ChronoStream: Elastic Stateful Stream Computation in the Cloud

Yingjun Wu, Kian-Lee Tan

School of Computing, National University of Singapore
{yingjun,tankl}@comp.nus.edu.sg

***Abstract*—We introduce ChronoStream, a distributed system specifically designed for elastic stateful stream computation in the cloud. ChronoStream treats internal state as a first-class citizen and aims at providing flexible elastic support in both vertical and horizontal dimensions to cope with workload fluctuation and dynamic resource reclamation. With a clear separation between application-level computation parallelism and OS-level execution concurrency, ChronoStream enables transparent dynamic scaling and failure recovery by eliminating any network I/O and state-synchronization overhead. Our evaluation on dozens of computing nodes shows that ChronoStream can scale linearly and achieve transparent elasticity and high availability without sacrificing system performance or affecting collocated tenants.**

I. Introduction

Distributed stream-processing systems (DSPSs) have been widely recognized as first-class citizens in the big-data analytics stack. Unlike large-scale batch-computation systems such as MapReduce [1], DSPSs support continuous low-latency complex analytics over massive data streams. Such online-computation capability is critical to many real-world applications, including transaction-log processing in financial markets, topic-trend detection in social media, and malicious-attack monitoring in telecommunication networks. A large body of research has designed and developed fault-tolerant scalable DSPSs for processing big streaming data in real time [2], [3], [4], [5], [6], [7], [8]. However, as the data-center environment and online-application requirements continue to evolve, new challenges have emerged to drive a complete redesign of DSPSs.

Large-state maintenance. Streaming applications commonly require stateful computations such as window operations and joins. The volume of the internal states manipulated by stateful operators in many real-world scenarios can expand to the order of hundreds of gigabytes, which is beyond the memory capacity of single machine or small computing clusters [9], [10]. Such memory-intensive streaming tasks require effective distribution of internal states to multiple nodes, achieving transparent failure recovery without putting heavy pressure on the in-progress computation tasks [6], [7].

Workload fluctuation. A long-running streaming application in the cloud can experience periodic or abrupt workload variations as well as unpredicted workload skews [11], which may overburden multiple computing nodes in the data center, causing severe straggler problems [1]. Resource planning and task deployment prior to job execution cannot adequately address such workload-related problems. Ideally, an efficient DSPS should flexibly leverage provisioned resources and transparently migrate workload hotspots at runtime in order to achieve balanced parallel computation.

Multi-tenant resource sharing. The next-generation cluster negotiators, such as Mesos [12] and Yarn [13], are taking over from slot-based job managers by providing fine-grained resource-sharing capability. In these resource-management frameworks, the cluster negotiator may dynamically preempt or grant resources, i.e., CPUs and RAMs, from or to systems at runtime for minimizing operating costs and preserving service-level objectives (SLOs). To support such multi-tenancy, the system runtime is expected to support flexible horizontal elasticity[1] and vertical elasticity[2] without degrading stream computation performance or affecting collocated tenants [14].

The challenges listed above call for a DSPS that is capable of supporting **elastic stateful stream processing** in a **multi-tenant environment**. Intensive research efforts have focused on elasticity for traditional database systems [15], [16], [17] as well as for data analytics systems [18], [8], [19]. In dataflow computation frameworks, elasticity is generally achieved by live state migration and complex state maintenance in active tasks. On facing workload spikes, the internal states of overly burdened tasks are repartitioned and transfered to lightly loaded computing nodes using live-migration techniques. Likewise, when resources are relinquished, the internal states of the involved tasks should be relocated and carefully consolidated without losing state consistency. This widely adopted strategy may perform well if the internal state in the active tasks is small. However, as the operator state expands to gigabytes or larger, the overhead caused by network load due to migration and state synchronization will increase, possibly violating the cluster SLOs and severely dampening the system performance. Our extensive experiments show convincingly that this classic paradigm is inefficient.

In this paper, we introduce *ChronoStream*, a system that supports transparent elasticity and high availability in latency-sensitive stream computation. ChronoStream integrates several key ideas that contribute to its effectiveness. First, ChronoStream aggressively divides the application-level states into a collection of *computation slices*, selectively distributes and checkpoints them into specified nodes; second, when node failure or workload redistribution occurs, ChronoStream transparently reconstructs and reschedules slice computation, eliminating any high and unpredictable overhead caused by network I/O and state synchronization; third, ChronoStream carefully balances the tradeoff between runtime performance and failure-

[1] add (remove) computing nodes in a system, aka scale out (in)

[2] add (remove) resources at a single node in a system, aka scale up (down)

recovery latency, minimizing potential system traps.

Our main contributions are as follows:

- We design ChronoStream, a distributed system that supports big stream processing in a multi-tenant environment. ChronoStream guarantees deterministic stream computation using locality-affinity checkpointing and lineage-free progress tracking. We present the system implementation and provide optimization solutions.
- We propose a lightweight state-management abstraction for big stateful computation. We illustrate how our system design detaches application-level computation parallelism from OS-level execution concurrency, achieving low-latency stream processing and transparent workload reconfiguration in an integrated model.
- We evaluate the performance of ChronoStream from several perspectives: scalability, elasticity, and fault tolerance. By comparing ChronoStream with several stream processing systems, we show that ChronoStream can scale linearly to dozens of nodes and achieve elasticity and high availability without sacrificing cluster SLOs or system performance.

