RQL: Retrospective Computations over Snapshot Sets

Nikos Tsikoudis
Brandeis University
tsikudis@cs.brandeis.edu

Liuba Shrira
Brandeis University
liuba@cs.brandeis.edu

Sara Cohen
Hebrew University, Jerusalem
sara@cs.huji.ac.il

ABSTRACT

Applications need to analyze the past state of their data to provide auditing and other forms of fact checking. Retrospective snapshot systems that support computations over data store snapshots, allow applications using simple data stores like Berkeley DB or SQLite, to provide past state analysis in a convenient way. Current snapshot systems however, offer no satisfactory support for computations that analyze multiple snapshots. We have developed a Retrospective Query Language (RQL), a simple declarative extension to SQL that allows to specify and run multi-snapshot computations conveniently in a snapshot system, using a small number of simple mechanisms defined in terms of relational constructs familiar to programmers. We describe RQL mechanisms, explain how they translate into SQL computations in a snapshot system, and show how to express a number of common analysis patterns with illustrative examples. We also describe how we implemented RQL in a simple way utilizing SQLite UDF framework in a Berkeley DB data store using Retro page-level incremental snapshot system. Multi-snapshot computations running over page-level incremental snapshots bring up interesting performance issues that have not been studied before. We present the first study defining a performance envelope for multi-snapshot computations over page-level incremental snapshots.

1 INTRODUCTION

To provide auditing and other forms of claim checking more and more applications need to answer questions, often formulated after the fact, about the past states of their data. To free applications from the burden of managing past states on their own, data management systems need to run ad-hoc computations over past states of the objects they store.

Computations over past states have been long supported by temporal databases, used by applications in specialized domains but not used by general applications because of cost and performance penalty for in-production operation. More recently, cheap storage and interest in using past state analytics for in-production operation led to development of systems that integrate past state analytics in a database [7, 11, 13], and all major vendors today offer products providing OLTP and OLAP processing in a single system [16]. These products however are not a good match for Internet applications that store their data in simple key value stores such as Berkeley DB (BDB) [15] or SQLite [8] and need past state analysis for on-line historical claim checking or auditing. Today however, even applications using key value stores can support past state analysis using snapshot systems that support retrospection, the ability of a data store to run queries over consistent snapshots of application past state as if they were the current state [22]. Retrospection makes it easy for programmers to provide expressive past state analysis since it allows to implement ad-hoc queries as general programs in the application language using the application code base and then run these programs on the snapshots of interest. For example, Retro [21], a snapshot system of BDB, allows to analyze the state of BDB SQLite applications at a particular point in time simply by running SQLite queries over the corresponding BDB snapshot.

As convenient as it is to analyze a single point in time using a snapshot system, many analyses concern multiple data points. Retro and other snapshot systems come short when it comes to analyzing multiple snapshots. A programmer needs to write a C script that manually identifies snapshots of interest, queries each snapshot separately, manually collects the results, and then processes the results. This approach is cumbersome, error prone and onerous for a SQL programmer who needs to learn a new language. The programmer would much prefer to specify the computation in a declarative manner using the language of the application.

To help with programming the desired computation logic for multiple snapshot analysis, we have developed a Retrospective Query Language (RQL), a simple declarative extension to SQL that allows to specify and run multi-snapshot computations without the need to use a low-level script. RQL mechanisms combine in a modular way high-level relational constructs to express general SQL computations over arbitrary sets of past BDB SQLite application snapshots. The constructs specify in SQL the set of snapshots that identify the past states of interest, the computations over each snapshot, and the computations that process the results.

We describe RQL mechanisms and explain how each high-level mechanism translates into a SQL computation over multiple snapshots in the Retro snapshot system. We also show how to express a number of common analysis patterns with illustrative examples. We then describe how we implemented RQL in a simple way using SQLite UDF.

RQL programs bring to light important performance considerations that arise when programs compute over multiple snapshots. For one, RQL mechanisms allow to specify computations over arbitrary size sets of snapshots. The number of snapshots stored by a snapshot system such as Retro, only limited by available storage, can be very high given today's low storage costs. Each snapshot includes the entire state of the database. RQL program therefore can compute over potentially very large amounts of data. A programmer needs to know how much CPU, memory and I/O resources his program requires, especially in today's utility computing environments. Furthermore, an important performance consideration in the design of snapshot systems like Retro is to avoid interfering with the data store performance so that snapshots can be created at required frequency without blocking or disrupting in-production application performance. Retro snapshot system achieves this by using a low-cost copy-on-write technique that creates an incremental page-level snapshot representation with a compact snapshot index [22, 23]. Such representation is known to be slower to compute with but the slowdown is considered to be an acceptable trade-off to preserve in-production performance. The reason a computation runs slower over a snapshot and incurs higher resource costs compared to the current database state is that the snapshot state needs to be assembled on-the-fly from the incremental representation. Inherently however, with a snapshot representation created using copy-on-write, consecutive snapshots share large parts of their state. An RQL program that iterates computations over multiple consecutive snapshots can therefore reduce its costs and improve performance by assembling such a shared state only once. The performance of programs computing over multiple *page-level copy-on-write* snapshots however has not been studied before.

To this end, we implemented RQL in SQLite BDB using Retro snapshot system and conducted an experimental study that characterizes the performance envelope of RQL programs. The goal of the study is to explain the performance in a way that is easy to understand by the programmer. Since the programmer specifies RQL mechanisms as a modular composition of relational constructs in SQLite, we relate the performance of RQL program to the performance of its SQL components, a cost that should be familiar to the programmer. Our experiments use workloads derived from the standard TPC-H benchmark to evaluate RQL performance so that even though our system is different from other past state systems, its performance is explained using a standard workload.

In summary, the paper makes the following contributions:

- RQL, a SQLite extension that allows to express computations over multiple snapshots in a convenient way in the language of the application. Our focus here is SQL extension but we believe a similar approach can be used for BDB applications in other languages since the BDB/Retro system is language-independent.
- An implementation of RQL system using SQLite UDF.
- A performance study of RQL programs including the first analysis explaining the performance of computations running over multiple page-level copy-on-write snapshots. While the performance results reported in our study are specific to our system, our performance analysis is more general and applies to other page-level copy-on-write snapshot systems.

The rest of the paper is organized as follows. Section 2 describes the RQL mechanisms, Section 3 outlines the salient points of RQL implementation using SQLite UDF, Section 4 briefly outlines the basic structure of the copy-on-write snapshot system Retro, providing the background for our performance analysis, Section 5 describes the experimental study, Section 6 considers the related work, Section 7 concludes.

2 RQL LANGUAGE

In this section we present RQL, the programming language for specifying SQL computations over sets of snapshots of past states in a data store. Our departure point is a transactional key value store with an integrated snapshot system that allows a SQL application to take snapshots of its state and subsequently run a SQL computation over a snapshot. Specifically, we assume the Retro snapshot system integrated with the BDB SQLite [21]. We first explain the snapshot computation model provided by Retro, and the concrete language constructs used by SQL programmers to create snapshots and to specify a program that runs over a snapshot.

Retro extends BDB/SQLite with a language construct that allows to declare a snapshot as part of normal transaction commit using the *BEGIN;* and *COMMIT WITH SNAPSHOT;* commands. The declaration command creates a transactionally consistent persistent snapshot that includes the state of the entire database (e.g., tables, indexes, system catalogs). The snapshot reflects the modifications committed by the declaring transaction T, and all the transactions committed before T. The declaration permanently associates with the snapshot a unique snapshot identifier that names the snapshot. The identifier and a current timestamp are entered in a table with name *SnapIds*. Although Retro uses integer sequence numbers as snapshot identifiers internally, a programmer can associate meaningful snapshot names with the identifiers.

l_userid	l_time	l_country
UserA	2008-11-09 13:23:44	USA
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 15:45:21	USA

(a) Snapshot S1

l_userid	l_time	l_country
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 21:33:12	USA

(b) Snapshot S2

l_userid	l_time	l_country
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 21:33:12	USA
UserD	2008-11-11 10:08:04	UK

(c) Snapshot S3

Figure 1: LoggedIn table in snapshots 1-3

snap_id	snap_ts
1	2008-11-09 23:59:59
2	2008-11-10 23:59:59
3	2008-11-11 23:59:59

Figure 2: SnapIds table

A query can run on a snapshot at any point following the snapshot declaration. To run a SQL program, such as "SELECT ..." over previously declared snapshot with identifier *sid*, the programmer simply specifies "SELECT AS OF sid ...". SQL queries and update transactions that do not declare snapshots remain unchanged by Retro.

