Piotr Podolski

TimeStream: Reliable Stream Computation in the Cloud

Problem

- 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
- 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-style (big) data processing and requires a different distributed-system infrastructure.
- 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.

- 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, where each incoming data item in an input stream is processed *exactly once*.

- Streaming database systems tend to use replication for reliability. Such a solution is often costly and relies on strong assumptions on how many failures can occur concurrently.

Solution

- TimeStream is a distributed system designed specifically for low-latency *continuous* processing of *big streaming* data on a large cluster of commodity machines.
- TimeStream manages to combine the best of both MapReduce-style batch processing and streaming database systems in one carefully designed coherent framework.
- 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.
- TimeStream is designed to faithfully preserve the programming model of StreamInsight, and as a result it can be used to scale out any existing StreamInsight applications to large compute clusters without any modification

- TimeStream makes the following technical contributions.
  - 1. 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
  - 2. Second, TimeStream adopts the programming model of StreamInsight 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.

Overwiew

- 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, S4, D-Streams, Puma and Flume.
- **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.
- **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, StreamBase, and StreamInsight) is appropriate. Unfortunately, these options have high overhead and limited flexibility when the system needs to scale up.
- 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 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.
- **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.
- 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.

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 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 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.
  - 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.

- 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).
- 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.

- 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.
  - *Count* windows are sliding windows, each including a fixed number of events.

