TimeStream: Reliable Stream Computation in the Cloud

Zhengping Qian[1] Yong He[1,2] Chunzhi Su[1,3] Zhuojie Wu[1,3] Hongyu Zhu[1,3]
Taizhi Zhang[1,4] Lidong Zhou[1] Yuan Yu[5] Zheng Zhang[1]

[1] Microsoft Research Asia, [2] South China University of Technology, [3] Shanghai Jiaotong University,
[4] Peking University, [5] Microsoft Research Silicon Valley

Abstract

TimeStream is a distributed system designed specifically for low-latency *continuous* processing of *big streaming* data on a large cluster of commodity machines. The unique characteristics of this emerging application domain have led to a significantly different design from the popular MapReduce-style batch data processing. In particular, we advocate a powerful new abstraction called *resilient substitution* that caters to the specific needs in this new computation model to handle failure recovery and dynamic reconfiguration in response to load changes. Several real-world applications running on our prototype have been shown to scale robustly with low latency while at the same time maintaining the simple and concise declarative programming model. TimeStream handles an on-line advertising aggregation pipeline at a rate of 700,000 URLs per second with a 2-second delay, while performing sentiment analysis of Twitter data at a peak rate close to 10,000 tweets per second, with approximately 2-second delay.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—*Distributed programming*

General Terms Design, Performance, Reliability

Keywords Distributed Stream Processing, StreamInsight, Cluster Computing, Real-time, Fault-tolerance, Dynamic Reconfiguration, Resilient Substitution

1. Introduction

We are witnessing the rise of a new type of data driven by rapidly expanding coverage of sensors, growing use of the mobile Internet, and increasing popularity of social media, such as Twitter and Facebook. Together with the availability of distributed-system infrastructure running on large clusters of commodity machines, this new type of data has further fostered a new class of applications for low-latency analytics on *big streaming* data, with the following defining characteristics: (i) incoming data arrives continuously at volumes that far exceed the capabilities of individual machines; (ii) *input streams* incur multi-staged processing at low latency to produce *output streams*, where any incoming data entry is ideally reflected in the newly generated results in output streams within seconds.

The new class of applications for real-time streaming analytics represents a significant departure from the traditional batch-oriented MapReduce [10]-style (big) data processing and requires a different distributed-system infrastructure. A MapReduce job runs on static input data and performs the computation once, whereas our computation has to run continuously as new data enters the system. When running on a large cluster of commodity machines, fault tolerance is of paramount importance, especially for continuously running computation. Recomputation-based failure recovery in MapReduce does not apply to stream computation because reloading data from the start of the computation is infeasible in practice. In addition, unlike one-time batch processing, long-running stream computation must cope with temporal dynamics related to load changes, variations in load distribution and balance, as well as failure and recovery.

Streaming data have been the subject of extensive research in the database community, where the modeling of both data and computation has been carefully studied. Clearly, we are now confronted with challenges at a significantly larger scale and with often abundant computation and storage resources in the cloud to leverage. Those seemingly quantitative differences have led to potentially qualitative advances in underlying mechanisms for big streaming data processing on a large commodity cluster, especially in terms of how to achieve reliable stream computation.

Stream computation is defined to be *reliable* if, despite dynamic reconfigurations in response to load fluctuations and failure recovery, the computation produces the same output streams as in a fail-free run with no reconfigurations,

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

where each incoming data item in an input stream is processed *exactly once*. Such strong semantics are clearly ideal if achievable at a reasonable cost; they might be required by applications such as an accounting pipeline for on-line advertising, where the exactly-once semantics are needed for correctness. Streaming database systems tend to use replication for reliability [7, 8, 17, 24, 26]; such a solution is often costly and relies on strong assumptions on how many failures can occur concurrently.

In this paper, we present a system, TimeStream, that takes on the challenge of building a distributed-system infrastructure for reliable low-latency continuous processing of big streaming data. TimeStream manages to combine the best of both MapReduce-style batch processing and streaming database systems in one carefully designed coherent framework, while at the same time it offers a powerful abstraction called *resilient substitution* that serves as a uniform foundation for handling failure recovery and dynamic reconfiguration correctly and efficiently.

In particular, TimeStream makes the following technical contributions. First, TimeStream develops a mechanism that tracks fine-grained *data dependencies* between the output and input streams to enable efficient recomputation-based failure recovery that achieves strong exactly-once semantics. Those dependencies are at the core of the new abstraction of resilient substitution. Second, TimeStream adopts the programming model of StreamInsight [6] for complex event processing (CEP), and extends it to large-scale distributed execution by providing automatic supports for parallel execution, fault tolerance, and dynamic reconfiguration. TimeStream is able to generate and track data dependencies automatically at runtime from the declarative operators provided by the language to enable reliable stream computation. Third, we have built several real-world applications on TimeStream and evaluated its ability to maintain a simple and concise programming model and to scale in a robust manner while providing real-time performance. For example, our system handles sentiment analysis of Twitter data at a peak rate close to 10,000 tweets per second, with approximately 2-second delay. It maintains this real-time characteristics with a moderate 20% resource over-provisioning, under single failure and multiple concurrent and correlated failures. For a real-time monitoring service, TimeStream scales elastically with load swings, without compromising the correctness of the computation during the reconfiguration.

In what follows, we begin with a discussion of the related work in Section 2, followed by an introduction of the programming model in Section 3. Sections 4 and 5 describe the design and implementation of TimeStream, respectively. Section 6 describes example applications and Section 7 covers detailed evaluation. We conclude in Section 8.

2. Related Work

In contrast to many other systems that deal with off-line batch processing, TimeStream focuses on large-scale low-latency stream computation, as with a number of recent systems including Storm [2], S4 [20], D-Streams [28], Puma [19] and Flume [14]. There is also a much broader context including the work from the more traditional streaming database community. Our contributions are multi-faceted, spanning across programming model and mechanisms for fault tolerance and dynamic reconfiguration. We discuss TimeStream's novelty in each of these areas.

Programming model. In TimeStream, as in other streaming systems, every new data entry triggers state change and possibly produces new results, while the computing logic, informally known as a *standing query*, may run indefinitely. Such a standing query can be translated into a query network. A number of existing systems such as Storm and S4 require that users wire the query network topology manually. Our philosophy is to hide such complexity using a declarative language, as seen in Trident [3], the recent extension to Storm, which has started to include some limited declarative features and APIs. TimeStream therefore preserves the elegant and well-formed data and computation model from the streaming database research in a declarative language that allows user-defined functions, much like DryadLINQ [27], PigLatin [21], and SCOPE [13] do for MapReduce-type computation. In addition, TimeStream uses *streaming DAG* (directed acyclic graph) as a lower-level computation model, further evolving the general DAG model in systems like Dryad. In a similar fashion as Dryad (or PIG) generalizes the more restrictive MapReduce (or Hadoop), TimeStream's streaming DAG enables more general computation than in Storm or S4.

More specifically, TimeStream leverages the StreamInsight API [6] and adopts the declarative style of LINQ [5]; the resulting code is concise and easy to understand. These building blocks enable us to program a variety of real-time applications, instead of simple aggregation logics as in Puma and Flume [14].

D-Streams takes a different approach treating a streaming computation as a series of deterministic batch computations on small time intervals, allowing the fault tolerance mechanism for batch processing to be reused naturally. However, D-Streams does force programmers to view stream processing as a series of batch jobs, which may be less expressive or natural for programmers. Moreover, in contrast to TimeStream's dependency tracking, D-Streams' computation dependency cannot be used to bound a recomputation if there are state dependencies between two mini-batch jobs: checkpointing is needed in such cases to avoid reloading from the start.

Fault tolerance. In the context of isolated deployments or when a single machine is sufficient to handle the load, using

hot standby technique such as process pairs (e.g., Flux [24], StreamBase [4], and StreamInsight [6]) is appropriate. Unfortunately, these options have high overhead and limited flexibility when the system needs to scale up.