Consider an example SQL application using a *LoggedIn* table that stores the users who are logged in a system, along with the time and the country from which each user has logged in. Whenever a user logs out, he is deleted from the table. Figure 3 shows a SQL program containing three consecutive snapshot declaration commands, (lines 1-2, lines 3-5, and lines 6-8), a snapshot query command that runs on the snapshot *sid* 1 (line 9), and the same query that runs on the current database state (line 10). Figure 1 shows the table state in three declared snapshots, and Figure 2 shows the *SnapIds* table. Note, that the state of *LoggedIn* table in the snapshot 2 declared in lines 3-5 does not include $UserA$ since a snapshot reflects updates of the declaring transaction.

Retro makes it easy to analyze a single snapshot but has no support for analysis concerning multiple snapshots. In order to provide this functionality we propose RQL, a simple language for specifying computations over a set of Retro snapshots.

```
Declare snapshot S1
1. BEGIN;
2. COMMIT WITH SNAPSHOT;

Update table and declare snapshot S2
3. BEGIN;
4. DELETE FROM LoggedIn WHERE l_userid = 'UserA';
5. COMMIT WITH SNAPSHOT;

Update table and declare snapshot S3
6. BEGIN;
7. INSERT INTO LoggedIn (l_userid, l_time, l_country)
     VALUES ('UserD', '2008-11-11 10:08:04', 'UK');
8. COMMIT WITH SNAPSHOT;

Retrospective query
9. SELECT AS OF 1 * FROM LoggedIn;

Query on current state
10. SELECT * FROM LoggedIn;
```

Figure 3: Retro SQL example

RQL queries specify computations using a small set of basic computational mechanisms that compose relational constructs familiar to a SQL programmer. A computation defined by an RQL query iterates over a set of snapshots, runs a SQL query on each snapshot, collects results of the query and performs on the results different combining computations determined by the specific mechanism used.

In specifying the set of snapshots to iterate over, and the snapshot query to be executed in each iteration, RQL uses two auxiliary constructs, a *snapshot table* that holds all the declared snapshot identifiers and timestamps, referred to as *SnapIds*, and a function *current_snapshot* that provides the identifier of the snapshot used in the iteration where the current computation is performed.

The RQL mechanisms *Collate Data*, *Aggregate Data In Variable*, *Aggregate Data In Table* and *Collate Date Into Intervals* are best described operationally by explaining the SQL computation each one performs.

2.1 Collate Data

The RQL mechanism *Collate Data* collects records from multiple snapshots into a table.

```
CollateData(Qs, Qq, T)
```

Collate Data requires three parameters, the queries Qs, Qq and table name T. Query Qs returns a single column containing snapshot identifiers selected from the snapshot table *SnapIds.* The Qs output represent the snapshot set (interval) the programmer is interested in. Query Qq is applied to every snapshot in the snapshot interval. T is the name of the table into which we collate the results of every Qq.

For the first snapshot identifier Sx returned by the Qs, the mechanism issues "CREATE TABLE T AS Qq" within the snapshot Sx. For all the subsequent Sy returned by Qs, "INSERT INTO T Qq" is issued, within snapshot Sy.

The following example collects all the user_ids and the snapshot identifier of the snapshot they appear in.

```
CollateData("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT l_userid, current_snapshot()
FROM LoggedIn", "Result")
```

Collate Data along with the snapshot table *Snapids* and the *current_snapshot()* function provide a general language for implementing any kind of computation by issuing SQL queries on the Collate Data output. However, Collate Data can have a large footprint in terms of the memory required to hold its result, especially when the Qs set and Qq result set are large.

The two aggregation mechanisms we present next, allow to reduce the memory footprint for RQL computations that need to aggregate results over snapshots.

2.2 Aggregate Data In Variable

The first aggregation mechanism *Aggregate Data In Variable* applies an aggregate function on a single element across multiple snapshots.

```
AggregateDataInVariable(Qs, Qq, T, AggFunc)
```

In addition to the queries Qs, Qq and table name T, it requires an aggregate function as a parameter and it expects from Qq to return a single row and a single column. For the first snapshot identifier Sx returned by Qs, we execute Qq on snapshot Sx and save the single value in a variable V1. For all subsequent identifiers Sy returned by Qs, we issue Qq on snapshot Sy, save the single value in a variable V2 and update V1 as AggFunc(V1, V2). Finally, we store the result in the table T.

The following example shows how we can count the number of snapshots in which a tuple appears. For example, we want to count the number of snapshots in which user *UserB* is logged in.

```
AggregateDataInVariable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT 1 FROM LoggedIn
WHERE l_userid = 'UserB', "Result", "sum")
```

In the next example, we want to find the first occurrence of the same user.

```
AggregateDataInVariable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT current_snapshot()  FROM LoggedIn
WHERE l_userid = 'UserB' ", "Result", "min")
```

2.3 Aggregate Data In Table

The mechanism *Aggregate Data In Table* provides the ability to apply aggregate functions on records of multiple columns across snapshots.

```
AggregateDataInTable(Qs, Qq, T, ListOfColFuncPairs)
```

The additional required parameter is a list of pairs of column names and aggregate functions.

For the first snapshot identifier Sx returned by Qs, we create a table T and insert the Qq output. For all the subsequent identifiers Sy returned by Qs we issue the query Qq and for each record in its output we search in table T to find a tuple with the same values in columns not included in the ListOfColFuncPairs. If such a record exists we perform the required computation on the values in columns of ListOfColFuncPairs, otherwise we insert into T the record returned by Qq. Note, the Aggregate Data In Table queries can be considered as across time GROUP BY queries where the grouping columns are the columns of the Qq output not appearing in the ListOfColFuncPairs. So, for the aggregation across snapshots to be well defined, Qq should never return two records that coincide on all the values in the grouping columns.

The following example shows how we can find the first time that each user has logged in.

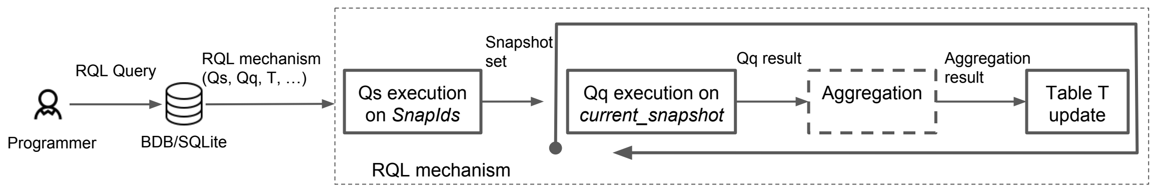

Figure 4: General structure of RQL computations

```
AggregateDataInTable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT l_userid, l_time FROM LoggedIn",
"Result", "(l_time,min)")
```

The next example shows how we can compute, for each country, what is the maximum number of users who are simultaneously logged in, from that country.

```
AggregateDataInTable("SELECT snap_id FROM SnapIds",
"SELECT l_country, COUNT(*) AS c  FROM LoggedIn
GROUP BY l_country",  "Result", "(c,max)")
```

In order for *Aggregate Data in Variable* and *Aggregate Data in Table* to work as described, we require the aggregate function to satisfy certain mathematical properties. Formally, the aggregate function must be definable by an abelian monoid (X, op, e) where *X* is the domain of values, *op* is an associative and commutative binary operation and *e* is the identity element. Most SQL aggregate functions e.g. min, max, count and sum, satisfy the requirement but some, e.g. average, and aggregations over distinct elements e.g. count distinct or sum distinct do not. Because average is widely used in SQL, our aggregation mechanisms implement a simple extension that support average as a special case. Aggregations over distinct elements can use the Collate Data mechanism to return a column containing the elements and then use SQL to perform the needed aggregation. Of course, such approach may not reduce the memory footprint of the result computation.