We organize the paper as follows: Section II introduces the system model of ChronoStream. Section III presents the state-management abstraction built inside ChronoStream. Implementation highlights are given in Section IV. We report results of an experimental study of ChronoStream in Section V. Section VI reviews some recently proposed related work. We conclude the paper in Section VII.

II. System Model

In this section, we introduce the system model of ChronoStream. We first present the high-level programming model that supports the runtime logic of upper-layer streaming applications, and then present the execution model that guides the job deployment on the computing cluster.

A. Programming Model

At a high level, a ChronoStream application takes as input a group of streams from various external event systems (e.g., sensor networks, logging systems) and generates multiple output streams as results. In ChronoStream, data is represented as *event streams*, each generated by an operator and consisting of a potentially unbounded sequence of *events*. An event contains two parts: a key field and a payload. The key field is assigned by the user for stream-partitioning purposes. Events in the same stream with different keys are considered computationally independent. The payload has a well-defined schema and bears the actual application data associated with the event. The programmer is responsible for implementing serialization functions for each user-defined event type. The pseudocode of the event-tuple abstraction is shown as follows:

```
class Event{
   virtual INT64 GetHashCode();
   virtual void Serialize(STRING &string);
   virtual void Deserialize(STRING &string);
};
```

A programmer composes a *query DAG* to deliver execution logic to ChronoStream. A query DAG is a directed acyclic graph consisting of a set of *logical operators* connected by event streams. Each stream is given an ID declared by the programmer. A logical operator is the basic programming unit that encapsulates processing logic. An operator consumes and produces a set of streams, and optionally maintains a computation state. Input streams with different IDs are consumed in a deterministic fashion in order to preserve computational determinism. Each operator implements an input function:

```
void ConsumeEvent(ID &id, EVENT &event);
```

On receiving an event from an input stream, the function `ConsumeEvent` is triggered, where `id` denotes the input stream ID that generates `event`. A programmer may call a system-provided function to emit output events to downstream operators:

```
this->ProduceEvent(ID &id, EVENT &event);
```

where `id` denotes the output stream ID that receives `event`.

The invocation of `ConsumeEvent` may result in an update to the user-defined computation state, which is maintained in system-level in-memory storage, resembling a key-value store. The computation state is allocated by calling a state registration function, with a state pointer as return value:

```
STATE* this->RegisterState();
```

A computation state is updated through a set of APIs:

```
bool Set(KEY &key, EVENT &event);
bool Get(KEY &key, EVENT &event);
bool Delete(KEY &key);
```

A query DAG contains one or more special operators called *stream extractors* that generate input streams for the DAG from various kinds of sources (e.g., sensor data, transaction logs, etc.) and one or more *stream outputters* to handle the output results of the DAG.

B. Execution Model

ChronoStream compiles a query DAG into an execution plan consisting of units of execution and dataflow relations that can be physically deployed in computing clusters. Each logical operator in the query DAG is transformed into an *operator stage*, which maintains a stage-level internal state that spans across multiple nodes as necessary. During the query deployment phase, these operator stages are distributed to the cluster successively. Each operator stage registers with its targeted nodes by creating a dedicated resource container. The resource container is stage-specific and each node may bear multiple resource containers from different stages. Resource containers for the same stage are mutually called *peer containers*, or *peers* for short.[3] These containers collaboratively hold the stage-level internal state, with each maintaining parts

[3] In an operator stage with n containers, each container has $n-1$ peers. A container cannot be a peer of itself.

of the state and accepting the corresponding input streams from upstream operator stages in a deterministic manner. The design of stage-level state management will be further elaborated in Section III. Figure 1 shows the operator-stage deployment phase presented above.

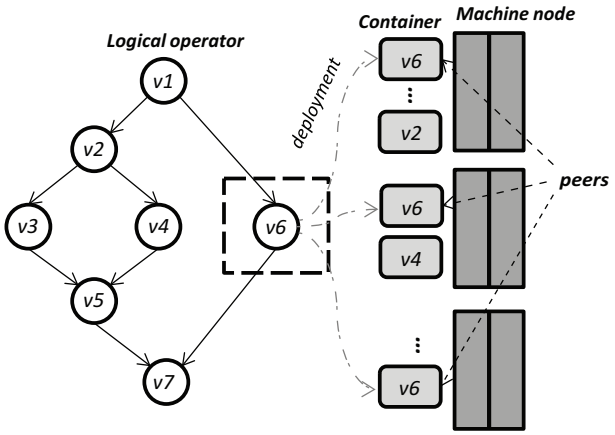

Fig. 1. Operator stage deployment.

Resource containers between neighboring stages are fully connected with each other, forming a large shuffle phase. The resource (i.e., CPUs and RAMs) granted to an operator stage is shared and managed jointly by the corresponding stage-level resource containers. A streaming execution runtime for a four-stage application is depicted in Figure 2.

