Cut-and-Rewind: Extending Query Engine for Continuous Stream Analytics

Qiming Chen(✉) and Meichun Hsu

HP Labs, Hewlett Packard Co., Palo Alto, CA, USA
{qiming.chen,meichun.hsu}@hp.com

Abstract. Combining data warehousing and stream processing technologies has great potential in offering low-latency data-intensive analytics. Unfortunately, such convergence has not been properly addressed so far. The current generation of stream processing systems is in general built separately from the data warehouse and query engine, which can cause significant overhead in data access and data movement, and is unable to take advantage of the functionalities already offered by the existing data warehouse systems.

In this work we tackle some hard problems in integrating stream analytics capability into the existing query engine. We define an extended SQL query model that unifies queries over both static relations and dynamic streaming data, and develop techniques to extend query engines to support the unified model. We propose the ***cut-and-rewind*** query execution model to allow a query with full SQL expressive power to be applied to stream data by converting the latter into a sequence of "chunks", and executing the query over each chunk sequentially, but without shutting the query instance down between chunks for continuously maintaining the application context across the execution cycles as required by sliding-window operators. We also propose the ***cycle-based transaction model*** to support Continuous Querying with Continuous Persisting (CQCP) with cycle-based isolation and visibility.

We have prototyped our approach by extending the PostgreSQL. This work has resulted in a new kind of tightly integrated, highly efficient system with the advanced stream processing capability as well as the full DBMS functionality. We demonstrate the system with the popular Linear Road benchmark, and report the performance. By leveraging the matured code base of a query engine to the maximal extent, we can significantly reduce the engineering investment needed for developing the streaming technology. Providing this capability on proprietary parallel analytics engine is work in progress.

1 Introduction

Streaming analytics is a data-intensive computation chain from event streams to analysis results. In response to the rapidly growing data volume and the pressing need for lower latency, Data Stream Management Systems (DSMSs) provide a paradigm shift from the load-first analyze–later mode of data warehousing [8, 16, 17, 19].

1.1 The Problem

However, the current generation of DSMS is in general built separately from the data warehouse query engine, due to the difference in handling stream data and static data;

A. Hameurlain et al. (Eds.): TLDKS XXI, LNCS 9260, pp. 94–114, 2015.
DOI: 10.1007/978-3-662-47804-2_5

as a result, the data transfer overhead between the two has become a performance and scalability bottleneck [4, 6, 10]. The standalone DSMS's also lack the full SQL expressive power and DBMS functionalities of managing persistent data. It does not have the appropriate transaction support for continuously persisting and sharing results along with continuous querying. As stream processing evolves from simple to complex, these functionalities are likely to be redeveloped.

In this paper we tackle the following technical challenges in integrating stream processing with data warehouse query engine:

- A query engine manages relations (tables) which contain well defined sets. However, a stream is unbounded, and never reaches the "end of data", which would pose problems with the existing query model and transaction model.
- Stream processing is often based on windows, and there is a need to apply a query repeatedly to chunks of unbounded stream data that fall in consecutive windows. Stream analytics also requires operators that are history sensitive, such as sliding window operators, and there is a need to continuously and efficiently maintain the state or a synopsis of the data that falls in the previous windows.
- During stream processing, there is a need to persist periodically to allow the analysis results to be visible to other concurrent applications, sometimes even to another branch of the same query. This will require extended transaction semantics that is not supported with existing query engines.

1.2 State of the Art

Since a stream query is defined on unbounded data and in general limited to non-transactional event processing, the current generation of DSMSs is mostly built from scratch independently of the database engine. Big players along this direction include System S (IBM) [15], STREAM (Stanford) [3], TelegraphCQ (Berkeley) [5], as well as Aurora, Borealis, etc. [1, 2, 7, 11, 17]. Two recently reported systems, the TruSQL engine [16] developed by Truviso Inc, USA, and the DataCell engine [19] developed by CWI, Netherlands, do leverage database technology but are characterized by providing a workflow like service for launching a SQL query for each chunk of the stream data during stream processing. To the best of our knowledge, none of the existing approaches has leveraged the query engine without introducing an additional loosely-coupled "middleware" layer. Oracle currently offers a "continued query" feature but it is based on automatic view updates and is not the same feature as stream processing.

Managing data-intensive stream processing outside of the query engine causes the data copying and moving overhead, and fails to leverage the full SQL and DBMS functionality.

Processing streams by multiple queries may incur performance penalty due to the overhead for frequent query setup and teardown, and more seriously, cause the semantic difficulty in chunk-wise data manipulation. Since the backend query execution processes are in isolated memory contexts, processing each data chunk by an individual query instance cannot maintain the application context, e.g. the data buffered with User Defined Functions (UDFs) continuously across multiple query instances, thus unable to deal with sliding-window like operations.

To the best of our knowledge, none of the existing approaches has solved the difficulty of processing stream in terms of truly continued SQL query with chunk-wise semantics but continuously tracked application context, by leveraging the query engine without introducing an additional loosely-coupled "middleware" layer.

1.3 The Solution

We view a query engine essentially as a streaming engine, although this potential has not been thoroughly explored. With this vision, we advocate an extended SQL model that unifies queries over both streaming and static relational data, and a new architecture for integrating stream processing and DBMS to support continuous, "just-in-time" analytics with window-based operators and transaction semantics.

Our proposed stream model is based on dividing an infinite stream of relation tuples with a criterion, e.g. by every 1-minute time window, into an unbounded sequence of chunks. The semantics of applying the query to the unbounded stream lies in applying the query to those infinite chunks which continuously generates an unbounded sequence of query results, one on each *chunk* of the stream data.

Our goal is to support the above semantics using a continuous query that runs cycle by cycle for processing the stream data chunks, each data chunk to be processed in each cycle, in a single, long-standing query instance. In this sense we also refer to the *data chunking criterion* C as the *query cycle specification*. The cycle specification can be based on time or a number of tuples, which can amount to as small as a single tuple, and as large as billions of tuples per cycle. The stream query may be terminated based on specification in the query (e.g. run for 300 cycles), user intervention, or a special end-of-stream signal received from the stream source.

Specifically, our solutions include the following.