2.4 Collate Data Into Intervals

The mechanism *Collate Data Into Intervals* creates an alternative, potentially more compact snapshot data representation that resembles the traditional record lifetime representation used by temporal databases, and could be used by applications that expect this kind of representation. When the snapshots are taken frequently, it is likely that the same record appears in many consecutive snapshots. The mechanism collects records from multiple snapshots into intervals which indicate the lifetime of the records. This is achieved by creating two attributes in our result table and storing the *start_snapshot* and the *end_snapshot* for each record's lifetime.

```
CollateDataIntoIntervals(Qs, Qq, T)
```

Collate Data Into Intervals requires the same parameters as Collate Data. For the first snapshot identifier Sx returned by the Qs, we create the table T and insert the records returned by Qq with start_snapshot and end_snapshot for each record set to Sx. For all the subsequent Sy returned by Qs we issue the query Qq and for each record in its output we search in table T to find a tuple with the same values in all columns except the start_snapshot and end_snapshot. If a tuple exists and if its end_snapshot value is the same as the snapshot identifier of the previous iteration we update the end_snapshot to Sy, otherwise we insert a new tuple with start_snapshot and end_snapshot set to Sy.

The following example calculates the interval during which each user was logged in.

```
CollateDataIntoIntervals("SELECT snap_id FROM SnapIds",
"SELECT l_userid FROM LoggedIn", "Result")
```

Figure 4 provides the general structure of our RQL computations. The *Aggregation* part is bypassed in case of *Collate Data.*

3 IMPLEMENTATION

An RQL computation iterates (loops) over the snapshots in the snapshot set defined by the parameter query Qs, and for each snapshot it executes a "loop body" that invokes the query Qq on this snapshot, processing its results in a way specific to each mechanism. This section explains how we implement this computation in SQLite/BDB Retro system using SQLite UDF. We highlight the salient points of the implementation, including the cross snapshot iteration with constructs *current_snapshot*() and *Snapids*, and the processing of snapshot query results.

We create the SQL program for RQL computation by composing the Qs and Qq query programs using the callback infrastructure provided by SQLite UDF. The infrastructure allows to interpose a UDF callback function on a SELECT statement so that the callback is invoked for each element of a set returned by the SELECT.

We define the "loop body" of our computation in a UDF callback function, providing for each mechanism a mechanism-specific callback, and iterate over snapshots by interposing the "loop body" callback on the SELECT statement for Qs.

The following SQLite statement shows the general syntax used by our implementation for an RQL mechanism.

```
SELECT rql_udf (snap_id, Qq, T, ...)
FROM SnapIds WHERE...;
```

By issuing this statement to SQLite, we achieve the iteration over the snapshot identifiers in the table *SnapIds* returned by the SELECT (i.e. Qs), where for each returned snapshot identifier, SQLite invokes the "loop body" defined by the UDF callback *rql_udf*. Figure 5 shows the general structure of the resulting computation.

The UDF argument *snap_id* is filled at runtime by SQLite with values returned by Qs in each iteration, the other parameters, including the string defining the Qq query, the table name T, and additional parameters needed for the aggregation mechanisms are specified by the programmer.

Inside the "loop body" UDF, we treat the parameter *snap_id* as "loop index". The "loop body" UDF uses Retro to run the query Qq on snapshot *snap_id* in every iteration it gets invoked.

To run on a snapshot *snap_id*, Retro requires a query to be in the form of "SELECT AS OF snap_id ...". Furthermore, the SQL program Qq may include the function *current_snapshot*(), explained in Section 2, that denotes the snapshot identifier of the current iteration. Therefore, as a first step, our "loop body" UDF rewrites the Qq, binding it to the value of "loop index" *snap_id*.

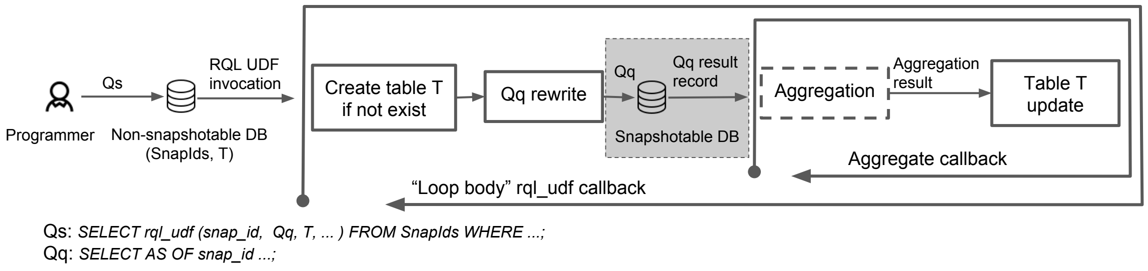
Qs
RQL UDF invocation
Programmer
Non-snapshotable DB (SnapIds, T)
Create table T if not exist
Qq rewrite
Qq
Qq result record
Snapshotable DB
Aggregation
Aggregation result
Table T update
Aggregate callback
"Loop body" rql_udf callback
Qs: SELECT rql_udf (snap_id, Qq, T, ...) FROM SnapIds WHERE ...;
Qq: SELECT AS OF snap_id ...;

Figure 5: Implementation structure of RQL computations

The rewriting involves adding the "AS OF snap_id" extension, and replacing every occurrence of current_snapshot() function with the value of *snap_id*.

For example, for the iteration with *snap_id* value of Si, the following Qq query typed by the programmer:

```
SELECT DISTINCT current_snapshot() FROM LoggedIn
WHERE l_userid = 'UserB';
```

will be rewritten by the RQL UDF to take the following form:

```
SELECT AS OF Si DISTINCT Si FROM LoggedIn
WHERE l_userid = 'UserB';
```

After Qq is rewritten, the UDF issues it to SQLite, invoking the *sqlite3_exec* function from the SQLite API. The "loop body" UDF then proceeds to process the results across snapshots in a mechanism-specific manner, as shown in Figure 5. Note the shaded area depicts the Qq computation run by Retro within a single snapshot.

Consider an example *Collate Data* query specified below:

```
SELECT CollateData(snap_id,
"SELECT DISTINCT l_userid, current_snapshot() AS sid
FROM LoggedIn", "Result")  FROM SnapIds;
```

Since the mechanism does not perform any computation besides Qq, its "loop body" UDF only utilizes the SQLite API. That is, Collate Data UDF callback executes SQL statements using the *sqlite3_exec* function to insert Qq results into the result table T.

The UDF callbacks that implement our aggregate mechanisms are more involved since they need to implement aggregation. Consider and example of *Aggregate Data in Variable* specified as follows:

```
SELECT AggregateDataInVariable(snap_id,
"SELECT DISTINCT current_snapshot()  AS sid
FROM LoggedIn WHERE l_userid = 'UserB' ",
"Result", "min")  FROM SnapIds;
```

In the first iteration, its "loop body" UDF creates a table Result with attribute sid. It then rewrites Qq and invokes the *sqlite3_exec* function from the SQLite API to run the Qq. One of the arguments of sqlite3_exec is a callback function which gets invoked for every tuple returned by the Qq, providing access to the returned tuple. In this callback function we implement the aggregate computation for Aggregate Data In Variable, directly following he specification in the previous section.

Consider next an example of *Aggregate Data in Table* specified as follows.

```
SELECT AggregateDataInTable(snap_id,
"SELECT l_country, COUNT(*) AS c FROM LoggedIn
GROUP BY l_country", "Result", "(c,max)")  FROM SnapIds;
```

Here the first "loop body" iteration creates the table Result with attributes l_country and c. It then rewrites and issues Qq and for every tuple returned by the Qq a callback function is invoked that inserts the tuple in the table Result. At the end of the first "loop body" iteration we also create an index on Result using as key the values in non-aggregating columns, in this case l_country. In subsequent iterations, for every record returned by the Qq, in the callback function we utilize the index to search in table Result. If a tuple with the same values in non-aggregating columns is present, we apply the aggregate functions on the columns specified in the ListOfColFuncPairs and update the record in the result table accordingly, otherwise we insert a new tuple.

We have also experimented with alternative Aggregate Data in Table implementation using a sort-merge based algorithm that turned out to be costlier.