The emerging big streaming data analytics in the cloud are driven by scenarios such as on-line service monitoring and real-time user-behavior analytics or advertising. In those settings, the latency requirement is somewhat lax, where a delay of seconds or tens of seconds is often acceptable. S4 achieves such kind of low-latency requirement by keeping computation in memory and uses data-partitioning for scalable stream computation. When a failure occurs, S4 simply restarts a computation with the potentially lost data/state, taking a best-effort approach. These weak semantics might be sufficient for some scenarios, but are often inadequate.

Compared to S4, Storm provides a stronger at-least-once semantics by tracking each (intermediate) output during computation back to the dependent input(s) with the help of functions that application programmers must provide. Once some output gets lost, it replays the dependent input(s) for recomputation. This approach guarantees that every input must be computed at least once despite failure. Trident for Storm supports transactional semantics; it currently supports only the simple operators and does not cover common streaming operators such as windowing and temporal joins. Storm does not enforce a deterministic order on how a computation consumes input entries, making it challenging to support strong semantics transparently at the system level; often programmers have to implement a set of APIs in order to support strong semantics.

D-Streams models a stream computation as a series of deterministic mini-batch jobs and leverages the MapReduce-style recomputation-based recovery for each mini-batch because each mini-batch captures the needed data dependencies between output and input entries. D-Streams imposes a programming model that forces programmers to think in terms of batches, rather than in a streaming model.

TimeStream differs from Storm and D-Streams by supporting precise failure-induced recovery with minimal recomputation. Unlike Storm, it enforces deterministic execution (when needed) that preserves a well-defined ordering of inputs to each computation, which makes it possible to do lightweight dependency tracking to provide the strong exactly-once semantics. The same mechanism also supports dynamic changes to the DAG for elasticity with the same strong semantics. TimeStream does so without changing the streaming computation model. This is in contrast to D-Streams, where programmers must think in terms of mini-batches. TimeStream also handles stateful computation naturally and can in many cases avoid checkpointing used to prevent cascading recomputation as needed in D-Streams. TimeStream further benefits from the use of a high-level declarative programming model from StreamInsight (with extensions) to hide the complexity of dependency tracking from programmers.

Tracking dependency is a powerful idea that has been exploited to compute what is absolutely necessary when there are limited changes in the input set (e.g., incremental computing as in Incoop [11], DryadInc [22], Nectar [15]), or frugal recomputation to repair lost state as in MadLINQ [23]. The streaming database community has proposed upstream backup [17] that deals with a single failure with two levels of acknowledgments.

Dynamic reconfiguration. In today's cloud environment, a system needs to cope with both resource and load fluctuations, as well as recovering from failures. Streaming systems must deliver results with low latency and are therefore more sensitive to such dynamism. In addition, an early study [26] shows that the cost of an individual operator may also change even when the event rate stays the same. Dynamic load management and balancing is a well-studied topic in the streaming database community (e.g., [26]). Though the latency requirements differ in our setting, many results on the policies to trigger reconfiguration can be applied. At the same time, it is important to have the capability of adapting the configuration on-demand without compromising computational integrity, which is the foundation to implement any appropriate reconfiguration policies.

Leveraging the efficient fault tolerance mechanism, a notable novelty in TimeStream is to enable robust elasticity. The system uses resilient substitution as a single unifying mechanism, allows runtime substitution of a portion of the computation DAG, and maintains data and state integrity despite changes. Such on-demand reconfiguration for scaling is absent in most distributed stream processing engines.

3. Programming Model

TimeStream is designed to faithfully preserve the programming model of StreamInsight [6], and as a result it can be used to scale out any existing StreamInsight applications to large compute clusters without any modification. In this section, we provide a high-level view of the programming model, highlighting the key concepts of the programming model including the data model and query language.

3.1 StreamInsight

StreamInsight is a powerful platform for developing complex event processing (CEP) applications, which is a Data Stream Management System (DSMS) based on the CEDR [9] research project. For CEP applications, data is represented as event streams, each describing a potentially infinite collection of events that changes over time. An event is the basic unit of data processed by a CEP engine. Each event consists of two parts: an event header and a payload. The event header defines the event kind and one or more timestamps that define the time interval for the event; the payload holds the actual application data associated with the event.

There are two kinds of events: INSERT and CTI. An INSERT event carries actual payload and can be in one of three shapes. An *interval* event represents an event that is valid for a given interval of time. The event header specifies the start and end time. A *point* event represents an event occurrence at a single point in time. An *edge* event specifies the occurrence of either the start or the end of an event. Both point and edge events can be regarded as special cases of interval events.

A CTI event is a special punctuation event that asserts the completeness of the event history before a given timestamp. It enables the processing of out-of-order events. TimeStream relies on CTI events to determine data granularity at runtime, as will be elaborated in Section 4.

StreamInsight queries are written in LINQ with a .NET language such as C#. LINQ introduces a set of declarative operators into .NET languages to manipulate collections of .NET objects. In StreamInsight, the base type for an event collection is `CepStream⟨TPayload⟩`, where `TPayload` is the .NET type of the payload, and the query operators are defined to perform transformations on `CepStream` collections. StreamInsight supports a comprehensive set of relational operators including projection (`Select`), filters (`Where`), grouping (`GroupBy`), and joins (`Join`). Their semantics are slightly adapted to handle events. For example, the join condition is changed to include/process only events with overlapping time intervals.

Windowing is another key concept in stream processing. A time window is just a finite collection of events, to which aggregations such as `Count` and `Sum` can be applied. StreamInsight supports several types of windows. They enable applications to perform operations such as aggregations over subsets of events that fall within some period of time. We briefly describe the types of windows used in the paper. *Hopping* windows are windows that "jump" forward in time by a fixed size. The windows are controlled by two parameters: the hop size H and the window size S. A new window of size S is created for every H units of time. *Tumbling* windows are a special case of hopping windows with $H = S$, representing a sequence of gap-less and non-overlapping windows. Similarly, *Count* windows are sliding windows, each including a fixed number of events.

Example. Figure 1 shows an example of a simple program that continuously computes for each time window the frequency of all words that have appeared in an infinite stream of tweets. The input of the program is a continuous stream of events with payload of type `Tweet`. The source of the input could be some live on-line server that continuously publishes the new tweets it collects. The program first calls the user-defined operator `GetWords` to turn each incoming tweet into a sequence of word events. It then applies the `GroupBy` operator to partition the words. The result of this grouping conceptually returns a set of `CepStreams`, one for each unique word denoted by `wordGroup` in the example. The words in each `wordGroup` are further transformed into time windows by the `TumblingWindow` operator, which are subsequently aggregated to compute the count of the events in the window.

```
// An input stream containing tweets
CepStream<Tweet> tweets = CepStream<Tweet>.Create(...);

// Counts word frequency in each time window
var counts = from w in tweets.GetWords()
                             .HashPartition(w => w, 3, uid)
          group w by w into wordGroup
          from win in wordGroup.TumblingWindow(winSize)
          select new { Word = wordGroup.Key,
                       Count = win.Count() }
```

Figure 1. Continuous word count that computes word frequencies in each time window with `HashPartition`.

3.2 TimeStream Extension

TimeStream preserves the StreamInsight/LINQ programming model and extends it to large-scale distributed execution by providing automatic support for parallel execution, fault tolerance, and dynamic reconfiguration.

TimeStream introduces a new re-partitioning operator `HashPartition⟨K,T⟩`, which is primarily used to re-partition a `CepStream` for parallel execution. It also gives the programmer more control to configure dynamically a continuously running application based on runtime information. For example, to enable parallel execution of continuous word count, the programmer simply adds a `HashPartition` to the word stream as shown in Figure 1.

The `HashPartition` breaks the word stream into three partitions so that the query can be executed for example on three machines in parallel. The `uid` argument of `HashPartition` is a unique identifier for this partitioning operation. TimeStream can use it to reconfigure dynamically the number of partitions at runtime. For example, a monitor of a continuously running query may identify a slow computation stage and improve the performance by increasing the parallelism dynamically with the identifier and a new partition count.