- We start with providing unbounded relation data to feed queries continuously. The first step is to integrate the notions of stream data source, and use function-scan instead of table-scan, for turning captured events into unbounded sequence of relation tuples to feed to stream queries without first storing them on disk.
- We develop UDF shells [9] to deliver operators with stream semantics (e.g. moving average, notification) that are not available in conventional SQL. We allow a UDF to cache the state in the application context for carrying out history-sensitive operations, such as sliding window oriented operations, along the stream processing pipeline. We also allow a UDF to emit the current or accumulated computation results continuously on the per-tuple basis - once a tuple from the stream has been received and/or processed.
- We propose the *cut-and-rewind* query model, namely, cutting a query execution based on some granule ("chunk") of the stream data (e.g. in a time window), and then rewinding the state of the query without shutting it down, for processing the next chunk of stream data. This mechanism, on one hand, allows applying a query continuously to the stream data chunks falling in consecutive time windows, within a single, long-standing query; on the other hand, allows retaining the application context (e.g. data buffered with UDFs) continuously across the execution cycles to perform sliding-window oriented, history sensitive operations.

– To support *Continuous Querying with Continuous Persisting* (CQCP), we introduce the cycle-based transaction model with the *cycle-based isolation* mechanism, which makes the heap-inserted, chunk-wise database updates accessible by other applications as soon as the corresponding cycle execution commits. Note however that a continuous query may emit non-transactional messages or events to external receivers before "commit" – such messages are not bound by transaction semantics.

A significant advantage of the unified model lies in that it allows us to exploit the full SQL expressive power on each data chunk. The output is also a stream consisting of a sequence of chunks, with each chunk representing the query result of one execution cycle. While there may be different ways to implement our proposed unified model, our approach is to generalize the SQL engine to include support for stream sources. The approach enables queries over both static and streaming data, retains the full SQL power, while executing stream queries efficiently.

The proposed *cut-and-rewind* approach enables us to support truly continuous query in a completely different way from other DSMSs, and seamlessly integrate the stream processing capability into a full–functional database system, creating a powerful and flexible system that can run SQL over tables, streams (tuple by tuple or chunk by chunk), and the combination of the two.

In this paper we have limited a query to refer to a single stream and thus a single cycle specification. In general, our model allows multiple stream queries to refer to the same source, and these queries can interact through database tables which may be memory resident; our model also allows a single query to refer to multiple stream sources with different cut criteria. Various pairing patterns [15] and the corresponding operations to allow multiple streams or hybrid queries to interact have been investigated and are to be reported separately.

We report our experience in leveraging the PostgreSQL engine for supporting stream processing. The proposed *cut-and-rewind* mechanism has been implemented with minimal engine extension, resulting in a tightly integrated, highly efficient platform with the advanced stream processing capability as well as the full DBMS functionality. We demonstrated the merit of our platform using the popular Linear Road benchmark. Providing this capability on a proprietary parallel database engine is currently being explored.

The rest of this paper is organized as follows: Sect. 2 reports our approach in handling stream source and stream analytic functions by extending a DBMS with new source functions and UDFs for stream operations; Sect. 3 proposes the *cut-and-rewind* approach; Sect. 4 deals with the transaction issues in cycle-based stream processing; Sect. 5 shows how the proposed approach is applied to the popular Linear Road stream processing benchmark, and discusses the experiment results; Sect. 6 concludes the paper.

2 Stream Processing as Continuous Querying

A SQL query is parsed and optimized into a query plan that is a tree of operators. The scan operator at the leaf of the tree gets and materializes a block of data to be delivered to the upper layer tuple by tuple. A non-blocking relational operator or a function, e.g. a UDF, is invoked multiple times in a query execution on the per-tuple basis, which forms a dataflow pipeline, and in this sense, similar to stream processing.

However, there exist some fundamental differences between the conventional query processing and the stream processing. First, a query is defined on bounded relations but stream data are unbounded; next, stream processing adopts window-based semantics, i.e. processing the incoming data chunk by chunk falling in consecutive time windows; however, the SQL operators are either based on one tuple (such as filter operators) or the entire relation; Further, stream processing is also required to handle sliding window operations continuously across chunk based data processing; and finally, endless stream analytics results must be continuously accessible along their production, under specific transaction semantics.

Let us use a simplified traffic system example to illustrate our unified query over stored and stream data, where the total amount of toll charged for each highway segment per minute are computed, given a segment toll table and events that report vehicles' entering a segment.

- *C* (*vid,sid,ts*), contains the event that a car (*vid*) enters a tolled segment (*sid*) with a timestamp in second (*ts*),
- *T* (*sid, charge*) contains the highway segment info where *charge* is the toll per car for segment *sid*.

We express the example first as a query over static relations only, and then as a hybrid query that includes a stream source. The graphical representation of the two queries is shown in Fig. 1.

For the first query *Q1* (shown on the left of Fig. 1), the inputs are two stored relations, *C* and *T*. However, if the table *C* above is not a stored relation, but replaced

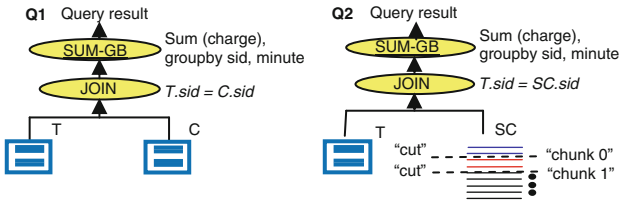

Fig. 1. Querying static table vs. querying data stream chunk by chunk

by a real-time stream source, while *T* remains a stored relation, then the above application becomes a streaming application. The above static SQL query is adapted to a streaming query simply by defining *SC* as a *stream* (instead of a table) with the same schema as *C* and changing the reference to *C* as follows (shown on the right of Fig. 1):

Q1:

```
SELECT sid, floor(ts/60) AS minute, SUM(charge)
FROM T, C WHERE C.sid = T.sid
GROUP BY sid, minute
```

Q2:

```
SELECT sid, floor(ts/60) AS minute, SUM(charge)
FROM T, STREAM (SC, cycle-spec) WHERE SC.sid = T.sid
GROUP BY sid, minute
```

In the above query, we replace the disk-resided database table by a special kind of table function *STREAM(),* called Stream Source Function (SSF), that listens or reads data/events sequence. Further, *STREAM(SC, cycle-spec)* specifies that the stream source *SC* is to be "cut" into an unbounded sequence of *chunks* SC_{C0}, SC_{C1}, ..., where all tuples in SC_{Ci} occur before any tuple in SC_{Ci+1} in the stream. The "cut point" is specified in the *cycle-spec*. Let *Q1* above be denoted as a query function over table *C*, i.e., *Q1(C).* The execution semantics of *Q2* is defined as executing $Q1(SC_{Ci})$ in sequence for all SC_{Ci}'s in the stream source *SC*.

