Stream-join Revisited in the Context of Epoch-based SQL Continuous Query

Qiming Chen
HP Labs
Palo Alto, CA, USA
qiming.chen@hp.com

Meichun Hsu
HP Labs
Palo Alto, CA, USA
meichun.hsu@hp.com

ABSTRACT

The current generation of stream processing systems is in general built separately from the query engine thus lacks the expressive power of SQL and causes significant overhead in data access and movement. This situation has motivated us to leverage the query engine for stream processing.

Stream-join is a window operation where the key issue is how to punctuate and pair two or more correlated streams. In this work we tackle this issue in the specific context of query engine supported stream processing. We focus on the following problems: a SQL query is definable on bounded relation data but stream data are unbounded, and join multiple streams is a stateful (thus history-sensitive) operation but a SQL query only cares about the current state; further, relation join typically requires relation re-scan in a nested-loop but by nature a stream cannot be re-captured as reading a stream always gets newly incoming data.

To leverage query processing for analyzing unbounded stream, we defined the *Epoch-based Continuous Query* (ECQ) model which allows a SQL query to be executed epoch by epoch for processing the stream data chunk by chunk. However, unlike multiple one-time queries, an ECQ is a single, continuous query instance across execution epochs for keeping the continuity of the application state as required by the history-sensitive operations such as sliding-window join.

To joining multiple streams, we further developed the techniques to cache one or more consecutive data chunks falling in a sliding window across query execution epochs in the ECQ instance, to allow them to be re-delivered from the cache. In this way join multiple streams and self-join a single stream in the data chunk based window or sliding window, with various pairing schemes, are made possible.

We extended the PostgreSQL engine to support the proposed approach. Our experience has demonstrated its value.

IDEAS12 2012, August 8-10 Prague [Czech Republic]
Editors: Bipin C. Desai, Jaroslav Pokorny, Jorge Bernardino

Categories and Subject Descriptors

H2.4 [**Query Processing**], D.1.3 [**Programming Techniques**]

General Terms

Management, Performance, Design, Experimentation.

Keywords

Continuous query, Stream join.

1. INTRODUCTION

Due to the massively growing data volumes and the pressing need for low latency, continuous data analysis applications are pushing the traditional data warehousing technologies beyond their limits [10]. Data Stream Management Systems (DSMSs) provide a paradigm shift from the load-first analyze-later mode of data warehousing; they can run many orders of magnitude more efficiently than disk based data processing systems.

However, the current generation of DSMS is in general built separately from the query engine thus lacks the functionalities of SQL and DBMS; such platform separation also causes significant overhead in data access and movement [5-7]. This situation has motivated us to develop techniques for analyzing stream data by query engines for leveraging the expressive power of SQL, the streaming functionality of query processing, and in general, the database innovations developed in decades. In this paper we revisit the stream join issue in the above context.

1.1 The Problem

In this work we focus on how to support stream-join in the specific context of query engine supported stream processing. Stream-join is a window operation where the key issue is how to punctuate and pair two or more correlated streams; to deal with it with the query engine, there exist several challenges.

First, a SQL query is definable only on bounded and finite data but stream data are unbounded and infinite. Nest, join two relations is a stateful, thus history-sensitive, operation that typically requires one relation to be re-scanned in a nested loop, but by nature a dataflow is captured on-the-fly and cannot be re-captured. While re-scanning a static table always gets the same data, reading a dynamic stream always gets new data.

1.2 The Prior 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[11], STREAM[1], Aurora, Borealis, etc [2]. 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. Oracle currently offers a "continued query" feature but it is based on automatic view updates therefore not really supporting continuous querying. Two recently reported systems, the TruSQL engine [10] and the DataCell engine [12] do leverage database technology but are characterized by providing a workflow [8] like service for launching a one-time SQL query to buffered stream data set iteratively in a non-dataflow fashion.

Join two or more input streams are supported by the DSMSs built from scratch, e.g. System-S [11]. However, how to join streams in a SQL query remains an open issue.

In contradistinction to the above, we study the stream join issue in the context of query engine supported stream analytics. Since stream-join is a history-sensitive window operation, using multiple one-time queries is inappropriate since the state cached in the query execution closure will be lost between query shutdown/restart. Thus our goal is to provide a truly continuous, long-standing query instance that deals with per-tuple processing, maintains application state continuously, and supports window semantics as well. In building this novel platform we formalized and extended several technologies we developed at HP Labs in dataflow language [7] and stream processing [5]; we also extended our solution for processing a single stream by a SQL query to the join of multiple streams.

1.3 The Solution

For integrating stream processing with query processing, we defined a unified query model over both stored relations and dynamic streams, and extended the query engine to support the unified model. In order to query unbounded stream data, we introduce the Epoch based Continuous Query (ECQ) to allow a SQL query to be executed epoch by epoch, with an execution epoch applying to a bounded chunk of stream data, but without shutting the query instance down between chunks in order to maintain the application state continuously across execution epochs. The above combination is essential for state-dependent operations including sliding-window based stream join. Apply an ECQ to data streams turns the query execution on the unbounded data into a sequence of query executions on the bounded data chunks.