4. TimeStream Design

TimeStream supports stream applications as continuous queries that compute over a potentially infinite sequence of input entries and produce output entries continuously. Unlike MapReduce-style computation, where a batch job typically involves a single multi-stage execution, TimeStream computation is continuous and long running, making it critical to handle runtime dynamics, such as load fluctuations and failures, and demanding different mechanisms from those used in batch processing. This section presents the system model, as well as the key concepts, abstractions, and mechanisms in the design of TimeStream to enable efficient and correct failure recovery and dynamic reconfiguration.

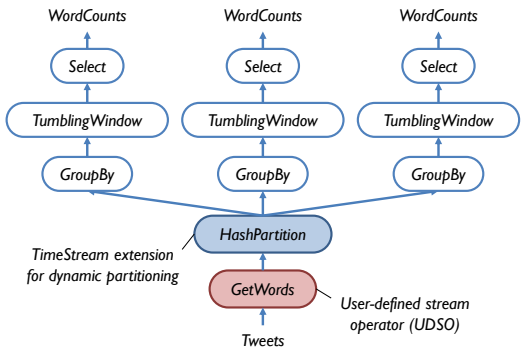

Figure 2. Streaming DAG for the continuous word count query in Figure 1.

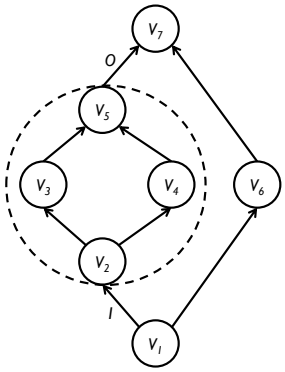

Figure 3. Streaming DAG and sub-DAG example: the circled subgraph is a sub-DAG with input stream I and output stream O.

4.1 Streaming DAG

TimeStream compiles a continuous query into a *streaming DAG* that can be mapped to physical machines for execution and dynamically reconfigured at runtime. Each *vertex* v in a streaming DAG takes a set of input streams, maintains an internal *state* (optionally), and implements a *streaming function* f_v. The computation in vertex v is triggered by an arriving entry i in an input stream, which updates v's state from τ to τ', and produces a sequence o of output entries as part of the output streams for downstream vertices, denoted as $(\tau', o) = f_v(\tau, i)$.

We map each stream operator in a StreamInsight query onto such a vertex. For `HashPartition`, the input stream is partitioned into multiple output streams and the segment of operators that follows are replicated for each of the partitions. To preserve the operator semantics and achieve reliability, the INSERT and CTI events (even from multiple inputs) to an operator have to be consumed in a deterministic order. We pack a segment of consecutive INSERT events with a CTI that follows into a single entry in a stream; such an entry constitutes the finest data granularity for transfer and computation. Each vertex then reorders the entries (when needed) according to the corresponding CTI timestamps before consumption. Both event packing and reordering are transparent to the operator. Figure 2 shows the streaming DAG corresponding to the query in Figure 1.

We assume that the computation f_v in each vertex v is *deterministic* in that the current state and the input entry decide the output and the state transition deterministically. We introduce *source vertices* as special input adapters that generate output from various sources (e.g., sensor readings) continuously to drive the computation of the rest of the DAG. We use *result vertices* to denote those that generate the output streams for the computation to be consumed elsewhere. The input streams to the source vertices and the output streams from the result vertices are assumed to be stored persistently and reliably, as they are the original input and final output of the computation.

A DAG itself has the same logical structure and function as a vertex. We further define a subgraph consisting of a subset of the vertices with the edges in between to be a valid *sub-DAG* if the following condition holds: for any vertices v_1 and v_2 in the sub-DAG and any v in the DAG, v must also be in the sub-DAG if v is on a directed path from v_1 to v_2. A sub-DAG is logically equivalent and can be reduced to one vertex. Figure 3 shows an example of a streaming DAG consisting of seven vertices, with v_1 being a source vertex and v_7 being a result vertex. The subgraph comprised of v_2, v_3, v_4, and v_5 (as well as all their edges) is a valid sub-DAG and can be reduced to a "vertex" with I as its input stream and O as its output stream. It may seem counter-intuitive, but the subgraph comprised of v_3, v_4, and v_6 is also a valid sub-DAG. A subgraph with only v_2, v_3, and v_5 is however *not* a valid sub-DAG because v_3 is on the path from v_2 to v_5, but not part of the subgraph; reducing that subgraph to a single logical vertex would create a graph with cycle, not a DAG.

4.2 Resilient Substitution and Dependency Tracking

TimeStream supports dynamic reconfiguration at runtime in response to server failures and load fluctuations through *resilient substitution* that can be applied to any vertex or sub-DAG. When a vertex in the DAG fails (e.g., due to the failure of the underlying machine), a new vertex is initiated to replace the failed one and continues execution, possibly on a different machine. TimeStream can also apply resilient substitution to a sub-DAG in order to adjust the number of partitions in a computation stage based on the incoming rate of the input streams. For example, in Figure 4, the sub-DAG comprised of vertices v_2, v_3 , v_4, and v_5 implements the three stages: hash partitioning, computation, and union. When the load increases, TimeStream can create a new sub-DAG (shown on the left), which uses 4 partitions instead of 2, to replace the original sub-DAG. Similarly, TimeStream can also replace that sub-DAG with a single vertex if the load is light through resilient substitution. The current implementation of TimeStream provides such dynamic recon-

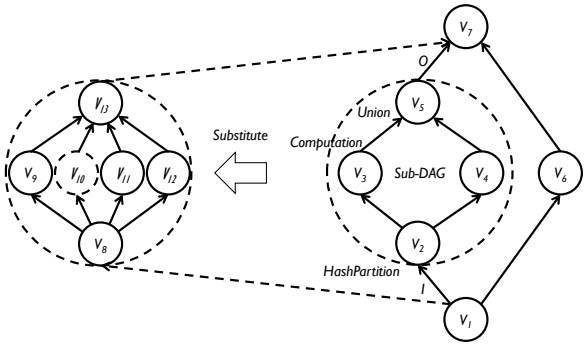

Figure 4. Resilient substitution example: the circled sub-DAG on the right can be substituted by the one on the left.

figuration. The policy by which such reconfiguration events are triggered is specified by applications.

To ensure resilience to faults, TimeStream requires that any modification made to the topology due to reconfiguration or maintenance does not affect the output data entries produced by the system. To achieve this goal, resilient substitution mandates that a new sub-DAG is *equivalent* to the one being replaced, where the notion of sub-DAG equivalence is defined as follows. Given two sub-DAGs, they are equivalent if and only if they compute the same function, i.e., given the same input stream(s), the two sub-DAGs will always produce the same output stream(s).

State and output dependencies. Implementing resilient substitution to be simultaneously efficient and correct is a challenging task. For example, when a vertex fails, its state may be lost and need to be reconstructed. While it is always possible to restart the entire computation from the beginning of the computation for correctness, this is clearly infeasible in practice. The difficulty lies in recovering the states and the missing entries correctly with a minimal amount of recomputation, while allowing dynamic reconfigurations during computation. Resilient substitution in TimeStream does so by tracking *state and output dependencies* for each vertex of a stream computation.

Given a vertex v that computes over a set of input streams, at any state τ, we define its *state dependency* (denoted as $dep_s(v, \tau)$) to be a subsequence of the input data entries, such that the vertex reaches the same state after consuming this subsequence of input data entries. For each output entry e that the vertex emits at state τ in response to a newly arrived input entry i, we define its *output dependency* $dep_o(v, \tau, e)$ to be $dep_s(v, \tau) \circ \{i\}$. To record the state and output dependencies, TimeStream labels data entries in an input or output stream using sequence numbers to identify them uniquely. State and output dependencies can be defined and constructed for any sub-DAG.

