Continuous Analytics Over Discontinuous Streams

Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, Neil Thombre

Truviso, Inc.
1065 E.Hillsdale Blvd, Suite #230, Foster City, CA 94404

ABSTRACT

Continuous analytics systems that enable query processing over steams of data have emerged as key solutions for dealing with massive data volumes and demands for low latency. These systems have been heavily influenced by an assumption that data streams can be viewed as sequences of data that arrived more or less in order. The reality, however, is that streams are not often so well behaved and disruptions of various sorts are endemic. We argue, therefore, that stream processing needs a fundamental rethink and advocate a unified approach toward continuous analytics over discontinuous streaming data. Our approach is based on a simple insight – using techniques inspired by data parallel query processing, queries can be performed over independent *sub-streams* with arbitrary time ranges in parallel, generating partial results. The consolidation of the partial results over each sub-stream can then be deferred to the time at which the results are actually used on an on-demand basis. In this paper, we describe how the Truviso Continuous Analytics system implements this type of *order-independent* processing. Not only does the approach provide the first real solution to the problem of processing streaming data that arrives arbitrarily late, it also serves as a critical building block for solutions to a host of hard problems such as parallelism, recovery, transactional consistency, high availability, failover, and replication.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *query processing, parallel databases, transaction processing.*

General Terms

Management, Performance, Design, Reliability.

Keywords

Streams, Order, Out-of-Order, Continuous Queries.

1. INTRODUCTION

Research and early commercial work in stream query processing viewed data streams as *continuous* sequences of data that arrived more or less in order. This assumption heavily influenced the architecture of first generation stream-processing systems. Such systems typically tolerate only small degrees of delay and out-of-orderness in the arrival of streaming data, and do so at the expense of introducing mandatory latencies and potentially expensive buffering into what should be high-speed, lightweight stream processing. In cases where the discontinuity of the input exceeds the limits that can be handled by such systems, the incoming data is often simply dropped, or at best, spooled to back-up storage for eventual (and often manual) integration into the system.

In practice, however, it turns out that streams are rarely continuous for several reasons. For example, in distributed environments, data from multiple sources can arrive in arbitrary order even under normal circumstances. Failure or disconnection followed by subsequent recovery of connectivity between remote sites causes even larger discontinuities as sites that have been down or disconnected for large periods of time finally wake up and start transmitting old, but important data. A similar pattern of events unfolds in the event of temporary disruptions such as network partitions between datacenters connected over WAN environments. Parallel execution techniques that are critical in scaling up and out in multi-core and cluster systems break the sequential/in-order nature of stream processing. Finally, high availability mechanisms for streaming systems, in which a recovering server must obtain missing data from other sites, create situations where data arrives piecemeal and not necessarily in order.

The problems described above are particularly acute in the emerging "Big Data" applications where stream processing systems are being increasingly used. Consider for example, the web-based digital media ecosystem of organizations delivering various services (e.g., social networking, advertising, video, mobile etc.) to internet-scale audiences. Such services operate in highly dynamic environments where monitoring and event data inherently arrives at multiple time-scales, and query results, analyses and predictions are needed across multiple time-scales. In such environments the source data is typically from log files spooled by large banks of distributed web/application servers. Failures in the source systems are commonplace and the log files are often delivered to the analytics system hours or sometimes days late. Finally, these services are more and more frequently deployed in cloud environments and have stringent availability requirements that must also be satisfied.

The limitations of traditional order-dependent stream processing systems, particularly for the growing class of applications in which the expectation of continuous streams is simply not realistic, has led us to develop a new approach to stream query processing that is *order-independent*. Our approach leverages key properties of relational query processing and the SQL language, without which, such an approach would be extremely difficult to realize. It relies on the flexibility of *data parallel* query processing mechanisms and uses sophisticated query rewriting techniques in order to hide the complexity of dealing with discontinuous streaming data from application developers and query writers.

The Truviso Continuous Analytics system inherently supports the processing of *discontinuous* streams of data. Any data that arrives too late to be handled by traditional schemes represents a

SIGMOD'10, June 6–10, 2010, Indianapolis, Indiana, USA.

discontinuity and is automatically partitioned and processed independently. This model of discontinuous streams maps much more naturally to the way streaming data arrives in large-scale deployments – chunks of data from individual log-files that are each mostly in order although they might arrive at widely disparate times.

As a result of this view of streams, the Truviso processing model differs significantly from that of other stream processing engines. In the following sections, we will show how our model greatly expands the environments and applications for which stream processing can be effective, and helps enable important data management services that are typically not associated with streaming and other real-time systems. In fact, a key insight is that once stream processing is freed from the assumption of continuous streams, many important features become more natural to implement, as they inherently disrupt ordering. Specifically, beyond enabling the handling of data that arrives significantly out-of-order, the order-independent approach is the used by Truviso Continuous Analytics System to support:

- Intra-query parallelism for both multi-core and cluster-based processing.
- Fault tolerance based on rapid recovery in which a recovering instance can begin processing newly arriving data while it is still re-establishing its state.
- High Availability deployments where recovering or new instances can retrieve state information and missed data from running instances.
- "Bootstrapping" continuous queries in which queries over streams can first be run over historical data while catching up to the current stream.

The remainder of the paper is structured as follows. In Section 2 we describe the state-of-the art in order-dependent stream processing, including the main tools that are used by systems to tolerate data that arrives slightly out-of-order. In Section 3 we introduce our order-independent processing approach, which is based on a unique dynamic stream partitioning approach and inspired by data parallel query processing techniques. In Section 4 we describe the query rewriting techniques we use to hide the underlying order-independent mechanisms. In Section 5 we present the architectural considerations of building a streaming system based on these techniques, including a description of how we support the features listed above. Section 6 presents our conclusions.

2. BACKGROUND: ORDER-DEPENDENT SYSTEMS

Input ordering has been a fundamental concern since the earliest stream processing systems, and there have been several different approaches in dealing with disorder. All of the previous systems of which we are aware, however, are ultimately sensitive to input order in a deep way, and as a result, they all must discard or divert records that arrive later than some threshold. We therefore describe such systems as being *order-dependent.* Li et al [13] further broadly categorized these systems as using either *in-order processing* (IOP) or *out-of-order processing* (OOP) techniques for records that arrive within acceptable time limits, and persuasively argue in favor of supporting the latter. Before describing our approach to processing arbitrarily late data, we first review previous techniques and summarize their limitations.

2.1 In-Order Processing

Most early streaming implementations were intolerant of disorder and expected data to arrive in a strictly time-oriented fashion. Some systems, like TelegraphCQ [6] and STREAM [3][15] enforced ordering on data input into the system as well as on output from any operator. Others, like Aurora [1] distinguished between *order-agnostic* (e.g., Filter) and *order-sensitive* (e.g., Aggregate) operators and enforced ordering on input to the latter.

