SimpleCQL: A Continuous Query Language for SimpleDB

Deepak Narayanan, Tuan Andrew Nguyen, Jeffrey Warren

Abstract

SimpleCQL is an extension to SimpleDB [4] that introduces continuous query semantics from CQL [2]. SimpleCQL supports a wide range of complex SQL-like queries over continuous, unbounded streams of data. The versatility of SimpleCQL is illustrated through three primary examples – aggregation of key statistics from real-time error logs, computation of trends of real-time tweets from Twitter, and computation of real-time advertisement statistics. SimpleCQL is measured against these examples, while other stream data management systems use the Linear Road benchmark [3]. It is important to note that minor adjustments render SimpleCQL capable of supporting the Linear Road benchmark.

We evaluate SimpleCQL and show that its performance is on par with most stream-processing systems. SimpleCQL achieves real-time processing speeds of up to 400k tuples/second based on our benchmarks. Furthermore, this system allows for the processing of complex queries against streaming data.

1. Introduction

Big data often arrives in the form of real-time streams and is generally most valuable at the time of arrival. For example, Advertisement Statistics, based on real-time data provide tremendous business value to an advertisement system. Real-time click-through rates can be used by advertisement auction models to maximize total profits; similarly, real-time budget analysis helps evenly distribute advertisements among available inventory.

One method to process streaming data involves creating custom solutions that fit the specific needs of the given problem. This solution however leads to complicated systems that are not easily applied to general problems. Another method involves using a general stream-processing system such as Apache Storm [5] or Spark Streaming [6]. These systems however can often become unwieldy to use because they lack familiar SQL-like syntax for querying streams of data, which reduces the benefit associated with using such a general stream-processing system.

In this paper we present SimpleCQL, which implements semantics that allows for precise, continuous, SQL-like queries against streaming data. SimpleCQL is built on top of SimpleDB [4], an extremely light-weight and easy-to-use database system; as such, it looks, feels, and performs much like a standard relational database with a structured query language. SimpleCQL supports familiar relational operations such as projections, filters, and joins on continuous, unbounded data streams.

Section 2 motivates the need for such a system and outlines a running example used throughout the paper. Section 3 defines the abstract semantics of a continuous language and provides definitions for streams and relations, the building blocks of SimpleCQL. Section 4 describes the implementation of SimpleCQL. Finally, section 5 evaluates the system using developed benchmarks.

2. Motivation: Advertisement Statistics

This section presents a running example, "Advertisement Statistics", that will be used throughout the paper. This example both motivates the need for a system such as SimpleCQL and illustrates the complex nature of querying continuous streams of data.

A naive advertising architecture involves two streams of data: *advertisement insertions* and *advertisement events*. Advertisement insertions are records of advertisements that have been inserted into a user's page or feed. Insertions include information such as the `AdvertisementId` as well as the `Cost` of the advertisement as determined by auction. Advertisement events are records of impressions and clicks generated by the user. Events include information such as an `InsertionId`. Advertiser and advertisement identifiers are not available in the advertisement events: sending this information to the end user or embedding it in the advertisement would leak critical and private data. Large tech companies like Google [1] and Pinterest [1] use a similar setup to compute important advertisement statistics.

[1]Based on knowledge gained by professional experience at Pinterest.

A typical advertising timeline generally mimics the following sequence,

- An *advertisement insertion* is made when an end-user requests a web page that has space for an ad. For example, when a user makes a search request, sponsored results (or ads) are mixed into the results. Insertions are logged and written to an `InsertionStream` with relevant information such as `InsertionId`, `AdvertisementId`, and `Cost`.
- An *advertisement impression* is made when an advertisement appears on a page for some specified amount of time (several hundred milliseconds), and can be visually seen by the user. Impressions are logged and written to an `EventStream` with relevant information such as `InsertionId` and have a `Type` equal to *impression*.
- An *advertisement click* is made if a user clicks on the advertisement. Clicks are logged and written to the `EventStream` with relevant information such as `InsertionId` and have a `Type` equal to *click*.