The ECQ query model provides the foundation for the query engine supported stream join - a fundamental operation for relating information from different streams, and a hard problem in SQL query based stream processing. We developed the techniques to buffer one or more consecutive stream data chunks falling in a sliding window across query execution epochs, to allow the buffered stream data chunks to be re-delivered from the cache as required by the join-operation. In this way join multiple streams and self-join a single stream in the data chunk based window or sliding window, with various pairing schemes, are made possible.

With the conventional query engines, the above capabilities are not provided and therefore we extend the query engine to offer them. One extension is focused on supporting truly continuous but epoch based query execution by the *cut and rewind* query execution mechanism, namely, cutting a query execution based on some chunking criterion (e.g. time window boundary), and then rewinding the state of the query without shutting it down, for processing the next chunk of stream data. The other extension lies in extending the User Defined Function (UDF) framework for capturing, chunking and buffering stream input data across query execution epochs. More specifically, supported by the extended query engine, one or more input streams are read by table UDFs through function-scan. The most recent data chunks of each input stream can be buffered in the function closure. The function scan is punctuated in each query execution epoch, and its behavior is switchable in the first scan (delivering new data) and the subsequent re-scans (delivering buffered data).

We have integrated the above stream processing functionalities into the PostgreSQL engine. Our experience shows the novelty of the proposed approach in handling multiple streams by an ECQ, and reveals its scalability and efficiency for stream-join.

The rest of this paper is organized as follows: Section 2 overviews our approaches in stream capturing and analysis based on SQL and UDF, as well as in supporting epoch-based query execution for chunk-wise stream processing; Section 3 deals with stream join using ECQs; Section 4 discusses experimental results; Section 5 concludes the paper.

2. STREAM ANALYTICS BY EPOCH BASED CONTINUOUS QUERY

We first describe the notion of ECQ which underlies query based stream-join. We view the pipelined query processing essentially as stream processing, and envisage the potential advantage of leveraging query engine for continuous stream analytics. We also realize the fundamental difference between the two as that a query is defined on bounded relations but stream data are unbounded. Based on the above 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 with query processing.

2.1 Model and Semantics

The difficulty of using regular SQL queries for stream processing is that a SQL query is not definable on unbounded stream data since in that case the query cannot return complete result, and if the query involves aggregation, it never returns any result. Our solution is to punctuate the input stream data into a sequence of chunks with each chunk representing a *bounded* data set on that a query is definable. This strategy fits stream analytics in general where applications often require the infinite data to be analyzed granularly.

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 bounded relation. The semantics of applying the query Q to the unbounded stream S plus 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 a sequence of query results, one on each *chunk* of the stream data.

To implement this model on a query engine, it is necessary for a query to capture stream elements on-the-fly, to punctuate input stream into chunks, to run the query epoch by epoch for processing the stream chunk by chunk. In doing so it is important to keep the continuity of the query instance for retaining the buffered data as required by the history-sensitive applications such as sliding window applications. It is easy to see that running an one-time query iteratively cannot meet the above requirement as the query state would be lost between shutting down and restarting the query. Below we describe our solutions.

2.2 Stream Capture Function

We start with capturing events from streams and turning them into 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 Capture Function (SCF) that captures the incoming data on-the-fly and returns a sequence of tuples to feed queries without first storing them on the disk. In the other words, we replace table scan by ***function scan***. A SCF can listen or read data/events sequence and generate stream elements tuple by tuple continuously. A SCF is called multiple, up to infinite, times during the execution of a continuous query, each call returns one tuple to fuel the query.

Note that fueling the query tuple by tuple immediately upon receipt of incoming event is not the standard function-scan mechanism found in existing DBMSs. In fact, the conventional function-scan first provides all the output tuples and then delivers them one by one which causes significant latency in processing data streams, and is semantically inconsistent with the infinity of stream data. To solve the above problem, we provided a special kind of function-scan for capturing and emitting data stream on the per-tuple basis.

We rely on SCF 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 SCF scan is supported at two levels, the SCF level and the query executor level. A data structure containing function call information, bridges these two levels, which is initiated by the query executor and passed in/out the SCF for exchanging function invocation related information. We use this mechanism for minimizing the code change, but maximize the extensibility, of the query engine.

2.3 Epoch based Continuous Query

To allow a query to apply to unbounded stream data chunk by chunk while keeping the query instance alive without shutdown/restart, our solution is to *cut* the input stream data into a sequence of chunks with each chunk representing a bounded data set on that a query is definable, and after processing a chunk of data, to *rewind* the query instance for processing the next chunk of data. More exactly, when the *end-of-epoch* event or condition is signaled from the SCF, the query engine completes the current query execution epoch in the regular way, but rewinds the query instance for the next execution epoch. As such the query is running epoch by epoch, and therefore we refer to it as Epoch-based Continuous Query (ECQ).