Figure 5 shows the dependency structures in four different streaming-computation examples. The first case is a stateless computation, where each entry in the output stream depends on the corresponding entry in the input stream; e.g., $dep_o(v, \_, o_3) = \langle i_3 \rangle$. The second case uses a tumbling window of size 3. We therefore have $dep_s(v, \tau_1) = \langle i_4, i_5 \rangle$, $dep_o(v, \tau_1, o_2) = \langle i_4, i_5, i_6 \rangle$, and $dep_s(v, \tau_2) = \langle i_5, i_6 \rangle$. The dependency structure has a clear pattern that can be inferred from the operator itself. The third case involves a join of two streams and its dependency structure depends on the values of the entries in the input streams. For example, we have $dep_s(v, \tau) = \langle a_2 \rangle$ and $dep_o(v, \tau, o_2) = \langle a_2, b_3 \rangle$. In the last case, a user-defined stream operator applies exponential smoothing to a stream of events. At any time, the current state depends on all consumed data entries in the input stream. Therefore, we have $dep_s(v, \tau) = \langle i_1, i_2, i_3, i_4 \rangle$, $dep_o(v, \tau, o_5) = \langle i_1, i_2, i_3, i_4, i_5 \rangle$. TimeStream is able to restore the state from the data entries in its state dependency. which requires only a subset of the input data entries in all the cases except the last one.

Recovery using dependencies. With state and output dependencies, TimeStream recovers output data entries from a vertex by initiating a *recovery task* on a given vertex, with the labels of the output data entries to be recovered. To recover output entry labeled o generated at state τ in response to entry i, the recovery task retrieves its output dependency and the corresponding state dependency, and recursively initiates a new recovery task for any input data entries that are needed, but not available, on the upstream vertex to supply these entries. When these data entries become available, this recovery task clones the vertex at its initial state and feeds this new vertex all these data entries to reach state τ and then produces output o. In practice, TimeStream recovers a set of data entries together, rather than for each individual one. Given that each stream computation is deterministic, it is easy to prove that the recovery task can produce the same output o based on the definition of state and output dependencies. For example, to recover output o_2 from v_5 in Figure 6, a recovery task can be initiated on v_5. According to the dependencies indicated by the directed dotted lines, this might further invoke the recovery task for $\langle m_1 \rangle$ on vertex v_4, which in turn might invoke the recovery task for $\langle k_1, k_2, k_3 \rangle$ on vertex v_2. This final recovery task can be completed by applying the streaming function of v_2 on input sequence $\langle i_1, i_3, i_4 \rangle$, as captured in the dependencies.

A recovery task can be initiated on any sub-DAG to recover output data entries for that sub-DAG by using the computed state and output dependencies for the sub-DAG. A recovery task on a sub-DAG becomes necessary after a substitution on a sub-DAG, as shown in the example of Figure 4. If v_7 fails, its recovery would request data entries from O, which cannot be recovered directly on vertex v_{13} in the new sub-DAG due to the lack of state and output dependency information: those data entries were computed in the original sub-DAG. In order to recover output data entries in O from input data entries in I, TimeStream computes the state and output dependencies for the sub-DAG using the state and

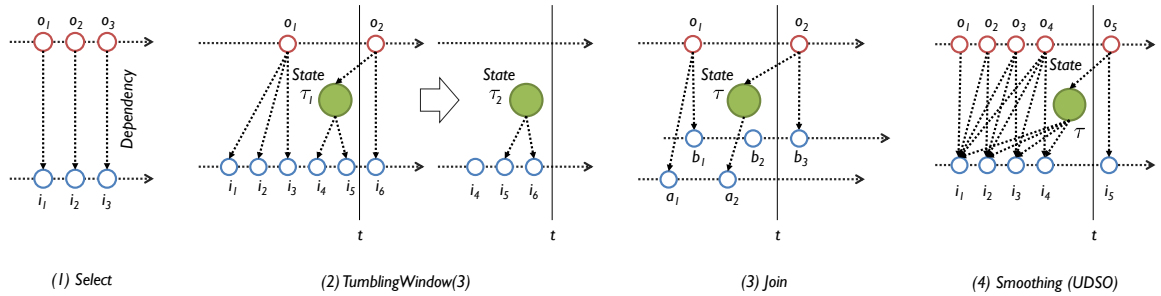

Figure 5. Dependency structure examples for common operators.

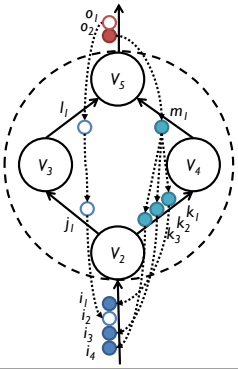

Figure 6. Dependency inference of a sub-DAG.

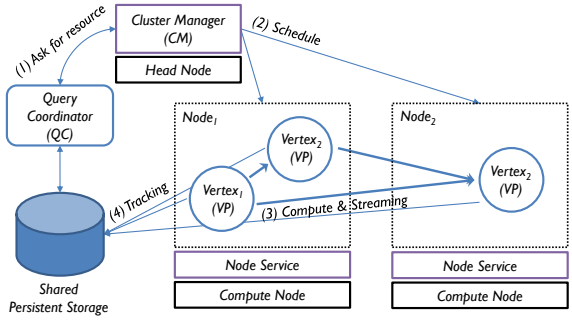

Figure 7. TimeStream distributed runtime overview.

output dependencies on the individual vertices, as illustrated in Figure 6. As a result, the output dependency of o_2 will be computed as $\langle i_1, i_3, i_4 \rangle$. To recover o_2, the new sub-DAG can simply take $\langle i_1, i_3, i_4 \rangle$. In fact, even with multiple substitutions, TimeStream can always find a sub-DAG with the computed state and output dependencies to recover data entries in an output stream.

Garbage collection using dependencies. Stream computation in TimeStream is continuous; garbage collection is used to remove information that is no longer needed, including the reliably stored source input data entries and the tracked dependency information for the execution. Garbage collection involves figuring out what information is no longer needed after some final output of the computation is stored reliably and will not be requested again. TimeStream again leverages the dependency information to do so. At any point, TimeStream can compute for each vertex in a reverse topological order the input data entries that are needed in order to recover any needed output entries that have been generated. Because result output entries are assumed to be reliably stored after they are produced, no output entries are needed for recovery from the result vertices. Given a vertex with a sequence O of output data entries needed from its downstream vertices—those data entries have been computed because of the reverse topological order of processing—TimeStream computes the input data entries that the vertex needs from its upstream vertices as the union of input data entries in the output dependency of each $o \in O$, with those in the state dependency of the current state. The process continues until it reaches the source streams, in which case all unneeded data entries in the source streams can be safely garbage collected. Any dependency for unneeded entries or state can also be garbage collected. For example, in Figure 6, when o_1 is no longer needed, l_1, j_1, and i_2 are unneeded, so are their dependencies. If o_2 is needed, then all its dependency and the source data entries $\langle i_1, i_3, i_4 \rangle$ remain needed.

5. Implementation

We have implemented TimeStream using C#/.NET. The prototype consists of a client library (7.2 KLoC) that provides a compatible query language/compiler for writing StreamInsight applications with TimeStream extensions, and a distributed runtime (8.2 KLoC) that executes a streaming DAG and supports dynamic reconfiguration.

Distributed runtime. The TimeStream runtime is built on top of a cluster service we developed for resource allocation, vertex scheduling, and failure detection. The cluster service consists of a cluster manager (CM) running on a *head* node and a node service (NS) running on each *compute* node in

the cluster. Figure 7 shows the flow of execution when a job enters the distributed runtime. A query coordinator (QC) is created for each job running on the cluster. The QC first talks to the CM for allocating resources to run the job DAG (Step 1) and then schedules the vertex processes (VP) through the cluster service onto the compute nodes (Step 2). We maintain the meta-data information of the QC in a reliable storage so that its fail-over is straightforward. We use standard Paxos-like approaches to ensure that there is a single QC at any time. In the current implementation, we allow each node to host a fixed number of VPs. Each VP may contain one or more tasks specified by the QC to carry out the stream computation and produce output to send to the downstream tasks running within other VPs (Step 3). During the execution, VPs track the progress and dependencies, and periodically write them to the same reliable storage (Step 4); the QC uses this information to manage any failure recovery and reconfiguration, as well as to coordinate garbage collection.

