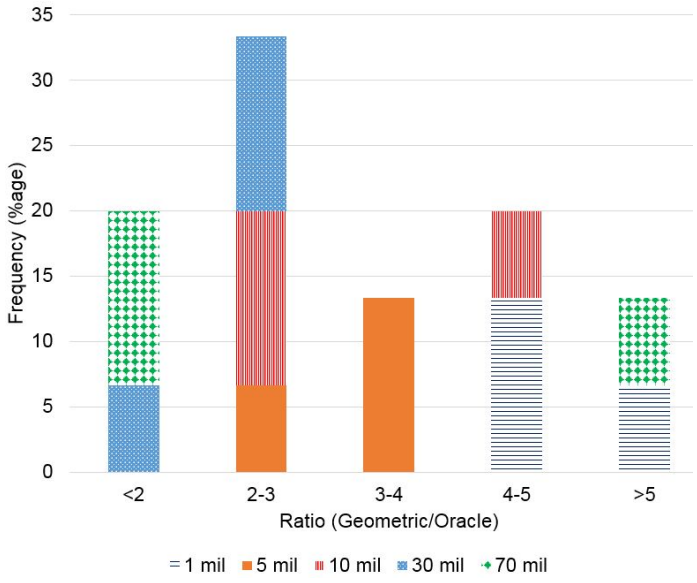


Figure 5.12: Histogram of Oracle v/s Geometric. Ratio=Total Time taken by Geometric/Total Time taken by Oracle.

5.5 Impact of Log Index

In this experiment, we try to understand the performance improvements of *GC* strategy in the presence of a Log Index (LI) on the database log. A log index brings us very close to the RDS, how close depends on how good the index is. We simulate this by jumping to the time stamp provided by the Log Index, i.e. loading the data upto the jump in one single chunk, performing 2 linear comparisons ($R_{out} \setminus Q(D_{jump})$ and $Q(D_{jump}) \setminus R_{out}$) at the end of the jump and then proceeding with the usual *GC* beyond this point. We run this experiment for LI jumps at 100k, 1 mil, 10 mil and 50 mil. Figures 5.13, 5.14 and 5.15 depict the ratio - running time of *GC* with Log Index to running time of *GC* without Log Index for Q15, 22 and 23 respectively. To understand the trends in these plots, we need to look at the horizontal chunks for each strategy.

Geometric chunking

[0, 10000, 30000, 70000, 150000, 310000, 630000, 1270000, 2550000, 5110000, 10230000, 20470000, 40950000, 81910000, 163830000]

GC with LI (100k)

[0, 100000, 110000, 130000, 170000, 250000, 410000, 730000, 1370000, 2650000, 5210000, 10330000, 20570000, 41050000, 82010000, 163930000]

GC with LI (1 mil)

[0, 1000000, 1010000, 1030000, 1070000, 1150000, 1310000, 1630000, 2270000, 3550000, 6110000, 11230000, 21470000, 41950000, 82910000]

GC with LI (10 mil)

[0, 10000000, 10010000, 10030000, 10070000, 10150000, 10310000, 10630000, 11270000, 12550000, 15110000, 20230000, 30470000, 50950000, 91910000]

GC with LI (50 mil)

[0, 50000000, 50010000, 50030000, 50070000, 50150000, 50310000, 50630000, 51270000, 52550000, 55110000, 60230000, 70470000, 90950000]

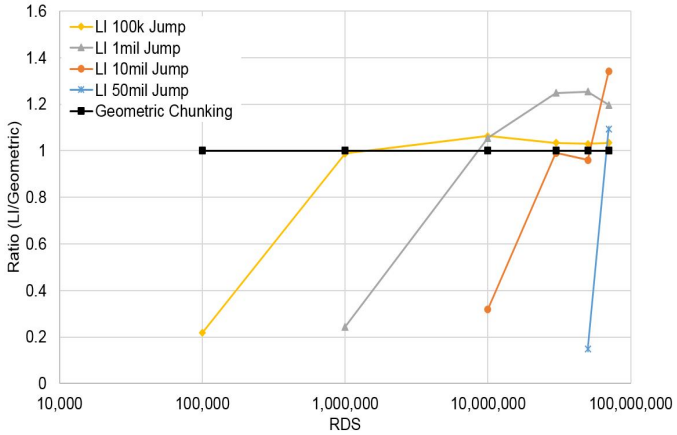


Figure 5.13: Log Index with different initial jumps versus Geometric for Q15. X-axis shown in log scale. Y-axis shows Ratio=Time taken by approach/Geometric