As a simplified example, consider a stream of network traffic packets with the following schema

[*pid, t_s, from-ip, to-ip, bytes*, …]

where *pid* stands for the ID of the packet, t_s for the source timestamp (according to the reliable TCP protocol, the stream of TCP/IP packets transmitted from a source to a destination should arrive in the order of their source timestamps).

The goal of querying is to capture IP-to-IP network traffic, convert them to host-to-host traffic, and measure the traffic volume between each pair of hosts. The mapping from IP to host is given in the table *hosts*.

We first show a regular one-time query defined on a bounded snapshot of the traffic flow stored in the table "*packet_table"*. Since this table is bounded, the query result involving aggregation (SUM) is well defined.

[One-Time Query: Q_A]

```
SELECT
   h1.host-id AS from-host, h2.host-id AS to-host, SUM(T.bytes)
FROM packet_table T, Hosts h1, Hosts h2
WHERE h1.ip = T.from-ip AND h2.ip = T.to-ip
GROUP BY from-host, to-host;
```

Then we look at the following query, Q_B, applied to the un-bounded stream data generated by a SCF; it receives packet stream from a socket, generates and delivers packet tuples to fuel the stream query. The stream is unbounded but we punctuate it to bounded per-minute chunks for processing. The goal of the query is to derive the host-to-host traffic volumes minute by minute.

[Epoch-based Continuous Query: Q_B]

```
SELECT
   floor(S.ts /60) AS minute, h1.host-id AS from-host, h2.host-id AS
   to-host, SUM(S.bytes)
FROM STREAM_get_packets(packet_stream, 'CUT ON ts BY
   60 SECS')) S, Hosts h1, Hosts h2
WHERE h1.ip = S.from-ip AND h2.ip = S.to-ip
GROUP BY minute, from-host, to-host;
```

In the above query, we replace the disk-resided database table by a SCF,

STREAM_get_packets (packet_stream, 'CUT ON t_s BY 60 SECS')

where *packet_stream* is the stream source and *'CUT ON t_s BY 60 SECS'* expresses the chunking criterion specifying that the stream source is to be "cut" into a sequence of bounded *chunks* every 60 seconds (1 minute) The execution of Q_B on an infinite stream is made in a sequence of *epochs*, one on each data chunk. In this way it returns a sequence of chunk-wise query results. The graphical representation of the two queries is shown in Fig. 1.

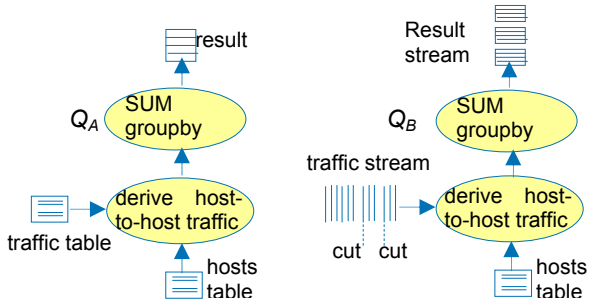

Figure 1. *Left*: apply query to static (bounded) tables; *Right*: execute continuous query epoch by epoch for processing stream chunk by chunk (each chunk is bounded)

2.4 Extend Query Engine for Supporting Epoch-based Query Execution

To support Epoch based query execution for chunk-wise data processing, we developed the *cut-and-rewind* query execution mechanism, namely, cut a query execution based on the epoch specification and then rewind the state of the query without shutting it down, for processing the next chunk of stream data in the next epoch.

Cut is originated in the SCF at the bottom of the query tree. SCFs have a general form of *STREAM(SS, epoch-spec)* specifies that the stream source *SS* is to be "cut" into an unbounded sequence of *chunks*. The "cut point" is specified in the *epoch-spec*. Upon detection of end-of-epoch condition, the SCF signals *end-of-epoch* punctuation to the query engine resulting in the termination of the current query execution epoch. In general the end of an epoch is determined when the first stream element belonging to the next epoch is received; that element will be cached to be processed first in the next epoch. This buffer can also be used to re-order the input elements to be processed for easing the across-chunks mis-order on which we do not discuss in detail here.

Upon termination of the execution of an epoch, 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. Since the query instance remains alive across epochs, data for sliding-window oriented, history sensitive operations can be kept continuously. Bringing these two capabilities together is the key in our approach.

We support multiple common chunking criteria for punctuating a stream:

- chunking by cardinality, i.e. the number of inputs;
- chunking by input range, e.g. by time-window; and
- chunking by "object" based on the *chunk-key* attribute (e.g. the *id* of a graph appearing in multiple consecutive stream elements).

We also support epoch-based transaction model coupled with the *cut-and-rewind* query model under which a stream query is "committed" one epoch at a time in a sequence of "mini-transactions", which makes the per-epoch stream processing results visible as soon as the epoch ends. We discussed this issue in [5] and skip the details here, as the focus of this paper is placed on query engine enabled stream join.

3. JOIN MULTIPLE INPUT STREAMS

Stream join is a fundamental operation for relating information from different streams [13]. Extending the above example, given two streams of packets seen by network monitors placed at two routers, we can join the streams on *packet id*s to identify those packets that flowed through both routers, and compute the time delta for them to reach these routers.