In general, given a query Q over a set of relation tables $T_1,..,T_n$ and an infinite stream of relation tuples S with a criterion ϑ for cutting S into an unbounded sequence of chunks, e.g. by every 1-minute time window, $< S_0, S_1, \ldots, S_i, \ldots >$ where S_i denotes the *i-th* "chunk" of the stream according to the chunking-criterion ϑ. S_i can be interpreted as a relation. The semantics of applying the query Q to the unbounded stream S plus the bounded relations $T_1,..,T_n$ lies in

$$Q(S, T_1, .., T_n) \rightarrow < Q(S_0, T_1, .., T_n), \ldots Q(S_i, T_1, .., T_n), \ldots >$$

which continuously generates an unbounded sequence of query results, one on each *chunk* of the stream data.

2.1 Stream Source Function

For providing unbounded relation data to fuel queries continuously, the first step is to replace the database table, which contains a set of tuples on disk, by the special kind of table function, called Stream Source Function (SSF) that returns a sequence of tuples to feed queries without first storing on disk. A SSF can listen or read data/events sequence and generate stream elements tuple by tuple continuously. A SSF is called multiple, up to infinite, times during the execution of a continuous query, each call returns one tuple. When the end-of-cycle event or condition is seen, the SSF signals the query engine to terminate the current query execution cycle.

We rely on SSF and query engine for continuous querying on the basis that "as far as data do not end, the query does not end", rather than employing an extra scheduler to launch a sequence of one-time query instances. The SSF scan is supported at two levels, the SSF level and the query executor level. A data structure containing function call information, *hFC*, bridges these two levels. *hFC* is initiated by the query executor and passed in/out the SSF for exchanging function invocation related information. We use this mechanism for minimizing the code change, but maximize the extensibility, of the query engine.

2.2 Stream Analytics Through UDF

One important characteristics of stream processing is the use of stream-oriented history-sensitive analytic operators such as moving average or change point detection. While the standard SQL engine contains a number of built-in analytic operators, stream history-sensitive operators are not supported. Using UDFs is the generally accepted mechanism to extend query operators in a DBMS. A UDF can be provided with a data buffer in its function closure, and for caching stream processing state (synopsis).

Furthermore, it is also used to support one or more *emitters* for delivering the analytics results to interested clients in the middle of a cycle, which is critical in satisfying stream applications with low latency requirement.

Stream processing involves operations on (time) windows, including sliding windows, and therefore is history sensitive. This represents a different requirement from the regular query processing that only cares about the current state. We use UDFs to add window operators and other history sensitive operators, buffering required raw data or intermediate results within the UDF closures.

A scalar UDF is called multiple times on the per-tuple basis, following the typical FIRST_CALL, NORMAL_CALL, FINAL_CALL skeleton. The data buffer structures are initiated in the FIRST_CALL and used in each NORMAL_CALL. A window function defined as a scalar UDF incrementally buffers the stream data, and manipulates the buffered data chunk for the required window operation. Since the query instance remains alive, as supported by our *cut-and-rewind* model, the UDF buffer is retained between cycles of execution and the data states are traceable continuously (we see otherwise if the stream query is made of multiple one-time instances, the buffered data cannot be traced continuously across cycle boundaries). As a further optimization, the static data retrieved from the database can be loaded in a window operation initially and then retained in the entire long-standing query, which removes much of the data access cost as seen in the multi-query-instances based stream processing.

We propose to run a SQL query cycle by cycle for deriving a sequence of data-chunk based results, but never shutting down the query instance in order to have the per-tuple based data processing history continuous tractable.

UDFs can be used to develop a library of reusable stream operators and further allow the unified query model to be extended. As will be illustrated in our Linear Road (LR) implementation, the 5-minute moving average speed is provided through a moving average UDF, atop the per-minute average speed, the latter computed using the standard SQL average-groupby function in one query cycle.

3 Cycle Based Continuous Query

To support the cycle based execution of stream queries, we propose the *cut-and-rewind* query execution model, namely, cut a query execution based on the cycle specification (e.g. by time), and then rewind the state of the query without shutting it down, for processing the next chunk of stream data in the next cycle.

Under this *cut-and-rewind* mechanism, a stream query execution is divided into a sequence of *cycles*, each for processing a chunk of data only; it, on one hand, allows applying a SQL query to unbounded stream data chunk by chunk within a single, long-standing query instance; on the other hand, allows the application context (e.g. data buffered within a User Defined Function (UDF)) to be retained continuously across the execution cycles, which is required for supporting sliding-window oriented, history sensitive operations. Bringing these two capabilities together is the key in our approach.

Cut *Cutting* stream data into chunks is originated in the SSF at the bottom of the query tree. Upon detection of end-of-cycle condition, the SSF signals *end-of-data* to the query

engine through setting a flag on the function call handle, that, after being interpreted by the query engine, results in the termination of the current query execution cycle.

If the cut condition is detected by testing the newly received stream element, the *end-of-data* event of the current cycle would be captured upon receipt of the first tuple of the next cycle; in this case, that tuple will not be returned by the SSF in the current cycle, but buffered within the SSF and returned as the first tuple of the next cycle. Since the query instance is kept alive, that tuple can be kept across the cycle boundary.

Rewind Upon termination of an execution cycle, the query engine does not shut down the query instance but *rewinds* it for processing the next chunk of stream data. Rewinding a query is a top-down process along the query plan instance tree, with specific treatment on each node type. In general, the intermediate results of the standard SQL operators (associated with the current chunk of data) are discarded but the application context kept in UDFs (e.g. for handling sliding windows) are retained. The query will not be re-parsed, re-planned or re-initiated.

Note that rewinding the query plan instance aims to process the next chunk of data, rather than re-deliver the current query result; therefore it is different from "rewinding a query cursor" for re-delivering the current result set from the beginning. For example, the conventional cursor rewind tends to keep the hash-tables for a hash-join operation but our rewind will have such hash-tables discarded since they were built for the previous, rather than the next, data chunk.

As mentioned above, the proposed *cut-and-rewind* approach has the ability to keep the continuity of the query instance over the entire stream while dividing it to a sequence of execution cycles. This is significant in supporting history sensitive stream analytic operations, as discussed in the previous section.

4 Continuous Querying with Continuous Persisting (CQCP)

One problem of the current generation of DSMSs is that they do not support transactions. Intuitively, as stream data are unbounded and the query for processing these data may never end, the conventional notion of transaction boundary is hard to apply. In fact, transaction notions have not been appropriately defined for stream processing, and the existing DSMSs typically make application specific, informal guarantees of correctness.