Formally, we define the following schema for advertisers, advertisements insertions streams and event streams:

Advertiser	AdvertiserId:INT, Name:STRING
Advertisement	AdvertisementId:INT, AdvertiserId:INT, Text:STRING
InsertionStream	InsertionId:INT, AdvertisementId:int, Cost:INT
EventStream	EventId:INT, InsertionId:int, Type:STRING

Table 1: Schema of relations and streams used in the "Advertisement Statistics" example.

The two streams, `InsertionStream` and `EventStream`, are joined on `InsertionId` in order to produce rich statistical information. For example, per-advertisement click-through rates are computed by joining `InsertionStream` and `EventStream` on `InsertionId`, grouping by `AdvertisementId`, and then computing the quotient of the sum of clicks and the sum of impressions.

It should be noted that there can often be a delay of up to several minutes between advertisement insertions and advertisement events. Advertisements may show up "below the fold" or a user may momentarily leave their computer leading to ad impressions and clicks being delayed by a couple of minutes.

3. Abstract Semantics

It does not seem straightforward to apply relational operators directly to streams. It seems necessary to reason about continuous semantics. Prior work defines a comprehensive abstract semantic foundation for continuous query execution [2]. The abstract semantics defined in the CQL paper define two data types and three classes of operators. SimpleCQL is built on these same constructs.

3.1. Data Types

Two data types that are important for stream processing are *streams* and *relations*. These two data types act as primatives for the streaming semantics and become building blocks for continuous queries. It should be noted that both streams and relations have fixed schemas. We now present formal definitions of a stream and a relation.

Stream
A stream S is an unbounded set of elements $< s, \tau >$ where s is a tuple belonging to the schema of S and $\tau \in T$ is the timestamp of the element.

Relation
A relation R is a mapping from T to a finite but unbounded set of tuples belonging to the schema of R.

The "Advertisement Statistics" example consists of two streams. `InsertionStream` and `EventStream` both contain structured tuples, each occuring at a discrete timestep τ. The streams have schemas defined in Table 1. A relation in this context consists of a time-based or tuple-based windowed-snapshot of the stream at some timestamp τ. Relations in the "Advertisement Statistics" example are used to compute joins and aggregate statistics.

3.2. Stream-to-Relation Converters

Converting a stream to a relation can be done by windowing the stream over some interval. This interval might be a range in terms of time or tuples. SimpleCQL implements two such converters.

Time-based window
A time-based window considers all tuples in the stream that have an associated timestep in the range $[t - \tau, t]$ where t is the current timestep and τ is a parameter that defines the size of the window. For example, a time-based window of size 10 seconds would return all tuples in the stream that appeared in the last 10 seconds.

Tuple-based window
A tuple-based window considers the last N tuples in the stream, where N is a parameter that defines

the size of the window. For example, a tuple-based window of size N tuples, would return the last N tuples that appeared in the stream.

The "Advertisement Statistics" example uses a time-based window on the insertion stream since advertisement events may only occur several minutes after an insertion event. This allows for events happening in real-time to be joined with insertions that existed several minutes ago.

3.3. Relation-to-Relation Converters

Relation-to-relation converters are used to map a relation R into a new relation R'. They include well-known relational operators such as *joins*, *filters*, and *aggregators* that produce new relations from existing ones.

3.4. Relation-to-Stream Converters

Converting from a relation to a stream can be done in several ways. A relation may change over time and can be thought of as having tuple insertions and/or tuple deletions from a previous timestep. As such, we look at three relation-to-stream converters.

Insertion Streams
At every timestep τ an insertion stream consists of all tuples that were in a relation at timestep τ but were not in the relation at timestep $\tau - 1$. Informally this is the set of tuples that were "inserted" into a relation at timestep τ. Formally,

$$\text{IStream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

Deletion Streams
At every timestep τ a deletion stream consists of all tuples that were in a relation at time $\tau - 1$ but are not in the relation at time τ. Informally this is the set of tuples that were "deleted" from a relation at timestep τ. Formally,

$$\text{DStream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

Relation Streams
At every timestep τ a relation stream consists of all of the tuples that were in a relation at time τ. Informally this is the set of tuples that were in a relation at timestep τ. Formally,

$$\text{RStream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

The "Advertisting Statistics" example makes use of insertions streams. After joining on `InsertionId` a stream of newly joined advertisements are produced. The data from these can then be computed on in order to produce real-time statistics.