For brevity, we omit the implementation details of *Collate Data Into Intervals*. It is implemented similarly to *Aggregate Data in Table* but instead of applying an aggregate function we check whether we need to update the record's lifetime or insert a new tuple in the result table.

Note that RQL mechanisms by default create the result table T as temporary non-snapshotable table. However, it can be created as persistent if the programmer decides otherwise.

We now briefly consider the construct *Snapids*. It is currently implemented at application level to support user friendly snapshot names. Also, it is stored in a separate SQLite database than application data because it is a non-snapshotable persistent table (whereas the rest of the data are snapshotable). Every time the application declares a snapshot and gets back the snapshot identifier, it inserts the identifier in the SnapIds along with a timestamp and any application meaningful information the programmer needs to later refer to the snapshot. The update operations on SnapIds table are transactional. Note, that concurrent updates to SnapIds table and RQL queries do not block each other since Retro runs snapshot queries as read-only snapshot transactions taking advantage of MVCC concurrency control in BDB, as we explain in section 4. Nevertheless, updating Snapids in a transaction adds overhead so we are currently working on an internal implementation to reduce this overhead.

4 RETRO SNAPSHOT SYSTEM

We briefly describe Retro, the snapshot system used in RQL. Our goal is to explain how the cost of snapshot query is impacted by the method of incremental page-level snapshot creation and indexing, and the different update workloads. The complete description of Retro system can be found in [21–23].

Retro snapshot system is implemented as a small set of modular extensions to the Berkeley DB transactional storage manager. The extensions interpose on transaction *commit, page flush, page fetch and recovery* operations. The implementation at the storage manager level allows to create transactionally consistent, recoverable snapshots efficiently, without blocking application transactions, by exploiting the BDB MVCC concurrency control and recovery mechanisms [22].

The snapshot system interface supports two operations, *snapshot declaration* and *snapshot query*, that can be exposed to applications in a language-specific way. Section 2 presented the SQL interface. A Retro snapshot is a set of immutable logical data pages that reflect the entire consistent database state, including the catalog and indexes, at snapshot declaration point. This allows to run on a snapshot any database query q that could run in the database at the snapshot declaration point. The snapshots are captured using a page-level copy-on-write technique (COW) that copies out and saves snapshot pages incrementally as transactions commit modifications to pages, following a snapshot declaration. At transaction commit time, Retro identifies any page P that is modified for the first time following the declaration of a snapshot S and copies out the pre-modification state (pre-state) of P. The pre-state corresponds to the state of P as-of snapshot S. If this modification of P is also the first since an earlier declared snapshot S', the pre-state is shared by S'.

Retro accumulates the copied-out pre-states in memory and writes them to an on-disk log-structured snapshot archive called *Pagelog* when the database flushes updates. The pre-states are indexed at low cost by simply recording a mapping that associates a snapshot page P with its *Pagelog* location. Retro writes the mappings to an on-disk log-structured list called *Maplog* [23]. When a snapshot S is needed, an efficient scan of Maplog allows to construct a snapshot page table $SPT(S)$ that maps every page P in snapshot S to its location in *Pagelog*. The scan length is $nlog(n)$ where n is the number of pages in the snapshot, independent of snapshot history length [23].

To run a query q on a snapshot S Retro interposes on the database page $fetch$ operation. When q requests a page P, Retro looks up page location in $SPT(S)$ and fetches P from Pagelog, the same way q would fetch P from the database if it was running on the current database state.

Retro caches snapshot pages in a buffer cache along with the database pages and needs extra cache memory to hold the snapshot pages when running snapshot queries. While we expect the database pages to reside in memory given today's large memories, we do not expect snapshot pages to fully reside in memory because with long snapshot histories Pagelog can grow very large, limited only by the available disk space. For this reason, even with a large snapshot cache, when a query runs on a snapshot that has not been accessed recently, we expect the snapshot page cache hit rate to be low. The I/O cost of a snapshot query therefore depends on the number of pages it fetches from Pagelog when the page is not present in the cache.

Retro allows to run snapshot queries concurrently with current state queries and update transactions. It relies on the BDB concurrency control scheme MVCC to avoid snapshot queries from interfering with updates. Consider a snapshot query q that runs over snapshot S following the snapshot declaration. At this point the snapshot shares all its pages with the database. Consider a page P requested by q and a concurrent transaction T that modifies page P. Retro runs q as a read-only MVCC transaction relying on MVCC to provide q with the unmodified pre-state of P to the end of q, without interfering with T.

We now consider the amount of Pagelog I/O needed by a snapshot query. Consider a set of update transactions $T1, ...Tn$ that updates every page in the database following the declaration of snapshot S. We call such transaction set an overwrite cycle of S. After the overwrite cycle of S completes, the entire state of S and all the snapshots declared before S, is copied out into Pagelog. A snapshot declared a long time ago is likely to have a complete overwrite cycle so a query running over old snapshots fetches all its pages not present in a cache from Pagelog. On the other hand, consider a snapshot S that has been declared recently. Some of the pages of the database likely have not been modified since snapshot S declaration, so its overwrite cycle is incomplete. If a page P has not been modified since snapshot S declaration, S shares P with the database. A snapshot query running on S and requesting a shared page P will fetch P from the database. Therefore, we expect a query running on recent snapshots to have less Pagelog I/O.

Finally, consider two snapshots $S1$ and $S2$ declared consecutively, and the set of update transactions $T1, ...Tk$ committed between $S1$ and $S2$. Note that snapshot $S1$ and $S2$ share all their pages except those pages modified by $T1, ..Tk$. Let $shared(S1, S2)$ be the set of pages shared by $S1$ and $S2$, and $diff(S1, S2)$ be the set of pages that are not shared by $S1$ and $S2$. Consider a snapshot query q running consecutively over two old snapshots $S1$ and $S2$ using a cache large enough to hold all the snapshot pages requested by q. If snapshots $S1$ and $S2$ have not been accessed for a long time, all pages of $S1$ will need to be fetched from Pagelog but any page P in $shared(S1, S2)$ needs to be fetched from Pagelog only once since P will be in the cache after q runs on $S1$. On the other hand, any page Q in $diff(S1, S2)$ requested by q running on $S2$ likely will need to be fetched from Pagelog. The size of $diff(S1, S2)$ will therefore determine the cache miss rate for q running on $S2$. The size of $diff(S1, S2)$ is determined by the transaction update workload, i.e. by how many pages the transactions modify, and by the frequency of snapshot declarations, i.e. how many transaction apart are the declarations $S1$ and $S2$. If transactions modify many pages and snapshot declaration are infrequent, $diff(S1, S2)$ will be large, but if transactions modify few pages and snapshot declarations are frequent, $diff(S1, S2)$ will be small and $S1$ and $S2$ will share most of the pages.

5 PERFORMANCE EVALUATION

This section presents an experimental study that characterizes the performance of RQL computations. We aim to explain RQL performance in terms that are familiar to a SQL programmer. Since the programmer specifies an RQL query r by providing SQL programs Qs, Qq and the aggregation functions, we consider how the performance characteristics of these SQL programs impact the performance of r.

To explain the performance of RQL we need to characterize the costs of a computation that iterates over snapshots. The performance of a snapshot computation that runs over a stand-alone single snapshot has already been studied [21]. However, as we explain in Section 4 because of snapshot page sharing, a snapshot computation that runs as one of the RQL iterations can have a different performance than a snapshot computation that runs on a stand-alone snapshot. Our study evaluates the benefit of page sharing for different transaction update workloads and explains how the benefit of sharing depends on the properties of the snapshot set Qs, and whether the snapshot computation Qq is I/O or computation intensive. Our experiments also analyze the memory requirements of different RQL mechanism and quantify the memory benefits of the aggregation mechanisms.

Our experiments run our implementation of RQL in the Retro snapshot system integrated with BDB SQLite version 5.3.21.

