Experience in Extending Query Engine for Continuous Analytics

Qiming Chen, Meichun Hsu

HP Laboratories
HPL-2010-44

Keyword(s):

In-Database Stream Processing

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 not able to take advantage of the functionalities already offered by the existing data warehouse systems. In this work we tackle some hard problems not properly addressed previously 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 generalize query engines to support the unified model. We propose the cut-and rewind query execution model to allow a query to be applied to stream data by converting the latter into a sequence of "chunks", and executing the query over each chunk sequentially without shutting the query instance down between chunks;, 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 more mature codebase 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 HP SeaQuest parallel analytics engine is work in progress.

External Posting Date: May 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: May 21, 2010 [Fulltext]

Experience in Extending Query Engine for Continuous Analytics

Qiming Chen
HP Labs
Palo Alto, California, USA
Hewlett Packard Co.
qiming.chen@hp.com

Meichun Hsu
HP Labs
Palo Alto, California, USA
Hewlett Packard Co.
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 not able to take advantage of the functionalities already offered by the existing data warehouse systems.

*In this work we tackle some hard problems not properly addressed previously 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 generalize query engines to support the unified model. We propose the **cut-and-rewind** query execution model to allow a query to be applied to stream data by converting the latter into a sequence of "chunks", and executing the query over each chunk sequentially without shutting the query instance down between chunks;, 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 more mature 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 HP SeaQuest parallel analytics engine is work in progress.

1. Problem Statement

Streaming analytics is a data-intensive computation chain from event streams to analysis results. In response to the rapidly growing data volume and the increasing need for lower latency, Data Stream Management Systems (DSMSs) provide a paradigm shift from the load-first analyze-later mode of data warehousing [13,16].

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; 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 analytics require operators that are history sensitive; these operators are often based on windows over the stream data, and there is a need to efficiently maintain the state or a synopsis of the data that falls within a window.
- 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.

2. Our 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.

We report our experience in leveraging the PostgreSQL engine for supporting stream processing. The proposed mechanism has been implemented with necessary engine extensions, 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 HP SeaQuest parallel database engine is currently being explored.

2.1 The Unified Query Model

We illustrate our approach to unifying queries over static and stream data with the following example application, first expressed as a query over static relations, and then as a hybrid query that includes a stream source. The graphical representation of the 2 queries is shown in Fig. 1.

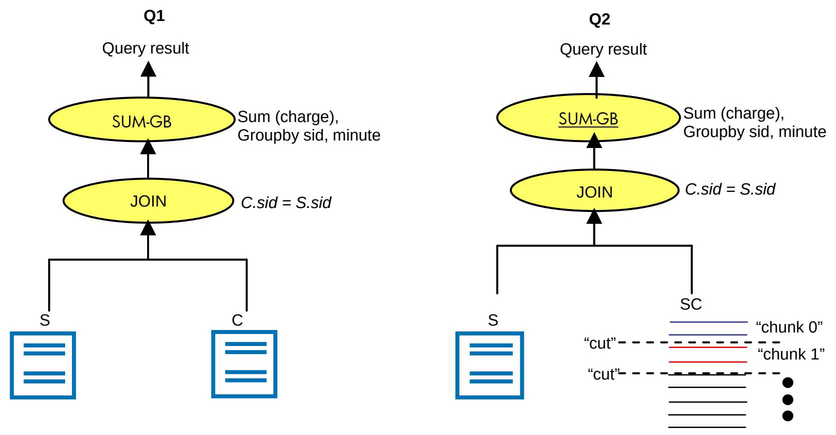

Fig. 1: An application computes the total amount of toll charged for each highway segment per minute. The inputs are two stored relations, *C* and S.

- *C* contains the event that a car enters a tolled segment, with the schema (*cid*, *sid, ts*), where *cid* is a unique identifier of the vehicle, *sid* is the identifier of the highway segment, and *ts* is the timestamp in second. For example, a tuple <12717, 13, 340> says that vehicle 12717 enters segment 5 during minute 5.
- *S* contains the highway segment info with schema (*sid, toll*) where *toll* is the toll per car for segment *sid*.