Even in this early work, it was clear that the ordering constraints necessary for progress and correctness were untenable in many real environments. Thus, simple buffer-and-reorder mechanisms were developed to increase systems' tolerance to data arrival order. These mechanisms tolerated a specified amount of "slack", basically a bound on the degree to which input could be unordered. Slack mechanisms basically delay processing for the specified slack period, buffering incoming data and reordering any out of order data that arrives within that period before the data is processed. Thus at time *t*, input for time *t* - *slack* can be sent to the query processor, while input that is timestamped after than *t* - *slack* is buffered and reordered if necessary.

Slack can be specified on a per-operator basis (e.g., Aurora) or on a per-stream basis. While the slack mechanism is easy to implement, it suffers from several major disadvantages: (a) it introduces a constant minimal latency into the processing pipeline, (b) data that arrives after its corresponding slack period has expired cannot be handled, (c) memory requirements for continuously buffering input data can be high, even when data is actually in order or when the disorder is entirely due to merging multiple ordered sources, and (d) it does not handle lulls in either some of the streams or some of the sources of a given stream making it challenging to implement non-unary operators (e.g., Join, Union).

Row Num	Input Timestamp	Slack Buffer	Output Timestamp
1	1	1	
2	2	1,2	
3	3	1,2,3	
4	2	1,2,2,3	
5	6	6	1,2,2,3
6	5	5,6	
7	~~1~~	5,6	
8	9	9	5,6
9	8	8,9	

Figure 1 Slack example with grace period of 3

The example in Figure 1 shows the use of a slack buffer with a grace period of 3 seconds. In the figure, input arrives in the order shown from top to bottom. There is a row in the figure for each input record – the three columns represent the input tuple, the contents of the slack buffer and the output tuples all identified by their timestamp. The example shows that output data is always sorted but that latency is introduced into the output, and that data that arrives later than the grace period (e.g., the 8th tuple with timestamp 1) is discarded.

As with other streaming systems, the Truviso system includes a slack-like functionality, which can indeed be effective in hiding

small degrees of out-of-orderness when relatively short grace periods are used. Our system also provides an additional mechanism called "drift" for streams that are populated by merging streams from multiple, independent data sources, each of which itself is ordered. Such cases arise frequently in distributed environments where events of a single logical stream are collected at multiple servers.

Such merged stream processing works well when the various servers produce events at similar speeds and in a regular fashion. In such cases, the system's input manager performs an ordered merge of the various data sources without any buffering. If, however, there is a significant delay in any individual source, the processing of input from all of the servers could be delayed. The "drift" mechanism solves this problem. In the event of a lull at a particular source, the input manager buffers a drift interval's worth of data from the active sources before proceeding without the disrupted source. In effect, the drift approach is designed to provide a grace period on a per source basis by modeling the actual inevitable drift in the system clocks of a distributed system.

Row Num	Source 1	Source 2	Drift	Output
1	(A,1)	(a,2)		(A,1)
2	(B,2)	(b,3)		(a,2),(B,2)
3	(C,3)	(s,3)		(b,3),(s,3)(C,3)
4	(G,4)	(e,5)		(G,4)
5	(D,6)			(e,5)
6	(E,7)		(D,6),(E,7)	
7	(R,8)		(E,7),(R,8)	(D,6)
8	(F,9)	~~(x,5)~~	(R,8),(F,9)	(E,7)
		(z,10)	(z,10)	(R,8),(F,9)

Figure 2 Drift example with grace period of 2

The example in Figure 2 shows the drift mechanism with two sources and a grace period of 2. Each row in the table represents a new tuple in the system from one or both of the two sources – the second two columns show a tuple-timestamp pair from each source, the fourth column shows the contents of any drift buffer, and the fifth column shows the output tuples that are emitted by the drift mechanism. Note that the second column is empty for rows 5,6, and 7 depicting a lull in the second source, which in turn leads to the drift buffer filling up with records until the (R,8) tuple at row 7 comes in from source 1 freeing the (D,6) tuple for processing. The lull in source 2 ends at row 8 with the input record `(x,5)` which is too late for processing and is not merged into the output stream.

Srivastava et al [17] proposed a more nuanced approach where an input manager considers various parameters including some specified by users and some inferred from the data to dynamically buffer and reorder data before processing and issue heartbeats in the style of punctuations [18] in order to ensure that the system can continue to make *progress* even when an individual source is delayed.

A common aspect of all the approaches described in this subsection is that their goal is to preserve ordering invariants on single source and/or multisource streams, so that streaming query operators can be implemented assuming proper ordering of the input. We refer to such approaches as In-Order Processing (IOP). These techniques, necessarily trade off latency and memory footprint for inclusiveness. That is, by delaying the delivery of some input to the processing engine, they can deliver more data to that engine in cases where ordering assumptions would otherwise be violated. Early stream processing engines made the bet that in many environments, delays would be short and rare, and thus, such mechanisms would be sufficient. In our experience however, this assumption proves to be overly optimistic for all but the most constrained situations.

2.2 Out-of-Order Processing

In contrast to the early systems that relied purely on IOP techniques, some next generation systems (e.g., including Truviso systems) employed independently developed variants of the OOP approach described by Li et al. The idea behind the OOP idiom is, where possible, to implement flexible and efficient query operators that can accept data out of order. For such operators to know when to emit results and purge state (e.g., when a window closes), a notion of stream progress is still required. The crux of the OOP technique is to provide this notion of stream progress at a system level without actually buffering or reordering the data. In such systems, stream progress is generally communicated using punctuations (*control tuples* in the Truviso system). Each individual operator is responsible for understanding these punctuations and for sending progress information to its downstream operators.

With this approach it is possible to implement certain types of operators in a way that is tolerant to moderate amounts of disorder in their input. For example, consider the tumbling (i.e., non-overlapping) window count query with a window of 5 minutes shown in Figure 3, which operates over a stream of data with many records arriving slightly out of order, and where an input manager provides progress information on some heuristic basis.[1] In the example there is a row for each input tuple (data or control) with columns depicting the timestamp of the tuple, the internal aggregate state for count, and the output tuple that is produced as and when control tuples arrive.

```
select   count(*), cq_close(*) t
from     S <slices '5 minutes'>
```

Row Num	Input Data	Count Control	Count State	Output Tuple
1	1		1	
2	3		2	
3	2		3	
4	4		4	
5	2		5	
6	1		6	
7		5		(6,5)
8	6		1	
9	~~2~~		1	
10	9		2	
11	7		3	
12	~~3~~		3	
13		10		(3,10)
14	12		1	
15	~~8~~		1	
16	~~4~~		1	
17	~~3~~			
18	~~9~~			
19		15		(1,15)

Figure 3 Order-Dependent Out-of-Order Processing

[1] The "cq_close(*)" aggregate function returns the timestamp at the close of the relevant window.