Parameters	Notations	Description
Update Workload	UW15	Delete and insert 15K orders and their lineitem records per snapshot
	UW30	Delete and insert 30K orders and their lineitem records per snapshot
Query Qs	Qs_N	Query that determines the snapshot interval length N
Query Qq	Qq_io	SELECT COUNT(*) FROM orders WHERE o_orderstatus = 'O' ;
	Qq_cpu	SELECT SUM(l_extendedprice) AS revenue FROM lineitem, part WHERE p_partkey = l_partkey and p_type = 'STANDARD POLISHED TIN';
	Qq_collate	SELECT o_orderkey FROM orders WHERE o_orderdate < '[DATE]';
	Qq_agg	SELECT o_custkey, COUNT(*) AS cn, AVG(o_totalprice) AS av FROM orders GROUP BY o_custkey;
	Qq_int	SELECT o_orderkey, o_custkey FROM orders;
RQL UDF		CollateData (Qs, Qq, T)
		AggregateDataInVariable (Qs, Qq, T, AggFunc)
		AggregateDataInTable (Qs, Qq, T, ListOfColFuncPairs)
		CollateDataIntoIntervals (Qs, Qq, T)
Aggregate function		MIN, MAX, SUM, COUNT, AVG

Table 1: Parameters and notations

The hardware platform consists of 2 hexa-core Intel Xeon CPU clocked at 2.50 GHz with Hyper-Threading enabled, 2 Intel 400 GB SATA SSD and 64 GB of RAM. The operating system is Red Hat Enterprise Linux Server release 6.8 (Santiago), x86_64 and the file system is formatted with ext4.

RQL performance depends on the database update workload, the in-snapshot query workload, and the type of RQL computation that combines the results of the snapshot queries. We expect the in-snapshot queries to include both *native* queries the application runs in the current state, e.g. when performing auditing, and *ad − hoc* queries formulated after the fact, e.g. when performing fact checking. The native queries are more likely to have native indexes captured in the snapshot. The ad-hoc queries may need to create the indexes at RQL execution time. Our experiments use both native and ad-hoc queries.

The database we use for our experiments is a TPC-H database. TPC-H [6] is a standard decision support benchmark consisting of tables designed to be business relevant and the database schema includes tables with information about Customers, Orders, Lineitems, etc. We create the database by using the TPC-H *dbgen* tool to produce the initial state of the database with size of 1.4 GB (the default size) without additional indices. The size increases accordingly when indices are included.

In order to create a snapshot history, we utilize the TPC-H refresh functions which produce a set of order identifiers for deletion and a set of order records along with Lineitem records associated with the orders for insertion. Our update workload program receives as input the TPC-H refresh function output, updates the database by deleting and inserting a certain number of Orders and their Lineitem records and creates snapshots. Between two consecutive snapshot declarations a constant number of orders and their associated records are inserted and deleted making it easier to interpret the performance results and memory requirements.

We consider two update workloads that delete and insert different amount of data, as defined in Table 1. These update workloads generate different $diff(S1, S2)$, the amounts of non-shared data between two consecutive snapshots S1 and S2, and affect how frequently the database gets overwritten, i.e. the length of the snapshot overwrite cycle. The UW30 overwrites the database every 50 snapshots while the UW15 overwrites every 100 snapshots, so the $diff(S1, S2)$ in UW30 is double the size of UW15.

We assume the current state database is memory resident, and the snapshot pages are stored in Pagelog on the SSD. We achieve the expected snapshot cache behavior by assuming the snapshot page cache is empty at the start of an RQL query, and assume the cache can hold the snapshot pages requested by a single RQL query, except when discussing memory costs.

For our experiments, we define custom queries that stress different RQL costs in both native and ad-hoc queries. Table 1 reports all the queries used throughout our performance evaluation. We explain the characteristics of each query when describing the experiments that use them. The reason we don't provide any experimental results using TPC-H queries is because their complexity makes them CPU intensive and does not allow us to stress and focus on a single RQL cost each time.

5.1 I/O intensive queries: Impact of snapshot sharing

Our first set of experiments considers the impact of snapshot sharing on the I/O costs of an RQL query for old and recent snapshots.

We first consider old snapshots. When all the snapshots included in the set defined by Qs are old, the first iteration fetches from the Pagelog disk all the pages it needs and is likely to fetch the highest number of pages compared to the subsequent iterations. We refer to the first iteration as *cold*, and to the subsequent iterations as *hot*. Note, the number of pages fetched by a cold iteration is identical to a stand-alone snapshot query and is determined by the code of *Qq*. The maximum number of pages potentially fetched by the subsequent *hot* iterations depends on two factors, the $diff(S1, S2)$ in the update workload explained in Section 4 and how far apart are the snapshots in the hot iterations, determined by the number of snapshots skipped in the Qs query. In the extreme case, if Qs defines a skip that exceeds the snapshot overwrite cycle length, the performance of a hot iteration will be no different than a performance of a cold iteration. We refer to an RQL query run where all iterations are cold

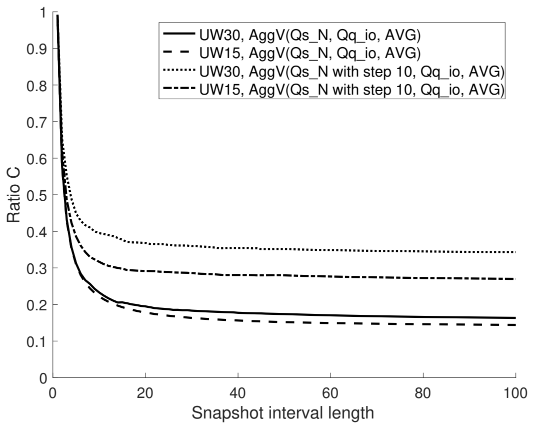

Figure 6: Ratio C with old snapshots: impact of sharing between snapshots.

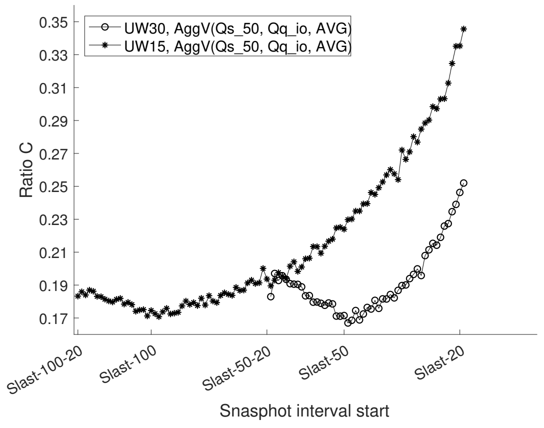

Figure 7: Ratio C with recent snapshots: impact of sharing with current state.

as $all - cold$. In general, we expect sharing to improve the RQL query performance compared to $all - cold$ but the amount of improvement depends on the snapshot interval length since for short intervals the performance of cold iteration may dominate.

The combined impact of sharing can be succinctly captured using a ratio C of latency of an RQL query r with a given Qq and Qs, to the latency of an $all - cold$ run with the same number of snapshots.

Figure 6 shows the ratio C as the number of snapshots in the interval increases, for update workloads with different amount of sharing, UW30 and UW15 and different distance between consecutive iterations. To isolate the impact of sharing on total I/O costs we use a computationally light RQL Aggregate Data in Variable query with an I/O intensive and computationally light *Qq_io*. At each RQL iteration, Qq scans the table Orders and returns the number of open orders for the current snapshot and the RQL query computes the average number of open orders per snapshot.

Since all snapshots are old, the cost of the $all-cold$ run remains constant. Sharing increases as we move from update workload UW30 to UW15 and when the number of skipped snapshots drops from 10 to 1. The ratio C drops with increased sharing reflecting the RQL query latency decrease compared to $all - cold$ run. Overall, C is high for short intervals since sharing makes little difference as the cost of the first cold iteration dominates the RQL query latency. For sufficiently long intervals however (more than 20 snapshots) C converges to a constant as the cost of the cold iteration stops being the dominant cost, and the RQL query latency is fully determined by sharing. The first two bars in Figure 8 break down the cost of the cold and hot iteration for UW30 showing the impact of sharing on the absolute I/O costs in the hot and cold iterations in this workload.

We next consider recent snapshots. When the set of snapshots defined by Qs includes recent snapshots, the number of pages fetched from *Pagelog* in a given snapshot iteration is impacted by an additional factor, namely by the number of pages the snapshot shares with the current state of the database since page shared with the current state is fetched from the main memory, as explained in Section 4. Therefore, the number of pages fetched by a snapshot iteration decreases as snapshot gets closer to the end of our snapshot history and snapshots share more pages with the current state.