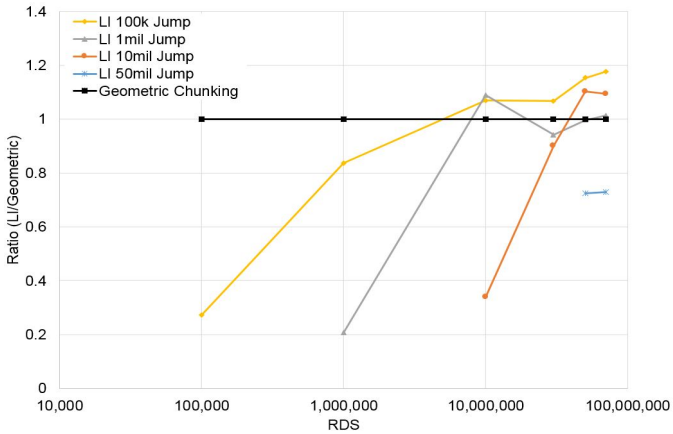


Figure 5.14: Log Index with different initial jumps versus Geometric for Q22. X-axis shown in log scale. Y-axis shows Ratio=Time taken by approach/Geometric

In Figure 5.13 for the 100k jump at RDS 100k, the ratio is very small as expected because the *GC* with LI-100k loads all the data upto 100k in one chunk and performs 2 comparisons ($R_{out} \setminus Q(D_{100k})$ and $Q(D_{100k} \setminus R_{out})$ whereas *GC* with default parameters performs 4 chunk loads and 3 binary searches to find the RDS.

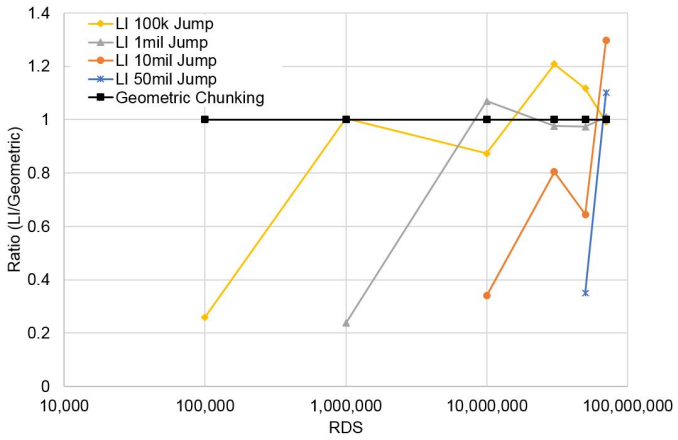


Figure 5.15: Log Index with different initial jumps versus Geometric for Q23. X-axis shown in log scale. Y-axis shows Ratio=Time taken by approach/Geometric

However, we observe that *GC* with default parameters catches up quickly for the later RDS values.

For *GC* with LI-1 million, a similar explanation holds for the small ratio at RDS 1 million. For the later RDS values, *GC* with the log index performs one additional chunk load to load data. This results in the ratios greater than 1. Please note the difference between the absolute running times is not very much.

GC with LI-10 million has ratios below 1 for RDS values 10, 30 and 50 million. At RDS 10 million, everything is loaded in one chunk as opposed to 10 chunk loads and 9 binary searches performed by *GC*. At RDS-30 million, *GC* with LI loads data upto 30470000 whereas the default *GC* loads a larger chunk upto 40950000. *GC* with LI performs one less binary search on a smaller chunk size. A similar explanation holds at RDS 50 million. *GC* with LI loads data upto 50950000 whereas the default *GC* needs to load a larger chunk upto 81910000. At RDS 70 million, *GC* with LI now needs to load one extra chunk upto 91910000. The linear comparisons here are more expensive as well since we do it on more data. This leads to the higher ratio at 70 million.

Note in Figure 5.14, the ratios for last two RDS values 50 million and 70 million are very similar for every strategy. Recall from our previous discussion in Section 5.3, R_{out} does not change beyond timestamp=32,386,674. So in essence, the al-

gorithm provides witness of validity at the same states for both 50 million and 70 million.

