Experience in Continuous analytics as a Service (CaaaS)

Qiming Chen
HP Labs
Palo Alto, CA, USA
1(650)857-4013
qiming.chen@hp.com

Meichun Hsu
HP Labs
Palo Alto, CA, USA
1(650)857-2219
meichun.hsu@hp.com

Hans Zeller
HP SW NED
Cupertino, CA, USA
1(408)447-6350
hans.zeller@hp.com

ABSTRACT

Mobile applications, such as those on WebOS, increasingly depend on continuous analytics results of real-time events, for monitoring oil & gas production, watching traffic status and detecting accident, etc, which has given rise to the need of providing Continuous analytics as a Service (CaaaS). While representing a paradigm shift in cloud computing, CaaaS poses several challenges in scalability, latency, time-window semantics, transaction control and result-set staging.

A data stream is infinite thus can only be analyzed in granules. We propose a continuous query model over both static relations and dynamic streaming data, which allows a long-standing SQL query instance to run cycle by cycle, each cycle for a chunk of data from the data stream, using a *cut-and-rewind* mechanism. We further support the *cycle-based transaction model* with cycle-based isolation and visibility, for delivering analytics results to the clients continuously while the query is running. To have the continuously generated analytics results staged efficiently, we developed the *table-ring and label switching* mechanism characterized by staging data through metadata manipulation without physical data moving and copying. To scale-out analytics computation, we support both parallel database based and network distributed Map-Reduce based infrastructure with multiple cooperating engines.

We have built the proposed infrastructure by extending the PostgreSQL engine. We tested the throughput and latency of this service based on a well-known stream processing benchmark; the results show that the proposed approach is highly competitive. Our experiments indicate that the database technology can be extended and applied to real-time continuous analytics service provisioning.

Categories and Subject Descriptors

H2.4 [**Query Processing**]

General Terms

Management, Performance, Design, Experimentation.

Keywords

Continuous query, Stream analytics, Cloud service.

EDBT'2011, March 22-24, 2011 --- Uppsala, Sweden

1. INTRODUCTION

Internet applications increasingly depend on the analytics results of real-time events, such as the traffic status summarized from the location and speed of many individual cars. New technologies like HTML5, which does local caching, could help mobile application to get past the internet access speed barrier in using the real-time information service. In the age of planetary computing, it is expected that people are always interconnected and rely on such continuous event analytics results for their work and life. This has given rise to the need of providing Continuous analytics as a Service (CaaaS).

CaaaS is a cloud computing model for enabling convenient, on-demand network access to a shared pool of event analytics results. Today, there are already some good examples of mobile cloud computing applications including mobile Gmail, Google Maps, location service and some navigation applications. This trend will continue. However, to the best of our knowledge, treating continuous analytics as a cloud service has not yet been addressed properly.

1.1 The Problem

CaaaS presents opportunities as well as challenges. The conventional data warehouse approach of store-first-analyze-later does not fit in CaaaS; instead, continuous analytics should be applied before the data are warehoused. The current generation of Data Stream Management Systems (DSMS) is not appropriate for CaaaS since they are in general built separately from the data warehouse and query engine, incurring significant overhead in data access and data movement, and is unable to take advantage of the functionalities already offered by the existing DBMS [2-5,13]. In general, to provide CaaaS, the following hard problems must be solved.

- Integrating continuous stream analytics with queries over stored data.
- Analyzing stream data chunk by chunk while maintaining the continuity of application context as required by sliding-window oriented operations.
- Supporting efficient data staging with respect to continuous, unbounded analytics results.
- Scaling out continuous data analytics with parallel and distributed infrastructure.

1.2 The Solution

Our support of CaaaS is characterized by the following.

- We define a unified query model over both stored relations and dynamic streaming data, and develop techniques to extend query engines to support the unified model. We introduce the *cut-and-rewind* query execution mechanism to allow a SQL

query to be executed cycle by cycle, each cycle for a chunk of data in the data stream, without shutting the query instance down between chunks, such that the application state can be continuously maintained across execution cycles to support state-dependent operations such as sliding-window operations. Correspondingly, we support the *cycle-based transaction model*, characterized by the cycle-based isolation and visibility, for making the analytics results visible to the clients of cloud service continuously while the query for generating these results is running.

- In order to support fast access to the analytics results we develop the "table-ring" mechanism which allows the infinite analytics results to be kept in a list of small-sized tables based on time sequence, and staged through "switching labels" without actual data copying and moving. As a result, these analytics results are easily manageable and downloadable to the mobile devices running WebOS and HTML 5.
- We scale-out CaaaS by having multiple engines cooperating based on the common data chunking criterion, on both parallel database and network distributed Map-Reduce infrastructures.