However, to allow a hybrid system where stream queries can refer to static data stored in a database, or to allow the stream analysis results (whether intermediate or final) to persist and be visible to other concurrent queries in the system in a timely manner, a transaction model which allows the stream processing to periodically "commit" its results and makes them visible is needed.

Note that if a stream application does not use static data in the database, or does not need to persist results and make them visible to other concurrent applications, then transaction semantics are not needed. In our design, the transaction semantics is used, and thus transaction management overhead is incurred, only when a stream application requires persistent data management.

4.1 Query Cycle Based Transaction Model

Lacking formal transaction semantics is a problem of the current generation of stream processing systems, as they typically make application specific, informal guarantees of correctness.

Conventionally a query is placed in a transaction boundary; the query result and the possible update effect are made visible only after the commitment of the transaction (although weaker transaction semantics do exist). Since the query for processing unbounded stream data may never end, the conventional notion of transaction boundary is hard to apply.

In order to allow the result of a long-running stream query to be incrementally accessible, we introduce the cycle-based transaction model coupled with the *cut-and-rewind* query model, which we call *continuous querying with continuous persisting*. Under this model a stream query is "committed" one cycle at a time in a sequence of "micro-transactions". The transaction boundaries are consistent with the query cycles, thus synchronized with the chunk-wise stream processing. The per-cycle stream processing results are made visible as soon as the cycle ends.

For example, in Q2 above, the query result, which is the total charge per highway segment, is made visible every cycle; if the cycle specification is per minute, then the total charge per segment is made visible per minute, and it can also be persisted at the minute boundary.

4.2 Staging Results Without Data Copy/Move

With the cloud service, the analytics results are accessed by many clients through PCs or smart phones. These results are read-only time series data, stored in the read-sharable tables incrementally visible to users as they become available. Since the analytics results are derived from unbounded stream of events, they are themselves unbounded and thus must be staged step by step along with their production. Very often, only the latest data is "most wanted". For scaling up CaaaS, efficient data staging is the key.

Data staging is a common task of data warehouse management. The general approach is stepwise archival of the older data, which, however, incurs data moving and copying overhead. While this approach is acceptable for handling slowly-updated data in data warehousing, it is not efficient for supporting real-time stream analytics.

To avoid the data moving and copying overhead in data staging, we have developed a specific mechanism characterized by *staging through metadata manipulation without real data movement*. As shown in Fig. 2, we provide a list of tables for keeping the stream analytics results generated in a given number of query execution cycles (e.g. generated in 60 per-minute cycles, i.e. one hour). These tables are arranged as a "table-ring" and used in a *round-robin fashion*. For example, to keep the results for the latest 8 h of notifications, 9 tables say T_1, T_2, …, T_9, are allocated in a buffer-pool, such that at a time, T_1 stores the results of the current hour, say *h*, T_2 stores the results of the hour *h-1*, …, T_8 stores the results of the hour *h-8*, the data in T_9 are beyond the 8-hour range thus being archived asynchronously during the current hour. When the hour changes, the archiving of T_9 has presumably finished and T_9 is reassigned for storing the results of the new, current hour.

The hourly based timestamp of these tables are maintained either in the data dictionary or a specifically provided system table. In the above data staging, only the "label" of a table is switched for representing the time boundary (i.e. the hour) of its content, without moving/copying the content to another table or file thus avoiding the read/write overhead.

Further, a stable SQL interface is provided for both the client-side users and the server-side queries. Assuming the table holding the summarized traffic status in the current hour is named "*current_road_condition*", this name remains the same at all the times but points to different physical tables from time to time. This may be accomplished by associating the table holding the latest results to "*current_road_condition*" through metadata lookup, or by system internal query modification.

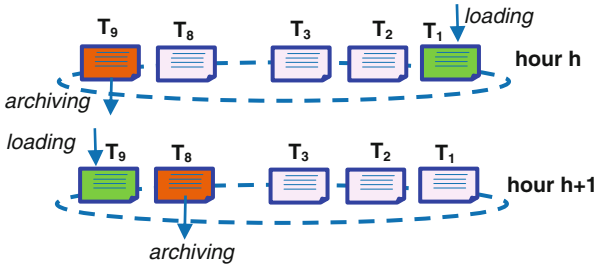

Fig. 2. Table-ring approach for staging analytics results through metadata manipulation without data copy/move

We have extended the query engine to support the above table ring for the client-side query. The continuous query uses the INSERT-INTO clause to capture the query results at each cycle. (See Sect. 2.5 for an example).The "into-relation" is closed prior to a cycle-based transaction commits and it re-opens after the transaction for the next cycle starts. Between the *complete_transaction()* call and the *reopen_into_relation ()* call, the number of execution cycles is checked, and if the specified staging time boundary is reached, the switching of "into-relations", i.e. the query destinations, takes place, where the above data dictionary or specific system table is looked up, and the "next" table ID is obtained and passed to the *reopen_into_relation()*. Thereafter another

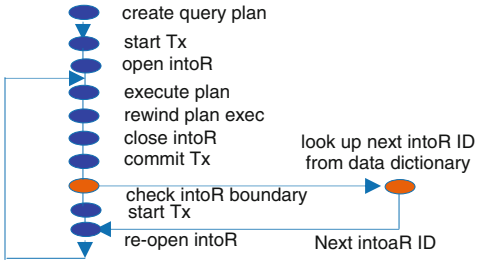

Fig. 3. Cycle-based query execution, transaction, staging

into-relation will act as the query destination. This way, the query runs cycle by cycle to process the input data stream chunk by chunk.

Overall, the cycle-based query execution, transaction commitment and multi-cycle based data staging are illustrated in Fig. 3.