To conclude, the *GC* catches up very quickly to the *GC* with Log Index. Moreover, maintaining a Log Index is also expensive. If we were to include the cost of maintaining a Log Index, *GC* with Log Index would not perform much better than the default *GC*.

5.6 Scalability in terms of available memory

We study the effects of our approach for limited memory scenarios. We limit the memory provided to SQL Server by modifying the maximum available memory to SQL Server. This experiment tries to simulate the practical scenario when the entire database does not fit in memory. We decrease the memory from 20GB to 2GB. Figure 5.16 shows an increase in running time for queries 15 and 23 as we decrease the memory (as expected because of the increase in I/O swaps). However, Q22 takes almost the same time to run up to 7GB after which decreasing memory increases the running time. This is because Q22 is a join between Customer and Orders relations. Since these tables are relatively small they fit in memory even when available memory is 7GB, after which there is an increase in I/O swaps reflected by the increase in running time.

To conclude, we show that our approach is robust in dealing with large databases that do not fit in memory. This is particularly useful in a streaming database setting where the entire database may not be available in memory.

5.7 Varying Configuration

5.7.1 Impact of Base Chunk Size

We run our experiments with default value of base chunk size as 10,000. In this experiment, we study the effects of varying the base chunk size at RDS of 10 million. We vary the base chunk size from 1k to 1 million. A very small base chunk size such as 1k does not scale up very well, the horizontal chunks being as follows: [0, 1000, 3000, 7000, 15000, 31000, 63000, 127000, 255000, 511000, 1023000, 2047000, 4095000, 8191000, 16383000, 32767000, 65535000, 131071000] takes

Figure 5.16: Varying the available memory provided to SQL Server.

14 chunk loads to load data up to 10 million compared to 10 taken by *GC* with default parameters. Moreover, *GC* with base chunk size 1k creates very large chunks towards the end which result in an increased number of binary searches. However, any reasonable base chunk size gives us very similar results as shown in 5.17 and our approach does not dramatically change with changes in base chunk size. Hence, our approach is robust to changes in base chunk size.

5.7.2 Tuning the geometric multiplier 'c'

Empirically, we have observed that geometric multiplier c=2 performs best in terms of striking a good balance between the number of linear loads and binary searches. However, changing the geometric multiplier does not drastically change results as seen from Figure 5.18. This shows that our approach is also robust to changes in c.

5.7.3 View Maintenance

We take advantage of the view maintenance functionality provided by the modern database systems and incrementally maintain the view for $Q(D_r)$. For systems which do not have a provision for view maintenance, we need to compute $Q(D_r)$ from scratch every time $R_{out} \setminus Q$ and $Q \setminus R_{out}$ needs to be computed. We show in

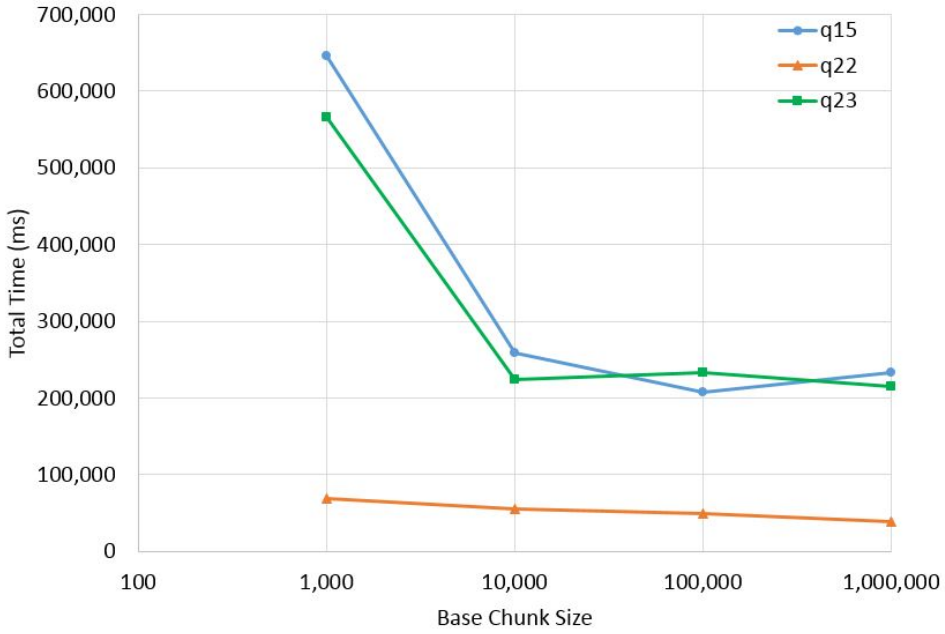


Figure 5.17: Varying base chunk size. X-axis is in log-scale.