A smart order-agnostic aggregate implementation can accept the incoming data in any order and still accumulate it correctly. When the progress information indicates that a given window is closed the aggregate operator can emit its output for that window. In the first window of the example, although rows 3, 5, and 6 (with timestamps 2, 2, and 1 respectively) are out-of-order they are still counted correctly, and when the control tuple with timestamp 5 arrives an output tuple with value (6,5) is emitted.

As demonstrated by Li et al., the advantages of such an approach are apparent – reduced memory and latency compared to the IOP approaches described above. In spite of these advantages, however, the OOP approach is fundamentally limited by the accuracy of the progress information that the input manager can construct. Any data that arrives sufficiently late, beyond an inferred or specified low water mark, will be discarded without being processed[2]. For example, in Figure 3 six of the sixteen data tuples arrive sufficiently late to be dropped. Furthermore, the incorrect results produced by this query will also affect the results of downstream queries. Suppose, for instance, the output of this query was being rolled up into a super-query computing a tumbling window count with a "15 minute" window (i.e., by summing up the three 5 minute totals produced here), the first output of the super-query will be (10,15) although there were actually 16 records in the 15 minutes.

2.3 Limitations of order-dependent systems

In essence, order-dependent systems are ideal for processing streaming queries on *mostly ordered input data* with a design point that assumes that out-of-order tuples are only slightly late. This is a reasonable assumption in some environments such as the financial trading space (the initial market for pure streaming systems) where trading algorithms have such small windows of opportunity that handling very late data is not particularly helpful.

In our experience, however, the requirements imposed by an order dependent system are fundamentally unrealistic and unacceptable in most enterprise environments because of the following reasons:

Historical integrity: It is vital to ensure that a streaming system correctly processes and accounts for every input tuple, no matter how late it arrives in the system, in order to preserve the integrity of stream histories. While this is obvious in the case of the stream-relational model that Truviso pioneered, it turns out that it is also true in pure streaming systems, which depend on the integrity of historical data either for replaying old data (e.g., for back-testing financial trading algorithms) or when comparing current results with a historical baseline in order to perform alerts.

Disconnected sources: In applications such as analytics over web/mobile/video, the data sources (web clients) are widely distributed and issue beacon requests in order to track activity. In such environments involving temporarily disconnected data sources it's quite impossible to infer any kind of progress information. For this reason, one of the biggest web-analytics vendors generally timestamps data on arrival to the system and cannot combine such data with previously timestamped data generated by disconnected clients (see "Sequential Data Requirements" on page 103 in [19]).

Scalability and Fault-Tolerance: It is extremely hard and quite impractical to deploy an order-dependent streaming system in a scalable and/or fault-tolerant cluster configuration. In such a configuration, data that is transferred between nodes can arrive significantly later than data that originated at a given node. Worse, in the event of a failure and subsequent recovery of a node, the newly recovered node needs to retrieve (from a peer) the data it missed during a crash, process that data, and only then "catch up" to live data – a process that requires a very complex and error-prone protocol. The fault-tolerant protocols developed as part of the Flux [16] and Borealis [4] systems are also quite complex for much the same reasons.

3. ORDER-INDEPENDENT SYSTEMS

Due to the inherent limitations in the traditional Order-Dependent approach to stream query processing outlined in the preceding section, we have developed and implemented a unique approach to Order-*Independent* stream processing. This technique leverages some of the unique aspects of our hybrid "stream-relational" architecture.

The Truviso stream-relational system [10] system can be used to manage historical as well as streaming data. The system supports two general classes of queries: (a) push-oriented live/continuous queries (CQs) that operate over either streams or a mix of streams and historical data stored in tables, and (b) pull-based static/snapshot queries (SQs) over historical data stored in tables. Note that the tables used to hold historical data are, in fact, full-blown SQL tables, so SQs are written in standard SQL, with all the bells and whistles allowed.

In this section we present the foundations underlying our order-independent query processing approach. We then build on this foundation to elaborate in more detail in the following sections.

3.1 Going, going, gone!

The central challenge in designing an order-independent system is what to do with very late data that arrives after a window closes. Of course, for certain monitoring/alerting queries that use only live data, this issue is largely moot – for such applications, once results are presented to the user or the application, the "ship has already sailed" and little can be done about it[3]. In such applications, for cases where the techniques described in the previous section do not work, application-level remedies such as compensations are required. One approach for this was described in Borealis [2] where the system replays history to generate revisions of query results, which applications must then handle.

Most real applications involving data streams, however, involve result delivery across a wide range of time scales: real-time alerts and monitoring, periodic reporting on an hourly, daily or even longer basis, as well as on-demand reports and queries over historical data. Furthermore, even "real-time" queries used for monitoring alerting typically compare incoming data to historical trends. Data arriving very late can impact the correctness of all of these types of queries, resulting in long-term inaccuracies due to short-lived problems that frequently occur in data-intensive environments.

The goal of an order-independent system, therefore, is to maintain the integrity of raw and derived stream histories. That is, the effects of any tuple that arrives too late to handle using standard

[2] Note that late data can be discarded by a window operator on a per-query basis rather than by a system-wide input manager.

[3] "The moving finger writes, and having written moves on. Nor all thy piety nor all thy wit, can conceal half a line of it" – Omar Khayyam.

OOP techniques must eventually be reflected in the historical archive. The techniques we describe below achieve this goal by leveraging key features of relational query processing in a unique way, in the streaming context.

3.2 Parallelism to the rescue

The key to our approach is a simple trick inspired from data-parallel processing paradigms. The goal is to process arbitrary partitions of an input stream in an isolated fashion independent of each other. At the heart of this approach is the concept of *partial processing*. More precisely, the partial processing of independent partitions of data produces partial results that can then be *consolidated* to produce final results for the queries. The consolidation can take place either immediately for a live CQ or *on-demand* when a portion of the archive is actually queried as part of either a live CQ or a historical SQ.

When the system's input manager receives data that is older than an already emitted progress indicator (i.e., tuples that arrive after the appropriate window has closed) the system organizes a separate partition for the data and processes this partition through a new instance of the original query plan. This new plan instance is automatically created on the fly by the system whenever a new partition is formed. Each individual partition is processed using standard OOP techniques where the input manager ensures that all tuples sent to a given partition can be safely processed without being discarded.