The following SQL query implements this application (as shown on the left of the above figure):

Q1:
SELECT sid, floor(ts/60) as minute, sum(charge)
FROM S, C
WHERE C.sid = S.sid
GROUPBY sid, minute

However, if the table *C* above is not a stored relation, but a real-time stream source, while the segment info *S* remains a stored relation, and the output is expected to appear with minimal latency every minute, then the above application becomes a streaming application. With our approach for integrated stream processing, 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* in *Q1* as follows (shown on the right of the above figure):

Q2:
SELECT sid, floor(ts/60) as minute, sum(charge)
FROM S, STREAM (SC, cycle-spec)
WHERE SC.sid = S.sid
GROUPBY sid, minute

In the above query, *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 $SC_{C\ i}$ occur before any tuple in SC_{Ci+1} in the stream. The criterion for determining the "cut point" is specified in the *cycle-spec*. For example, in this application, the cycle-spec is set to be "per minute". 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 $SC_{C\ i}$'s in the stream source *SC*.

In general, our proposed unified model is defined as follows:

Given a query Q over a set of relations $R_1,..,R_n$ and an infinite stream of relation tuples *S* with a criterion C for cutting *S* into an unbounded sequence of chunks, e.g. by every 1-minute time window,

$$<S_{C0}, S_{C1}, \ldots, S_{Ci}, \ldots>$$

where S_{Ci} denotes the *i-th* "chunk" of the stream according to the chunking-criterion C. S_{Ci} can be interpreted as a relation. The semantics of applying the query Q to the unbounded stream *S* plus bounded static relations $R_1,..,R_n$ lies in

$$Q(S, R_1,..,R_n) \rightarrow < Q(S_{C0}, R_1,..,R_n), \ldots Q(S_{Ci}, R_1,..,R_n), \ldots >$$

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

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 (e.g. run for 300 cycles), user intervention, or a special end-of-stream signal received from the stream source.

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. Extensions to allow multiple streams or hybrid queries to interact without going through database tables are being investigated.

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 elements of our approach are introduced in the following subsections.

2.2 Stream Source Function

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. The scan operator 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.

We start with providing unbounded relation data to feed 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[1], called Stream Source Function (SSF) that listens or reads data/events sequence and returns a sequence of tuples to feed queries without first storing the tuples on disk. A SSF is called multiple, up to infinite, times during the execution of a stream query, each call returns one tuple. When the pre-specified end-of-cycle condition is detected, the SSF signals the query engine to terminate the current query execution cycle.

2.3 Cycle Based Streaming Query Execution with Cut-and-Rewind

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 next subsection.

2.4 Stream Analytics and Window Operators based on UDFs

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 standard SQL engine contains a number of built-in analytic operators, stream history-sensitive operators are not supported. User Defined Functions (UDFs) are 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.

[1] A *table function* in a relational database system is a user-defined function (UDF) that returns a table.

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[2] 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.

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.

2.5 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 ends, 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 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.

Query Cycle based Transaction Model. Conventionally a query is placed in a transaction boundary. In general, the query result and the possible update effect are made visible only after the commitment of the transaction (although weaker transaction semantics do exist). In order to allow the result of a long-running stream query results to be incrementally accessible, we introduce the cycle-based transaction model incorporated with the *cut-and-rewind* query model, which we call *continuous querying with continuous persisting*, (CQCP). Under CQCP, 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. The isolation level is Cycle based Read Committed (CRC). To allow the cycle results to be continuously visible to external world, regardless of the table is under the subsequent cycle-based transactions, we enforce record level locking.

We extended both SELECT INTO and INSERT INTO facilities of the PostgreSQL to support CQCP. We also added an option to force the data to stay in memory, and an automatic space reclaiming utility should the data be written to the disk.