—Query—	—Total Time (in ms)—		—Percentage Improvement—
	Without View Maintenance	With View Maintenance	
q15	344,818	259,267	24.81047973
q22	65,825	55,278	16.02278769
q23	316,567	224,480	29.08926073

Table 5.2: Performance improvement using View Maintenance

Table 5.2 that there is a significant performance gain achieved by using incremental view maintenance over computing the query from scratch. The numbers shown here are compared at `RDS` of 10 million. Percentage improvement is calculated as: (Time taken without view maintenance - Time taken with view maintenance)/Time taken with view maintenance * 100.

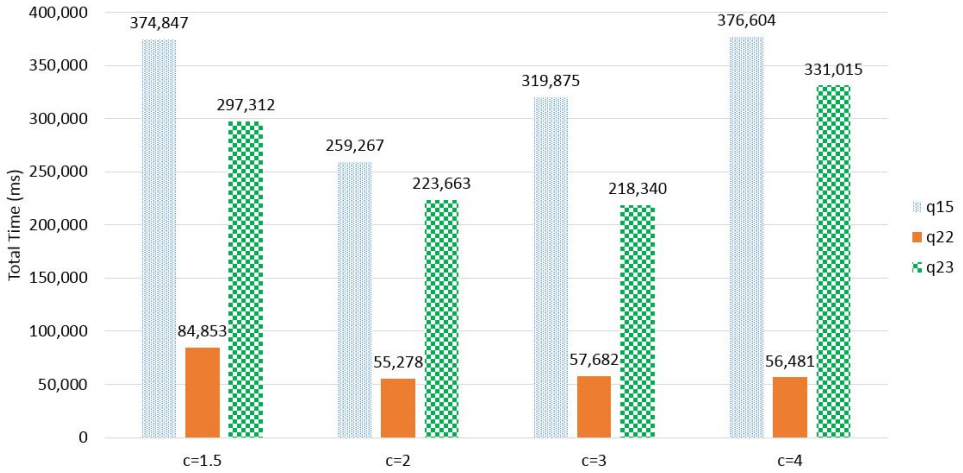


Figure 5.18: Varying c. X-axis showing RDS is in log-scale.

5.7.4 Full query workload versus right query workload

In our experiments, each full query workload consists of the right query, one "incorrect" subset query, one "incorrect" superset query.

Superset queries As discussed earlier, superset queries are faster to invalidate. Consider the following two scenarios:

- RDS *lies in the first chunk*: For a superset query, $R_{out} \setminus Q_{sup}(D_{end})$ is empty and $Q_{sup}(D_{end}) \setminus R_{out}$ is non-empty at $\tau = end$, where *end* is the last timestamp of the first chunk. According to our algorithm, we perform a binary search between $\tau = 1$ and $\tau = end$. Say at $\tau = mid$, both the conditions become non-empty and we give the witness of invalidity i.e., $wit_Q = \langle (D_{mid}, D_{mid+1}) \rangle$. In the worst case, this will take $2 * log_2$(base chunk size) number of comparisons. [3].

- RDS *lies beyond the first chunk*: For a superset query, $R_{out} \setminus Q_{sup}(D_end)$ and $Q_{sup}(D_end) \setminus R_{out}$ both are non-empty at $\tau = end$, where *end* is end of the first chunk. Therefore we can give the witness of invalidity at $\tau = end$ itself, without having to perform any binary search comparisons. The expense for

[3] since base chunk size=size of first chunk

query invalidation here is minimal- one chunk load and 2 comparisons at the end of the chunk.

Generally, we encounter the second scenario where we can quickly invalidate the superset queries at the end of first chunk.