The ratio C, defined as ratio of measured RQL query cost to the cost of $all - cold$ run, can be used to explain the additional impact of sharing pages with the current state database. Due to the database overlap, the cost of the cold iteration depends on the starting point of the interval. Therefore, the cost of $all - cold$ run for two intervals with the same number snapshots drops when interval starts at a more recent snapshot. Figure 7 shows the ratio $C(x)$ for fixed size interval of consecutive snapshots (skip 1) starting at snapshot x, for intervals that include recent snapshots. We show two update workloads UW30 and UW15 exhibiting different inter-snapshot sharing $diff(S1, S2)$.

Assuming Slast is the most recent declared snapshot, and OverwriteCycle is the overwrite cycle length for a given update workload UW, the interval starting at snapshot Slast-OverwriteCycle-20 is the earliest interval to include a snapshot sharing pages with the database. OverwriteCycle is 100 in case of UW15 and 50 in case of UW30. We consider therefore intervals starting at $x = Slast - 100 - 20$ and later for UW15, and $x = Slast - 50 - 20$ and later for UW30.

For intervals starting with an old snapshot x, C(x) drops as x becomes more recent since the measured RQL cost decreases but the cost of all-cold run remains constant. For intervals starting with a recent snapshot x, C(x) increases since the cost of all-cold run decreases and converges to the measured RQL cost, as both cost drop as intervals become more recent.

In absolute terms, RQL cost decreases sharply as we move to more recent intervals as shown in Figure 8, where an iteration on a more recent snapshot Slast-25 performs significantly better that on older Slast-50 (UW30).

Where cold iteration cost can be a dominant factor for old snapshot intervals since it can fetch substantial number of pages from Pagelog, this is not so for recent intervals where cold iteration fetches a substantial number of pages from the database so the dominating factor for intervals of recent snapshot is the sharing with the current state of the database.

5.2 CPU intensive queries

We expect snapshot page sharing to have less impact on CPU intensive RQL queries.

We consider two kinds of CPU intensive RQL queries. In one case, an RQL query issues a computationally heavy Qq so that SQL query execution time is the dominant cost, in the other case

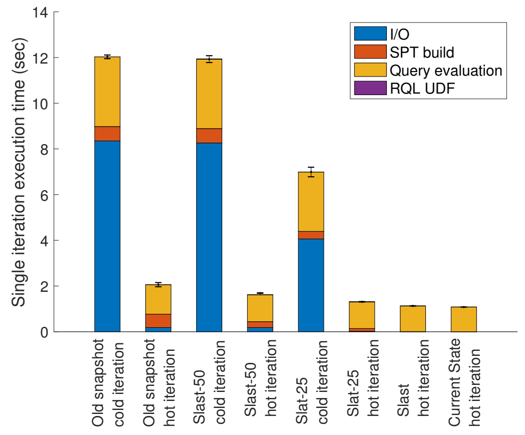

Figure 8: Single iterations cost for the RQL query AggregateDataInVariable(Qs_50, Qq_io, T, AVG) with update workload UW30.

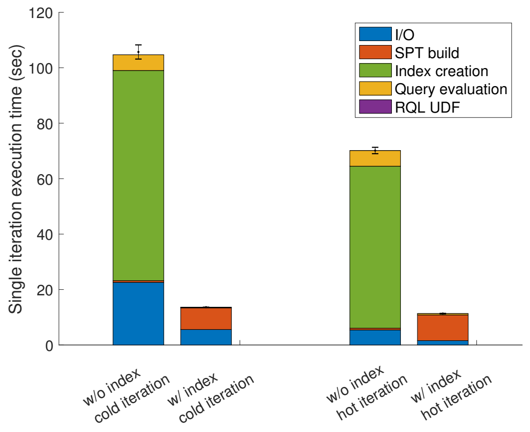

Figure 9: Single iteration cost for the RQL query AggregateDataInVariable(Qs_50, Qq_cpu, T, AVG) with update workload UW30.

RQL defines a Qq with a large output size. In the latter case the RQL UDF becomes the dominant cost.

For computationally intensive Qq, especially with join operations, if there is no native index in the snapshot, SQLite creates covering indices to assist the query evaluation. An experiment reported in Figure 9 shows that index creation always dominates RQL cost. Our experiment uses Aggregate Data in Variable to avoid introducing significant RQL UDF cost and Qq_cpu. The Qq performs a join operation on tables Part and Lineitem and returns the revenue of the orders that include an item of a certain type. SQLite decides the building of a covering index on table Lineitem as part of the query execution plan. Note, unlike for I/O intensive queries, here the cost difference between a cold and hot iteration is less since I/O cost is small part of total Qq execution cost.

The Qq will not always be an $ad-hoc$ query in the database workload and may have a native index built by the programmer. We evaluate the cost of the same RQL query where a native index is available in the snapshot. As shown in Figure 9 the I/O cost due

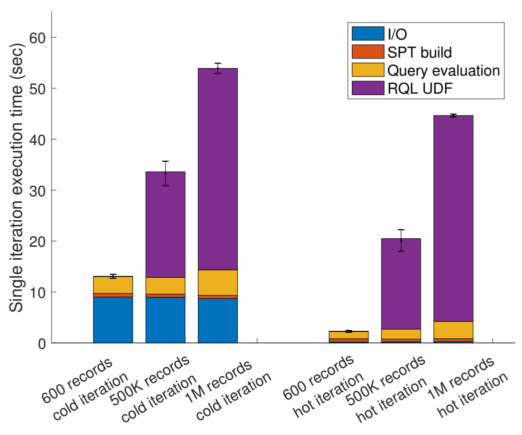

Figure 10: Single iteration cost for CollateData(Qs_50, Qq_collate, T) with varying Qq output size with UW30.

to index creation drops but instead the SPT build cost increase since an index increases the size of the database and the Pagelog.

Our last experiment considers Qq that returns as a result a large number of records. This increases RQL UDF cost since SQlite UDFs invokes a callback function to perform operations for every record returned by the Qq. These operations are either insert operations in case of Collate Data or aggregations in case of Aggregate Data in Table.

Figure 10 shows the RQL performance where the CPU cost is dominated by the cost of UDF for Collate Data with a computationally light Qq query (Qq_collate in Table 1). The query scans the table Orders and returns the orders with order date less than a certain date. It has a single predicate which we vary to control the query output size. As in earlier CPU intensive queries, sharing has minimal impact on RQL cost.

5.3 Memory costs

The memory requirements of an RQL computation include two parts, the memory to hold the snapshot pages requested by Qq iterations, and the memory needed for result computation and to hold the result table T. Since Qq iterates over snapshots sequentially, the first part is independent of snapshot set size, and is essentially the memory needed to run Qq over a single snapshot, which includes the snapshot pages holding the working set of Qq plus the snapshot metadata structures such as Maplog and SPT(S). The memory costs of single snapshot computation have been studied before [21]. Here we consider the memory costs of RQL mechanism result computation for different mechanisms.

RQL can support general computations over snapshots using Collate Data and running SQL computations over the results. However, such a method could incur high memory cost when Qs requires to compute over large number of snapshots. Memory cost can be reduced for RQL computations that perform aggregations on records across snapshots by using RQL aggregation mechanisms.

Consider the case where given the table Orders of TPC-H the user wants to find out, for each customer, what is maximum number of orders placed in a single snapshot by the customer and their average total price. This can be accomplished by running a single Aggregate Data in Table.

```
AggregateDataInTable(Qs_50, Qq_agg, T, (MAX,cn))
```

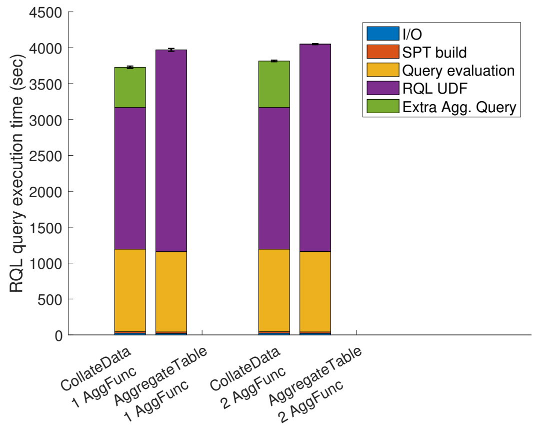

Figure 11: Comparison of RQL queries that producing the same result using Aggregate Data in Table and Collate Data for different number of aggregations.