In principle, the system can maintain an arbitrary number of concurrent partitions and associated query plans. In practice we use some heuristics and maintain no more than a handful of active partitions – if there is no progress on a given partition it is sent heartbeats to complete its processing and then eliminated. In general, when a new tuple arrives and is seen as being too late to process for the original partition, there can be several partitions that are candidates for the tuple and the system therefore employs a simple rule to choose an appropriate partition: it picks the partition whose latest data is closest to the new tuple.

```
          select   count(*), cq_close(*) t
          from     S <slices '5 minutes'>
Row      Input       State Partitions          Output
Num    Data  Ctrl  part-1  part-2  part-3      Tuples
-------------------------------------------------------
 1     1            1
 2     3            2
 3     2            3
 4     4            4
 5     2            5
 6     1            6
 7           5                                 (6,5)
 8     6            1
 9     2            1       1
 10    9            2       1
 11    7            3       1
 12    3            3       2
 13          10             2                 (3,10)
 14    12           1       2
 15    8            1       1                 (2,5)
 16    4                    1       1
 17    3                    1       2
 18    9                    2       2
 19          15             2       2         (1,15)
 20       flush-2                   2         (2,10)
 21       flush-3                             (2,5)
```

Figure 4 Order-Independent Partial Processing

Figure 4 demonstrates the use of order-independent partial processing with the exact same data set shown earlier in Figure 3, which resulted in six out-of-order tuples being discarded. In this figure we have 3 columns each representing the state of an individual partition. The behavior in the first window is identical to that of the OOP approach described earlier and the result returned in this example is identical to that shown in Figure 3. In the second window, however, arrival of the out-of-order tuple with timestamp 2 (row 9) causes the system to spin up a second partition. When the out-of-order tuple with timestamp 3 arrives during that same window, it is handled in the second partition, as it is still in-order relative to that partition.

When the tuple with timestamp 8 (shown in row 15 in the figure) comes in during the third window, its timestamp is high enough to cause the open window of the second partition to close, producing a partial result of (2,5) and processing the new tuple in the second partition associated with the second window ending at time 10. When the next two tuples (at rows 16 and 17) with timestamps 4 and 3 come in, they are too late to be processed in the second partition and require the system to spin up a third partition where they are sent. Next, the tuple with timestamp 9 (row 18) comes in and is sent to the second partition. When the system receives a control tuple with timestamp 15 it flushes the second and third partitions producing partial results of (2,10) and (2,5). Now if as in the example of Figure 3, the output of this query was rolled up into a super-query computing a tumbling window count with a "15 minute" window, using this Order-Independent Partial Processing method, the first output of the super-query will in fact be (16,15) – the correct answer.

The technique of combining partial results to form final results is widely known and is used in parallel systems ranging from MPP databases [9] to MapReduce (MR) [8] implementations. There is, however, a key difference in the way we apply this technique compared to how it is done in those types of systems. In systems like MPP databases and MapReduce, the partial processing (e.g., the *map* in MR) is immediately followed by the consolidation (e.g., the *reduce* in MR): the two phases are tightly integrated whether by being placed in the same thread of execution (in MPP databases) or more loosely integrated using intermediate files (in MR systems). In contrast, in our approach we decouple these two phases. That is, we apply consolidation lazily, only when results are needed. Lazy consolidation on an on-demand basis is sufficient for an SQ or portions of a CQ that operate on archived histories, and is accomplished by using a set of novel query rewrites and views we detail in the following Section. In order to evaluate CQs over parallelized derived streams, however, consolidation across multiple concurrent stream partitions must happen in runtime on an online basis. This online consolidation is in fact a CQ that is run in serial fashion as detailed in Sections 4.3 and 5.2.

Apart from being a critical building block of the Truviso order-independent system, our partial processing approach is particularly appealing because: (a) it is an ideal fit for the Big Data world where data often comes in the form of chunks of data that are in order but where any individual chunk may arrive significantly late and (b) it is naturally amenable to parallelism in general and scale-up in particular since each partition is processed independently by a separate dataflow, making it very easy to exploit modern multi-core and many-core hardware.

4. Parallel Partial Processing

In the previous section we described the underlying concepts behind our order-independent architecture. In this section we provide more details on our parallel partial processing technique that was sketched out in Section 3 by illustrating our innovative query rewrite technology with examples.

First, we check if a query can operate in terms of partial aggregates and if so transform it appropriately. Next we generate views that encapsulate deferred consolidation of partial aggregates. We then focus on more complicated scenarios where a query has to be broken up to operate in partials form. Finally, we consider how to rollup partials via successive reduction – a form of consolidation that leaves the results in partial state. For simplicity, we show each of these techniques using descriptive examples instead of presenting the algorithms we have developed for this purpose. For the same reason, we have also taken liberties with the syntax of the Truviso query language in these examples.

4.1 The Intuition

At a high level the rewrites involve various transformations of queries submitted to the system. These transformations enable the queries to correctly execute over streams that are dynamically partitioned. Relational views (both static and streaming) are then put in place to hide the underlying partitioning from query writers and application developers. That is, queries are written assuming traditional, in-order streams, even though the streams may actually be made up of overlapping partitions, created by the late data handling mechanisms of the previous section, or for other reasons related to parallelism, distribution, fault tolerance or high-availability, as discussed in the subsequent Section.

For ease of exposition, we focus on aggregate queries, which are our most common use case. More specifically, we consider *distributive* (e.g., max, min, sum, and count) as well as *algebraic* (e.g., avg) aggregates as classified by Gray et al [12]. These are typical aggregates in database systems, and can be computed using *final aggregation* over *partial aggregates*[4] over disjoint partitions of their input, a technique used with parallel databases (e.g., Bubba [5]) as well as streaming systems (e.g., STREAM, Telegraph etc.).

The approach requires us to implement aggregation functions using three kinds of aggregates. A *partial aggregate* runs over a partition of base data, and computes an intermediate result (or "partial state record") for that partition. A *reduce aggregate* runs over a set of partial state records and produces another partial state record, which summarizes its input. A *final aggregate* runs over a set of partial state records and produces a final result, of the same type as the originally specified aggregation function.

Aggregate Fn	Implemenation
count(x)	
count_partial(x)	count(x)
count_reduce(c)	sum(c)
count_final(c)	sum(c)
avg(x)	
avg_partial(x)	<sum(x), count(x)>
avg_reduce(<s,c>)	<sum(s),sum(c)>
avg_final(<s,c>)	sum(s)/sum(c)

[4] In general, the functions used for the partial aggregates can be different from those for the overall aggregate.

The chart above shows three kinds of aggregates (partial, final, and reduce) that are associated with each of the count and avg aggregate functions. In the case of count there is no last step for the final aggregate since the intermediate state is itself the answer and therefore the final and reduce aggregate are identical: the sum aggregate function. In the case of avg, however, the final aggregate is different from the partial aggregate. The partial state produced by each separate avg_partial is in the form of <sum,count> pairs. While these can be rolled up to get an actual average using the avg_final aggregate, they can also be rolled up into a single <sum,count> pair. In general, if it is possible to express any aggregate using partial and final aggregates, it is also possible to derive the reduce aggregate – simply leave out the last "finalization" step of the final aggregate.