TimeStream

- 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 repartition a CepStream for parallel execution.
- The HashPartition breaks the word stream into three partitions so that the query can be executed for example on three machines in parallel.

```
// 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`.

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.

TimeStream design- 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.

- The computation in vertex v is triggered by an arriving entry i in an input stream, which updates v's state from T1 to T2, and produces a sequence o of output entries.

- 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.

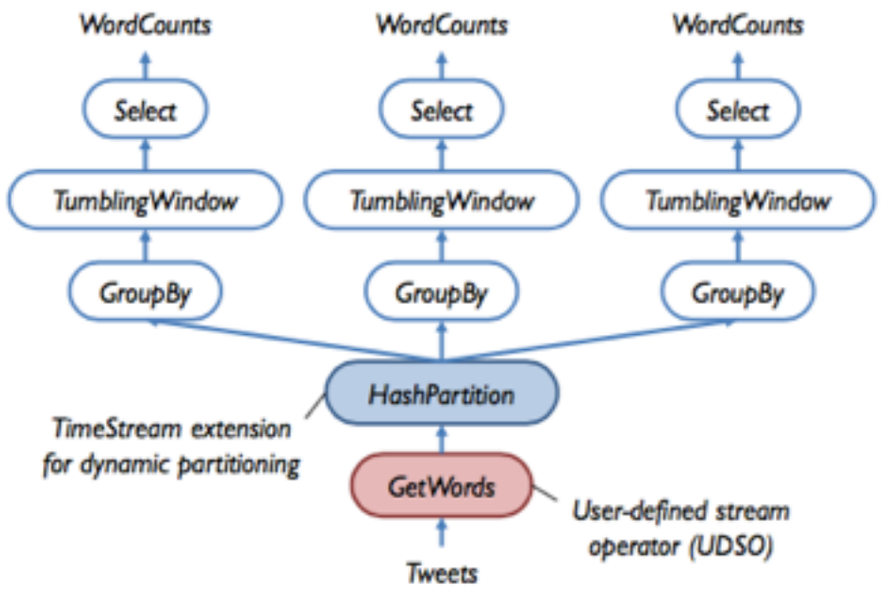

Figure 2. Streaming DAG for the continuous word count query in Figure 1.

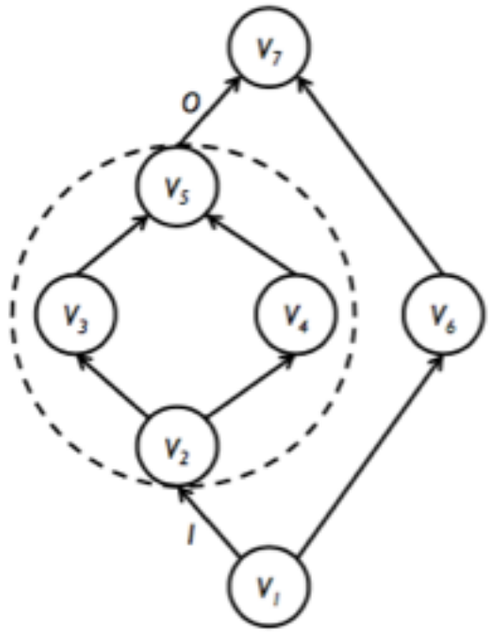

Figure 3. Streaming DAG and sub-DAG example: the circled subgraph is a sub-DAG with input stream I and output stream O.

- 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.
- S*ource vertices are* special input adapters that generate output from various sources continuously to drive the computation of the rest of the DAG.
- *Result vertices* 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.

- 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.

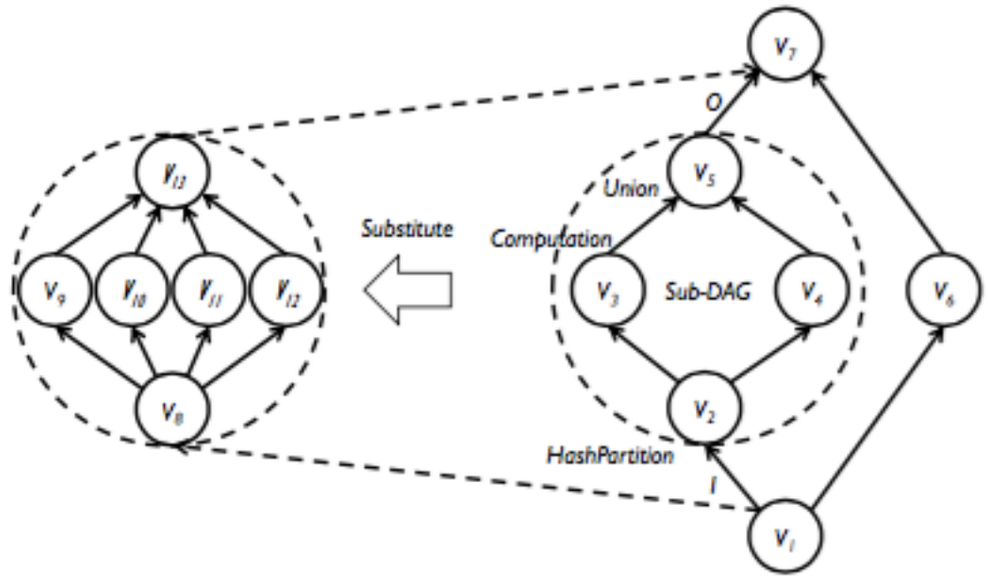

Figure 4. Resilient substitution example: the circled sub-DAG on the right can be substituted by the one on the left.

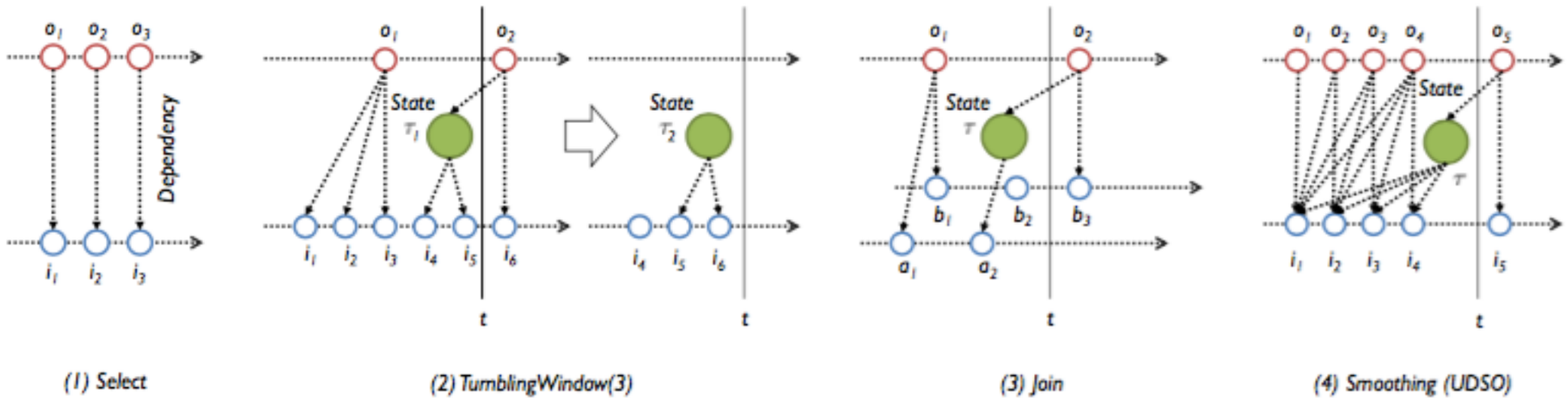

Figure 5. Dependency structure examples for common operators.

- 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.
- Resilient substitution in TimeStream does it 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 *st*, we define its *state dependency* (denoted as $dep_s(v,st)$) 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.
- To record the state and output dependencies, TimeStream labels data entries in an input or output stream using sequence numbers to identify them uniquely.

- 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 *st* 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 *st* and then produces output *o*.
- In practice, TimeStream recovers a set of data entries together, rather than for each individual one.