Subset queries Subset queries take $\geq$ number of chunk loads as the right query. In the worst case, subset query can take $2 * (log_2(last) - log_2(end))$ number of comparisons to give a witness of invalidity where last refers the last TS of the log and end refers to the last TS of the chunk containing RDS. In general, we have observed that it takes one extra chunk load than the right query to give a certificate of invalidity. This is because $R_{out} - Q_{sub}(D_r)$ is non empty (because Q_{sub} is a subset query) and $Q_{sub}(D_r) - R_{out}$ is also non-empty (because r lies beyond RDS, hence incorporating extra tuples from time stamps between RDS and r). Therefore, subset queries take longer to receive a certification of invalidity.

Since there is no way of knowing beforehand which category the query belongs to, we assume a mix of all three in our workload. We maintain three views for each query in the workload. Each chunk is loaded only once and all the queries (whose validity is yet to be determined) are evaluated at the end of the chunk. As soon as any query receives a witness, we drop the view for that query and go forward with the remaining queries.

Figures 5.19, 5.20 and 5.21 show the absolute running times for the right query and the full query workload. Observe that the full query workload does not involve a tremendous amount of overhead for the extra two queries. Figure 5.22 depicts the percentage increase in the running time for the full query workload over just the right query. Barring the first RDS at 10k, all the others lie within 33% of increased running time. For the first RDS, the startup cost of loading the chunk and end of chunk comparisons outweigh the advantage received by maintaining the different views. However, the absolute times here are very small.

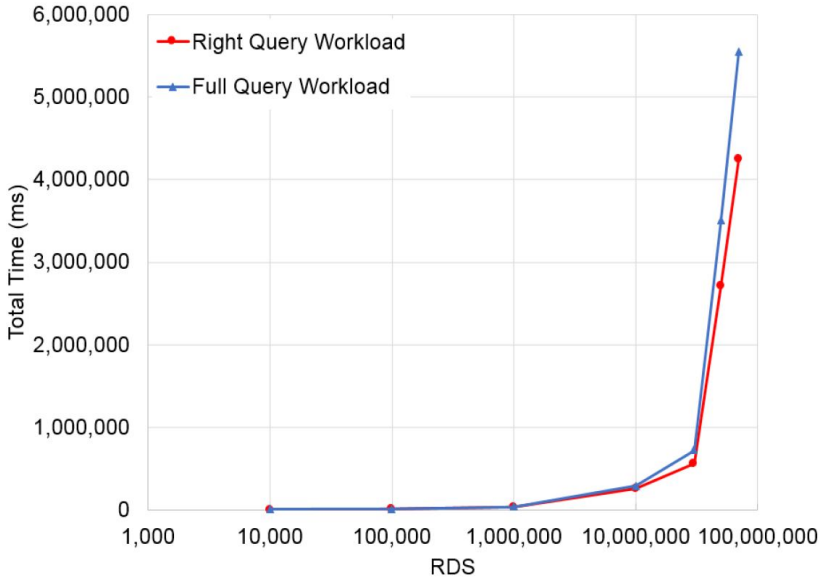


Figure 5.19: Right Query versus Full Query Workload: q15. X-axis showing RDS is in log-scale.

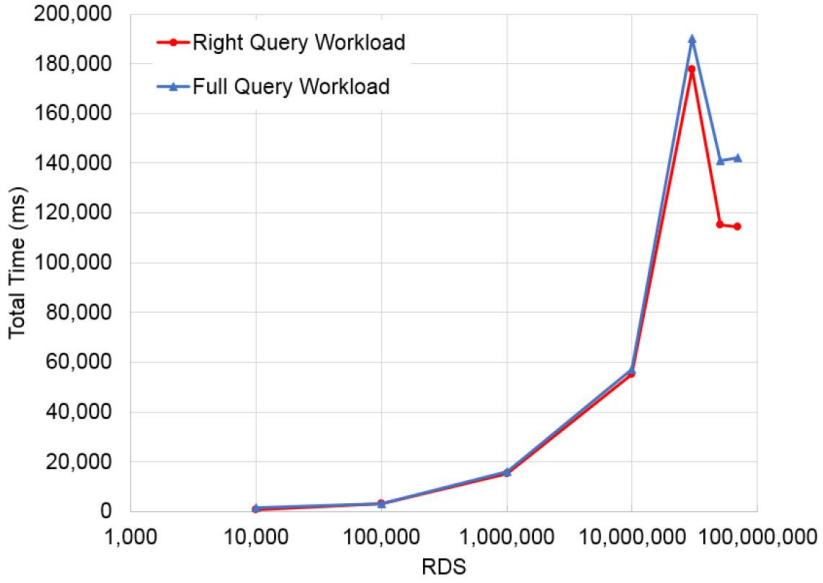


Figure 5.20: Right Query versus Full Query Workload: q22. X-axis showing RDS is in log-scale.