DAG generation and dependency extraction. When a query is executed, the client library compiles it into a streaming DAG and submits it to the distributed runtime. Dependency tracking logic is embedded by the compiler when generating the vertex code. TimeStream automatically tracks fine-grained dependency for all the built-in operators including projection, filters, windowing, and temporal joins. For stateless computations like projection and filters, dependency tracking is simply done by propagating the input labels to output. For stateful computations like windowing and temporal joins, we have modified the standard operators to expose dependent input labels necessary for reconstructing the current state. TimeStream also allows users to introduce new operators, but requires that users expose dependencies for their operators.

Dependency tracking and maintenance. During the execution, each vertex tracks the meta-data information of dependencies and progress, represented in a compact form using range sets of entry labels, and saves them to the reliable storage periodically. The dependencies are kept in a key-value store with the keys corresponding to the output labels of a vertex. Persisting the dependencies asynchronously reduces the overhead by removing part of the tracking logic out of the critical path of the computation. Such asynchrony could lead to the QC having a (slightly) stale view of the execution. This does not lead to any correctness problems, even though the QC could under-estimate the current progress with a stale view and trigger more recomputation than necessary during failure recovery/reconfiguration. Any duplicate output produced in such a case will later be removed by downstream vertices.

5.1 Optimizations

We highlight some of the important optimizations we have implemented.

Operator fusion. To reduce unnecessary I/O, we map a segment of the stream operators in a query onto a same vertex in the DAG. For example, a `Where` operator can be merged with a preceding `Select` operator; a `Window` operator can be merged with the subsequent aggregator. In the current implementation, such fusion of operators is statically determined by the type of operators. When fusing two or more operators together, the compiler automatically propagates and merges the dependencies accordingly.

Dependency batching. The dependencies exposed by the operators are at the granularity of an event segment including only one CTI. Such fine granularity, albeit configurable, might incur too much overhead in cases where CTI events appear frequently for low latency. To address this, TimeStream automatically batches the dependencies for a consecutive segment of vertex output within a fixed time period and merges them into one coarser-grained dependency. Such batching effectively reduces the amount of dependencies at the (often acceptable) cost of extra (but bounded) recomputation during failure recovery/reconfiguration.

Output buffering. TimeStream also supports output buffering on each vertex as a "caching" mechanism. A vertex keeps a buffer of recently generated data entries of its output streams, so that they do not need to be recomputed when requested by the downstream vertices.

State checkpointing. For vertices where the state dependency always includes all the input data entries, as in the last case of Figure 5, TimeStream supports checkpointing to avoid replaying a stream computation always from the start. A checkpoint can be done asynchronously on an individual vertex, which simply records the current state and the positions of the input streams that have been consumed up to this point. TimeStream can always restore the state of the vertex from a checkpoint and continue from there. Consider, for example, the resilient substitution on a sub-DAG for repartitioning a computation stage as shown in Figure 4. Here, checkpoints for the vertices being replaced can no longer be used unless the application knows the semantics of the substitution and is able to derive checkpoints for the new vertices from those for the old ones.

6. Applications

Complex event processing has a wide range of applications including areas such as web analytics, financial trading, and service monitoring. This section illustrates the power of TimeStream with several real-world applications. While they are developed for different domains, their logics all share common patterns. The continuous events are cut into a train of windows, and within each window the events are grouped according to domain-specific correlations. The order of windowing and grouping is typically interchangeable. Following that, analytic logics are applied to the grouped events in the window. The real-time conditions refer to

the ability to output result immediately after the window is closed.

Network Monitoring. Network monitoring is a classic application of stream processing, and the goal is to inspect network latency as seen from the machines in a data center. The complete query is shown below:

```
from c in input.HashPartition(c => c.FromCluster, 40)
group c by new { c.FromMachine, c.ToMachine } into ctmp
from w in ctmp.TumblingWindow(10000)
select new QueryResult() {
   FromMachine = ctmp.Key.FromMachine,
   ToMachine = ctmp.Key.ToMachine,
   Latency = w.Average(a => a.Latency) }
```

The input is a continuous stream of measurements from the end machines. The payload of an event contains the latency of a data transfer from `FromMachine` to `ToMachine`. The query groups the latency events by each pair of machines. For each pair, it computes the average latency in every tumbling window of 10 seconds.

Audience Insight. This application is designed to provide a real-time view of the quality of a cloud-based video delivery service. The input is a continuous stream of monitoring events, collected from media players at the clients.

```
from e in inputStream.HashPartition(v => Key(v), 8)
group e by Key(e) into videoGroup
from w in videoGroup.TumblingWindow(10000)
select ComputeQualityStatistics(videoGroup, w)
```

Similar to the network monitoring application, we first group the input events by a user-defined key function `Key`, and compute the quality of each group for every tumbling window of 10 seconds. The function `Key` could be any interesting attributes of the input events, such as the video program, the Geo-location of the client, and the type of the media player, depending on signals included in the event payload.

TopK and Distinct Count. The queries described earlier can be seen as a simple pre-processing. Following that initial step, we will typically want to perform more advanced filtering, as demonstrated by the TopK and Distinct Count. These two queries take the observations of different objects in a moving time window, and report two different statistics in real time: the top K most frequent and the total number of distinctive objects, respectively. In the following example we use URL as the object of the query. The input is a continuous stream of URLs, each of them is attached with a monolithically increasing timestamp. The URL may correspond to search engine clicks, web site page views, or tweets that contain it.

```
// Compute top URLs:
from win in input.HashPartition(u => u, 16)
            .Where(u => !IsBot(u))
            .HoppingWindow(30000, 2000)
            .Select(w => w.TopK(100))
            .Union().Scan(new MergeSortOperator(100))
// Compute # unique URLs:
input.HashPartition(u => u, 16)
    .Where(u => !IsBot(u))
    .HoppingWindow(30000, 2000)
    .Select(w => w.DistinctCount())
    .Union().Scan(new SumOperator())
```

The query first partitions the input stream into 16 sub-streams, each containing a disjoint subset of the URLs. Pattern matching is applied to each sub-stream to detect and remove bot-generated queries, followed by a window operator `HoppingWindow`. `HoppingWindow`, also known as sliding window, is similar to `TumblingWindow` except that the two adjacent windows may overlap with a distance given by the second parameter. For example, in the previous query, a new window is created every 2 seconds and contains events within the last 30 seconds. A user-defined operator `TopK` computes the top 100 URLs of each window, followed by a union on the sub-streams of partial results, and ended by a user-defined operator that performs the final merge sort.

Distinct Count is similar, except that it applies a user-defined aggregator to compute the distinct count in each window instead of sorting. The last operator simply aggregates distinct counts from different partitions.

Sentiment Analysis of Tweets. This query explores a different domain, and the logic is much more complex: monitor a continuous stream of incoming tweets, and detect social opinion swings over certain user-dictated topics, at a specified temporal granularity (i.e., `wSize`). For changes that are deemed significant, the query reports the change of hot-words that may suggest the cause of change.

After grouping the tweets by topics, we get to the heart of the computation:

```
// Compute sentiment changes:
var scores = from w in topicTweets.TumblingWindow(wSize)
             select w.Average(t => Sentiment(t));
var change = from ss in scores.CountWindow(2)
             where ss.IsChanged()
             select ww.Events.Last();
// Compute top word changes:
var words = from t in topicTweets.TumblingWindow(wSize)
            select Aggregate(WordCount(t));
var delta = from ww in words.CountWindow(2)
            select Delta(ww);
// Relate sentiment changes to word changes:
from c in change
from r in delta
select ChangeWithReason(c, r)
```

The query computes the sentiment changes (`change`) and top word changes (`delta`), and then join them to analyze the possible reasons of sentiment changes. To compute `change`, we first compute the average sentiment of the tweets in every tumbling windows of `wSize`, and then detect if the sentiment is changed for every sliding window of size 2. The computation of `delta` is similar.