Continuous Persisting. In a regular database system, the queries with SPJ (Select, Project, Join) operations and those with the update (Insert, Delete, Update) operations are different in the flow of resulting data. In a SPJ query,

[2] A *scalar* UDF is a user-defined function that takes a single tuple as input and returns a single tuple as result.

the destination of results is a query receiver connected to the client. In a data update query, such as insert, the results are emitted to, or synched to, the database.

In stream processing, such separation would be impractical. The analytic results must be streaming to the client continuously as well as being stored in the database if needed for other applications to access. Therefore, we extended the query engine to have query evaluation and results persisting integrated and expressed in a single query. This two-receiver approach makes it possible to have the results both persisted and streamed out externally.

Certain intermediate stream processing results can be deposited into the database from UDFs. To do so the UDF must be relaxed from the read-only mode, and employ the database internal query facility to form, parse, plan and execute queries efficiently. In our prototype, the PostgreSQL SPI (Server Program Interface) is used.

3. Evidence the Solution Works

3.1 Modeling the Linear Road Benchmark

We use the widely-accepted Linear-Road (LR) benchmark [15] to demonstrate our extended query engine. The LR benchmark models the traffic on express ways for a 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 seconds 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 seconds 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. 2 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;
```

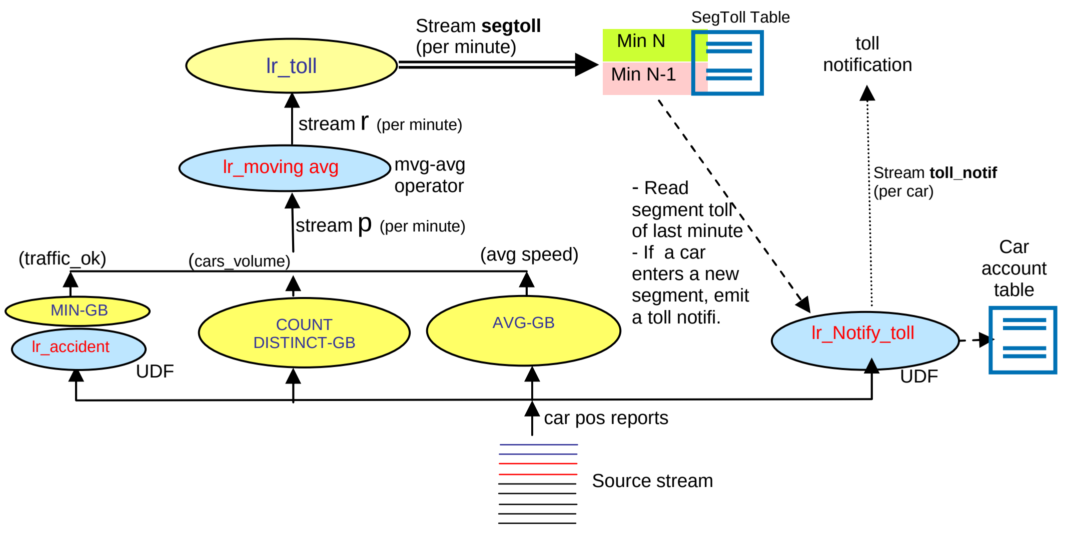

Fig. 2. Cycle based stream query for LR benchmark, for both the generation of per-minute, per cycle tolls common to all cars, and the per car based retrieval of resulting tolls

This 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 timestamps, 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 seconds), and performs a "cut", then onto the next cycle, for a total of 180 cycles (for 3 hours).

- **Segment statistics and toll generation -** As illustrated by the left hand side of Fig. 2, 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 seconds 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. 2, the per-car toll notification is provided by the UDF *lr_notify_toll()* appearing in the following phrase