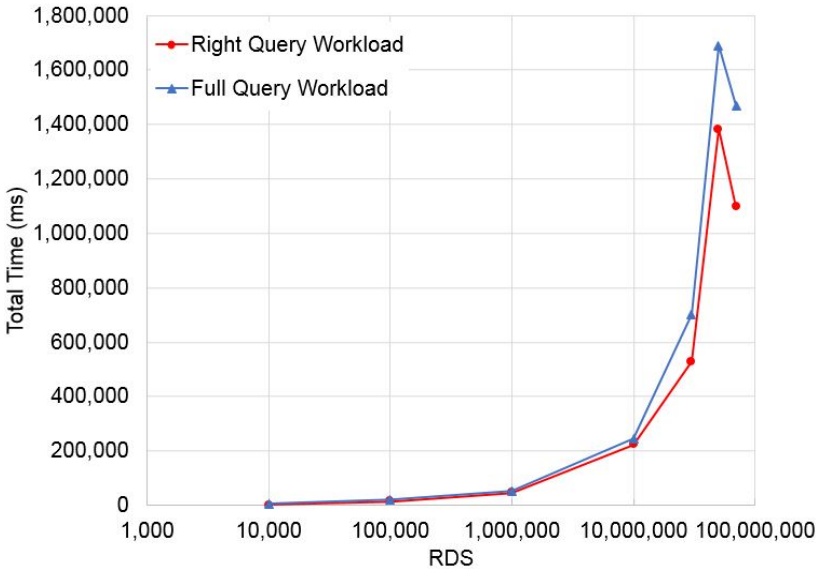


Figure 5.21: Right Query versus Full Query Workload: q23. X-axis showing RDS is in log-scale.

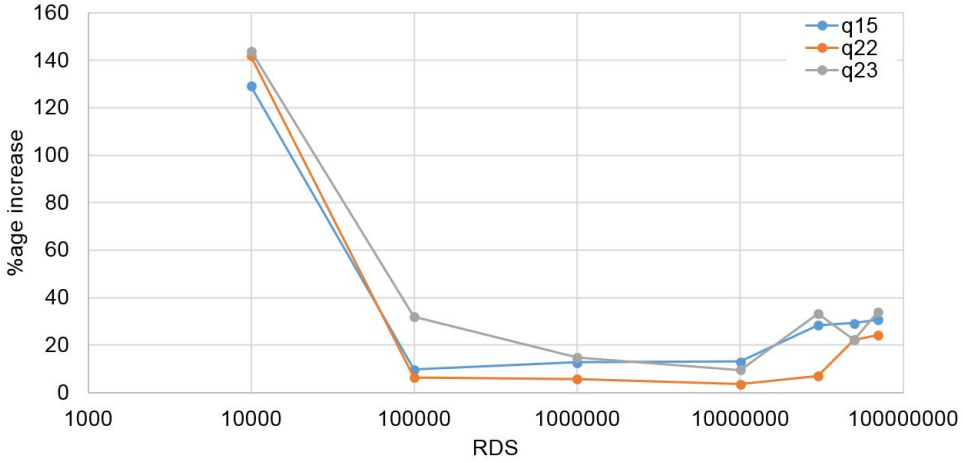


Figure 5.22: %age increase=(Full Query Workload-Right Query Workload)/Right Query Workload * 100. X-axis showing RDS is in log-scale.

Chapter 6

Related Work

If I have seen farther it is by standing on the shoulders of Giants.
— Sir Isaac Newton (1855)

The related work can be broadly classified into 2 areas - Query Reverse Engineering and Reverse Query Processing.

6.1 Query Reverse Engineering (QRE)

This body of work focuses on obtaining a SQL query that generates a specified result set when evaluated on a given database state. There are three recent directions that fall under the umbrella of Query Reverse Engineering but their goals and techniques have significant differences.