Many stream processing operators are defined on the buffered stream data, known as synopses. In the SQL engine based stream-join, we use UDFs to provide the data buffer in the function closures, for caching the window state in stream processing. Briefly speaking, a UDF is called multiple times following the typical FIRST_CALL, NORMAL_CALL and FINAL_CALL skeleton, where the data buffer is initiated in the FIRST_CALL and used in each NORMAL_CALL. However, the multi-call skeletons of the scalar UDF and the table UDF have different scopes.

- For the scalar UDF, each call is for processing a tuple which returns a single value; thus multiple calls correspond to the processing of multiple tuples. Therefore, a data processing state initiated in the FIRST_CALL can be carried on and updated in each NORMAL_CALL, i.e. used in the processing of multiple tuples.
- For the table UDF that returns a set of values out of each input tuple, each call corresponds to a return, and the multi-call skeleton corresponds to the processing of a single input tuple. In this way the data processing state initiated in the FIRST_CALL can only be used in processing a single input tuple.

Since the window UDF for stream join is a table function, and it must maintain the data processing state across multiple input tuples, we extend the query engine to allow such *multi-call-process* of a table function to span multiple input tuples in the way similar to a scalar UDF. Such a window UDF incrementally buffers the stream data, and manipulates the buffered data chunk for the required window operation. Although the proposed ECQ runs epoch by epoch for processing stream data chunk by chunk, the query instance remains alive, thus the UDF buffer is retained between epochs 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 epoch 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 stream processing systems which launch one-time query iteratively [10,12].

Below we discuss the typical cases of ECQ based stream-joins.

3.1 Join Static Relations

A SQL *join* between two non-streaming relations R and R' returns the set of all pairs $<r, r'>$, where r ∈ R, r' ∈ R', and the join condition $\theta(r, r')$ evaluates to *true*. For example, in the query Q_A shown above, the tables *packet_table* and *hosts* are joined for getting the host-to-host traffic volume where the hosts table is also self-joined.

Three fundamental algorithms exist for performing a join operation: Nested loop join, Sort-merge join and Hash join. Choosing the most efficient algorithm is the duty of the query optimizer based on the sizes of the input tables, the number of rows from each table that match the join condition, and the operations required by the rest of the query; such a choice is generally transparent to the SQL query developer.

A nested loop join is the basic join algorithm, that requires re-scan the table R' above for each tuple of R. In the above example query, Q_A, the *hosts* table is nested at two levels therefore if the *packet_table* has n tuples, then the *hosts* table is scanned n^2 times. With any join algorithm, a participating table may be materialized in memory for either semantic reason (e.g. sorting) or performance reason (avoid re-scan disk); but the bottom line is that re-scan a relation returns the same result as the initial scan.

For the function scan used in capturing streams for join, the nested loop join is most likely chosen by the system query planner.

3.2 Join a Stream Window and a Static Table

Join the stream elements falling in a time window, say S_t, and a relation R returns the set of all pairs $<s, r>$, where $s \in S_t$, $r \in R$, and the join condition $\theta(s, r)$ evaluates to *true*.

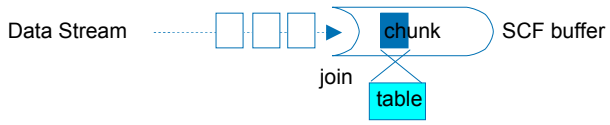

Figure 2. Join a table with a buffered stream data chunk that can be re-scanned for returning the same data as the original scan

As illustrated by Fig. 2, for example, in the above query Q_B, the input stream data generated by the SCF *STREAM*_get_packets*()* are joined with table *hosts* in the per-minute chunk to derive the host-to-host traffic volume on the minute basis. Since in each epoch the chunk of stream data is bounded, Q_B can generated query results epoch by epoch; otherwise with unbounded input stream the query semantics is un-definable. Regarding to the join of a stream data chunk returned from *STREAM_ get_packets()* with the table, *hosts*, it is the operation taking place in each query execution epoch with three nested loops.

for each tuple r in the chunk of the stream do
  for each tuple h_1 in Hosts do
    for each tuple h_2 in Hosts do
      if r and h_1, h_2 satisfy the join condition
        then output the tuple <r, h_1, h_2>

Besides of the performance concern (whether re-scan is made by cache access or by disk access), the critical issue is whether re-scan a data source can get the same data access result. It can be seen that re-scan a table, such as the *hosts* above, results in the same data content as the original scan. However, in case the SCF, *STREAM_get_packets*(), is re-executed on the "re-scan" demand in an query execution epoch, it should not read in new stream data but deliver the original chunk of data received in the current query execution epoch. Reading in new stream data during re-scan is inconsistent with the semantics of re-scan and generates incorrect join results. This problem is potential but may not really show up in joining a stream with a static table, because the cardinality of a stream data source, represented by a SCF, is unknown, the query optimizer tends to choose to re-scan the table with known cardinality. However, if both data sources are streams, we must extend the query engine to deal with this issue, which is described in the next section.