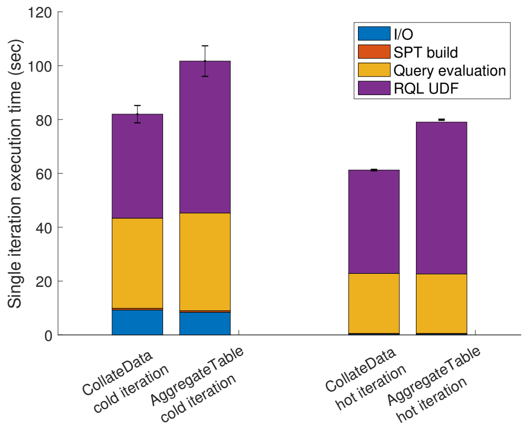

Figure 12: Single iteration cost for the RQL queries CollateData(Qs_50, Qq_agg) and AggregateDataInTable(Qs_50, Qq_agg, (MAX,cn)) and update workload UW30.

The same result can be produced using Collate Data by first collecting how many orders each customer has placed in every snapshot in the given snapshot interval along with their average total price.

```
CollateData(Qs_50, Qq_agg, T)
```

A single SQL query can then compute the final result.

```
SELECT o_custkey, MAX(cn), av FROM T;
```

In Figure 11, the first two bars compare the performance of the two approaches for Qs that iterates over 50 snapshots. The RQL UDF cost dominates the RQL cost because the query's output is approximately 1M of records for every snapshot. The Collate Data performs slightly better than Aggregate Data in Table.

However, Aggregate Data in Table has a significantly smaller memory footprint. Where Collate Data result table is more than 1GB, the Aggregate Data in Table result table is less than 100MB. Aggregate Data in Table achieves the memory footprint reduction for only 6% overhead in total execution time. Importantly, the memory footprint of Aggregate Data in Table is independent of Qs since we don't expect the result table to grow significantly after each iteration, whereas Collate Data inserts the entire Qq output at each iteration.

The reason why Aggregate Data in Table is costlier than Collate Data can be explained in Figure 12 which shows single cold and hot iteration of the two RQL queries issuing the same Qq_agg. The first cold iteration is more expensive in case of Aggregate Data in Table even though they insert exactly the same number of records in the result table, because the Aggregate Data in Table creates an index on its result table. In addition, the insert operations in Collate Data is slightly cheaper than in Aggregate Data in Table because the result table of Collate Data does not have a primary key. Maintaining a primary key on a table introduces overhead to the insert operations. The reason why hot iteration is more expensive in case of Aggregate Data in Table is because we overall perform more operations. The Qq query returns approximately 1M records so, Collate Data in a single iteration executes 1M insert operations. Aggregate Data in Table on the other hand executes 1M select operation on the result table and a number of inserts or updates given the aggregation it performs. In this experiment is perform approximately 22K inserts or updates.

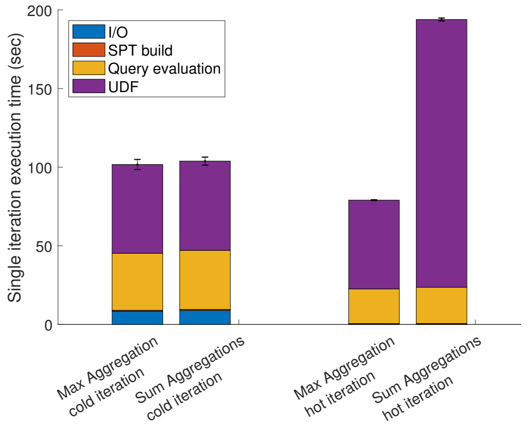

Figure 13: Single iteration cost for Aggregate Data in Table for different aggregate function

Since the Aggregate Data in Table is able to aggregate on multiple columns Figure 11 also evaluates the cost of additional aggregation. The second aggregation calculates the maximum among averages of the total price using the following Aggregate Data in Table query:

```
AggregateDataInTable(Qs_50, Qq_agg, (MAX,cn):(MAX,av))
```

And the extra query required by the Collate Date will be:

```
SELECT o_custkey, MAX(cn), MAX(av) FROM collate_result;
```

As we can see adding extra aggregation does introduce significant overhead.

For some aggregate functions like SUM and COUNT the Aggregate Data in Table have to update its result table for every record returned by the Qq. For these aggregations, we expect the hot iterations to have increased cost. Figure 13 compares Aggregate Data in Table RQL queries that apply different aggregate functions, MAX and SUM on the results of the Qq_agg. The cold iterations perform the same because they do the same initial insert operations and index creation. The hot iterations do the same number of search operations on the result table but in case of SUM they will do significantly more updates, 1M versus 22K in case of MAX.

Next, we briefly consider the Collate Data Into Intervals mechanism, comparing it to Collate Data. We focus on the memory costs since the non-memory costs of Collate Data Into Intervals closely resemble Aggregate Data In Table.

The result table size of Collate Data Into Intervals depends on the size of Qq, and the lifetimes of records, i.e. the number of updated and deleted records between consecutive snapshots. The mechanism also needs memory to store the index. The result table size of Collate Data only depends on the size of Qq. Our experiment uses a Qs that defines an interval of 50 consecutive snapshots, a Qq that returns 1.5 millions records in each snapshot (Qq_int in Table 1), and four different update workloads UW7.5, UW15, UW30, and UW60 that respectively modify between consecutive snapshots 7,500, 15,000, 30,000, and 60,000 order records.

For Collate Data mechanism, the result table has 75M records of total size more than 3GB. For Collate Data Into Intervals, the result table has respectively 1.86M records (89 MB), 2.3M records (105MB), 2.97M records (138 MB) and 4.4M records (204 MB) for UW7.5, UW15, UW30 and UW60 update workloads. For each workload, the mechanism requires about 50% additional memory to hold the index during the computation. Interestingly, with our workloads, increasing the number of modified records between snapshots does not increase the result table size proportionally. Overall the experiment shows Collate Data Into Intervals can substantially reduce memory costs, confirming the known space saving properties of record lifetime based snapshot representation compared to naive page-level representation [24].

6 RELATED WORK

Computations over past state have been investigated in depth by a rich body of work on temporal databases [17, 27]. The most common data model adopted by the temporal databases is the extension of the relational data model with time-stamps that record lifetimes of record values via attributes indicating the start and end time [17]. The temporal dimensions can vary depending on whether a database supports the *transaction time* or the *valid time*, or both (bi-temporal databases). Temporal databases have not been widely adopted by general applications because of poor performance during normal in-production operation [10] and portability concerns.

One of the first commercial databases to support historical data management was Oracle Flashback [18] which allows to store all the modifications in a compressed format using the undo tablespace without affecting application portability. IBM provides support for bi-temporal tables in DB2 [19] by allowing the declaration of additional attributes in temporal tables to indicate the time dimension. Both Flashback and DB2 provide time travel operation to a single point in the past but offer no support for temporal computations over multiple past points which is the main focus for our work. Teradata [1] supports bi-temporal tables by extending the tables with columns representing the time domain and supports a set of temporal computations called temporal aggregations [30]. These computations scan the values which participate in the aggregation at the logical level, to determine how they have changed between different timestamps, and then compute the aggregation on the values changes.

In contrast to the logical record level approaches to past state management, RQL computations run in a snapshot system that manages the past state using a different low-level page-based approach where snapshots expose to application the entire past state of the database, and portions of the state required by the snapshot computation are assembled on demand from the low-level representation. Using the snapshot system RQL provides multi-snapshot computations to applications in a seamless manner as we have explained. Moreover, in terms of expressive power, these computations can compute anything a record-level past state system can compute assuming snapshots capture all updates to the database. In particular, our Collate Data Into Intervals mechanism can create the same time-stamped representation used by the temporal databases. A potential downside of the snapshot system approach is that snapshot representation is less compact than logical record level representation and adds space overhead. However, prior work has shown that a snapshot system can reduce the space overhead substantially without impacting normal in-production application performance, using an adaptive low-level page-diff approach [24], that offers a convenient trade-off between more compact snapshot representation and a higher cost of snapshot reconstruction.