- 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.

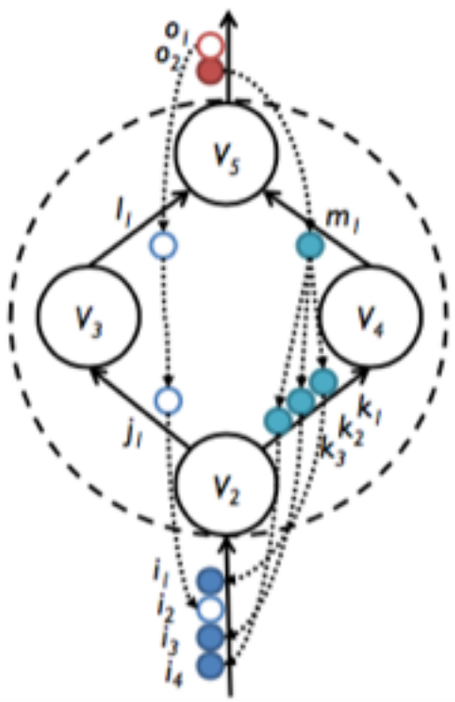

Figure 6. Dependency inference of a sub-DAG.

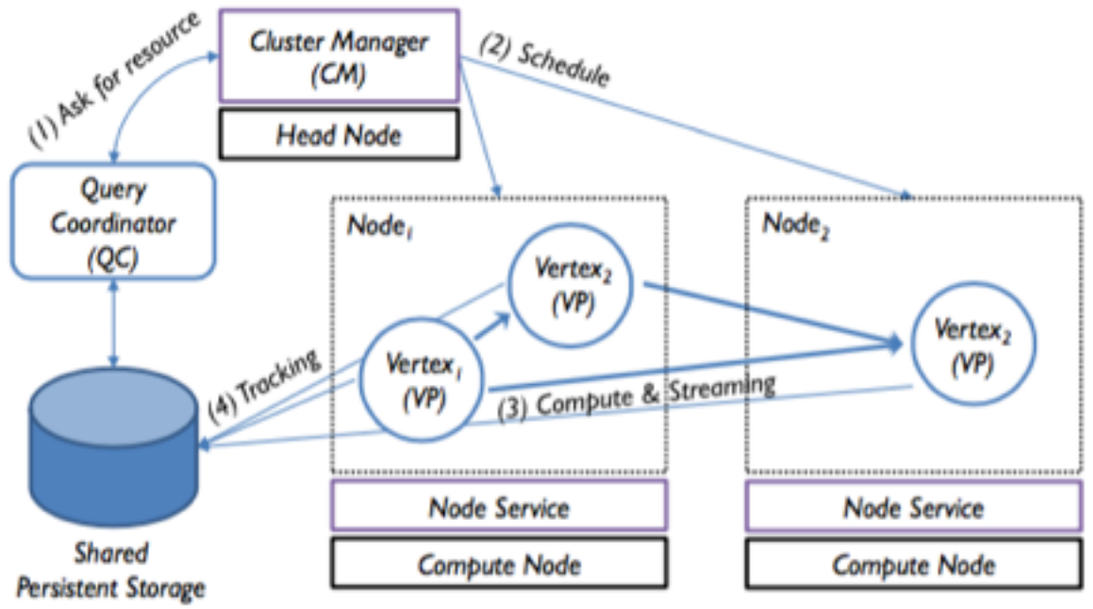

Figure 7. TimeStream distributed runtime overview.

- The cluster service consists of:
  - Cluster Manager(CM) running on a *head* node.
  - Node service(NS) running on each *compute* node in the cluster.
  - Query coordinator(QC) is created for each job running on the cluster.

Implementation- distributed runtime

- C# / .NET
- The TimeStream runtime is built on top of a cluster service we developed for resource allocation, vertex scheduling, and failure detection.
  - 1. The QC first talks to the CM for allocating resources to run the job DAG
  - 2. QC schedules the vertex processes (VP) through the cluster service onto the compute nodes
    - Meta-data information of the QC is maintained in a reliable storage so that its fail-over is straightforward
- 3. 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
- 4. During the execution, VPs track the progress and dependencies, and periodically write them to the same reliable storage.
  - QC uses this information to manage any failure recovery and re- configuration, 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.

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

Optimizations

- **Operator fusion**- To reduce unnecessary I/O, a segment of the stream operators in a query is mapped 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)
- **Dependency batching**- The dependencies exposed by the operators are at the granularity of an event segment including only one CTI. 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.
- **Output buffering**- TimeStream also supports output buffer- ing on each vertex as a "caching" mechanism. A vertex keeps a buffer of recently generated data entries of its out- put streams, so that they do not need to be recomputed when requested by the downstream vertices.
- **State checkpointing**- 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.

Applications

- 1. Network monitoring
- 2. Audience insight
- 3. Sentiment analysis of tweets
- 4. TopK and Distinct Count