5 Example and Experiments

5.1 Modeling the Linear Road Benchmark

We use the widely-accepted Linear-Road (LR) benchmark [18] to demonstrate our extended query engine. The LR benchmark models the traffic on express ways for the 3-hour duration; each express way has two directions and 100 segments. Cars may enter and exit any segment. The position of each car is read every 30 s and each reading constitutes an event, or stream element, for the system. A car position report has attributes *vid* (vehicle ID), *time* (seconds), *speed* (mph), *xway* (express way), *dir* (direction), *seg* (segment), etc. The benchmark requires computing the traffic statistics for each highway segment, i.e. the number of active cars, their average speed per minute, and the past 5-minute moving average of vehicle speed. Based on these per-minute per-segment statistics, the application computes the tolls to be charged to a vehicle entering a segment any time during the next minute, and notifies the toll in real time (notification is to be sent to a vehicle within 5 s upon entering the segment). The application also includes accident detection; an accident occurring in one segment will impact the toll computation of that segment as well as a few downstream segments. An accident is flagged when multiple cars are found to have stopped in the same location. The graphical representation of our implementation of the LR stream processing requirement is shown in Fig. 4 together with its corresponding stream query.

```
INSERT INTO toll_table SELECT minute, xway, dir, seg, lr_toll(r.traffic_ok, r.cars_volume)
FROM (
  SELECT minute, xway, dir, seg, cars_volume,
         lr_moving_avg(xway, dir, seg, minute, avg_speed) as mv_avg, traffic_ok
  FROM (
    SELECT floor(time/60)::integer AS minute, xway, dir, seg,
           AVG(speed) AS avg_speed, COUNT(distinct Vid)-1) AS cars_volume,
           MIN(trffic_ok) AS traffic_ok
    FROM (
      SELECT xway, dir, seg, time, speed, vid,
             lr_acc_affected(vid,speed,xway,dir,seg,pos) AS traffic_ok
      FROM STREAM_CYCLE_lr_data(60, 180)
      WHERE lr_notify_toll(vid, xway, dir, seg, time)>=0
    ) s
    GROUP BY minute, xway, dir, seg
  ) p
) r
WHERE r.mv_avg > 0 AND r.mv_avg < 40;
```

This query provides the following major functions.

- **Stream Source Function** - The streaming tuples are generated by the SSF *STREAM_CYCLE_lr_data(time, cycles)*, from the LR data source file with time-stamps, where parameter "*time*" is the time-window size in seconds; "*cycles*" is the number of cycles the query is supposed to run. For example, *STREAM_CYCLE_lr_data(60, 180)* delivers the position reports one-by-one until it detects the end of a cycle (60 s), and performs a "cut", then onto the next cycle, for a total of 180 cycles (for 3 h).
- **Segment statistics and toll generation** - As illustrated by the left hand side of Fig. 4, the tolls are derived from the segment statistics, i.e. the number of active cars, average speed, and the 5-minute moving average speed, as well as from detected accidents, and dimensioned by express way, direction and segment. We leveraged the *minimum, average* and *count-distinct* aggregate-groupby operators built into the SQL engine, and provided the *moving average* (lr_moving_avg) operator and the *accident detection* (lr_accident) operator in UDFs.
- **Toll persisting** - Required by the LR benchmark, the segment tolls of minute *m* should be generated within 5 s after *m*. The toll of a segment calculated in the past minute is applied to the cars currently entering into that segment. The generated tolls are inserted into a *segment toll table* (SegToll) with the transaction committed per cycle (i.e., per minute). Therefore the tolls generated in the past minutes are visible to the current minute.
- **Toll notification** - As shown on the right side of Fig. 4, the per-car toll notification is provided by the UDF *lr_notify_toll()* appearing in the following phrase

$$\text{WHERE lr\_notify\_toll(vid, xway, dir, seg, time)} >= 0$$

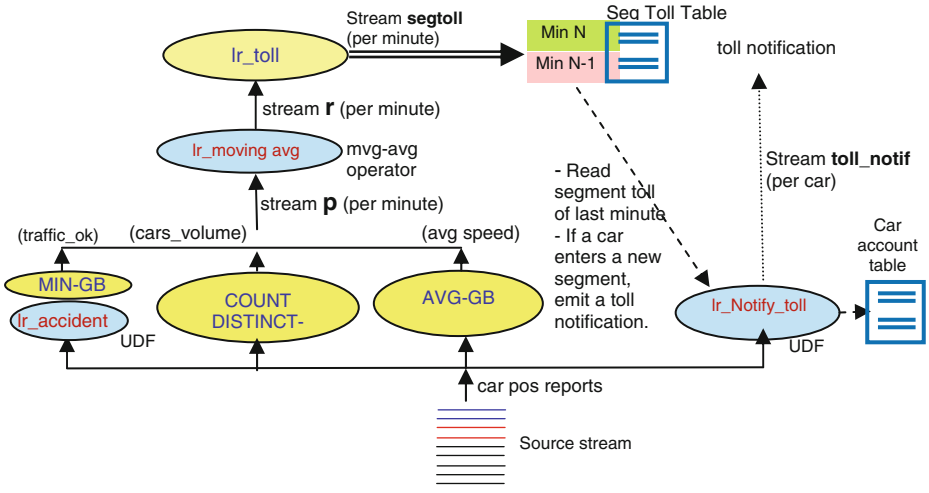

Fig. 4. Cycle based stream query for LR benchmark, for both the generation of per-minute, per cycle tolls common to all cars, and the per car based retrieval of resulting tolls

This UDF keeps enough information about active cars so as to detect the event of a car entering a new segment; and for each car entering a new segment, it emits a toll notification while persisting the toll to a table (carAccount table) for future account balance queries. This UDF reads the segment tolls of the previous minute within the FIRST_CALL part of the UDF (represented by the dash line), enabling it to use the information produced by the previous cycle of the stream query. Since this UDF is a per-tuple UDF (i.e., the NORMAL_CALL part of the UDF is invoked per input tuple), the toll notification is emitted immediately after the position report is received from the source stream, and does not wait for the current cycle (minute) to terminate. This UDF also persists the toll into the car account table. While the toll is notified immediately upon receiving the car position report, persisting the toll is committed once per cycle, in accordance to our CPCQ model.

Multiple features of our cycle-based stream processing approach are illustrated in this query:

- **Cut-and-Rewind.** This query repeatedly applies to each data chunks falling in 1-minute time-window as an execution cycle, and rewinds 180 times in the single query instance; the sub-query with alias *p* uses the standard SQL aggregate-groupby function to yield the number of active cars and their average speed for every minute dimensioned by segment, direction and express way. The SQL aggregate functions are computed for each cycle with no context carried over from one cycle to the next.
- **Sliding Window Function (per-tuple history sensitive).** The sliding window function *lr_moving_avg()* buffers the up to 5 per-minute average speed for accumulating the dimensioned 5-minute moving average; since the query is only rewound but not shut down, this buffer is retained continuously across query cycles – this is a critical advantage of cut/rewind over shutdown/restart.
- **Continuous Querying with Continuous Persisting.** The top-level construct of the LR query is actually the INSERT-SELECT phrase; with our engine extension, it persists the result stream returned from the SELECT query (r) to the toll table on the per-cycle basis. The transactional LR query commits per cycle to make the cycle based result accessible to subsequent cycles or other concurrent queries after the cycle ends. This cycle-based isolation level is supported with the appropriate locking mechanism.
- **Self-Referencing**. The per-car toll notification is generated by the UDF *lr_notify_toll()*. It efficiently accesses the segment toll in the *last minute* directly from the toll table. This kind of self-referencing provides a handshake mechanism for the *producer* part and the *consumer* part of the same query to rely on the query engine to synchronize, to perform history sensitive stream analytics, and to gain extremely high performance due to their seamless integration. We believe that such self-referencing represents a common paradigm in stream processing.