Temporal aggregations apply aggregate functions on relations that evolves over time, e.g. the average salary of an employee over the past certain years. Efficient methods for computing temporal aggregations have been the subject of numerous studies in temporal database research. The implementation of temporal aggregations is challenging in the logical temporal models because in order to aggregate along the time dimension a temporal computation may need to assemble the consistent state of multiple records at each time. Many of the proposed techniques in the literature accelerate this process using sophisticated data-structures that impose a variety of constraints to achieve efficiency. An early work by Kline and Snodgrass uses a non-indexed approach based on a data structure called Aggregation Tree [12]. The approach requires the entire structure to be memory resident, and has a worst case complexity of O(n^2) when the tree is unbalanced. The complexity can be improved in special cases of ordered data. SB-Tree [28] and MVSB-Tree [29] relax the in-memory limitation by introducing disk-based indexes for temporal aggregations. The approach limits the type of aggregate functions if data deletion is expected to SUM, COUNT and AVG. Moreover, the index size can be larger than the database. The TMDA [3] and Timeline Index [9] do not limit the aggregation functions. TMDA however does not support time travel queries. The Timeline Index is an in-memory system that requires large amount of memory to perform well. In contrast, RQL relies on state reconstruction at a lower level, and since each Retro snapshot includes the entire state of the database, RQL computations can aggregate over consistent state seamlessly without restricting types of aggregation and without requiring special temporal indexes. Nevertheless, auxiliary temporal data-structures can potentially be beneficial for repeated RQL computations and we consider this future work.

Similarly to temporal aggregations, temporal join is another challenging computation for temporal databases due to the need to identify all the versions of the join candidates that overlap in the time domain. Like with aggregation, in a snapshot system temporal join poses no addition challenge because the join candidates that overlap in time exist in the same snapshots and the temporal join is executed as if they were in current state.

Several temporal query languages have been adopted by temporal systems. The most notables are the TSQL2 [26] and TQuel [25] which extend SQL and Quel languages respectively. Some of the TSQL2 temporal properties have been adopted in SQL:2011 standard and the Teradata database. T4SQL [5] and TENORS [4] are more recent languages designed based on the time-stamping data model. These languages are not suitable for our snapshot based system because of the different data model. Our SQL UDF based query language is fully compatible with SQL and does not require any change to the current ISO standard version. Languages designed for streaming data management such as CQL [2], resemble temporal computations since stream tuples include timestamps but since they are specialized for computations subject to real-time performance requirements they offer less expressive power compared to the temporal languages and RQL.

Although, RQL computations are most closely related to temporal databases, the snapshot system RQL extends is also similar to versioning systems with linear branching. These systems manage historical data by creating versions of the dataset somewhat similarly to the way Retro uses compact diff-based snapshot [24]. Array data versioning system [20] supports time travel queries and uses a SELECT primitive similar to ours to collect data from multiple versions. Decibel [14] is a branching system for relational datasets and supports time travel queries and multi-version aggregation. Our snapshot system does not support non-linear branching so many of the computation primitives and methods they explore are not applicable to RQL.

A snapshot system called Hyper [11] utilizes hardware-assisted memory snapshots to implement a hybrid OLTP and OLAP relational database system. OLTP and OLAP queries can run in parallel, in separate processes. The analytical queries access transaction consistent snapshots of the current state which are discarded after the queries are evaluated. Hyper does not consider a language for snapshot computations.

7 CONCLUSION

Auditing and other forms of claim checking require applications to compute over multiple past states of their data. The current systems supporting past state computations cannot be easily used by general applications using simple data stores. These applications however can easily use a snapshot system but current snapshots systems do not provide convenient support for multi-snapshot computations. Trying to bridge this gap, we proposed RQL, a retrospective query language that allows programmers to specify multi-snapshot computations in a snapshot system. Our language, implemented as a SQL extension using SQL UDF callbacks in SQLite BDB with Retro snapshot system, provides programmers a convenient way to express computations using the language of the application. Our experimental study evaluates the performance of RQL computations and explains how RQL program parameters, the SQL programs Qq and Qs interact with the page-level copy-on-write snapshot representation. This is the first study explaining the performance of programs running on multiple page-level copy-on-write snapshots. Our future work includes performance optimizations for RQL programs exploring how computations can be shared across multiple snapshots and whether parallelization can be applied.

8 ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation award CNS-1318798.

REFERENCES

[1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 573–578.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal?The International Journal on Very Large Data Bases* 15, 2 (2006), 121–142.

[3] Michael H Böhlen, Johann Gamper, and Christian S Jensen. 2006. Multi-dimensional aggregation for temporal data. In *EDBT*, Vol. 3896. Springer, 257–275.

[4] Cindy Xinmin Chen, Jiejun Kong, and Carlo Zaniolo. 2003. Design and Implementation of a Temporal Extension of SQL. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 689–691.

[5] Carlo Combi, Angelo Montanari, and Giuseppe Pozzi. 2007. The T4Sql Temporal Query Language. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management (CIKM '07)*. ACM, New York, NY, USA, 193–202.

[6] Transaction Processing Performance Council. 2010. TPC-H: Decision Support Benchmark. http://www.tpc.org/tpch. (2010).

[7] William Endressi. 2013. On-line Analytic Processing with Oracle Database 12c. *An Oracle White Paper* (2013).

[8] Hipp, D. R., Kennedy, D. and Mistachkin, J. 2017. SQLite (Version 3.21.0) [Computer software]. (2017). https://www.sqlite.org/

[9] Martin Kaufmann, Peter M Fischer, Norman May, Chang Ge, Anil K Goel, and Donald Kossmann. 2015. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 471–482.

[10] Martin Kaufmann, Peter M Fischer, Norman May, and Donald Kossmann. 2014. Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?. In *EDBT*. 738–749.

[11] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 195–206.

[12] Nick Kline and Richard T Snodgrass. 1995. Computing temporal aggregates. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*. IEEE, 222–231.

[13] David Lomet, Roger Barga, Mohamed F Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2005. Immortal DB: transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 939–941.

[14] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment* 9, 9 (2016), 624–635.

[15] Olson, M. A., Bostic, K. and Seltzer, M. I. 1999. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*. Monterey, CA, USA.

[16] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1771–1775.

[17] Gultekin Ozsoyoglu and Richard T Snodgrass. 1995. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 7, 4 (1995), 513–532.

[18] Ravi Rajamani. 2007. Oracle total recall/flashback data archive. *An Oracle White Paper* 12 (2007).

[19] Cynthia M Saracco, Matthias Nicola, and Lenisha Gandhi. 2010. A matter of time: Temporal data management in DB2 for z. *IBM Corporation, New York* (2010).

[20] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient versioning for scientific array databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 1013–1024.

[21] Ross Shaull. 2013. *Retro: a methodology for retrospection everywhere*. Brandeis University.

[22] Ross Shaull, Liuba Shrira, and Barbara Liskov. 2014. A Modular and Efficient Past State System for Berkeley DB.. In *USENIX Annual Technical Conference*. 157–168.

[23] Ross Shaull, Liuba Shrira, and Hao Xu. 2008. Skippy: a new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 637–648.

[24] Liuba Shrira and Hao Xu. 2006. Thresher: An Efficient Storage Manager for Copy-on-write Snapshots.. In *USENIX Annual Technical Conference, General Track*. 57–70.

[25] Richard Snodgrass. 1987. The temporal query language TQuel. *ACM Transactions on Database Systems (TODS)* 12, 2 (1987), 247–298.

[26] Richard T. Snodgrass (Ed.). 1995. *The TSQL2 Temporal Query Language*. Kluwer.

[27] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. 1993. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc.

[28] Jun Yang and Jennifer Widom. 2001. Incremental computation and maintenance of temporal aggregates. In *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 51–60.

[29] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. 2001. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*.

[30] Xin Zhou. 2011. Processing a temporal aggregate query in a database system. (Aug. 30 2011). US Patent 8,010,554.