```
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) }
```

```
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)
```

```
// 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())
```

```
// 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)
```

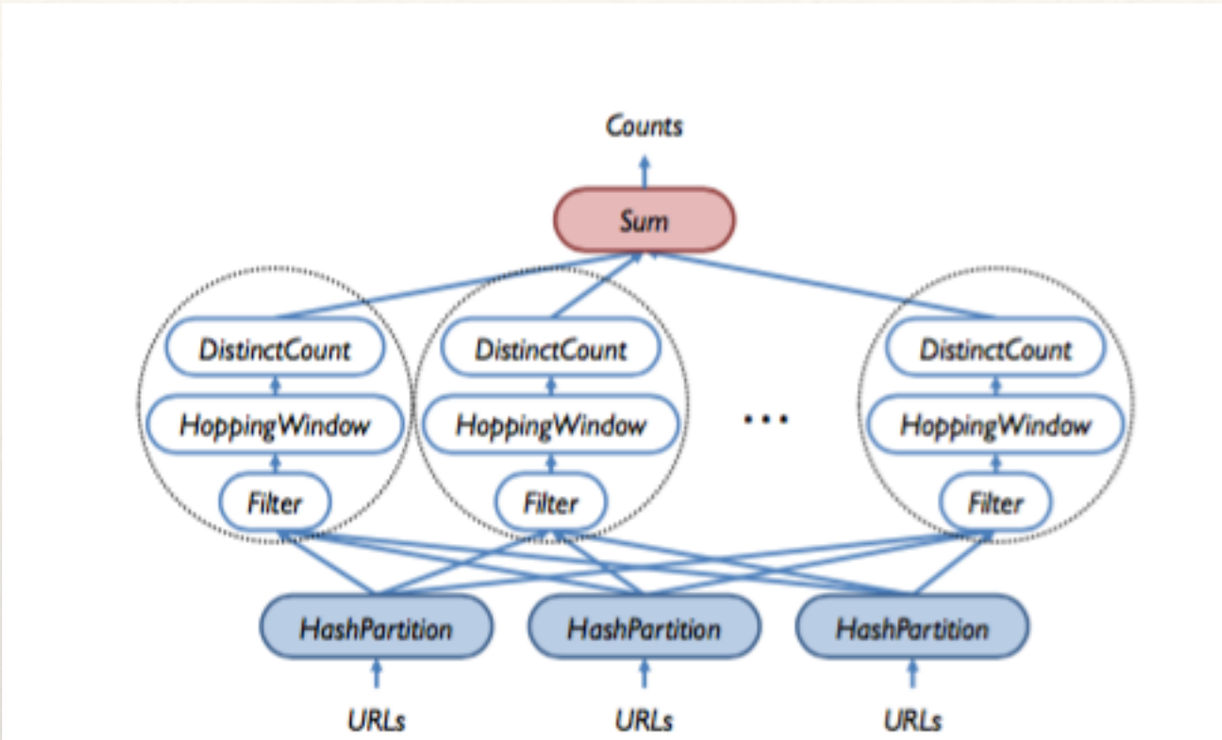

Figure 8. Streaming DAG for the Distinct Count query.

Evaluation

- Special attention was paid to scalability, fault tolerance, consistency, and adaptability to load dynamics.
- Overhead introduced by dependency tracking for achieving strong semantics was also tracked.