3.3 Window-Join Multiple Streams

Stream join is a fundamental operation for relating information from different streams. Like relation join, stream-join is a stateful operation. Un-windowed stream joins are impractical and lead to memory outgrowing. In the time-based dataflow context, joining two streams chunk-wise is a block operation – it takes place only after two chunks of stream data are received. In general, given two streams S and S' punctuated by the same chunking criterion, in each query execution epoch, join the most recent chunks of them, S_t and S'_t, returns the set of all pairs $<s, s'>$, where $s \in S_t$, $s' \in S'_t$, and the join condition $\theta(s, s')$ evaluates to *true*. This is a special case shown in Fig 3 where only the most recent chunks of each stream are joined.

As mentioned above, for function scan (the access method of stream query), nested loop join is the default system choice that potentially involves "re-scan a stream source". With non-stream data, re-scan always gets the same set of input data. With SCF, the key issue is to ensure that "scan" a stream source initially receives the newly incoming stream data, but "re-scan" returns the same data as the above initial scan, and these two behaviors are automatically switchable during stream-join.

Let us extend the above example by considering two streams, S_1 and S_2, of packets seen by network monitors placed at two routers, RT_1 and RT_2, we can use ECQ to join the streams *in the minute based chunks*, on *packet id*s to identify those packets that flowed through both routers, and compute the average time for such packets to reach RT_2 from RT_1. The two streams have the same schema

[*pid, t_s, from-ip, to-ip, bytes, t_{rt}…*]

where t_{rt} is the timestamp captured at the router, and the stream punctuation point is determined by t_s, the source timestamp (based on the reliable TCP protocol, the stream of TCP/IP packets transmitted from a source to a destination should arrive in the order of their source timestamps).

[Epoch-based Continuous Query for Stream-Join : *Qc*]

SELECT floor($S_1.t_s$ /60) AS minute, AVG($S_2.t_{rt}$ –$S_1.t_{rt}$)
FROM *STREAM_get_packets* (RT_1, *'CUT ON t_s BY 60 SECS', 'BLOCK'*)) S_1, *STREAM_get_packets* (RT_2, *'CUT ON t_s BY 60 SECS', 'BLOCK'*)) S_2,
WHERE S1.pid = *S2*.pid GROUP BY minute;

We enforce the re-scan semantics by extending the buffer hierarchy and invocation pattern of the table functions serving as SCFs, to allow the data chunk read in each query execution epoch to be *buffered* and *re-delivered* in the subsequent "re-scans" which is indicated by the SCF's parameter 'BLOCK'.

In each query execution epoch, the SCF as a table function is called multiple times for returning multiple tuples generated from the received events; the associated buffer state is initiated at the beginning of the first call and finalized (e.g. cleanup) at the end of the last call; these calls make up the ***multi-call-process*** of a ***function scan*** in the query execution epoch.

- The initial function-scan, the SCF returns multiple tuples generated from the received stream elements to fuel the query; it takes place only once per query-epoch.
- If re-scan is necessary, in the subsequent function-scans, the SCF returns the buffered tuples.

To support the above mechanism, two extensions to the query engine are required: first, buffering SCF's input data across multiple function-scans in an epoch as well as across multiple epochs; and second, switching the SCF's behavior for delivering the initially captured data in the first scan and delivering the buffered data in all the subsequent re-scans.

In summary, for handling one or multiple streams, the SCF is provided with the following capabilities.

- *Chunking* incoming data by cardinality or other punctuation criterion;
- Generating and *streaming out* tuples one by one continuously to fuel the ECQ;
- *Signaling* the query engine with "end-of-epoch" message to make up a query execution epoch.
- *Caching data chunk for re-scan* in or across query execution epochs (if there is a stream join).
- *Supporting initial function-scan and re-scans* where the initial scan takes place only once in a query execution epoch, and the re-scans simply returns the buffered data.

To support re-scan and to distinguish the initial scan and the re-scan, a Boolean variable is provided with state retained across multiple function scan and re-scans in a query execution epoch. This variable is set during the initial scan, indicating "the initial scan has been done". For each function scan, if this variable is not set, the initial scan process is invoked for getting new data from the stream source; otherwise the re-scan process is invoked for delivering the already buffered data.

While our approach is applicable to the nested-loop join, it is naturally applicable to other join types as well.

3.4 Join Streams in Sliding-Windows

In general, by adjusting the buffer boundary, join two streams on the *chunk-based* sliding windows with various pairing schemes is possible (note that a chunk-based sliding window shifts chunk by chunk, a chunk can be as small as a single tuple). Consider two streams S and S' captured by *scf* and *scf'* and commonly chunked by timestamp; *scf* keeps a sliding window buffer for M data chunks of S, and *scf'* holds N data chunks of S'. In each query execution epoch, join $m \le M$ chunks, S_m, held in *scf* and $n \le N$ chunks, S'_n, in *scf'* returns the set of all pairs $<s, s'>$, where $s \in S_m$, $s' \in S'_n$, and the join condition $\theta(s, s')$ evaluates to *true*. In case $M = 1$ or $N = 1$, the joins are not overlapped between query execution epochs. The stream join takes place every query execution epoch. This is illustrated in Fig. 3; the ECQ, *Qc*, described in the last subsection, is a special (but most frequently used) case where $M = N = 1$.