The key insight is the following: using this partial aggregation approach with the query and view rewrites described in this Section, our system is able to run in "partial aggregation mode" as its normal mode of operation. Since partial aggregation allows computation to be done over different partitions independently, this approach enables us to handle very late data simply by dynamically creating partitions as described in the previous Section. Partial aggregation results are computed as data arrives and are stored in their partial state. Results are then finalized *on demand* at runtime, using the reduce and final aggregate functions. Most importantly, using the rewrites described next, query writers (whether human or algorithmic) can code without any concern for such partitioning.

4.2 The Initial Rewrite

We start with a simple scenario modeling clickstream data for online advertising (see Figure 5) with a raw stream called impsns (short for "impressions") where each tuple records the fact that an ad from a certain campaign was shown to a given user at a specified price and time. There is also a derived stream (i.e., a stream that is defined as the results of a CQ) called impsn_count that contains the number of ad impressions for each campaign over successive "1 minute" slices using a straightforward grouped aggregate query whose results are archived into a table called impsn_count_ar.

```
create stream impsns(
       user     int,
       campaign int,
       price    int,
       ts       timestamp cqtime
);
create stream impsn_count as
 (select  campaign, count(*) c, cq_close(*) t
  from    impsns <slices '1 minute'>
  group by campaign)
archive into impsn_count_ar;
```

Figure 5 Example scenario

The first step of the rewriting process is to determine whether or not the query that produces impsn_count can be rewritten in a partial aggregation form. This check is substantially similar to decisions made in a query optimizer for a parallel RDBMS. In this case, it's easy to see that the computation in question (the

count(*) aggregate function) can be split into two phases: a partial phase that computes a per-campaign count for each individual run, and a final phase that consolidates these results across all runs (by summing the per-run per-campaign counts on a per-campaign basis) to compute global per-campaign counts across runs. The first part of this split is accomplished by creating a derived stream of partial aggregations as shown in Figure 6.

```
create stream i_c_p as
 (select campaign, count(*) pc, cq_close(*) t
  from   impsns <slices '1 minute'>
  group by campaign) archive into i_c_p_ar;
```

Figure 6 Example scenario rewritten as partial

While in this case the CQ on which the system-generated stream i_c_p is based is identical to that of the original user-created stream impsn_count, the rewrite is still necessary for two reasons. First, the partial aggregate function and the resulting per-run aggregate values could be different from the original aggregate function when using aggregates such as avg (see Section 4.5), as opposed to the count(*) used in this example. Second, a new name for the stream is required because the system generated i_c_p stream contains the results of the partial phase, i.e., per-campaign counts on a per-partition basis and not what the user originally defined in the impsn_count stream.

4.3 Consolidating static and streaming views

Recall that a key requirement of our rewriting approach is that there should be no changes required to application-level code. Thus, part of the rewriting process is to create special static and streaming views with names identical to that of the original stream and archive, which hide the rewriting of the original stream. In our advertising example, such views (Figure 7) consolidate the per-partition per-campaign counts across all independent and possibly concurrent partitions in order to compute the per-campaign counts for each 1 minute slice, as would have been produced by the original query. As a result, applications require no modifications; they query the views just as if they were dealing with the original user-created stream and table objects, unaware of any rewrites.

```
create view impsn_count as
  (select campaign, sum(pc) c, cq_close(*) t
   from   i_c_p <slices '1 minute'>
   group by campaign);

create view impsn_count_ar as
  (select campaign, sum(pc) as c, t
   from   i_c_p_ar
   group by campaign, t);
```

Figure 7 Consolidated streaming and static views

It is important to understand that these consolidated views are not materialized, so that final aggregation happens on demand in response to the execution of a query over one of these views. When an application issues a query on impsn_count_ar associated with the impsn_count stream it is actually issuing a static query against the impsn_count_ar view, which in turn is rewritten automatically by the system to operate on the i_c_p_ar object. Similarly, if an application issues a live continuous query on the results of the impsn_count stream (as originally defined) it us actually posing an appropriate continuous query over the impsn_count streaming view. In response, the system rewrites the reference to impsn_count to include the definition of the view, which in turn consolidates results from multiple concurrent runs.

The definitions of both views are fairly straightforward and merely involve the use of the reduce aggregate function (sum in this case) to combine the partial aggregate results produced by the i_c_p stream. One important nuance is in the impsn_count_ar static view where the timestamp column t is included in the group by clause. This is necessary in order to combine partial results of a given window. Note that the column t is actually populated by the cq_close(*) aggregate function in the rewritten derived stream i_c_p.

4.4 Stacking up queries

We now consider more complicated examples where only parts of a query may be amenable to partial processing. In such cases, it is often possible to carve out a sub-query block that can be written in data parallel mode. We identify the dividing line between the sub-query and the rest of the query as a "gather" point. A gather point corresponds quite closely to points in parallel DBMS query plans where results from various nodes need to be brought together. We demonstrate this approach using examples of individual derived streams written on top of impsn_count and identify cases where these second order streams are candidates for the partial transformation.

```
create stream impsn_sum as
 (select sum(c) s, cq_close(*) t
  from   impsn_count <slices '1 min'>);
```

Figure 8 Super-query that can be deployed in RB mode

We begin with the stream impsn_sum (Figure 8) that computes the total number of impressions across all campaigns over successive 1 minute slices. It turns out that this stream can indeed be rewritten with the partials transformation shown earlier because it is possible to compute partial aggregates for the sum function over partial aggregates of the underlying count function. In this case, impsn_sum would be automatically rewritten to operate over the previously generated stream of partials i_c_p as opposed to the consolidating streaming view impsn_count. Note that this happens although impsn_sum is declared to be over impsn_count, because (as shown above) impsn_count is actually a view.

```
create stream impsn_max as
 (select   max(c) m, cq_close(*) t
  from     impsn_count <slices '1 min'>)

create stream impsn_thresh as
 (select   campaign, sum(c) c, cq_close(*) t
  from     impsn_count <slices '5 min'>
  group by campaign
  having   sum(c) > 50);
```

Figure 9 Queries that cannot be deployed in partial mode

Next we consider the create statements for streams impsn_max and impsn_thresh shown in Figure 9, neither of which can be rewritten using this technique. In the case of impsn_max (which, computes the maximum number of impressions across all campaigns in successive 1 minute slices) this is because the aggregate function "max" cannot be composed on top of the underlying count since the max of partial counts for various groups in a single partition is not the same as the max of the complete counts for the various groups across partitions. More

formally, let there be n distinct campaigns, k partitions of data, leading to the i^{th} partition producing partial counts $c_{i1},c_{i2},\ldots,c_{in}$ for each of the n campaigns. In this case, deploying impsn_max using the partial parallel processing rewrite would end up computing $\max(\max(c_{11},\ldots,c_{1n}),\max(c_{21},\ldots,c_{2n}),\ldots,\max(c_{k1},\ldots,c_{kn}))$ as the result value instead of the correct answer whish is $\max(\text{sum}(c_{11},c_{21},\ldots,c_{k1}),\text{sum}(c_{12},c_{22},\ldots,c_{k2}),\ldots,\text{sum}(c_{1n},c_{2n},\ldots,c_{kn}))$. More generally, there are legal compositions (e.g., sum on top of count) and illegal compositions (e.g., max on top of count) that the system tracks in order to determine which rewrites are legal.