Our CaaaS architecture for delivering real-time event analytics results involves multiple cloud components communicating with each other over APIs; it unifies the capabilities of stream processing and query processing in its specific way. Mobile applications can use the continuous analytics results generated by pre-defined continuous queries, or create specific continuous queries on the event streams. In this sense, our work actually covers SaaS (Software as a Service) as well as PaaS (Platform as a Service). There exist a number of common issues for managing connection, security, privacy, etc, which are not discussed in this report.

The rest of this paper is organized as follows: Section 2 deals with the convergence of stream analysis and query processing, the cycle based query model and transaction model, as well as the staging mechanism for efficiently handling continuous and infinite stream analytics results. Section 3 describes two kinds of parallel infrastructures for CaaaS. Section 4 discusses related work and concludes the paper.

2. CYCLE-BASED CONTINUOUS STREAM ANALYTICS

We view a pipelined query engine essentially as a streaming engine. We advocate the use of an extended SQL model that unifies queries over both streaming and stored relational data, and an extended query engine for integrating stream processing and DBMS.

To illustrate the extended model and mechanisms, we will use a traffic system hypothesized in the Linear-Road (LR) benchmark [15] throughout this section. In the LR benchmark, vehicles are equipped with GPSs and emit signals to report their positions and speeds every 30 seconds. Each reading constitutes an event with attributes *vid* (vehicle ID), *time* (seconds), *speed* (mph), *xway* (express way), *seg* (segment of the express way), *dir* (direction), etc. The benchmark requires computing the traffic statistics for each highway segment. 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.

2.1 Stream Analytics as Continuous Query

We will first 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.

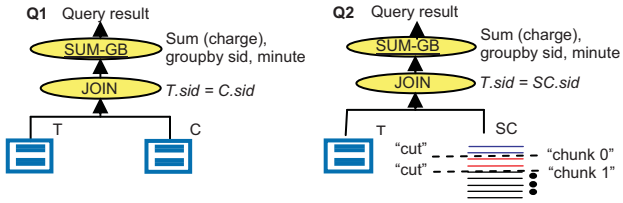

Figure 1. Querying static table vs. querying data stream chunk by chunk

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 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):

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

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

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

2.2 Cycle-based Continuous Query Model

For supporting the above semantics, we adopt User Defined Functions (UDFs) to code the stream analysis not directly expressible by SQL operators and functions, on which we had extensive studies [7-9]. We allow a UDF to be provided with a data buffer in its function closure, for caching stream processing state (synopsis), and in this way to support consecutive or sliding window based computation.

We developed the *cut-and-rewind* query execution mechanism, namely, cut a query execution based on the cycle specification and then rewind the state of the query without shutting it down, for processing the next chunk of stream data in the next cycle.

Cut is originated in the SSF at the bottom of the query tree. Upon detection of end-of-cycle condition, the SSF signals *end-of-cycle* to the query engine resulting in the termination of the current query execution cycle.

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) is retained. The SSF is specifically registered or named with a prefix (e.g. "STREAM") to instruct the query engine to use the cut-and-rewind mechanism. Since the query instance remains alive across cycles, data for sliding-window oriented, history sensitive operations can be kept continuously. Bringing these two capabilities together is the key in our approach.

2.3 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.

2.4 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 hours 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.

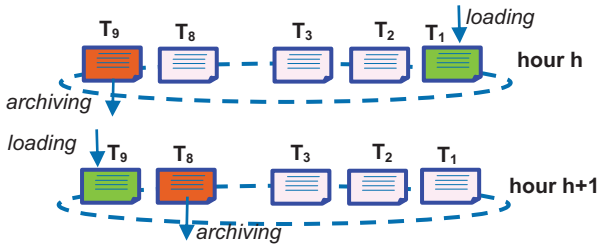

Figure 2. Table-ring approach for staging analytics results through metadata manipulation without data copy/move

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.

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 Section 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 into-relation will act as the query destination.

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

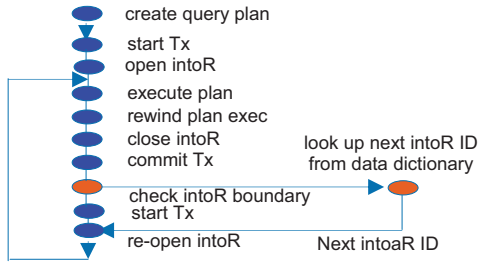

Figure 3. Cycle-based query execution, transaction, staging

2.5 Experimental Results with Linear Road

We implemented the Linear-Road (LR) benchmark [15] using the above model to test the CaaaS scalability and performance. Our stream query is listed below; its graphical representation is shown in Fig. 4.

```
INSERT INTO toll_table SELECT minute, xway, dir, seg,
toll_comp(c.no_accident, c.cars_volume) FROM (
  SELECT minute, xway, dir, seg, cars_volume,  5_minutes_moving_avg(xway,
  dir, seg, minute, avg_speed) as mv_avg,   no_accident FROM (
    SELECT floor(time/60)::integer AS minute, xway, dir, seg, AVG(speed) AS
    avg_speed, COUNT(distinct Vid)-1) AS cars_volume, MIN(no_accident) AS
    no_accident FROM (
      SELECT xway, dir, seg, time, speed, vid,
      accident_affected (vid,speed,xway,dir,seg,pos) AS no_accident
      FROM STREAM_CYCLE_lr_data(60, 180) ) s
   GROUP BY minute, xway, dir, seg ) r
 ) c
WHERE c.mv_avg > 0 AND c.mv_avg < 40;
```