3.5. Combining Operators

Streams can be processed by combining the aforementioned operators. An incoming stream may be converted to a relation using a stream-to-relation operator. This relation might be processed through several relation-to-relation operators in order to get more useful information, and the results can be continuously output in a new stream. Figure 1 shows a representation of these steps.

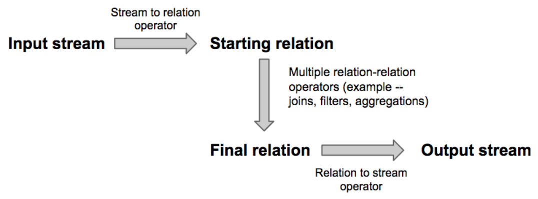
Stream to relation operator
Input stream
Starting relation
Multiple relation-relation operators (example -- joins, filters, aggregations)
Final relation
Output stream
Relation to stream operator

Figure 1: Combining operators to take a stream as input and emit a stream as output with multiple relation-to-relation operators in between.

4. Implementation

SimpleCQL is built on top of SimpleDB, a lightweight database implemented in Java. The *relation* data primitive is already well-defined in SimpleDB, as are all of the relation-to-relation converters. As such, we only need to augment SimpleDB with implementations of the *stream* data primitive, stream-to-relation operators, and relation-to-stream operators to fully support stream processing in SimpleCQL.

4.1. Discretized Time

SimpleCQL introduces a discretized notion of time in order to process continuous streams. Our system maintains an internal *timestep* that is used to label tuples from continuous streams. This *timestep* assignment does not correlate directly to cpu-time, application-time, or wall-clock time, but serves to preserve the relative time ordering of incoming data. We require such an ordering to reason about and operate on tuples and relations in our concrete semantics.

The internal *timestep* is updated each time our operators are invoked to consume from an input stream or emit to an output stream. Each update to *timestep* represents the passage of one unit of discretized time – our input and output streams also use this notion of discretized time.

4.2. Stream Readers

A *StreamReader* is an implementation construct we developed to represent an abstract *stream* primitive. *StreamReaders* allow us to discretize timesteps and read incoming tuples from an otherwise amorphous stream of data.

We implemented a generic StreamReader interface – specific types of StreamReaders need to implement this interface. All StreamReaders need to implement a `getNext(int timestep)` function – this function throws an exception if the requested timestep is too far out in the future, and otherwise returns the next tuple that appears at that timestep (null if no such tuple exists).

We will now discuss two specific StreamReaders we implemented that are interesting, and relevant to the results presented in section 5.

FileStreamReader

This StreamReader reads a static file that is not changing to produce a stream – each line in this static file is associated with a user-specified timestep, and the FileStreamReader will make these tuples ready to be consumed by downstream jobs at that specified timestep.

The implementation of this StreamReader is very simple – the entire file is read into memory at initialization, and a map of timesteps to array of tuples is created. Once this is created, we can answer queries for tuples at any timestamp.

LiveStreamReader

This StreamReader tries to mimic tuples coming into the system in real time. A python script is used to produce a steady stream of tuples into a file; our LiveStreamReader then polls this file at regular intervals of time, reads in the new tuples written to the file by the Python script, and then makes these new tuples available to downstream jobs.

Our LiveStreamReader discretizes continuous time – the time interval associated with a single timestep is configurable. In addition, to ensure that we have seen all tuples associated with a particular timestep ts, we need to see at least one tuple with timestep $ts+1$.

4.3. Stream-to-Relation Converters

We implemented both a time-based Stream-to-Relation converter and a tuple-based Stream-to-Relation converter.

Time-based Stream-to-Relation Converter

The time-based converter is parameterized by a *stream*, which is the input data source, and a *windowSize*, which is the number of most recent timesteps to window the stream per advance in unit time. A time-windowed relation at $t = \tau$ with $windowSize = w$ will include all tuples from the stream with time labels $\in [\tau - w, \tau]$. The converter maintains the time-windowed relation as a list of tuples, and this data structure is initialized to be empty until consumption from the stream begins. We invoke the converter once to consume one timestep from the stream and incrementally update the internal relation data structure, adding only the new tuples that enter the window and removing only the old tuples that leave the window. Repeated invocation simulates the continuous forward motion of discretized time. Figure 2 illustrates the relation data structure state with the passage of time.