The situation for impsn_thresh (which returns the campaigns that have impression counts exceeding a threshold of 50 over successive 5 minute windows) is similar – the expression in the having clause must be evaluated *after* the partial results are consolidated.

4.5 Rollups: partial consolidation

We now consider the important use case of rollups where the user's intent is to compute the same aggregate at various levels in a hierarchy.

```
create stream impsn_count_5min as
 (select  campaign, count(*) c, cq_close(*) t
  from    impsns <slices '5 min'>
  group by campaign);
```

Figure 10 Rollup of aggregates (simple case)

Although the stream impsn_count_5min (Figure 10) computes per-campaign counts over successive 5 minute slices and is written directly against the raw stream impsns, it is clearly a candidate to be rewritten over the derived stream impsn_count which computes the identical measure over successive 1 minute slices. It's easy to see how impsn_count_5min can exploit partial rewrites: compute per-campaign partial counts over a 5 minute window by adding up the constituent partial counts of 1 minute windows. In other words, the rollup involves combining partial results to form other partial results for the same exact aggregate function.

```
create stream impsn_avg as
(select campaign, avg(price)p, cq_close(*) t
 from   impsn <slices '1 min'>
 group by campaign);

create stream impsn_avg_5min as
(select campaign, avg(price) p, cq_close(*) t
 from   impsn <slices '5 min'>
 group by campaign);
```

Figure 11 Complex rollup

Things are considerably more complex, however, if the avg aggregate function is used in place of count in the impsn_avg and impsn_avg_5min streams as shown in Figure 11.

```
create stream i_a_p as
(select campaign,avg_partial(p) p, cq_close(*) t
 from   impsns <slices '1 min'>
 group by campaign);

create stream i_a_p_5 as
(select campaign, avg_reduce(p) p, cq_close(*)t
  from  i_a_p <slices '5 min'>
  group by campaign);

create view impsn_avg as
(select campaign,avg_final(p) p,cq_close(*) t
 from   i_a_p <slices '1 min'>
 group by campaign);

create view impsn_avg_5min as
(select campaign, avg_final(c), cq_close(*) t
 from   i_a_p_5 <slices '5 min'>
 group by campaign);
```

Figure 12 Rewrites for complex rollups

In such a situation, the rewrite of the lower-level stream would use the avg_partial aggregate, which returns a composite type consisting of a <sum,count> pair. The consolidating views would thus use the avg_final final aggregate to roll up these pairs to produce an actual average on demand. The rewritten higher level stream must, however, combine partial values produced by avg_partial (a set of <sum,count> pairs) and produce a result which is also in the partial form (a single <sum,count> pair). Therefore, this requires the higher-level stream to be rewritten using the reduce aggregate (avg_reduce in this case), which performs the partial state to partial state reduction. These rewrites are shown in Figure 12.

4.6 Background Reducer

In situations where there are large amounts of out-of-order data, the basic approach described in this Section and the previous one could result in large numbers of different partial state records for a given window. This is especially true if query windows are relatively small since such windows are more likely to close before the late data is processed, thereby triggering fresh partitions which in turn cause more partial state to be written out. Large numbers of partial state records increase the cost of evaluating the consolidating views. We therefore use a background process that periodically employs the "reduce" aggregate to keep the partial state small. Note that we need to use the reduce aggregate for two reasons: (a) the reducer updates the contents of a partial archive operating over partial state records and must therefore also generate partial state records, and (b) there may be extremely late data which only arrives after the background reducer has summarized some partials and so the on-demand consolidating views will need to combine the latest new partials from the late data with the reduced partials from earlier data. Thus the reducer's contract is to keep the data in partial aggregate form for deferred on demand consolidation.

In this section we described the implementation techniques that can be used to realize the order-independent processing model for parallel processing streams. We build on this in the next section as we combine parallel processing with serial processing streams.

5. PUTTING IT ALL TOGETHER

In this section we describe the overall architecture of the Truviso system in order to provide guaranteed order independence.

The data-parallel approach described in Sections 3 and 4 offers a powerful new processing model for handling discontinuity in data thereby enabling the development of a scalable and reliable system. Unfortunately, this approach is by itself insufficient for the following two important reasons:

1. There is still very much a need for live CQs that are able to consolidate partial results on the fly and provide

immediate alerts, often by comparing the combined data with historical values.

2. Just as not all query operators in an MPP database system can be parallelized, not every CQ operator is amenable to the partial processing idiom. In fact the operators for which parallelization is effective are the ones for which partial processing is most applicable.

The above reasons suggest that a complete solution must combine the parallel processing model with the traditional serial-processing approach, while still offering order-independent behavior to applications. Supporting the latter is particularly challenging and also vital in building a scalable and reliable system as explained earlier in Section 2.2.

In what follows we describe the overall Truviso architecture followed by a sketch of the various other problems we solved in order to build an enterprise-class system.

5.1 System Architecture

The basic idea in our combined approach is to split the entire CQ workload into a *parallel* phase that relies on partial processing and a *serial* phase that operates using a more traditional OOP based approach as follows:

1. Automatically identifying the portions of the workload that are suitable for parallel versus serial processing.
2. Pushing down partial CQ processing right to the input manager stage of the data loading pipeline.
3. Hooking up the results of concurrent partial processing to the traditional OOP-based serial processing.

In what follows, we will use the terms parallel processing (PP) streams for the raw and intermediate derived streams that are earmarked for the parallel phase, and serial processing (SP) streams for derived streams and other CQs that operate in the serial phase.