With the above query, the streaming tuples are read by the SSF, *STREAM_CYCLE_lr_data(w, cycles)*, from the LR data source file with timestamps, where parameter "*w*" is the time-window size in seconds; "*cycles*" is the number of cycles the query is supposed to run (setting cycle to 0 means running the query indefinitely). 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). We leveraged the *minimum, average* and *count-distinct* aggregate-groupby operators built in the SQL engine, and provided the moving average function, *5_minutes_moving_avg*(), and the accident detection function, *accident_affected*() as UDFs. The generated tolls are inserted into one of the segment toll tables based on the staging mechanism described earlier.

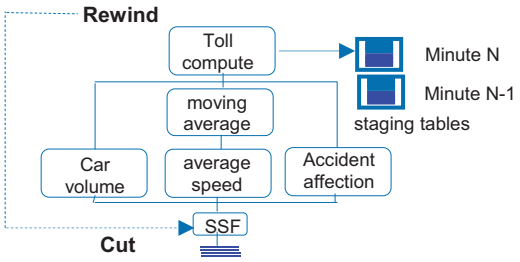

Figure 4. Cycle based continuous query for LR benchmark

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). Below we present two important experiment results.

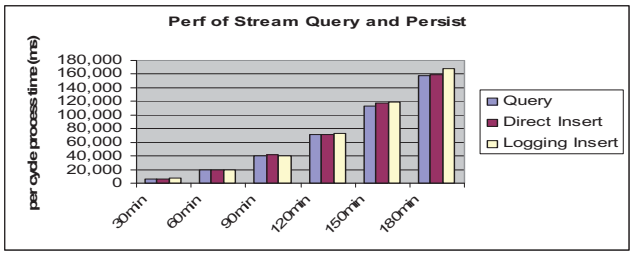

Figure 5. Performance comparison: cycle based continuous query-only vs continuous query with continuous persisting

From Fig. 5 we can see that persisting the cycle based stream processing results either by inserting with logging (using "INSERT INTO" or by direct inserting (using "SELECT INTO", both with extended support by the query engine internally), does not add significant performance overhead compared to querying only. This is because we completely push stream processing down to the query engine as efficient heap operations.

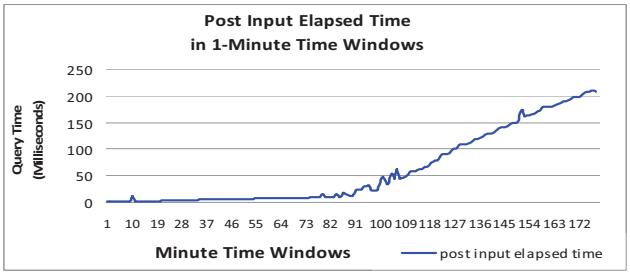

Figure 6. Query time and PCET for minute time windows

Fig 6 shows that the per-cycle (one minute) results can be made accessible in around 0.2 second, after the cycle ends; this time delta, called Post Cycle Elapse Time (PCET), is actually the maximal response time for retrieving the computation results (segment toll) generated by the past query cycle. The increase of

the responding time in later time windows is caused by the increased data volumes provided by the benchmark.

As mentioned above, LR benchmark requires the segment tolls in each minute to be available within 5 second after that minute, and most reported results fall in the range from 1 second to 5 seconds. Our experimental result (0.2 second) indicates that our approach is highly competitive to any reported one.

3. CYCLE BASED MAP-REDUCE

We rely on the Map-Reduce (MR) computation to scale out CaaaS. With the original MR model [10], 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.

3.1 Cut-Rewind a Parallel Query

We have shown in [8] that 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 should not 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 7, 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.

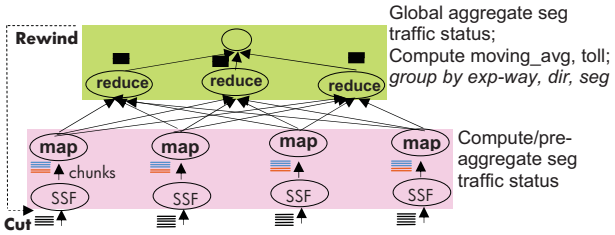

Figure 7. Parallel DB based streaming Map-Reduce

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.