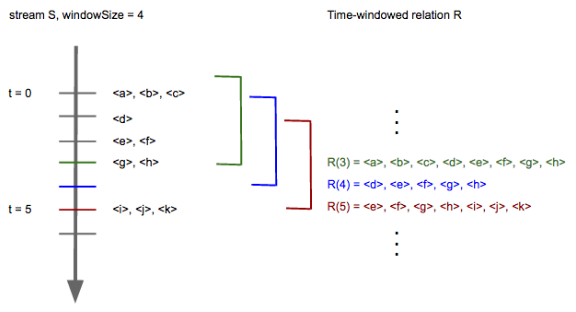
stream S, windowSize = 4
Time-windowed relation R
t = 0
t = 5
<a>, <b>, <c>
<d>
<e>, <f>
<g>, <h>
<i>, <j>, <k>
R(3) = <a>, <b>, <c>, <d>, <e>, <f>, <g>, <h>
R(4) = <d>, <e>, <f>, <g>, <h>
R(5) = <e>, <f>, <g>, <h>, <i>, <j>, <k>

Figure 2: Example time-based Stream-to-Relation converter: windowed relation at three consecutive discretized timesteps with $windowSize = 4$.

Tuple-based Stream-to-Relation Converter

The tuple-based converter is similarly parameterized by a *stream* and a *tupleCount*, which is the number of most recent tuples to include per advance in unit time. A tuple-windowed relation at $t = \tau$ with $tupleCount = c$ will include the most recent c tuples from the stream since time τ. We similarly invoke the converter to incrementally consume one timestep from the stream, adding the tuples at the new timestep and removing old tuples until the relation size is c. Figure 3 illustrates the relation data structure with the passage of time.

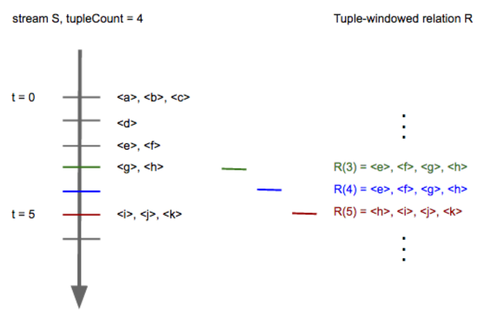
stream S, tupleCount = 4
Tuple-windowed relation R
t = 0
<a>, <b>, <c>
<d>
<e>, <f>
<g>, <h>
t = 5
<i>, <j>, <k>
R(3) = <e>, <f>, <g>, <h>
R(4) = <e>, <f>, <g>, <h>
R(5) = <h>, <i>, <j>, <k>

Figure 3: Example tuple-based Stream-to-Relation converter: windowed relation at three consecutive discretized timesteps with $tupleCount = 4$.

Both converters return, per advancing timestep, a *DbIterator* of the windowed relation that is compatible with existing relational operators in SimpleDB.

4.4. Relation-to-Stream Converters

The Relation-to-Stream converters are similar to the Stream-to-Relation converters in that they are also repeatedly invoked to simulate the continuous forward motion of discretized time. Relation-to-Stream converters generate an output *stream*; each update invocation takes in a relation at time $t = \tau$ and compares it to the previous input relation at time $t = \tau - 1$, emitting the corresponding tuples to the output stream.

Insertion Stream
The Insertion Stream emits to the output stream all of the tuples that are present in the new $relation_\tau$ but that were not present in the previous $relation_{\tau-1}$. Thus, IStream(τ) can be generalized to be defined as $relation_\tau - relation_{\tau-1}$.

Deletion Stream
The Deletion Stream emits all tuples that are not present in the new $relation_\tau$ but that were present in the previous $relation_{\tau-1}$. Dstream(τ) is defined as $relation_{\tau-1} - relation_\tau$.

Relation Stream
The Relation Stream emits at time $t = \tau$ all tuples in the $relation_\tau$ that is passed into the converter update. RStream(τ) is exactly $relation_\tau$.

We implement these diffs-based computations by maintaining the `prevRelation`, hashing the tuples of `prevRelation`, and then probing into the hash set with tuples of `nextRelation` to determine the differences across timesteps. Figure Z illustrates the emitted tuples for each of the Relation-to-Stream converters with respect to time; the IStream is highlighted because it is most illustrative and most significant in our system.