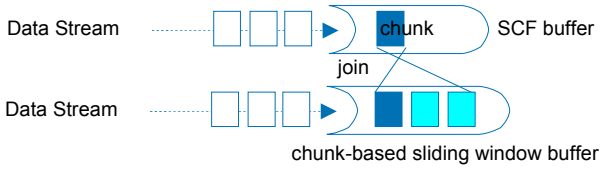

Figure 3. Join two streams by pairing their chunks buffered in chunk-based sliding windows; these buffered data chunks can be re-scanned

3.5 Self-join a Stream in a Sliding Window

Self-join different chunks, typically the most recent chunk and certain past chunks in a sliding window, of the same stream for correlation purpose, is useful in many applications. This problem can be described as follows. Given a stream S and a sliding window S_w containing the consecutive chunks of S, $<S_0, S_1, \ldots S_k>$ where S_0 is the most recent chunk. Join S_0 with S_w returns the set of all pairs $<s_0, s_w>$, where $s_0 \in S_0$, $s_w \in S_w$, and the join condition $\theta(s_0, s_w)$ evaluates to *true*. This is conceptually shown in Fig. 4.

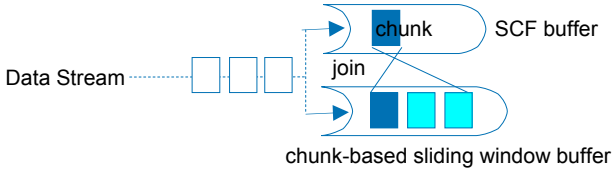

Figure 4. Correlating (join) the most recent chunk with the previous chunks of the same stream kept in a sliding window, using two SCFs which buffer that stream differently

For example, in telecommunication monitoring applications, a CDR (Call Detail Record) stream often contains the duplicate CDRs representing the same phone call but generated by different probes with slightly different start-call timestamps t_s (e.g. 0.2 sec delta). Identifying such duplicate CDRs can be treated as join the CDR captured in a 5 seconds chunk with those captured in the past 1 minute (12 chunks), as expressed by the following ECQ. To fuel this query, the CDR stream is ***split*** and fed in two SCF instances, say scf_1 and scf_2, where both chunk the input stream in every 5 seconds by timestamp; scf_1 only keeps the current chunk of data, but scf_2 continuously maintains a sliding window containing 12 chunks (1 minute) of data. The sliding window shifts chunk by chunk. In each 5 second epoch the current chunk of data captured by scf_1 are joined with the 12 chunks of data kept in scf_2. Since each new data chunk appears in scf_1 only once, join it with the data chunks held in scf_2 is not duplicated. There exist other steps to eliminate the duplicates, which is not discussed here.

```
SELECT DISTINCT *
FROM STREAM_get_cdrs (CDRS, 'CUT ON ts BY 5 SECS',
    'BLOCK',  1)) scf1, STREAM_get_cdrs (CDRS, 'CUT ON ts BY 5
```

*SECS', 'BLOCK', **12**)) scf_2,*
WHERE scf_1.caller# = scf_2.caller# AND scf_1.calee# = scf_2.calee#
AND ((scf_1.t_s > scf_2.t_s AND scf_1.t_s – scf_2.t_s < 0.2) OR (scf_2. t_s > scf_1.t_s AND scf_2.t_s – scf_1.t_s < 0.2))

This technique is very useful for correlating the most recent stream data with the previous ones in a sliding window boundary.

3.6 Extend Query Engine for SCF Re-Scan

As described earlier, for handling ECQ based stream processing, we have extended the query engine to support the cut-and-rewind approach. For enabling ECQ based stream join, we take one step further to support generalized UDF buffer management. Below we outline it briefly, more details are reported separately.

Like a regular program, a UDF can maintain a data buffer; unlike a regular program, a UDF is called multiple times in a single host query, with respect to each input and output; therefore, managing the *life-span* of the data buffer with respect to these invocations has become an important system issue.

The current DBMSs support scalar, aggregate and table functions. The input of any of these functions is bound to the attribute values of a single tuple, where an aggregate function is actually implemented with incremental per-tuple calculation. A scalar or aggregate function is called multiple times, one for each input with a single return value (or a tuple as a composite value); a table function, however, can return a set out of a single input; accordingly, on each input, called multiple times, one for each output. In the other words, the *multi-call-process* of a scalar/aggregate function spans over all the input tuples, but the *multi-call-process* of a table function limits to one input only, while across multiple returns out of that single input.

A SCF is a table function with parameters bearing the chunking condition, stream source, etc. Please note that the captured stream data are not the input tuples of the SCF. A function scan by a SCF is a *multi-call-process*. In each query epoch, there is an initial function scan, and possibly multiple subsequent re-scans.