- Distinct Count-evaluation Distinct Count using a URL stream generated according to the statistics based on sample logs from the Bing search engine as the input dataset.
- Both systems(Storm, TimeStream) 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.

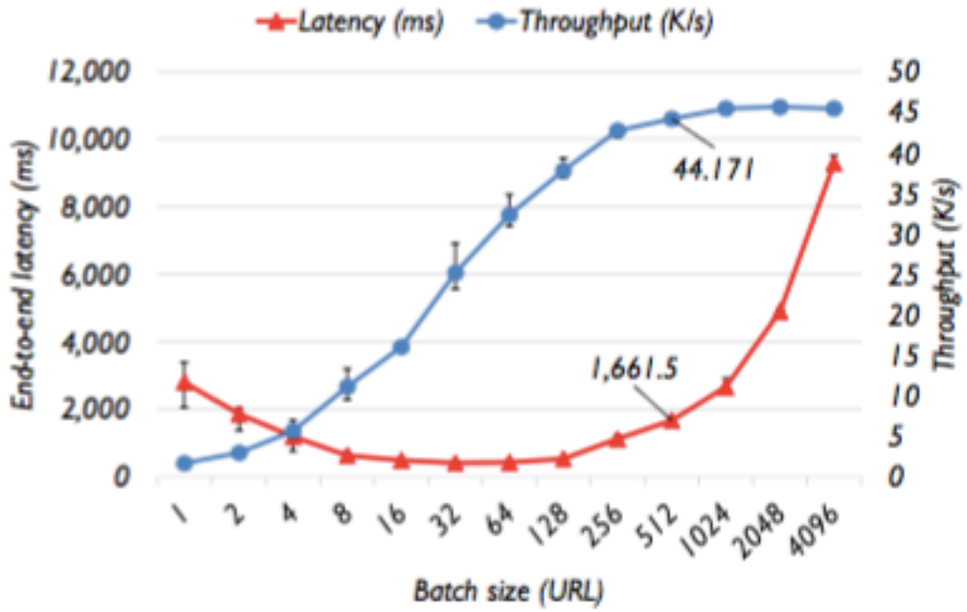

Figure 9. Latency and throughput trade-offs with different batch sizes for the Distinct Count query. At batch size 512 URLs, the throughput reaches 44.171K/s with a latency of 1,661.5ms.

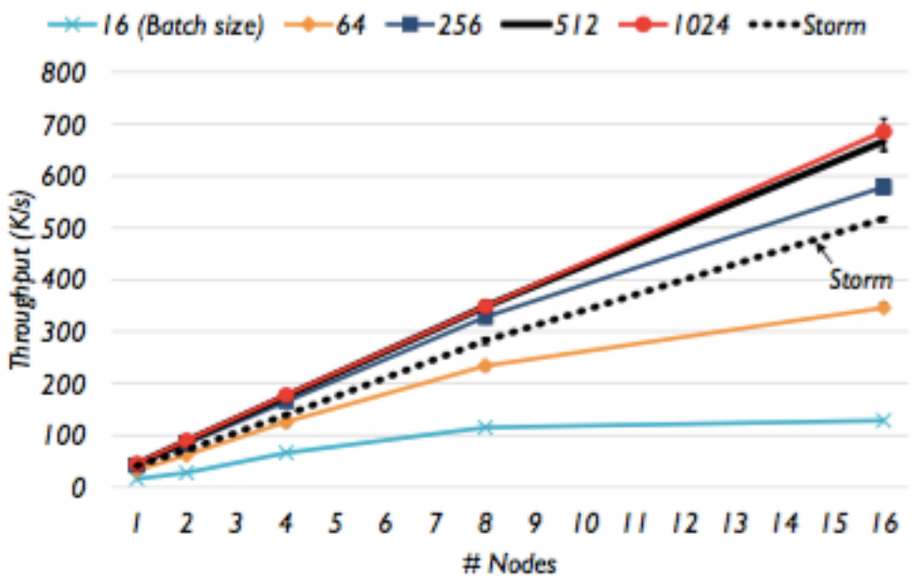

Figure 10. Scalability of Distinct Count using different batch sizes, with a comparison against Storm.

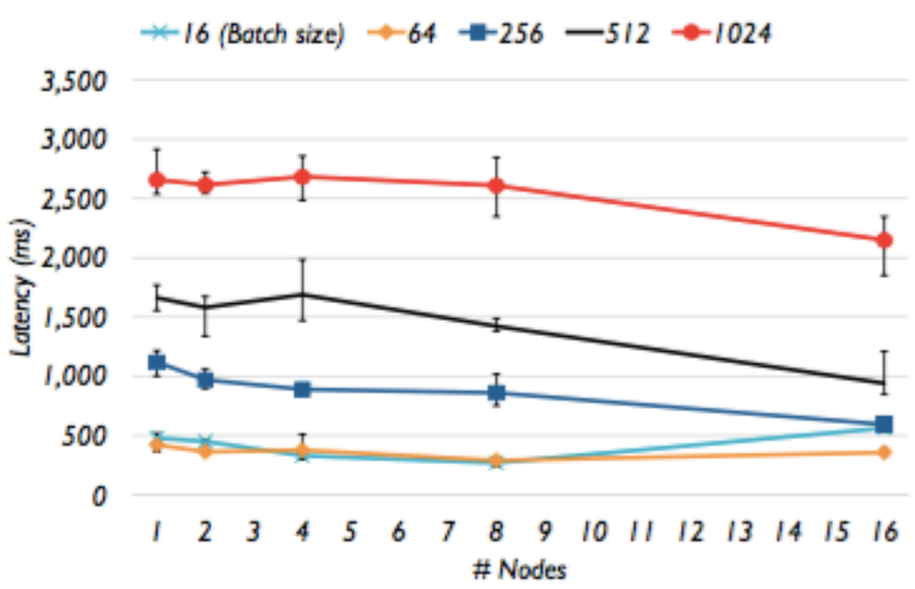

Figure 11. Latency of Distinct Count using different batch sizes.