7. Evaluation

In this section, we evaluate TimeStream's capability of delivering reliable low-latency computation for the real-world applications described in Section 6. We pay special attention to scalability, fault tolerance, consistency, and adaptability

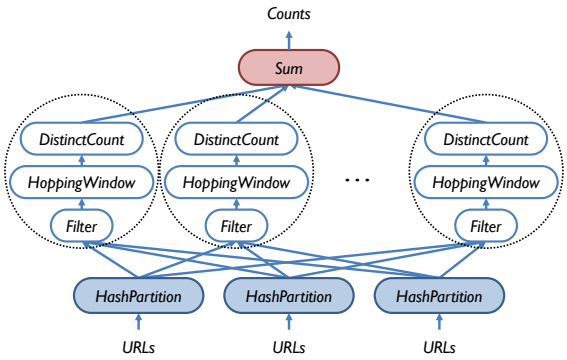

Figure 8. Streaming DAG for the Distinct Count query.

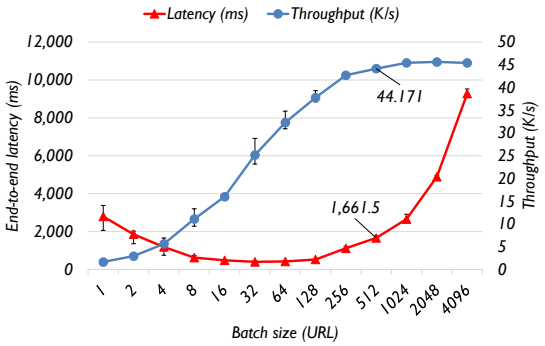

Figure 9. Latency and throughput trade-offs with different batch sizes for the Distinct Count query. At batch size 512 URLs, the throughput reaches 44.171K/s with a latency of 1,661.5ms.

to load dynamics. We further report the overhead introduced by dependency tracking for achieving strong semantics.

We perform the experiments on a computer cluster running Windows Server 2008 R2. Each server has dual Intel Xeon X3360 2.83GHz CPUs (a total of 4 cores), 8GB memory, and two 1.0TB SATA disks. All servers are interconnected with 1Gigabit Ethernet switches.

7.1 Scalability

Distinct Count. We evaluate Distinct Count using a URL stream generated according to the statistics based on sample logs from the Bing search engine as the input dataset. The log contains 30,766,355 distinct URLs, each associated with a frequency. The average URL length is 57 characters. Figure 8 shows the streaming DAG. The source streams are partitioned on 4 nodes, which stream the data to the compute nodes based on their hash code.

For this query, scaling is achieved by changing the number of compute nodes in the middle tier. Each compute node first matches the URLs against a list of 100 patterns of domain names to filter out likely bot queries and then performs distinct count in a train of 30-second sliding windows with a 2-second sliding step. These three steps are all fused into one physical vertex, which is the bottleneck of the pipeline. The final stage performs a summation and is never the bottleneck. The application is a simplification of a real-time counting pipeline for on-line advertising.

We implement the same benchmark in TimeStream and Storm[1], and compare the two systems using the same dataset. Both systems expose an event-based interface for expressing the logic. For execution, TimeStream packs the events into batches according to the special punctuation events (i.e., CTIs), which are then used as the unit of computation, data transfer, and dependency tracking. Storm does batched data transfer, but uses events as the granularity for the rest[2]. To make the comparison fair, we measure the throughput of Storm (using the default configuration) and allows an end-to-end latency of up to 2 seconds (a reasonable value for many scenarios). We then tune the batch size in TimeStream to match that latency.

We use a 6-node configuration (including 4 data sources, 1 middle-tier compute node, and 1 final merge node) for the experiments. Storm can handle 29,824 and 41,374 URLs per second with fault-tolerance (i.e., ACK) on and off, respectively. We try both cases and pick the maximum throughput values for the comparison. Figure 9 shows the latency-throughput correlation in TimeStream by varying batch sizes. Normally, we would expect to see the latency go up when the batch size increases because of batching. However, we notice that the latency also goes up when the batch size decreases to small values (1-8 URLs). This is primarily due to the high overhead of enforcing deterministic execution and tracking when the batch size is that small. With a batch size of 512 URLs, the latency remains below 2 seconds. We use this as the configuration for comparison and highlight it in the figure. The throughput (44,171) is comparable to that of Storm with ACK off (41,373), which shows that, with a reasonable batch size, the overhead of deterministic execution and dependency tracking is acceptable and does not negatively affect system performance in any significant way. We use 512 URLs as the default batch size in the following experiments.

We then repeat the experiment with different numbers of compute nodes to test the scalability. Figure 10 shows the throughputs using different batch sizes. We also show the maximum throughputs from Storm. The corresponding latency is shown in Figure 11. We repeat the runs for 5 times to get the error bars for both throughput and latency. As shown in the figures, with a latency up to 2 seconds, TimeStream scales linearly and performs comparably to Storm.

[1] URLs may arrive out of order in Storm. In order to achieve the same semantics of windowing, the applications on Storm must buffer and reorder the input when necessary; this is not necessary when writing the applications on TimeStream because the compiler generates such logic automatically from the query.

[2] We program against Storm directly and have not used Trident [3] in our evaluation.

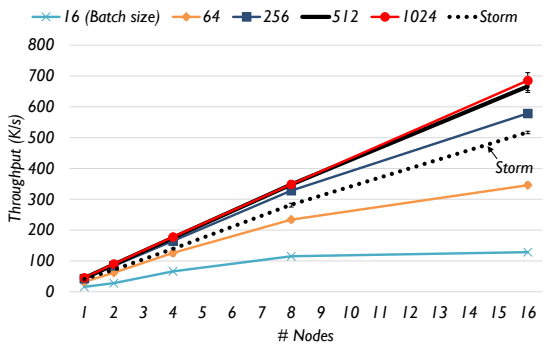

Figure 10. Scalability of Distinct Count using different batch sizes, with a comparison against Storm.

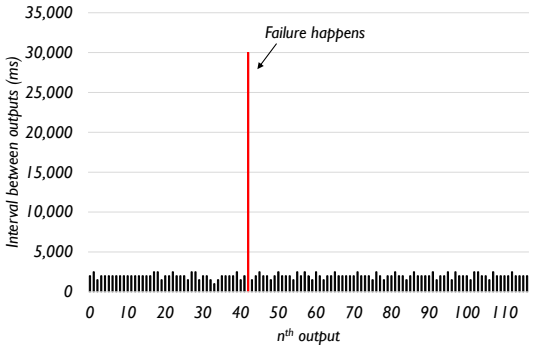

Figure 12. Failure recovery in a Distinct Count run.

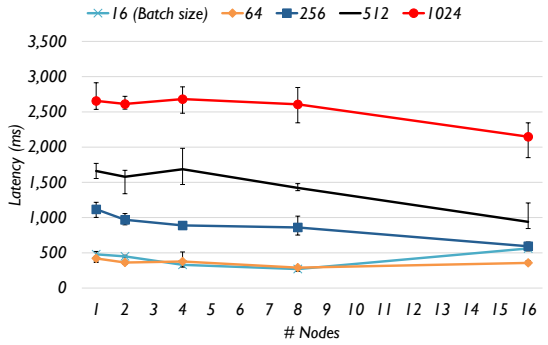

Figure 11. Latency of Distinct Count using different batch sizes.

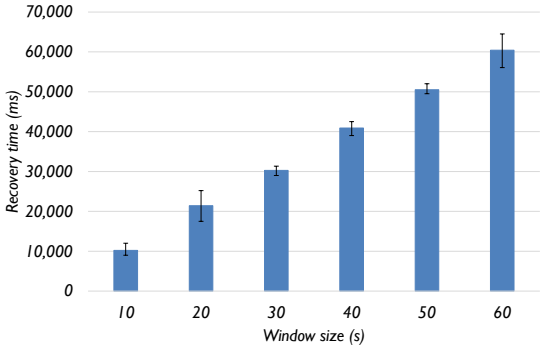

Figure 13. Failure recovery time with different window sizes for a Distinct Count query.