5.2 Experimental Setup

The experimental results are measured on HP xw8600 with 2 x Intel Xeon E54102 2.33 Ghz CPUs and 4 GB RAM, running Windows XP (x86_32) and PostgreSQL 8.4.

The input data are downloaded from the benchmark's home page. The "L = 1" setting was chosen for our experiment which means that the benchmark consists of 1 express way (with 100 segments in each direction). The event arrival rate ranges from a few per second to peak at about 1,700 events per second towards the end of the 3-hour duration. Figure 6 (Left) shows the distribution of data volume per minute, i.e. the per-minute throughput.

The LR data can be supplied in the following two modes:

- Stress test mode: the data are read by the SSF from a file continuously without following the real-time intervals (continuous input)
- Real-time input: the data are received from a data driver outside of the query engine with real-time intervals. Each car position report carries a system timestamp assigned by the data driver when the event is generated, which could be compared with the system timestamps generated during when toll notification is emitted, for measuring the response time.

We report our experimental results in these 2 different modes.

5.3 Performance Under Stress Test Mode

Time for computing segment tolls. Calculating the segment statistics and tolls has been recognized as the computation bottleneck of the benchmark in the literature. The LR benchmark requires the segment toll to be calculated based on the segment statistics and traffic status (whether affected by accidents) every minute. We took the left-hand-side of our LR model in Fig. 4 and ran that branch of the query up until the toll is computed, under the stress test mode. The total computation time with L = 1 setting is shown in Fig. 5 (Left). It shows that our system is able to generate the per-minute per-segments tolls for the total 3 h of LR data (approx. 12 Million tuples) in a little over 2 min.

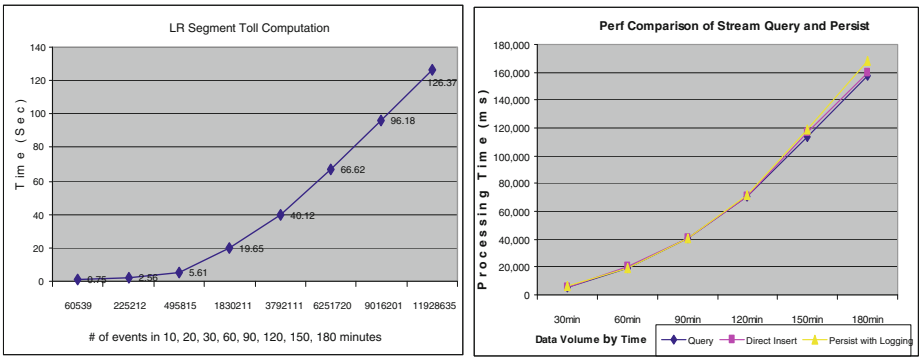

Fig. 5. **(Left)** Total time of toll computation. **(Right)** Performance comparison of querying-only and query + persisting (with continuous input)

Performance of Persisting with Heap-Insert Unlike other reported DSMSs where the stream processing results are persisted by connecting to a separate database and issuing queries, with the proposed cycle-based CQCP approach, the continuous, minute-cycle based query results are stored through efficient heap-insert.

From Fig. 5 (Right) we can see that persisting the cycle based stream processing results either by inserting with logging (using INSERT INTO with extended support by the query engine) or by direct inserting (using SELECT INTO with extended support by the query engine – not shown in this query), does not add significant performance overhead compared to querying only. This is because we completely push stream processing down to the query engine and handle it in a long running query instance with direct heap operations, with negligible overhead for data movement and for setting up update queries.

Post Cut Elapsed Time. In cycle-based stream processing, the remaining time of query evaluation after the input data chunk is cut, called Post Cut Elapsed Time (PCET), is particularly important since it directly affects the delta time for the results to be accessible after the last tuple of the data chunk in the cycle has been received.

Figure 6 (Left) shows the input data volume over 1-minute time windows (i.e., the stream workload). Figure 6 (Right) shows the query time, as well as the PCET, for processing each 1-minute data chunk. It can be seen that the PCET (the blue line) is well controlled around 0.2 s., meaning that the maximal response time for the segment toll results, as measured from the time a cycle (a minute) ends, is around 0.2 s.

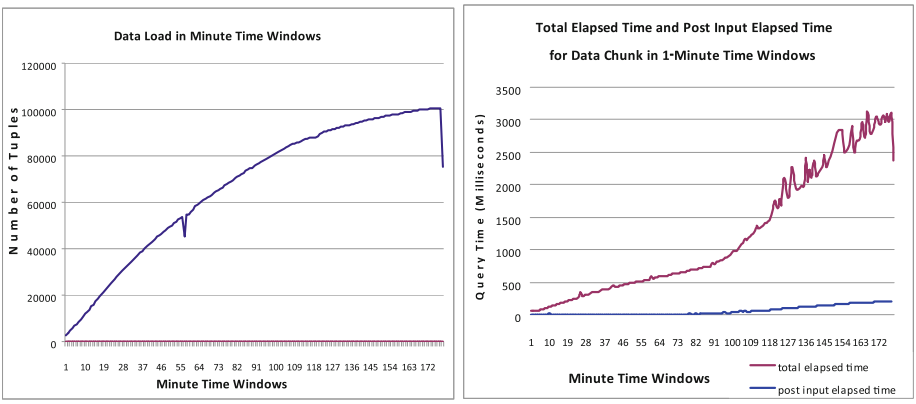

Fig. 6. (Left) Data load distribution over minute time windows **(Right)** Query time as well as PCET on the data chunk falling in each minute time window

5.4 Performance Under the Real-Time Input Mode

With real-time input, the events (car position reports) are delivered by a data driver in real-time with additional system-assigned timestamps. The query runs cycle by cycle on each one-minute data chunk. Figure 7 shows the maximal toll notification response time in each of the 180 1-minute windows.