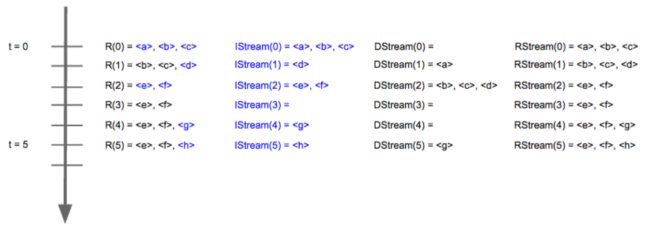

Figure 4: Changes in the input Relation R over time and the corresponding updates to IStream, Dstream, and RStream.

4.5. Garbage Collection

We implemented garbage collection in our system to ensure that streams are not stored in their entirety at any point in time – this is desirable since streams can become arbitrarily large, especially if tuples are accumulated for a long time.

Two possible garbage collection policies are possible – a time based garbage collection policy, where tuples from too far out in the past are discarded (this is suitable for queries featuring streams that are windowed using a time-based Stream-to-Relation converter) and a tuple based garbage collection policy, where tuples are discarded once the number of tuples in the system passes a certain threshold.

For now, we have only implemented the time based garbage collection policy. Effects on memory footprint and performance are talked about in the performance portion of this paper.

5. Evaluation

Benchmarks of SimpleCQL were performed on an AWS Compute Optimized (c3.8xlarge) instance with 60 GiB of memory, 32 vCPUs, 640 GB of SSD-based local instance storage, and a 64-bit platform. When applicable, SimpleCQL performance was compared to that of SimpleDB and the continuous workload of SimpleCQL is shown to outperform equivalent discretized workloads of SimpleDB.

SimpleDB was intended to be a toy database designed for teaching core internal database concepts. As such, the performance of both SimpleDB and SimpleCQL suffer from a lack of optimizations. However, we made sure to use the same underlying operations when benchmarking both SimpleDB and SimpleCQL; we also held many other factors constant to make for a fair comparison. For example, we configured SimpleDB to not have to write inserted tuples to disk because SimpleCQL operates entirely in-memory. Further, SimpleDB does not use an index because there is no indexing in SimpleCQL. With these kinds of controls in place, we anticipate our experimental results will closely delineate the performance differences between SimpleDB and SimpleCQL.

Many of the benchmark tests were run over $D = 1800$ timesteps, and the number of tuples per timestep R was used to vary the work density per timestep. We measured the time T it took to process D timesteps with respect to R. The following equation is used to estimate the projected throughput of the system.

$$\text{Throughput} = R \text{ tuples/timestep} \times \frac{D \text{ timesteps}}{T \text{ seconds}}$$

5.1. Log Analysis: Detecting Attacks

Most web companies wish to keep their users' accounts safe and out of reach from hackers and spammers. On occasion though, malicious parties do manage to obtain lists of millions of usernames. These users can then be victims of a brute-force attack where attackers try to brute force through all possible permutations to guess a user's password, resulting in a spike in the number of failed login attempts. Access to real-time error log statistics allows spam and abuse teams to easily detect such an attack and respond as necessary.

In this example, we consider a stream of log messages, `LogStream`. We formally define the schema of the stream as: `LogStream{Message:STRING}`. Given this schema, a rolling count of log messages that contain the string "failed login" provides the relevant information for the above situation. The following query computes a rolling count of failed login attempts.

```
SELECT ISTREAM(
    count(*)
)
FROM LogStream [Range 1 seconds]
WHERE LogStream.Message
    LIKE "%failed login%"
GROUP BY LogStream.Message;
```

Query 1: SimpleCQL query that computes number of times "failed login" was found in LogStream

Data for the `LogStream` was generated prior to running tests and read using a custom `FileStreamReader`. The data generated represents 30 minutes of log messages.

As seen in Figure 5, SimpleCQL drastically outperforms SimpleDB when running queries over relatively small time windows (several seconds). As the number of tuples per timestep is increased, the difference in execution time between SimpleCQL and SimpleDB also increased.