Sentiment Analysis of Tweets. We perform Sentiment Analysis on real Twitter data collected during August 2010. The trace contains a total of 1.2 billion tweets, with an average and a peak rate around 600 and 2,000 tweets/s, respectively. The application selects tweets according to specified topics, cuts the tweet stream into a train of non-overlapping windows, computes sentiment score of each window with a sentiment labeling engine, and monitors the sentiment-score change across two consecutive windows. In parallel, the query inspects the hot words of these windows as a clue of significant sentiment changes.

This is a highly complex query, using 6 different types of operators. Scaling this application is achieved by adding partitions to the sentiment-labeling stream. However, at a high rate, other operators can become overloaded as well (e.g., the operator that performs a cross-join of the tweet stream and the stationary topic stream).

In our experiment, we select 14 topics to monitor, ranging from IT industries to popular political figures. The event rate after topic filtering is about 2% of the original. A test-bed of 9 nodes can sustain the rate of 9.6K tweets/s with an average CPU utilization of 70%; this rate is above the publicly reported peak rate[3]. Scaling down the input rate sees a corresponding drop of cluster utilization: the utilization is 10%, 20%, and 36% for an input rate of 1.2K/s, 2.4K/s, and 4.8K/s, respectively. Note that, because we are feeding the trace at a much higher rate than the real posting rate, we need to reduce the window size as well. The window size we use is 6 minutes. Even at the peak rate of 9.4K/s, we are able to maintain a low latency at around 2 seconds.

[3] http://yearinreview.twitter.com/en/tps.html

7.2 Fault Tolerance

Distinct Count. We run Distinct Count on 10 nodes with the input rate controlled at around 20K/s, and inject a failure to one of the compute nodes in the middle tier. Figure 12 shows the intervals between the consecutive (windowed) outputs during such a run with a failure. Note that the delay of the next new output after the failure is roughly the size of the window (30 seconds). This is not a coincidence: the recovery time largely depends on the size of the state (i.e., the window size) in that the state needs to be reconstructed by recomputing the dependent set of input. We verify the correlation between recovery time and window size by repeating this experiment using different window sizes (while keeping the sliding step of 2 seconds). Figure 13 shows the result.

Dependency tracking in TimeStream makes it possible to recover from a state checkpointing whenever available. For fast recovery, checkpointing needs to happen frequently. Fortunately, checkpointing can be performed in the back-

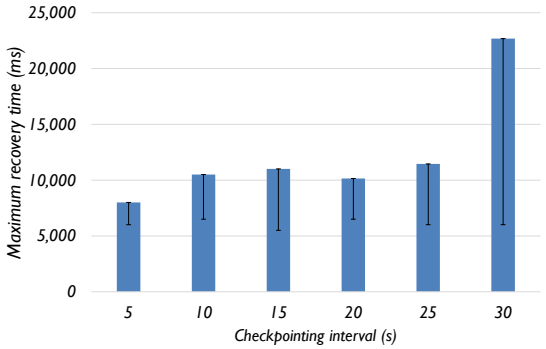

Figure 14. Maximum failure recovery time with different checkpointing intervals for a Distinct Count query. The black vertical lines show the range of the recovery time from 10 independent runs.

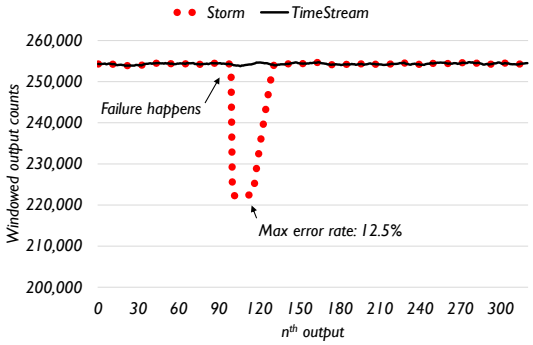

Figure 15. Distinct Count output with failure in TimeStream and Storm.

ground asynchronously to the main computation. Our measurement shows that on average checkpointing in Distinct Count takes 73ms to complete for the whole working set of about 8MB on each node (with 30-second window size). Figure 14 shows the correlation of checkpointing frequency versus recovery time. For these experiments, we repeat the run for 10 times for each checkpointing interval and calculate the *maximum* recovery time.

Unlike TimeStream's precise recovery, Storm provides a weaker guarantee on fault tolerance and may introduce errors due to state loss or duplicate input from replay when failures happen, or both. We repeat the early fault-tolerance experiment on Storm (with a single failure) and compare the output against that from TimeStream. Figure 15 shows the differences, where the output from TimeStream is exactly the same as a run without any failure. However, Storm outputs 31 incorrect window counts with a maximum error rate of 12.5% and then restores after about 60 seconds, which is actually the period of time for the window to slide out of the incorrect state.

Sentiment Analysis of Tweets. We performed extensive failure injections to Sentiment Analysis with an input rate of 9.6K/s. The failure patterns include both isolated and multiple correlated failures. All the operators are either stateless or bounded stateful. We therefore use dependency tracking and output buffering for recovery, without resorting to checkpointing.

A chain in the DAG: F ⇨ HP ⇨ GB ⇨ ST ⇨ U ⇨ W	
Vertex Fails	Recovery Time (s)
F×1	5.2
ST×1	12.2
W×1	6.9
F×1, ST×1, W×1	12.3 (F: 7.9, ST: 12.3, W: 8.6)
All (×6)	12.3

Figure 16. Recovery time of different failure patterns: single failure (first three rows; $\times 1$), multiple concurrent but isolated failure (the fourth row) and chained failure involving all operators (the last row; $\times 6$). These six vertices form a chain in the DAG: F(filter), HP(partition), GB(grouping), ST(sentiment engine), U(union), W(windowing).

We use a configuration with a 20% over-provisioning: 9 nodes can handle a peak load at around 12K/s. The over-provisioning is required to mask failures, and is much cheaper than a hot-standby solution [24] that would require 100% over-provisioning. The impact of failures varies with respect to operator types and failure patterns. To crash an operator, we kill the operator on the node where it is running. The failure detection mechanism will kick in and reschedule the operator to a spare node.

We pick 6 different operators that form a chain in the DAG (Figure 16; first row). The failure pattern includes crashing one operator, three operators simultaneously, or the entire chain. For a contiguous failure segment, the recovery time is the elapsed time from the time of the crash to the time when the last operator starts to generate new output.

The average recovery times of filter, sentiment engine, and aggregation window are 5.2, 12.2, and 6.9 seconds, respectively. The average fault-detection and rescheduling time is around 4.7 seconds. Thus, the time to repair the state is moderate. Filter recovers quickly even though it is a stateful operator (a cross-join), because the topic stream is small and stationary. The window operator takes slightly longer to recover, because it has to rebuild a larger state. Although sentiment engine is stateless, its initialization takes several seconds. Note that, because the recoveries of multiple operators occur concurrently, the total recovery time is always close to the most expensive one (i.e., the sentiment engine).

In all these experiments, we do not observe any visible change of the real-time behavior due to failures: the latency stays around 2 seconds for all these failure patterns. This is because the result is produced at the closing of each non-overlapping 6-minute window while the failure recov-

ery usually happens within that window, and there is enough resource over-provisioning to mask the failure(s). Unlike in the experiment on Distinct Count, the recovery time is much shorter than the duration of the window, because in this case the bottleneck is the computation-intensive sentiment labeling engine: the state is small and can be reloaded quickly from upstream vertices (e.g., buffered output) instead of a rate-controlled data source as in the experiments on Distinct Count. In the worst case where the failure and recovery happen near the closing of a 6-minute window, we could see the full effect of failure recovery of up to 12.3-second delay, although the chances of that happening are low.

7.3 Dynamic Reconfigurability

We use Audience Insight to test TimeStream's ability to reconfigure and adapt to the load fluctuation. This application implements a monitoring service on the quality of a cloud-based video delivery service. The quality statistics are gathered and streamed in from the client media players. The demographic as well as geographic distributions of the viewing crowd can change dynamically according to the video program, causing the load to the monitoring service to fluctuate as well. In addition, the monitoring service needs to be as real-time as possible, because it guides the dynamic resource provisioning of the video service.