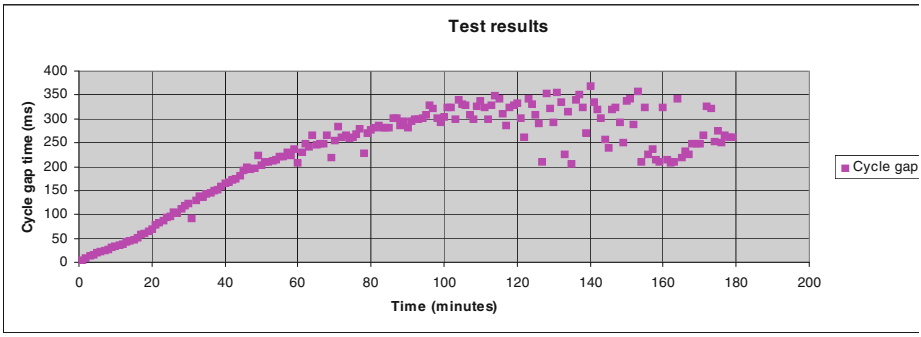

Fig. 7. Maximal toll notification response time in consecutive one-minute time windows

The maximal response time of toll notification really depends on the PCET measure introduced above, i.e. it is essentially the delay after a cycle is "cut" in completing the segment toll part of the query of that cycle. This is because in the beginning of each cycle, the toll notification cannot be emitted until the segment toll generation of the last cycle completes. In the first 2 h the toll notification response time is rather small, and with the increased data load in the last hour, it reaches the maximal value of about 0.3 s, which is still well within the 5-second latency requirement of the benchmark. Note that the maximal notification latency is not the average response time of notification. On the average, the notification response time is near zero, as the ones after the beginning of each cycle are not measurable by millisecond.

The experimental results indicate that our approach is highly competitive to any reported one. This is because we completely pushed stream processing down to the query engine with negligible data movement overhead and with efficient direct heap-insert. We eliminated the middleware layer, as provided by all other systems, for scheduling time-window based querying.

6 Cycle Based Map-Reduce

We rely on the Map-Reduce (MR) computation to scale out CaaaS. With the original MR model; static data are partitioned "horizontally" over cluster nodes for parallel computation; while enhancing the computation bandwidth by divide-and-conquer, it is not defined on unbounded stream data.

We envisage that Cut-and-Rewind (CR) provides a powerful mechanism for MR to reach stream analytics. We have investigated the combination of MR and CR on parallel database platform as well as on network distributed MR infrastructure.

6.1 Cut-Rewind a Parallel Query

A parallel query with UDFs can naturally express Map-Reduce computation. To explain how to apply CR to a parallel query engine for stream processing, let us review the parallel query execution process. A SQL query is parsed and optimized into a query

plan that is a tree of operators. In parallel execution multiple sub-plan instances, called fragments, are distributed to the participating query executors and data processors on multiple server nodes; at each node, the scan operator at the leaf of the tree gets and materializes a block of data, to be delivered to the upper layer tuple by tuple. The global query execution state is kept in the initial site.

To handle streaming data in parallel, the input stream is partitioned over multiple machine nodes, in the way similar to hash partitioning static data.

To support Cut-and-Rewind on a parallel database, every participating query engine is facilitated with the CR capability. The same ***cut*** condition is defined on all the partitioned streams. Note that if the cycle based continuous querying is "cut" on time window, the stream cannot be partitioned by time, but by other attributes.

A query execution cycle ends after *end-of-cycle* is signaled from *all* data sources, i.e. all the partitioned streams are "cut". As the *cut* condition is the same across all the partitioned streams, the cycle-based query executions over all nodes are well synchronized through data driven.

To parallelize the LR stream analysis, we hash partition the data stream by vehicle-id (vid); use the Map function to compute and pre-aggregate the segment traffic statistics per minute (without accident detection); use the Reduce function to globally aggregate the segment statistics, group by express-way, direction and segment, then calculate per segment moving average speed and finally the toll. The whole map-reduce implementation of the application is expressed in a ***single query*** running in the per-minute cycle.

As shown in Fig. 8, the LR stream is partitioned "horizontally" over Map nodes; all partitions are *cut* on the same one-minute boundary; the chunk-wise local results are shuffled to the Reduce nodes for global aggregation. The data partition of Map results is based on the standard parallel query processing of "group-by". The system runs cycle by cycle with Map-Reduce applied to data streams in each cycle, hence supporting scaled-out query processing over unbounded data streams.

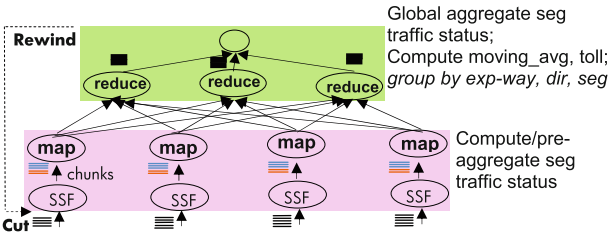

Fig. 8. Parallel DB based streaming map-reduce

This design is being integrated into a commercial parallel database engine where SSF is handled by the storage engine layer at each node, while the Map function and Reduce function are handled by query executers.

6.2 Network-Distributed Map-Reduce Scheme

In network distributed MR scheme, query-engine based stream engines are logically organized in the Map-Reduce style as illustrated in Fig. 9. The separation of "Map" engines and "Reduce" engines are logical, since an engine may act as a "Map" engine, a "Reduce" engine, or both.

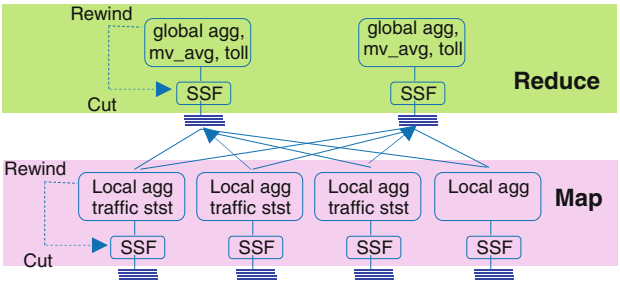

Fig. 9. Network distributed streaming map-reduce

Different from parallel database oriented MR, with the network distributed MR, a specific application is expressed in terms of two cycle based continuous queries, say CQ_{map} and CQ_{reduce}. The same CQ_{map} run at all the Map engines, and the same CQ_{reduce} at all the Reduce engines. The streams are partitioned and fed in multiple CQ_{map}; the resulting streams from CQ_{map} are shuffled to and fused by multiple CQ_{reduce} based on certain grouping criteria specified in the network replicated hash-tables. Those CQ_{map} and CQ_{reduce} synchronized by the same ***cut*** criteria, which determines the boundaries of input streams as well as the resulting streams.