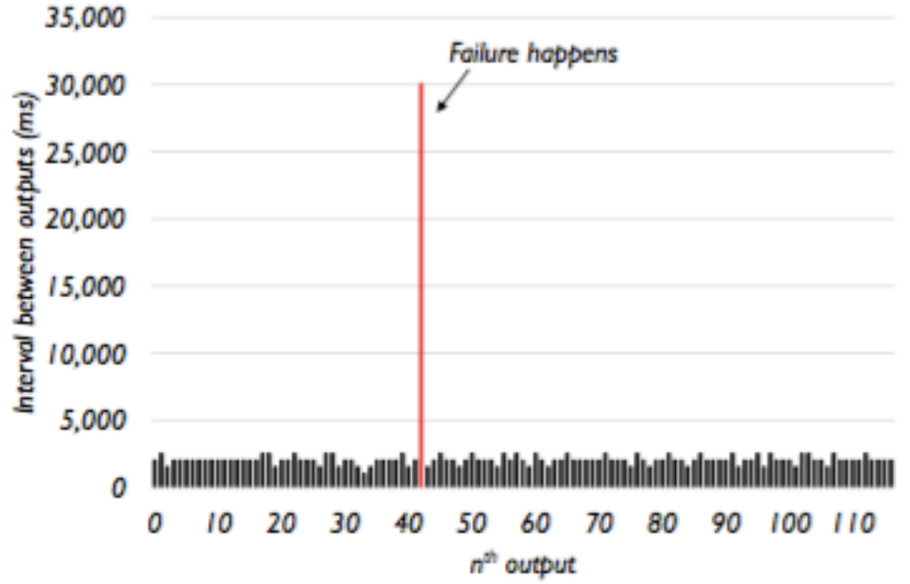

Figure 12. Failure recovery in a Distinct Count run.

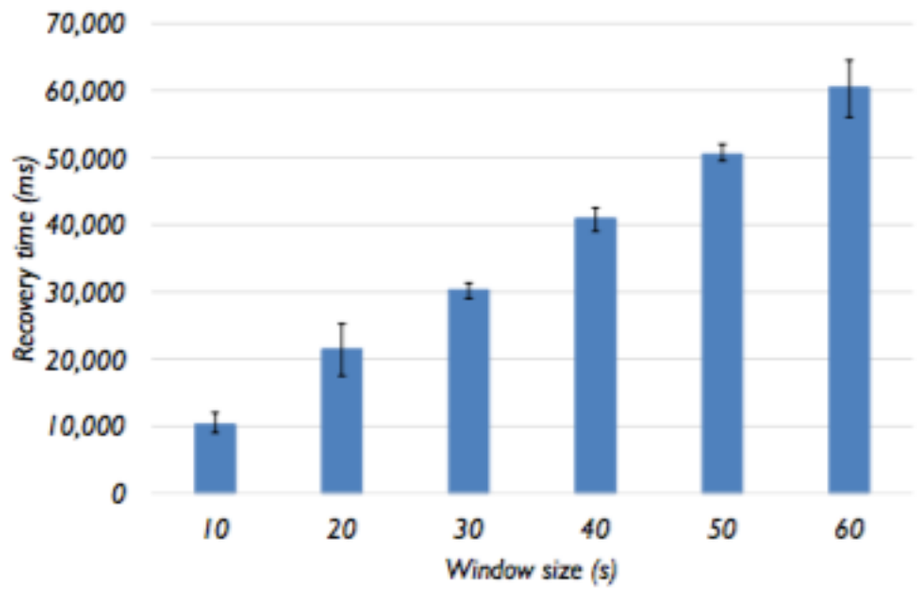

Figure 13. Failure recovery time with different window sizes for a Distinct Count query.

- Figure 10 shows the throughputs using different batch sizes.
- The maximum throughputs from Storm- the corresponding latency is shown in Figure 11.
- TimeStream scales linearly and performs comparably to Storm.
- Figure 12 shows the intervals between the consecutive (windowed) outputs during such a run with a failure.
- 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.