```
WHERE lr_notify_toll(vid, xway, dir, seg, time) >= 0
```

  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 FirstCall 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 NormalCall 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.

3.2 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 Window XP (x86_32), running 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. Fig. 4(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.

3.3 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 2 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. 3 (Left). It shows that our system is able to generate the per-minute per-segments tolls for the total 3 hours of LR data (approx. 12 Million tuples) in a little over 2 minutes.

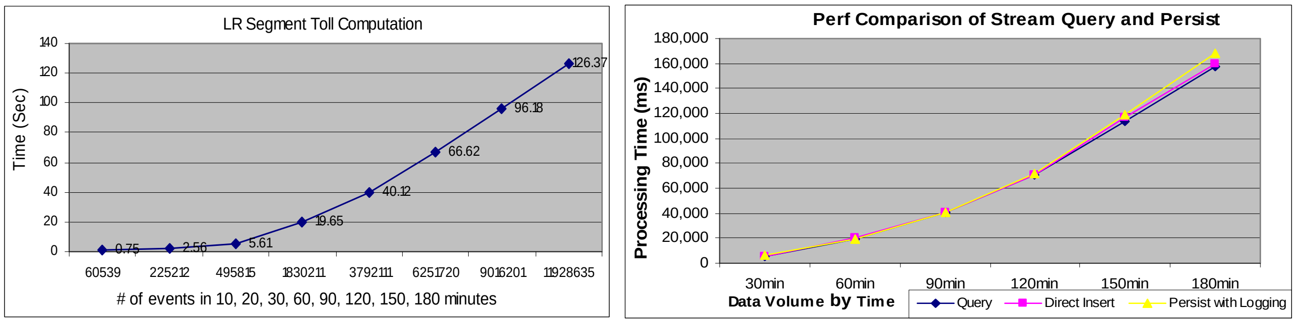

Fig. 3 (Left) Total time of toll computation. **(Right)** Performance comparison of querying-only and query+persisting (with continuous input)

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. 3 (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.

Fig 4 (Left) shows the input data volume over 1-minute time windows (i.e., the stream workload). Fig. 4 (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 below 0.2 seconds, 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 seconds.

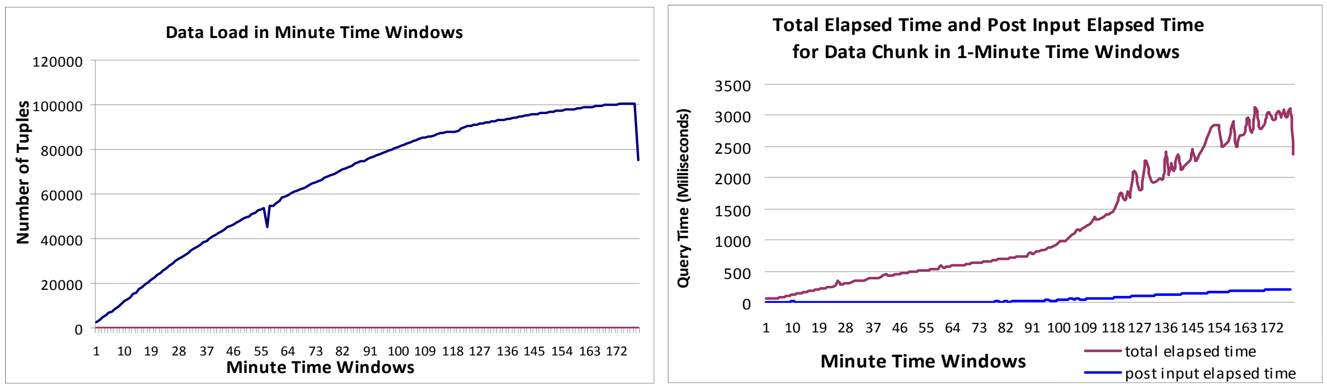

Fig. 4 (Left) Data load distribution over minute time windows **(Right)** Query time as well as PCET on the data chunk falling in each minute time window

3.4 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. Fig 5 shows the maximal toll notification response time in each of the 180 1-minute 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 hours 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 second, which is well below 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 much smaller since except the first few in a minute, most of them are near 0.

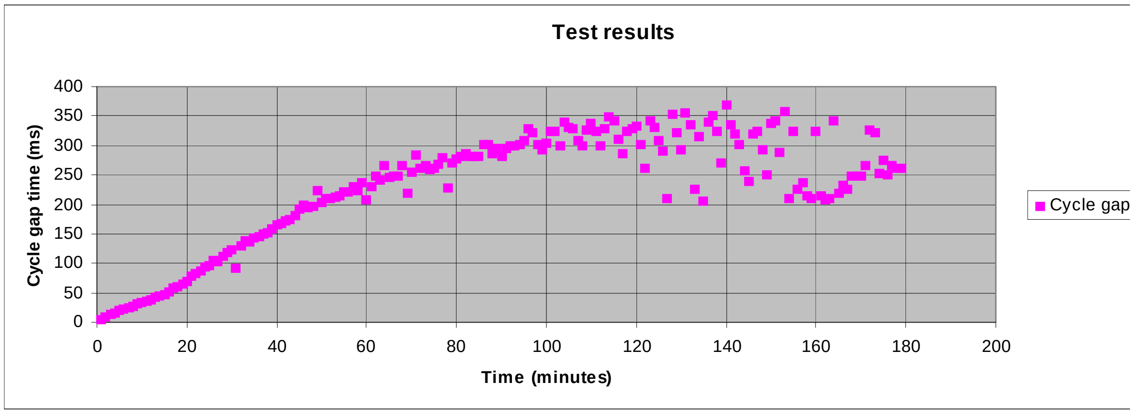

Fig. 5 Maximal toll notification response time in consecutive one-minute time windows

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.

4. Competitive Approaches