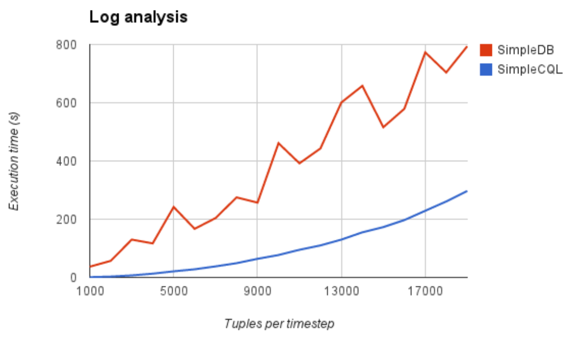

Figure 5: Performance of SimpleDB and SimpleCQL detecting attacks.

SimpleCQL was able to achieve about an order of magnitude speedup over SimpleDB. The projected throughput of SimpleCQL is approximately 400k tuples/second, whereas the projected throughput of SimpleDB is approximately 50k tuples/second.

5.2. Trending Tweets

Trending tweets on Twitter are useful for a variety of reasons. The company may use them to increase user engagements, optimize advertisements, or analyze the state of current events.

In this example, we consider a stream of Twitter tweets, `TweetStream`. We formally define the schema of the stream as: `TweetStream{Text:STRING, Hashtag:STRING, Location:STRING}`. Given this schema, a rolling count of log messages that contain the string "failed login" provides the relevant information for the above situation. The following query computes the rolling count.

```
SELECT ISTREAM(
    TweetStream.Hashtag,
    MAX(COUNT(*))
)
FROM TweetStream [Range 30 seconds]
WHERE TweetStream.Location = "Boston"
GROUP BY TweetStream.Hashtag;
```

Query 2: SimpleCQL query that computes the trending hashtags in Boston given an input TweetStream

Data for the `TweetStream` was generated prior to running tests. The data generated represents 30 minutes of tweets with text, hashtags, and locations.

As seen in Figure 6, SimpleCQL again outperforms SimpleDB when running queries over larger time windows. As the number of tuples per timestep is increased, the difference in execution time between SimpleCQL

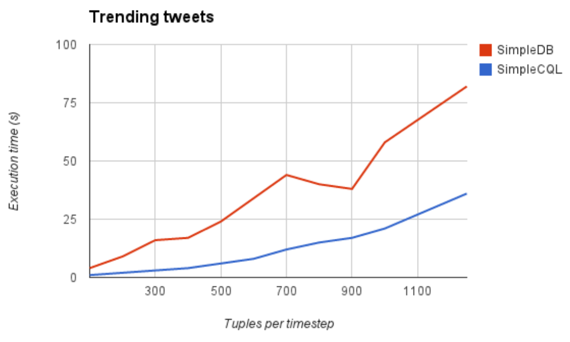

Figure 6: Performance of SimpleDB and SimpleCQL calculating trending tweets.

and SimpleDB also increased.

The projected throughput of SimpleCQL is approximately 100k tuples/second, whereas the projected throughput of SimpleDB is approximately 35k tuples/second.

5.3. Advertisement Statistics

The Advertisement Statistics example, presented in detail in section 2, involves joining advertisement insertions across two disparate streams. Specifically a stream of advertisement insertions, known as `InsertionStream`, and a stream of advertisement events, known as `EventStream`, are joined on `InsertionId` in order product real-time per-advertisement click-through rate. The following query computes the join.

```
SELECT RSTREAM (
    advertisement_id,
    SUM(type = 'click' ? 1 : 0) /
    SUM(type = 'impression' ? 1 : 0)
)
FROM (
    SELECT ISTREAM(advertisement_id, type)
    FROM InsertStream [Range 600 seconds],
         EventStream [Range 1 seconds]
    WHERE InsertStream.insertion_id =
    EventStream.insertion_id
)
GROUP BY advertisement_id;
```

Query 3: SimpleCQL query that computes CTR of advertisements given an EventStream and an InsertStream

Data for the `InsertionStream` and `EventStream` was generated prior to running tests. The data generated represents 30 minutes of advertisement insertions. Each insertion in the `InsertionStream` may have a corresponding impression and click event in the EventStream that appears up to 10 minutes after the insertion.