6.1.1 Deriving Instance Equivalent Queries (IEQs)

Two queries Q_1 and Q_2 are considered IEQs w.r.t. a database state D_r if their results are equal, i.e. $Q_1(D_r) = Q_2(D_r)$. Authors initial work — Query By Output(QBO), is a data driven approach that focuses on generating IEQs from a known input query and its result set on a given database state [19].

They extend QBO to derive IEQS where the input query is unknown and provide support for multiple database states on a continuous log. Specifically, given a result set R_{out} and a sequence of database states $< D_1, D_2, .., D_r >$, determine

the most recent database state r and a query Q such that $Q(D_r) = R_{out}$. This work [20] is most relevant to our work and we will discuss their work in greater detail. Their work aims to solve a more general problem, where they aim at query discovery rather than query validation. Their work considers SPJ queries as well as aggregates and unions. They model the different database states using a backward delta storage organization [17]. Given a sequence of database states $< D_1, D_2, .., D_r >$, the database stores the most recent state D_r together with backward deltas $\delta_{r(r-1)}, .., \delta_{21}$. A backward delta models a set of insert and delete operations such that D_{r-1} is derived from D_r and $\delta_{r(r-1)}$. Since their work considers update operations involving additions and deletions, they are unable to take advantage of the natural monotonicity in a data streaming environment. Our baseline—*Backward naïve strategy* is inspired by their approach. From the family of queries that they cover, we focus on SPJ queries and show in Section 5.1.3 that their approach is unable to scale up beyond a few database states. However, their discovery of IEQs can serve as an input to our candidate query set Q.

A subproblem of QBO is considered in View Definitions Problem (VDP) [9]. This is a selection condition discovery problem for the view V for a single relation in R, where both R and V share the schema. Their discovery of Q equates to discovering selection predicates on R to generate V without any joins and projections. Our work is able to validate queries involving selection conditions and hence, VDP can serve to generate candidate queries for our work.

Another work that belongs to QRE and is seemingly relevant to our work is Reverse Engineering Top-k Database Queries [15]. This work addresses the problem of reverse engineering top-k queries, given a result set R_{out} and a relation R. This work also deals with a single relation and tries to find selection conditions to produce R_{out}. The authors present a probabilistic model that assesses the chance of a candidate query Q to evaluate exactly as or close to R_{out}. We are particularly interested in their approach for query validation. They order the candidate queries by their expected suitability to evaluate to R_{out}. They claim that this approach promises to find a valid query early. They also discuss smart query validation where, instead of evaluating each query sequentially in the order provided by the ranking, they take advantage of the information learnt from executing the previous query. Consider query Q_1 was executed (based on the ranking) and it produced

results very similar to R_{out} but not an exact match. In such a scenario they consider validating queries that are similar to Q_1 and skip those in the ranked list. The drawback of their approach is that they incorporate many false positive predicates in their candidate queries. For evolving databases (because of updates, inserts or deletes in a data warehouse scenario), their approach further introduces false negatives. They claim to combat this using the previously mentioned 'smart validation.' However, even in the smart validation strategy, they resort to evaluating the candidate queries on the entire database state, which is an expensive operation. Instead, our GC approach could work on top of their smart validation strategy and help discard the invalid queries early on in the log by failing fast.

6.1.2 Query from Examples (QFE)

This body of work was introduced in [12] and helps non-expert database users construct SQL queries through user feedback. The user initially provides a database state-result set pair (D_r, R_{out}) as input. QFE iteratively presents the user with database state-result set pairs which are close to the input pair. The user is required to determine if the new database state-result set pair is consistent with her desired query Q. A similar approach has been discussed in AIDE [10] where the desired query Q is developed based on the users feedback on samples of database tuples. Based on user feedback, the system generates a new set of database tuples. The query is presented to the user when she wishes to terminate this process after a few iterations. These approaches are similar to our work in that the query is unknown. However these are all query generation approaches and our work can be appended to validate the candidate queries generated by the discussed approaches.

Abouzied et al. discuss learning and verifying a special class of Boolean queries called qhorn queries. We are particularly interested in the verification part of their work. This is also a query from example approach, where the learning algorithms pose membership questions to the user and she classifies each data object as an answer or a non-answer. The input to the verification model is a candidate query Q_c and the user's intended query Q_i. The algorithm generates a set of membership questions for each query. If Q_i is semantically different from Q_c, then their answers would differ on at least one question and we can safely invalidate the query.