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) [12], STREAM (Stanford) [3], TelegraphCQ (Berkeley) [5], as well as Arora, Borealis, etc [1,2,7,11]. 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. Two recently reported systems, the TruSQL engine [13] developed by Truviso Inc, USA, and the DataCell engine [16] 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.

5. Current Status and Next Steps

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 paper 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.

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 an investigation into using HP SeaQuest for providing a massively parallel processor (MPP)-based, data-intensive streaming analytics platform.

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, 2003.
2. D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In CIDR, 2005.
3. Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal, (15)2, June 2006.
4. R.E. Bryant. “Data-Intensive Supercomputing: The case for DISC”, *CMU-CS-07-128*, 2007.
5. Chandrasekaran, S., et. al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003.
6. Chaiken, B. Jenkins, P-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, “SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets”, *VLDB* 2008.
7. J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In SIGMOD, 2000.
8. Qiming Chen, Meichun Hsu, “Cooperating SQL Dataflow Processes for In-DB Analytics”, CoopIS 2009.
9. Qiming Chen, Meichun Hsu, Rui Liu, "Extend UDF Technology for Integrated Analytics", DaWaK 2009.
10. B. F. Cooper, et.al, “PNUTS: Yahoo!’s Hosted Data Serving Platform”, *VLDB* 2008.
11. C. D. Cranor et al. Gigascope: A Stream Database for Network Applications. SIGMOD, 2003.
12. Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, Myung Cheol Doo, “*SPADE*: The *System S* Declarative Stream Processing Engine”, *ACM SIGMOD 2008*.
13. Michael J. Franklin, et al, “Continuous Analytics: Rethinking Query Processing in a Network Effect World”, CIDR 2009.
14. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks”, *In EuroSys 2007*, March 2007.
15. N. Jain et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In SIGMOD, 2006.
16. E. Liarou et.al.“Exploiting the Power of Relational Databases for Efficient Stream Processing”, EDBT 2009.
17. H. Zeller. “NonStop SQL/MX Publish Subscribe: Continuous Data Streams in Transaction Processing”, SIGMOD Conference 2003.