- The correlation between recovery time and window size was verified by repeating this experiment using different window sizes (while keeping the sliding step of 2 seconds). Figure 13 shows the result.

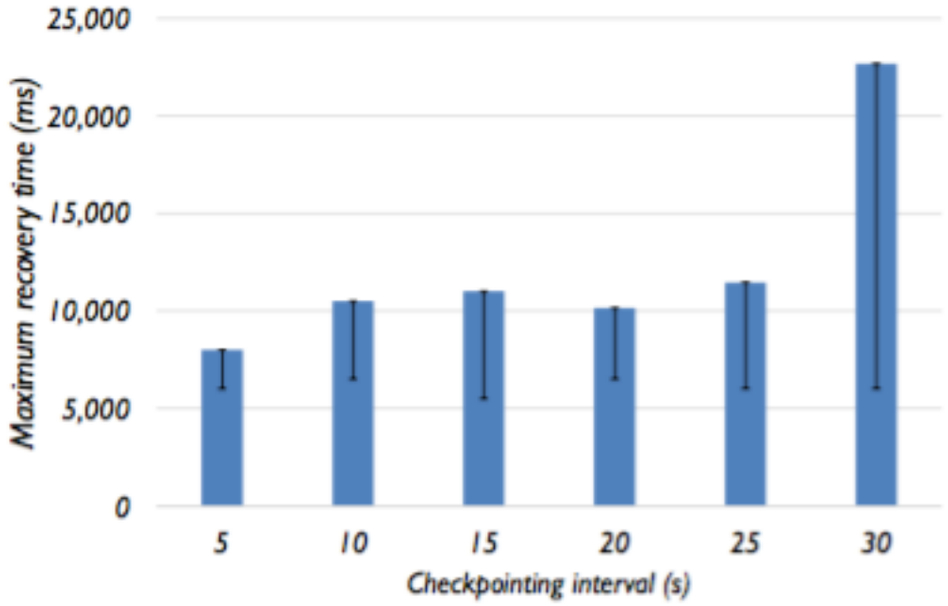

Figure 14. Maximum failure recovery time with different checkpointing intervals for a Distinct Count query. The black vertical lines show the range of the recovery time from 10 independent runs.

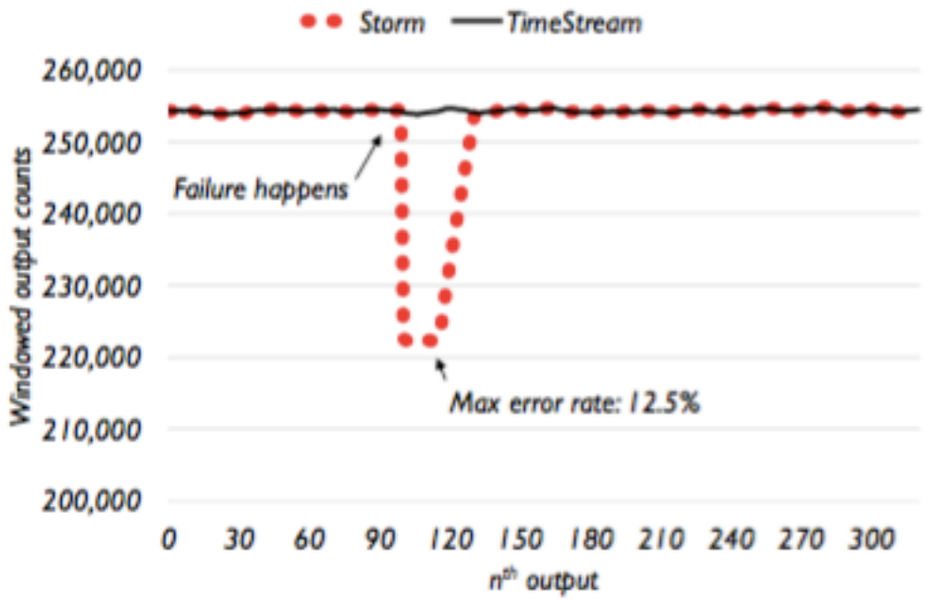

Figure 15. Distinct Count output with failure in TimeStream and Storm.

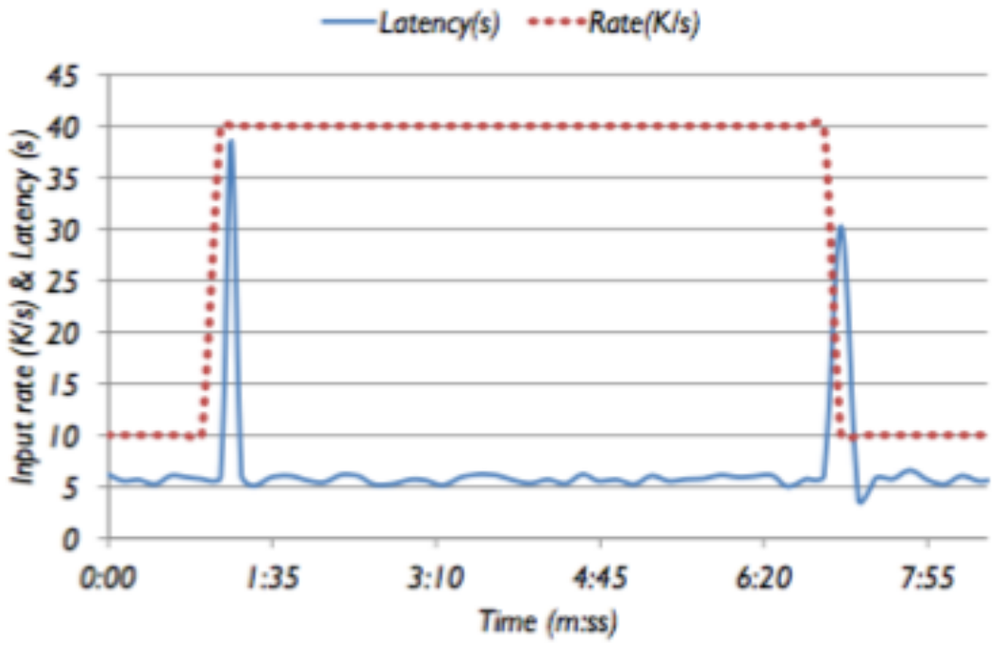

Figure 17. Dynamic reconfiguration in response to load changes: load is elevated 4 times from minute 1 to minute 6.