6.1.3 Targeted Query Generation

Bruno et al. and Mishra et al. study the problem of generating test queries to satisfy certain cardinality constraints on their subexpressions.

6.2 Reverse Query Processing

A slightly tangential body of work discusses Reverse Query Processing. The problem addressed here is generating databases to satisfy a set of constraints given a set of queries $\mathscr{Q}$ and a corresponding result set $\mathscr{R}_{out}$. Binnig et al. introduced the problem: given a query $Q \in \mathscr{Q}$ and the corresponding result set $R_{out} \in \mathscr{R}_{out}$, generate a database D such that $Q(D) = R_{out}$.

QAGen and its various variants discuss the problem: given an input query Q and a set of cardinality constraints on the subexpressions for Q's evaluation plan P, generate a test database D such that P's execution on D satisfies those constraints [6][13][3]. Unlike the above works on database generation, our work focuses on finding a database state-query pair in a continuous log.

Chapter 7

Conclusion

In this work, we have proposed a framework to validate monotone SPJU queries on a continuous log. This problem can be used as a standalone application to assist in data mining and data analysis tasks or it can be used in conjunction with the various Query Reverse Engineering (QRE) problems. From the previous work on QRE, we have observed that most focus on query discovery tasks, where query validation becomes a bottleneck. Our proposed approach Geometric Chunking aims to solve that by failing fast on the invalid queries and quickly finding the RDS for valid queries. We have performed experiments on a 10 GB data set and shown that our approach is scalable and robust to changes in parameters. Our approach deals well with limited memory conditions and can perform for scenarios where the memory available is one-fifth the size of the database. We also compare with baselines and static chunking and show that they do not scale well. As future work, a challenging issue that requires further study is to validate queries involving non-monotonicity, aggregation operators and arithmetic expressions.

Bibliography

[1] S. Abiteboul. *Foundations of databases*. Addison-Wesley, Reading, Mass, 1995. → pages 6

[2] A. Abouzied, D. Angluin, C. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, pages 49–60, New York, NY, USA, 2013. ACM. → pages 53

[3] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 685–696, New York, NY, USA, 2011. ACM. → pages 54

[4] A. Baid, A. Balmin, H. Hwang, E. Nijkamp, J. Rao, B. Reinwald, A. Simitsis, Y. Sismanis, and F. van Ham. Dbpubs: multidimensional exploration of database publications. *PVLDB*, 1(2):1456–1459, 2008. → pages 2

[5] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 506–515. IEEE, 2007. → pages 54

[6] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 341–352, New York, NY, USA, 2007. ACM. → pages 54

[7] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM. → pages 11, 22, 24

[8] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. In *Transactions on Knowledge and Data Engineering*. IEEE Computer Society, January 2006. → pages 54

[9] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 89–103, New York, NY, USA, 2010. ACM. → pages 52

[10] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. AIDE: an automated sample-based approach for interactive data exploration. *CoRR*, abs/1510.08897, 2015. → pages 53

[11] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable scheduling of updates in streaming data warehouses. *IEEE Transactions on Knowledge and Data Engineering*, 24:1092–1105, 2011. → pages 5

[12] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015. → pages 53

[13] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. *Proc. VLDB Endow.*, 3(1-2):848–859, Sept. 2010. → pages 54

[14] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 499–510, New York, NY, USA, 2008. ACM. → pages 54

[15] K. Panev, E. Milchevski, and S. Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In *Data Engineering Workshops (ICDEW), 2016 IEEE 32nd International Conference on*, pages 181–188. IEEE, 2016. → pages 52

[16] R. Schumacher. An extended version of faulhabers formula. *J. Integer Sequences*, 19, 2016. → pages 22

[17] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM. ISBN 0-89791-191-1. → pages 23, 52

[18] TPC. Tpc-h benchmarks. (visited on 24/08/2017). URL http://www.tpc.org/. → pages 25, 29

[19] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 535–548, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. → pages 51

[20] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, Oct. 2014. → pages 2, 10, 52

[21] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 809–820, New York, NY, USA, 2013. ACM. → pages 26