3.2 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. 8. The separation of "Map" engines and "Reduce" engines are logical, since an engine may act as a "Map" engine, a "Reduce" engine, or both.

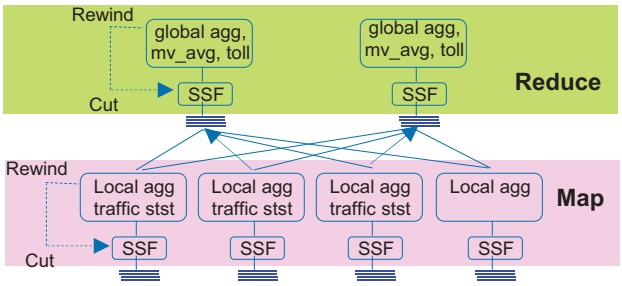

Figure 8. Network distributed streaming Map-Reduce

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 later is more flexible and has obvious cost benefits.

4. CONCLUSION AND RELATED WORK

CaaaS represents a paradigm shift and opens a new field in cloud computing as more and more applications depend on the analytics results of real-time events. In this work we have addressed several specific challenges and reported our solutions. Our thesis is that database technology can be extended and applied to real-time continuous analytics service provisioning.

We start with unifying dynamic stream processing and static data management for data intensive analytics. To capture the window semantics, we have introduced the cycle-based query model and transaction model which allows SQL queries to run and to commit cycle by cycle for analyzing unbounded stream data chunk by chunk, thus making the analysis results visible to the clients of cloud service timely while the continued query for generating them is still running. We then proposed the table-ring mechanism for staging analytics results without physical data copying and moving, that is especially efficient in coping with continuous information generation. Finally, we have developed two kinds of parallel computing infrastructures, one based on parallel database engine with performance consistent with the study given in [11]; and another based on network distributed Map-Reduce that is architecturally similar to HadoopDB [1], but with extended streaming capability.

Our platform significantly differs from the current generation of DSMSs which are in general built separately from the database systems [2-5,13]. As those systems do not have the full SQL expressive power and DBMS functionalities, incur significant overhead in data access and movement [7-9,11], and lack the appropriate transaction support for continuously persisting and sharing results, they fail to meet the requirements for providing high-throughput, low-latency CaaaS.

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 DSMSs [12,16]. 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.

Metadata manipulation is also used in Oracle's table partition management [17]; that, however, is not designed for round-robin table staging. Our staging mechanism avoids the overhead of physical data copying and moving thus ensures low-latency data retrieval without being interrupted by data staging. For mobile applications such as those running on WebOS with HTML 5 caching capability, keeping analytics results in small sized tables makes them easily downloadable for batch usage with reduced internet connections.

5. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads", VLDB 2009.

[2] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal, (15)2, June 2006.

[3] 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. In VLDB J (12)2: 120139, August 2003.

[4] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In CIDR, 2005.

[5] Chandrasekaran, S., et. al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003.

[6] Qiming Chen, Meichun Hsu, "Continuous MapReduce for In-DB Stream Analytics", Proc. CoopIS 2010.

[7] Qiming Chen, Meichun Hsu, "Integrate Analytic Streaming into Query Engine", Tech Rep HPL-2010-44, 2010.

[8] Qiming Chen, Andy Therber, Meichun Hsu, Hans Zeller, Bin Zhang, Ren Wu, "Efficiently Support Map-Reduce alike Computation Models Inside Parallel DBMS", Proc. IDEAS'09, 2009.

[9] Qiming Chen, Meichun Hsu, Rui Liu, "Extend UDF Technology for Integrated Analytics", Proc. DaWaK 2009.

[10] J. Dean., "Experiences with MapReduce, an abstraction for large-scale computation", Int Conf on Parallel Architecture and Compilation Techniques. ACM, 2006.

[11] D.J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, A. Krioukov, "Clustera: An Integrated Computation And Data Management System", *VLDB* 2008.

[12] Michael J. Franklin, et al, "Continuous Analytics: Rethinking Query Processing in a Network-Effect World", CIDR 2009.

[13] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, Myung Cheol Doo, "*SPADE*: The *System S* Declarative Stream Processing Engine", *ACM SIGMOD 2008*.

[14] HP Neoview enterprise data warehouse, http://h71028.www7.hp.com/nterprise/w1/en/software/business-intelligence-neoview.html

[15] N. Jain et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. SIGMOD 2006.

[16] Erietta Liarou et.al. "Exploiting the Power of Relational Databases for Efficient Stream Processing", EDBT 2009.

[17] SageLogix, Inc, "Scale to infinity: partitioning in Oracle Data Warehouses", http://www.sagelogix.com/idc/groups/public/documents/sagelogix-whitepaper/sage016100.