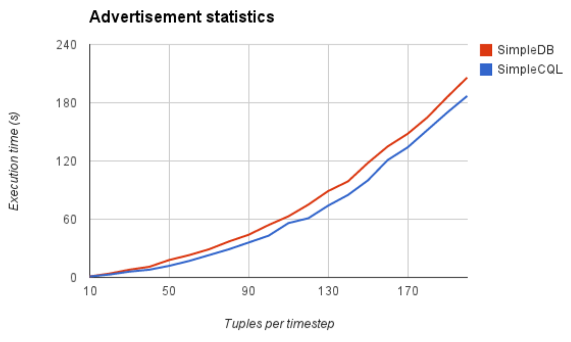

Figure 7: Performance of SimpleDB and SimpleCQL computing real-time click-through rates.

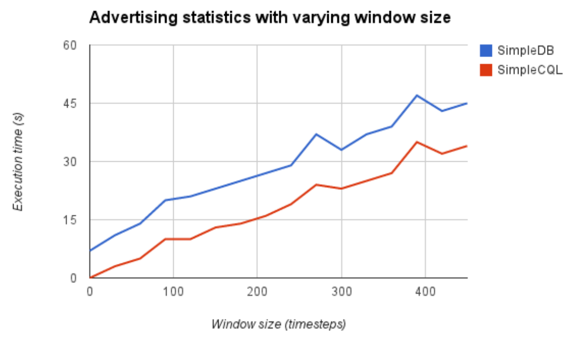

Figure 8: Performance of SimpleDB and SimpleCQL computing real-time click-through rates with varying window.

As seen in Figure 7, SimpleCQL does not notably outperform SimpleDB when running queries over very large time windows with complex queries. It is important, however, to note that SimpleCQL's performance did not degrade.

The projected throughput of SimpleCQL is approximately 8k tuples/second, whereas the projected throughput of SimpleDB is approximately 6.5k tuples/second. Both systems are severely hindered by an unoptimized nested-loop join implementation.

5.4. Garbage Collection

Garbage collecting tuples that are no longer needed using a time-based collection policy drastically reduced the memory footprint of the system. This is expected as keeping tuples in memory indefinitely will continuously consume more memory. There is a performance tradeoff to discarding old tuples, but we show that this outweighs the memory usage in Figure 9. The impact to execution time per timestep when garbage collection is enabled is approximately 20ms, or roughly 4%.

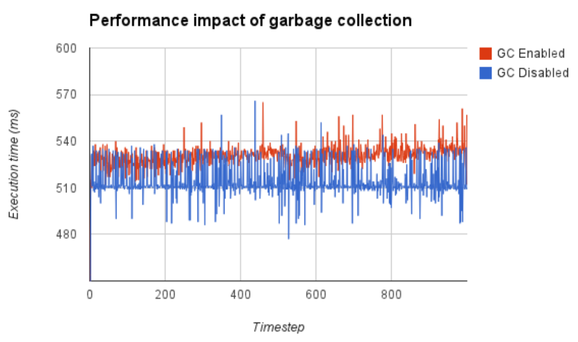

Figure 9: Plot of execution time per timestamp versus timestamp with and without garbage collection. With Garbage Collection enabled, the system performs marginally worse than with Garbage Collection disabled.

6. Conclusion

We have augmented SimpleDB, a static relational database, with implementations of the abstract semantics in a Continuous Query Language. The resulting SimpleCQL is our successful implementation of a streaming database management system that supports SQL-like relational operations on real-time streams of structured data. We have exemplified the motivation for such a continuous query processing system through our examples of error logs aggregations, trending tweets computations, and Advertisement Statistics calculations. Despite being built on SimpleDB, a foundation with plenty of room for improvement and optimizations, SimpleCQL performs considerably well on these three tested workloads, all the while leaving a negligible memory footprint when processing unbounded streams of data.

The code for SimpleCQL is hosted on GitHub. A demo is available on YouTube.

References

[1] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD '13: Proceedings of the 2013 international conference on Management of data*, pages 577–588, New York, NY, USA, 2013.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[3] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[4] Edward Sciore. Simpledb: A simple java-based multiuser system for teaching database internals. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 561–565, 2007.

[5] Apache Storm. https://storm.apache.org/.

[6] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.