The quality-of-service (QoS) requirement is that the monitoring service must itself be able to handle 4 times the regular load, on-demand. Our goal is to let the amount of resources to "ride the tide" automatically. Also, because video playback time is typically fairly short and that the video popularity is unpredictable, adaptation should happen fairly frequently and do so without compromising data integrity. In contrast, a static worst-case configuration without using TimeStream would require an over-provisioning of 8 times the capacity, including the hot-standbys.

In our implementation, scaling is achieved by using different numbers of partitions (Section 3), a common practice. The configuration has a front-end server handling the input and partitioning, followed by one server per partition. For 1, 2, 4, and 8 partitions, the near-saturation throughputs are 28K/s, 54K/s, 110K/s, and 200K/s, respectively. Latency before the saturation point is not sensitive to the number of partitions and stays around 5 seconds.

As shown in Section 3, the named hash-partition gives the handle to adjust resources dynamically to cope with load fluctuations. We implement a simple policy to perform the adjustment on the fly. To be conservative, we choose the *safe* operating rate to be half of the saturation input rate. A 5% difference from the current safe operating rate triggers reconfiguration.

In Figure 17, the input rate is initially 10K events/s, handled by one partition. Between minute 1 and minute 6, we elevate the rate to 40K/s, and then return to 10K/s afterwards. During the high load period, the system reconfigures to use 4 partitions and continues to maintain real-time and low latency, except for a brief latency spike at the edge of the adjustment. Similarly, the system reconfigures back to use one partition after the rate drops back, causing the second brief latency spike.

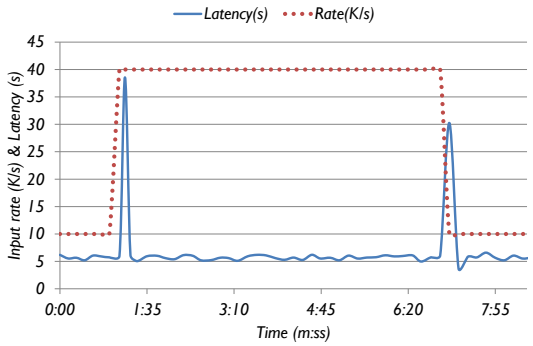

Figure 17. Dynamic reconfiguration in response to load changes: load is elevated 4 times from minute 1 to minute 6.

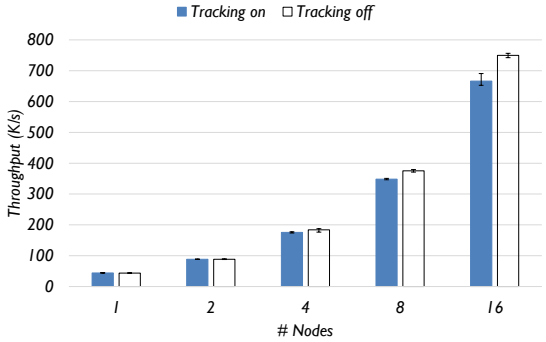

Figure 18. Distinct Count performance with and without dependency tracking.

7.4 Overhead of Dependency Tracking

Tracking dependencies is not free; there is both computational and network overhead. It turns out that computing dependency incurs negligible overhead: stateless operator (e.g., projection) has nothing to track, and the semantics of stateful operators (e.g., windowing and cross-join) are typically clear enough that their dependencies can be computed efficiently. Communicating the dependency to the reliable storage incurs CPU overhead for serialization and consumes network bandwidth. We amortize the overhead by batching.

To understand the overall impact, we compare two sets of runs of Distinct Count on different numbers of nodes, with and without dependency tracking, with a default batch size of 512 URLs. As shown in Figure 18, dependency tracking is lightweight and does not incur much overhead.

We also measure the overhead of a complete run of Sentiment Analysis at 9.6K/s input rate running on 9 nodes. The CPU and network overhead is 1.55% and 2.23%, respectively. Most of the operators have negligible CPU overhead, except the window operator which currently has an

inefficient implementation. For the network overhead, the filter operator—a fused operator including cross-join, filter and projection—incurs the highest network overhead, or 14.82%.

8. Conclusion

We are witnessing the emergence of a new class of applications that involve continuous complex data processing on huge volumes of streaming data. In this paper, we present a system, TimeStream, that takes on the challenge of building a distributed-system infrastructure for reliable low-latency continuous processing of big streaming data. TimeStream manages to combine the best of both MapReduce-style batch processing and streaming database systems in one carefully designed coherent framework, while at the same time it offers a powerful abstraction called resilient substitution that serves as a uniform foundation for handling failure recovery and dynamic reconfiguration correctly and efficiently. We believe this new class of applications will drive the design of next-generation distributed systems and demand new abstractions to be developed to meet the new challenges.

Acknowledgments

We thank Furu Wei and Ming Zhou for the Sentiment Analysis application, Jie Tong, Jun Qian, Kang Ji and Mao Yang for the on-line advertising scenario, and Tong Jin for the Storm deployment. We are indebted to our reviewers for their insightful comments on the paper, and particularly to our shepherd Ymir Vigfusson for his feedback and help in writing the final version of this paper.

References

[1] Hadoop. http://hadoop.apache.org/.

[2] Storm. https://github.com/nathanmarz/storm/wiki.

[3] Trident. https://github.com/nathanmarz/storm/wiki/Trident-tutorial.

[4] Streambase systems. http://streambase.com/.

[5] MEIJER, E., BECKMAN, B., AND BIERMAN, G. LINQ: Reconciling object, relations and xml in the .NET framework. In *SIGMOD*, 2006.

[6] ALI, M. H., GEREA, C., RAMAN, B. S., SEZGIN, B., TARNAVSKI, T., VERONA, T., WANG, P., ZABBACK, P., ANANTHANARAYAN, A., KIRILOV, A., LU, M., RAIZMAN, A., KRISHNAN, R., SCHINDLAUER, R., GRABS, T., BJELETICH, S., CHANDRAMOULI, B., GOLDSTEIN, J., BHAT, S., LI, Y., DI NICOLA, V., WANG, X., MAIER, D., GRELL, S., NANO, O., AND SANTOS, I. Microsoft CEP server and online behavioral targeting. In *VLDB*, 2009.

[7] ANDRADE, H., GEDIK, B., WU, K. L., AND YU, P. S. Processing high data rate streams in system S. *J. Parallel Distrib. Comput. 71*, 2 (2011), 145–156.

[8] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD* 2005.

[9] BARGA, R., GOLDSTEIN, J., ALI, M., AND HONG, M. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUIN, R. Incoop: MapReduce for incremental computations. In *SOCC*, 2011.

[12] BIEM, A., BOUILLET, E., FENG, H., RANGANATHAN, A., RIABOV, A., VERSCHEURE, O., KOUTSOPOULOS, H., AND MORAN, C. IBM InfoSphere Streams for scalable, real-time, intelligent transportation services. In *SIGMOD*, 2010.

[13] CHAIKEN, R., JENKINS, B., LARSON, P., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: Easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[14] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[15] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

[16] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIXATC*, 2010.

[17] HWANG, J. H., BALAZINSKA, M., RASIN, A., CETINTEMEL, U., STONEBRAKER, M., AND ZDONIK, S. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[18] LAMPORT, L. Paxos made simple, fast, and byzantine. In *OPODIS*, 2002.

[19] LIU, C., CORREA, R., GILL, H., GILL, T., LI, X., MUTHUKUMAR, S., SAEED, T., LOO, B. T., AND BASU, P. Puma: Policy-based unified multi-radio architecture for agile mesh networking. In *COMSNETS*, 2012).

[20] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed stream computing platform. In *ICDM Workshops*, 2010.

[21] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[22] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *HotCloud*, 2009.

[23] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. MadLINQ: Large-scale distributed matrix computation for the cloud. In *EuroSys*, 2012.

[24] SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.

[25] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a MapReduce framework. In *VLDB*, 2009.

[26] XING, Y., ZDONIK, S., AND HWANG, J. H. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.

[27] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[28] ZAHARIA, M., DAS, T., LI, H., SHENKER, S., AND STOICA, I. Discretized Streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.