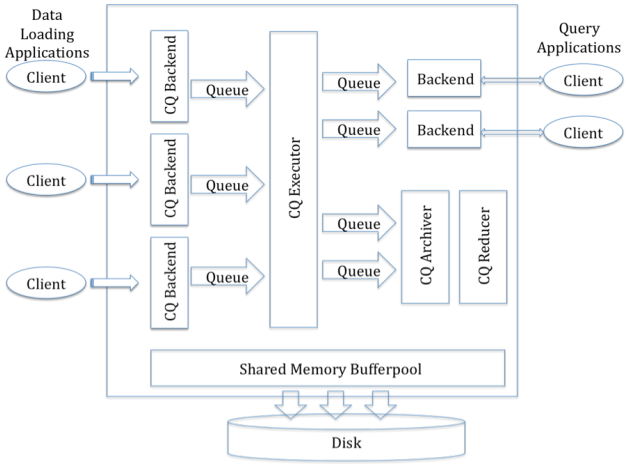

Figure 13 Truviso System Architecture

The architecture of the Truviso system is shown in Figure 13. Data sources connect to the system using standard protocols and start pumping data into streams, typically using a bulk loading API. In the usual manner, the system forks an associated CQ Backend thread that is dedicated to each such connection. This thread lazily instantiates a local data flow network of CQ operators appropriate for the stream on which data arrives. These operators have specifically been generated for the PP streams that correspond to a particular raw stream and in turn produce runs of results. In addition, based on the specified DDL and query rewrites, the local data flow network is responsible for:

1. Archiving into tables the raw data it processes as well as the corresponding PP results it produces.
2. Sending the PP data to the shared CQ executor for use in processing SP streams via shared memory queues. The CQ executor also receives similar data from Backend threads that services other data sources. Note that the CQ executor is part of a single very long-running transaction for its entire existence.

The CQ executor thread fetches PP records from input queues and processes them through a data flow network of CQ operators using the OOP techniques described earlier in Section 2.2. The CQ executor can itself exploit multiple available cores in the system by splitting either operators or entire queries across multiple threads. Finally the system also includes a CQ archiver thread that is responsible for writing out windows of data that were produced for SP streams by the CQ executor to tables, a CQ reducer thread that is responsible for eagerly combining partial results in the background, and a CQ repair thread that continually fixes the contents of archives of SP streams. Note that it is possible to spawn several instances each of the executor, archiver, reducer and repair threads.

5.2 Order-Independence

In the case of PP streams, we get order-independence for "free" as described in Sections 3 and 4. For SP streams however, our technique of implementing order-independence is based on processing very out-of-order data (too late for OOP) by periodically *repairing* any affected archived SP data. The archives are therefore guaranteed to be correct with respect to out-of-order data on an eventual consistency basis. The mechanics involve two pieces: (a) spooling all tuples that arrive too late into an auxiliary structure (a system-generated corrections table), and (b) a CQ repair process that periodically scan records from the auxiliary table and combines them with an appropriate portion of originally arrived tuples in order to be able to recomputed and update affected portions of archived SP data. This approach is in many respects similar to the dynamic revision of query results in Borealis [2] with one difference – the Truviso system updates the archives of streams in response to late data, whereas Borealis updates the results of a streaming query.

5.3 Unit of Work

A unit of work, or a transaction [11], is generally associated with the well-known ACID properties: atomicity, consistency, isolation and durability. We focus here on atomicity and durability and have addressed isolation and consistency in earlier work [10][7]. Durability is critical in order to be able to recover state after a crash. Atomicity is vital in order to more easily undo the effects of failures in either individual applications or the system. In fact, these properties are the key to the typical way in which data is loaded into analytic systems where the loading application batches up a data set and loads it in a single UOW. This model is vital in connecting a transactional message queue with the streaming system in a manner that guarantees that no records can ever be lost.

The idiom described above depends on the ability to abort a data loading transaction – either based on an error condition (e.g., in the loader, network, or system) or because the loader chose to abort the transaction for whatever reason. On abort it is vital that all modifications to raw and derived stream histories must be rolled back, at least eventually. It is very challenging to support this abort requirement in a pure streaming system because (a) waiting until data is committed before processing leads to significant extra latency and defeats the purpose of a streaming system, and (b) commingling and then processing dirty uncommitted data from multiple transactions makes it hard to unwind the effects of a single transaction. The latter is particularly hard because archiving of SP streams is the responsibility of separate Archiver threads that run in their own transactions and are independent of the Backend thread that manages the UOW in which data is loaded.

Our solution to this problem involves two parts: (a) we push down the partial processing of CQs (and archiving query results) to the input manager thread that handles the data loading transaction as described in Section 5.1, and (b) we organize the data-loading application into several, possibly concurrent, units of work each of which loads one or more chunks of data that we call "runs". More precisely, a "run" is a finite sub-part of a stream that arises naturally as a by-product of the way that data is collected and sent to a streaming system. Typically, individual systems (e.g., application servers or web servers) spool data into log files they keep locally. These log files are often split at convenient boundaries based on number of records, size, or time (e.g., 50K records, every two minutes, etc.) and are then sent separately to the stream-processing engine: in Truviso the files are actually bulk-loaded through standard interfaces such as JDBC/ODBC. Such log files serve as natural units for "runs", with clear boundaries. In other situations, where data is sent in a true streaming fashion, a run can be created by periodically committing the data-loading transaction – a mechanism that is akin to inserting punctuation records indicating when a run begins and when it ends.

5.4 Recovery

The database recovery problem [11][14] is generally defined in terms of bringing the system back up to a sane and consistent state after a crash when all in-flight transactions during the crash are deemed aborted. The recovery of archives of PP streams comes for "free" since all writes of raw and corresponding derived data happen as part of the same transaction. We benefit not only from the robust recovery architecture of the underlying PostgreSQL storage subsystem but also from other enterprise-class features of PostgreSQL such as online backup mechanisms.

Recovery for serial processing is a more challenging requirement because of the large amounts of *runtime state* managed in main-memory structures by the operators in the CQ executor as well as the decoupled nature in which durable state is written out originally by the CQ archiver. Crash recovery therefore involves three steps: (a) standard database style recovery of all durable state, (b) making all SP archives self-consistent with each other and the latest committed data from their underlying archive and therefore the archive of the raw stream, and (c) rebuilding the runtime state of the various operators in the CQ executor. Only after all of these are accomplished can the CQ executor be declared to be "open for business" again.

The ability to have a robust recovery implementation that is capable of quickly recovering from a failure is essential. Furthermore, the longer it takes to recover from a failure, the more the amount of pent-up data that has gathered and the longer it's going to take to catch-up to live data.

5.5 Atomicity

All data that is generated in a PP stream archive is automatically consistent with respect to the UOW in which data is loaded into the underlying base stream. It is critical that the same UOW atomicity property is also supported for archives of SP streams in order for recovery and high-availability to work correctly. One simple but clearly unsuitable approach to facilitate atomicity is to delay the processing of any data until it has been committed. Waiting for commits unfortunately introduces latency with the system devolving into a traditional RDBMS. What's really required (and implemented in the Truviso system) is the ability to offer a strong guarantee about the atomicity and durability of any and all data loaded into a system within a single UOW without compromising on immediate processing of data. This calls for speculatively processing dirty uncommitted data in a *laissez-faire* fashion based on the assumption that errors and transaction aborts are few and far between. When a transaction is actually aborted the system will asynchronously *repair* the associated SP archives in a manner similar to how repair guarantees order-independence on an eventual consistency basis.