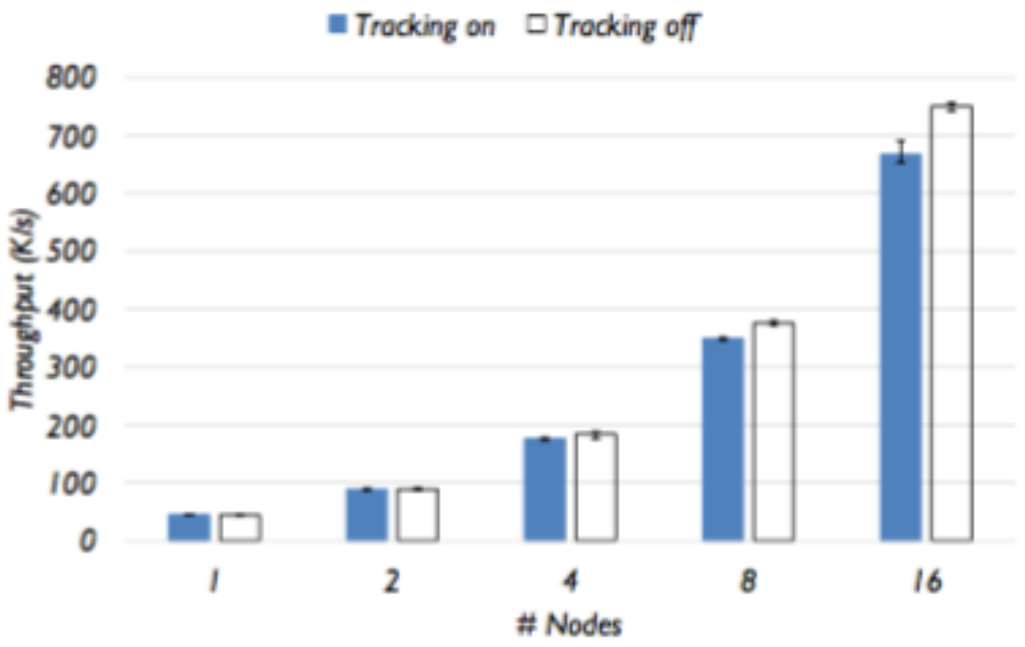

Figure 18. Distinct Count performance with and without dependency tracking.

- 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.
- Figure 15 shows the differences, where the output from TimeStream is exactly the same as a run without any failure.
- 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

Conclusion

- 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 of- fers a powerful abstraction called resilient substitution that serves as a uniform foundation for handling failure recovery and dynamic reconfiguration correctly and efficiently