With the current technology, the "top-level" data buffer of a table UDF is related to the *multi-call-process* of it, thus to a single function scan only. To support stream join, it is necessary to extend table function's data buffering across multiple function scans in an epoch for a chunk of data, and, across multiple chunks, in addition to the existing per-input/multi-retruns data buffering in a single function scan.

Therefore, as the technical backbone, we have extended the query engine and the Table UDF framework to allow a SCF to retain a data buffer across multiple function-scans wrt the processing of one or more chunks of data.

A query is parsed, optimized and planned, forming a query plan tree. To be executed, an instance of the query plan is initiated with nodes representing operators and their states; a UDF is represented by a function node that is the root of its function closure where the information about the function and function-invocation, as well as the data buffers, are provided. Please note that the function node is external to the function (e.g. UDF), therefore, the data linked to the function node may be sustained across multiple function calls. We extended table UDF buffer management by the following.

- Extend a data structure under the function node with an additional pointer, and allocate a new buffer under it. The life-span of this buffer sustains across multiple function-scans.
- Support multi-layers of buffers, or memory context – across multiple chunks, per-chunk, per-input tuple (i.e. per-scan with multi-returns), and per-return (Fig. 5).
- Control the scope and time of memory de-allocation properly in terms of system internal utilities, e.g. after a returned tuple, a function-scan, or the whole query has been processed.

Accordingly, new APIs for creating data buffer with per-query, per-chunk and per-scan initial states are distinguished. In general, the buffers of a UDF at all levels are linked to the system handle for function invocation, and accessible through system APIs.

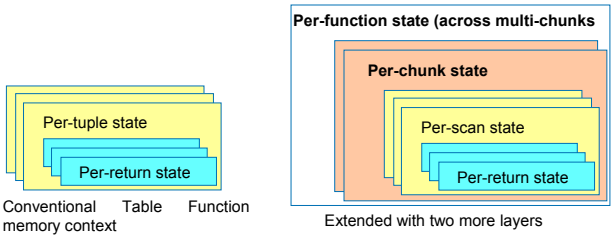

Figure 5. Extend the memory context of table function to support multi-layer data buffering

4 EXPERIMENTS

In measuring the performance of single stream processing, for fair comparison we used the widely-accepted Linear-Road (LR) benchmark, which requires computing the per-minute traffic statistics from the GPS reports of multiple cars, and making that information available to each car within 5 seconds. Tested on our extended query engine, the experimental results measured on a single node, HP xw8600 with 2x Intel Xeon E54102 2.33 Ghz CPUs and 4 GB RAM, running Window XP (x86_32), running PostgreSQL 9.0.3, show that the above response time can be controlled within 0.2 second; but the other published results generally range over 1 to 5 seconds; the details were given in [6].

As the focus of this work is placed on stream join, we compare the performances of

- join streams in a single long-standing ECQ;
- join the results of iteratively executed one-time query as temporary tables (in memory) holding chunks of stream data.

In measuring the stream join performance in a continuous query, we focus on the post cut elapsed time of an ECQ, namely the remaining time of query evaluation after the input data is punctuated, which reflects the delta time for the results to be accessible after receiving the last tuple of the data chunk in the epoch, and therefore reflects the stream-join time. Our experiment results show that the former outperforms the later significantly (Fig. 6). The performance difference can be explained as follows: first, using ECQ eliminates the frequent query startup and shutdown; next, using ECQ avoids the IPC cost for accessing the data buffered outsides of the query; further, using ECQ retains the pipeline feature of query processing while using temporary tables makes the consecutive query executions as strict "block" operations.

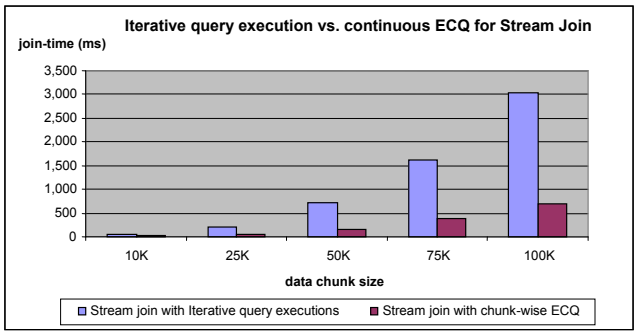

Figure 6. Comparison between (a) chunk-wise join using a single continuous ECQ; (b) joining iteratively executed query results as temporary tables holding chunks of stream data.

Additionally, we are interested in the "cost" of stream-join in continuous stream processing with ECQ. Particularly, stream-join is a block operator and we want to know how much cost it is higher than a pipelined operator, as a guidance of the potential optimization. For this purpose we compare the performances of single stream aggregation (***Qd***) and stream join (***Qc***).