With the above simplified LR example, the stream data are hash partitioned by vehicle ID; the stream data corresponding to express-ways, directions and segments are crossing Map nodes.

- The Map query, CQ_{map}, covers partitioned stream processing, up to the local aggregation of car-volume, speed-sum, group-by time and location.
- The results of CQ_{map} are treated as the input streams of the Reduce query, CQ_{reduce}, partitioned by express-way, direction and segment, based on the network replicated hash tables. Each CQ_{reduce} is also equipped with a SSF for receiving the Map results.
- CQ_{reduce} aggregate segment traffic statistics globally, calculate the segment moving average speed, and then the segment toll.

Both Map and Reduce queries run in the per-minute cycle.

Note the difference CR/MR schemes for parallel DB based and network-distributed MR infrastructure. Since the parallel query engine naturally supports reduce with aggregate-groupby, the MR is expressed by a single query, in each CR cycle the whole MR computation is iterated. With the network distributed MR infrastructure, the Map engines and the Reduce engines run separate cycle-based continuous queries; they

process the stream data chunk by chunk based on the common window boundary, or cut criterion, thus cooperate without centralized scheduling. The parallel DB based MR infrastructure generally over-performs the network-distributed one due to efficient data transfer from the Map nodes to the Reduce nodes, but the latter is more flexible and has obvious cost benefits.

7 Conclusions

Due to the growing data volume and the low-latency requirement, the *platform separation* of analytics and data management has become the performance bottleneck, and their integration offers great potential in real-time, data-intensive analytics.

In this work we have addressed several specific challenges. Our thesis is that database technology can be extended and applied to real-time continuous analytics service provisioning.

We reported our experience in leveraging the DBMS for continuous stream analytics. We tackled the key technical issues for integrating stream analytics capability into the existing query engine, and built an integrated, efficient and robust system with stream processing capability while retaining the full DBMS functionality, giving the query engine a new role. We proposed the *cut-and-rewind* query execution model for chunk-wise continuous stream processing with the full SQL power, while enabling history-sensitive stream operations. We provided advanced stream processing capability by extending the existing query engine directly without introducing separate executor or additional "middleware". With this approach we have bridged SQL and stream processing in a single engine.

Our platform significantly differs from the current generation of stream processing systems which are in general built separately from the database systems. As those systems do not have the full SQL expressive power and DBMS functionalities, incur significant overhead in data access and movement, and lack the appropriate transaction support for continuously persisting and sharing results, they fail to meet the requirements for providing high-throughput, low-latency service provisioning.

Further, the cycle-based query model allows multiple query engines to synchronize and cooperate based on the common window boundaries. Such data-driven cooperation is very different from the workflow like centralized scheduling used in other stream processing systems. This feature allows us to apply MR cycle by cycle continuously and incrementally for parallel and distributed continuous analytics, in the way not seen previously. Accordingly, we investigated two kinds of parallel computing infrastructures, one based on parallel database engine; and another based on network distributed Map-Reduce but with extended streaming capability.

The proposed approach has been implemented on the PostgreSQL engine. Our future work includes further refinement of our unified query and transaction model, further characterization and classification of UDFs (to enable optimization) and building out stream analytics operators, additional extensions required for the optimizer and query pipeline, and providing a front-end for demonstrating the live stream analytics. As pointed out in [12], big data visualization issues are tightly coupe with analytics. We are also investigating the use of a massively parallel processor

(MPP)-based, data-intensive streaming analytics platform, and looking into the issues of privacy preservation which plays a critical roles in analytics in both centralized and distributed environments [13, 14].

References

1. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
2. Abadi, D.J., et al.: The design of the borealis stream processing engine. In: CIDR (2005)
3. Arasu, A., B, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)
4. Bryant, R.E.: Data-intensive supercomputing: the case for DISC. In: CMU-CS-07-128 (2007)
5. Chandrasekaran, S., et al.: TelegraphCQ: continuous dataflow processing for an uncertain world. In: CIDR (2003)
6. Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. VLDB **1**(2), 1265–1276 (2008)
7. Chen, J., et al.: NiagaraCQ: a scalable continuous query system for internet databases. In: SIGMOD (2000)
8. Chen, Q., Hsu, M.: Cooperating SQL dataflow processes for In-DB analytics. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009, Part I. LNCS, vol. 5870, pp. 389–397. Springer, Heidelberg (2009)
9. Chen, Q., Hsu, M., Liu, R.: Extend UDF technology for integrated analytics. In: Pedersen, T. B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 256–270. Springer, Heidelberg (2009)
10. Cooper, B.F., et al.: PNUTS: Yahoo!'s hosted data serving platform. VLDB. **1**(2), 1277–1288 (2008)
11. Cranor, C.D., et al.: Gigascope: a stream database for network applications. In: SIGMOD (2003)
12. Cuzzocrea, A., Mansmann, S.: OLAP visualization: models, issues, and techniques. In: Wang, J. (ed.) Encyclopedia of Data Warehousing and Mining, 2nd edn, pp. 1439–1446. IGI Global, Hershey (2009)
13. Cuzzocrea, A., Saccà, D.: Balancing accuracy and privacy of OLAP aggregations on data cubes. In: Proceedings of the 13th ACM International Workshop on Data Warehousing and OLAP (DOLAP 2010) in conjunction with 19th ACM International Conference on Information and Knowledge Management (CIKM 2010), Toronto, pp. 93–98, 26–30 October 2010
14. Cuzzocrea, A., Bertino, E.: A secure multiparty computation framework for privacy preserving OLAP over distributed XML data. In: Proceedings of the 25th ACM International Symposium on Applied Computing (SAC 2010), Sierre, pp. 1666–1673, 22–26 March 2010
15. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.C.: SPADE: the system s declarative stream processing engine. In: ACM SIGMOD (2008)
16. Franklin, M.J., et al.: Continuous analytics: rethinking query processing in a network-effect world. In: CIDR (2009)

17. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys 2007, March 2007
18. Jain, N., et al.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: SIGMOD (2006)
19. Liarou, E., et.al.: Exploiting the power of relational databases for efficient stream processing. In: EDBT (2009)
20. Zeller, H.: NonStop SQL/MX publish subscribe: continuous data streams in transaction processing. In: SIGMOD Conference (2003)