5.6 Fault-Tolerance and High-Availability

While our data-parallel approach provides a natural path to scalability, we now explain how to enhance it in order to provide for Fault-Tolerance (FT) and High-Availability (HA). We define FT as the ability of a system to react well from some kind of extreme or catastrophic error – whether in the streaming engine itself, in the application, or in some aspect of the hardware and/or software environment. In particular, quick recovery from a failure state is critical in realizing FT. We characterize HA as the ability of a system to remain up even in the face of a catastrophic error. HA is generally realized using additional back-up resources that are organized together in either an "active-standby" or "active-active" configuration.

The UOW and Recovery functionality sketched above serve as key building blocks for HA and FT in the Truviso system. Our implementation supports a comprehensive HA/FT solution by organizing a cluster of Truviso nodes in a multi-master active-active configuration.

In this setup, the same CQs are typically running on all nodes of the cluster. Any incoming run of data can be sent to any - but only one - node in the cluster.[5] It is then the responsibility of a special stream replicator component in each Truviso node to communicate the *complete* contents of each run to the peers in the cluster. The runs of data that are populated into a stream by a peer are treated just like any other incoming data except for one thing – they are not further re-replicated to other nodes.

Our model here is one of eventual consistency – in other words, the run replication procedure happens on an asynchronous basis and not part of each commit of a UOW. The only downside of eventual consistency is the very small risk of data loss in the event

[5] In the interest of simplicity we are leaving out a discussion of our horizontal scale-out functionality within a cluster.

of any catastrophic media failure between a run getting committed and replicated to a peer.

It is important to understand the critical role that our order-independent infrastructure plays in realizing a simple and sane HA/FT architecture. Since each individual node in the cluster can accept data in any order whatsoever, the different nodes can stay loosely coupled and implement simple and easy to verify protocols. What's more when a node recovers from failure it is immediately able to start accepting new transactions and patch up the data it has missed asynchronously. In contrast, alternative approaches to FT such as Borealis [4] require that a node recovering from failure refuse new clients until it has processed enough data to reach a consistent state.

On failure of a node in the cluster it is the responsibility of the application layer to direct all CQs and SQs to other nodes. Furthermore, after the failed node is brought back online it needs to capture all the data that it missed while being non-functional. This is also accomplished by the replicator component using a simple protocol that tracks the runs that have and have not been replicated.

5.7 Bootstrapping and Nostalgia

When a live CQ is added to a streaming system on an ad hoc basis, it is easiest for the query to only see data that arrives in the system *after* the query is submitted. This easy approach is quite reasonable if the query in question involves small windows (e.g., a window over the last 5 seconds that advances every 5 seconds). In many situations, however, this is not the case (e.g., a sliding window over the last 24 hours that advances every minute) and a naïve streaming implementation will not produce complete results until steady state (24 hours in the example) is reached.

What is required from the system is a feature that we call "bootstrapping", which is the ability to exploit any available archives of the underlying raw or derived stream that the CQ is based on by reaching into the archive and replaying history in order to build up the runtime state of the query and start producing complete results as early as possible.

In addition, if a new CQ has an associated archive there is often a need for the system to populate this archive with all the data already in the system prior to the query's addition. We call this feature "nostalgia" and when specified we asynchronously catch-up the associated SP archives in a manner similar to how repair enforces order-independence.

In this section we provided an overview of the Truviso approach to Continuous Analytics over Discontinous Streams. The bedrock of our system is the ability to offer real uncompromising order-independence. This provides not just correctness but the building blocks for realizing a simple and easy to verify distributed configuration offering scalability, availability and fault-tolerance. In the next sections we present more details of the various components sketched out above.

6. CONCLUSION

Research and early commercial work in stream query processing viewed data streams as *continuous* sequences of data that arrived more or less in order. This assumption heavily influenced the architecture of first generation stream-processing systems. The reality, however, is that streams are not often so well-behaved and disruptions of various sorts are endemic. In this paper we described the solution taken to this problem in the Truviso Continuous Analytics System. The approach is based on a simple insight – queries can be performed over independent *sub-streams* with arbitrary time ranges in parallel, generating partial results. The consolidation of the partial results over each sub-stream can then be deferred to the time at which the results are actually used on an on-demand basis. We refer to this style of processing as *order-independent* processing. We explained the approach and how it leverages properties of the SQL language and relational query processing to be the first real solution to the problem of processing streaming data that arrives arbitrarily late. Perhaps more significantly, we described how it also serves as a critical building block for solutions to a host of hard problems in stream processing such as parallelism, recovery, transactional consistency, high availability, failover, and replication.

7. REFERENCES

[1] Abadi, D., Carney, D., and Cetintemel, U., et al. Aurora: A Data Stream Management System. In Proc. CIDR 2003.

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

[3] Arasu, A., Babcock, B., Babu, S., et al. STREAM: The Stanford Data Stream Management System. In Data Stream Management: Processing High-Speed Data Streams, Springer, 2009.

[4] Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M. Fault-tolerance in the Borealis Distributed Stream Processing System. In SIGMOD 2005.

[5] Bancilhon, F., et al. FAD, a powerful and simple database language. In VLDB 1987.

[6] Chandrasekaran, S., Cooper, O., Deshpande, A., et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertan World. In Proc. CIDR 2003.

[7] Conway, N. Transactions and Data Stream Processing. http://neilconway.org/docs/thesis.pdf. April 2008.

[8] Dean, J., Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.

[9] DeWitt, D., Gray, J. Parallel Database Systems: The Future of High Performance Database Systems. CACM 35(6) 1992.

[10] Franklin, M., Krishnamurthy, S., et al. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In CIDR 2009.

[11] Gray, J., Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993, ISBN 1-55860-190-2.

[12] Gray, J., et al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab and Sub-Total. In ICDE 1996.

[13] Li, J., Tufte, K., Shkapenyuk, V., Papdimos, V., Johnson, T., Maier, D.. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. In Proc. VLDB Endowment (2008), 274-288.

[14] Mohan, C., Haderle, D., et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS 17(1): 94-162 (1992).

[15] Motwani, R., Widom, J., Arasu, A., et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In Proc. CIDR 2003.

[16] Shah, M., Hellerstein, J., Brewer, E. Fault-Tolerant Parallel Dataflows. In SIGMOD 2004.

[17] Srivastava, U., Widom, J. Flexible Time Management in Data Stream Systems. In PODS 2004.

[18] Tucker, P., Maier, D. Exploiting Punctuation Semantics in Data Streams. In ICDE 2002.

[19] Omniture Web Services API. https://sc.omniture.com/p/l10n/1.0/en_US/docs/WebServices_API_14_Implementation_Manual.pdf