[Epoch-based Continuous Query for Stream-Join : *Qc*]

SELECT floor($S_1.t_s$ /60) AS minute, AVG($S_2.t_{rt}$ –$S_1.t_{rt}$)
FROM *STREAM_get_packets* (RT_1, *'CUT ON t_s BY 60 SECS', 'BLOCK'*)) S_1, *STREAM_get_packets* (RT_2, *'CUT ON t_s BY 60 SECS', 'BLOCK'*)) S_2,
WHERE S1.pid = *S2*.pid GROUP BY minute;

[Single stream ECQ for chunk-wise aggregation: *Qd*]

SELECT floor(t_s /60) AS minute, AVG(*S*.bytes)
FROM *STREAM_get_packets*(packet_stream, *'CUT ON t_s BY 60 SECS'*)) S,
GROUP BY minute;

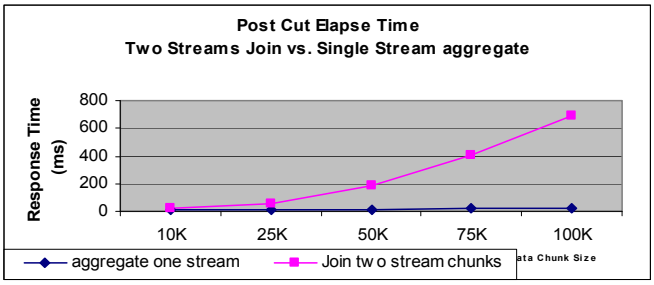

Figure 7. Comparison between the chunk-wise join of two streams (blocked) and the chunk-wise aggregation of a single stream (pipelined)

As illustrated in Fig. 7, for a single stream aggregation (***Qd***), the per-epoch (e.g. 1 minute) query evaluation can be made incrementally with the per-tuple processing pipeline, and therefore the post-cut query time is well controlled, particularly when the stream elements for a query epoch arrive in real-time (e.g. in 1 minute). However, joining two streams (***Qc***) is a block operator that can proceed but cannot generate result until the end of the epoch; which leaves much of the work to the post-cut time period. The comparison of single stream aggregation (***Qd***) and stream join (***Qc***) is shown in Fig 7 where the test is under the *stress mode*, namely, the data are read by the SCFs from files continuously without following the real-time intervals; the chunk size ranges from 10K to 100K; the stream-join ranges from 10 ms to 600 ms.

5 CONCLUSIONS

In response to the massively growing data volumes and the pressing demands for low latency, Data Stream Management Systems (DSMSs) have provided a paradigm shift from data warehousing with orders of magnitude efficiency gain. However, since the current generation of DSMSs is in general built separately from the query engine, it lacks the functionalities of SQL and DBMS and incurs significant overhead in data access and movement. Motivated by bridging the above gap we have integrated stream analytics with query processing. In this context we conducted the research on stream join and reported our findings in this paper. We tackled the following challenging issues: how to handle window based stream join with unbounded stream data using a SQL query that is definable only on bounded data; how join two or more streams for correlating them, and how to handle re-scan for joining streams captured on-the-fly.

To leverage query processing for analyzing unbounded stream with join operations, we defined the Epoch based Continuous Query (ECQ) model for turning the query execution on an unbounded stream into a sequence of query execution epochs on the sequence of bounded data chunks. While allowing a SQL query to be executed epoch by epoch on the stream data chunk by chunk, we have the query instance kept alive across execution epochs to maintain the continuity of the application state as required by the state-dependent join operations.

In exploring the join of multiple streams based on the combination of the epoch-based query execution and the sustainable query instance, we further extended the query engine to allow one or more consecutive data chunks falling in a sliding window to be buffered across query execution epochs for re-scan, and to allow the function scan and re-scan behavior to be switchable. In this way join multiple streams and self-join a single stream in the data chunk based window or sliding window, with various pairing schemes, are made possible.

We integrated the above stream processing functionalities into the PostgreSQL engine. Our experience reveals that the proposed approach is highly scalable and efficient; it supports a unified query processing over both stored relations and dynamic streaming data, and provides a paradigm shift from the store-first, analyze-later mode of data warehousing.

6 REFERENCES

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

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

[3] R.E. Bryant. "Data-Intensive Supercomputing: The case for DISC", *CMU-CS-07-128*, 2007.

[4] 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.

[5] Qiming Chen, Meichun Hsu, Hans Zeller, "Experience in Continuous analytics as a Service", EDBT 2011.

[6] Qiming Chen, Meichun Hsu, "Experience in Extending Query Engine for Continuous Analytics", Proc. 11th Int. Conf. DaWaK, 2010.

[7] Qiming Chen, Meichun Hsu, "Query Engine Net for Streaming Analytics", Proc. 19th International Conference on Cooperative Information Systems (CoopIS), 2011.

[8] Qiming Chen, Meichun Hsu, "Inter-Enterprise Collaborative Business Process Management", Proc. of 17th Int'l Conf on Data Engineering (ICDE), 2001.

[9] 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.

[10] Michael J. Franklin, et al, "Continuous Analytics: Rethinking Query Processing in a NetworkEffect World", CIDR 2009.

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

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

[13] Junyi Xie , Jun Yang, "a survey of join processing in data streams", In Chapter 10, "***Data Stream***: Models and Algorithms", (Springer).