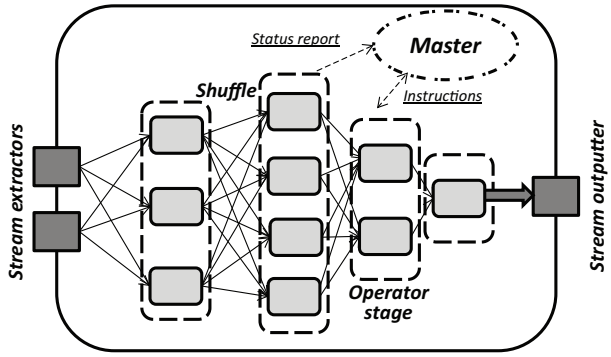

Fig. 2. Execution runtime of a four-stage application.

Each resource container reports its current status such as heartbeat and computation progress to a job master periodically. The master issues instructions to certain resource containers when dynamic scaling or failure recovery is required.

III. State Management

In this section, we present the state-management abstraction in ChronoStream. We first introduce the overall design of the state-management abstraction that powers parallel stateful computation, and then elaborate fundamental mechanisms for enforcing system determinism. We further demonstrate how ChronoStream transparently leverages two-dimensional dynamic scaling and parallel failure recovery by eliminating network I/O and synchronization overhead. We close this section with comprehensive system comparisons.

A. Design Overview

ChronoStream parallelizes and distributes stateful computation in large-scale computing clusters. In ChronoStream, each logical operator in the query DAG is modeled as a state machine that accepts a set of input streams, manipulates its internal states, and generates a set of output streams. We generally classify the internal states in each logical operator into two forms: *computation state* and *configuration state*.

Computation state is defined as a collection of application-level data structures that can be directly accessed and manipulated according to user-defined execution logic. Without losing generality, ChronoStream models each user-defined data structure as a key-value store, and any mutation is correspondingly represented as `get`, `set`, or `delete` operations. During the deployment phase, ChronoStream aggressively hash-partitions the computation state maintained in an operator stage into an array of constant number of fine-grained *computation slices*, and distributes them to multiple resource containers in a balanced fashion. Each slice is a computationally-independent unit associated with a subset of input streams and generating corresponding output streams. We say the slices are container-oblivious as they can be transparently *relocated* to peer containers in the same operator stage without affecting the consistency of its output streams.

Configuration state is the set of container-level states that maintains the runtime-relevant parameters. The configuration state is tied to each container and its contents may vary between different containers. The configuration state maintained in each resource container includes three components: an *input routing table*, which directs the input events into associated slices; an *output routing table*, which routes output events from an internal slice to the corresponding resource container in the downstream operator stage; and a *thread-control table*, which records how the OS-level threads are scheduled to support the computation of upper-layer slices. In a nutshell, configuration state actually plays the role of bridging from application-level parallelism to local OS-level multithreads.

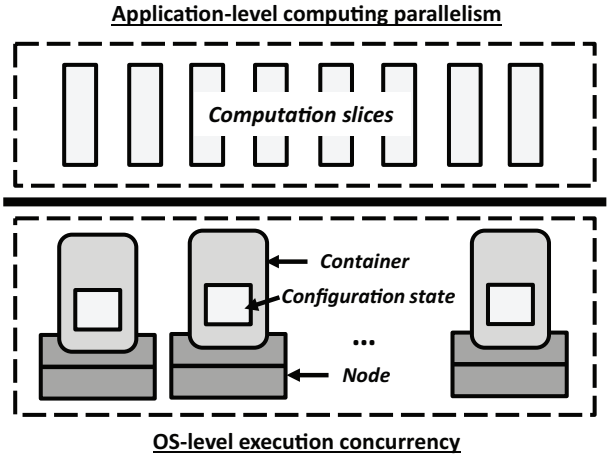

Fig. 3. Design overview of the internal state management abstraction.

Figure 3 depicts the relation among internal states, resource containers, and the underlying computing nodes in an operator stage. ChronoStream enables horizontal and vertical elasticity by *logically* scaling the computing nodes and manipulating

the corresponding configuration states instead of adjusting the upper-level computation states. Each container periodically checkpoints its active slices to remote peer containers. Any update to the configuration state in any container is recorded to the job manager persistently.

B. Fundamental Mechanisms

Streaming computation in ChronoStream is deterministic, elastic, and highly available. This subsection describes several advanced mechanisms that support these characteristics.

Chained Backup. ChronoStream periodically checkpoints the active computation slices to remote nodes for supporting elasticity and high availability. While it is always possible to rely on an underlying replicated storage system such as HDFS for durability, that approach is generally ineffective for a latency-sensitive computation framework due to its poor data locality and expensive replication overhead. ChronoStream addresses this problem by backing up the active slices in each resource container to its peer containers with a locality-sensitive data placement scheme, described as follows:

> *In a given operator stage, with a backup factor of L, for a resource container that bears M active slices $S_1, S_2, S_3, ..., S_M$ and has N corresponding peer containers $P_1, P_2, P_3, ..., P_N$, the l^{th} $(1 \leq l \leq L)$ backup for the m^{th} slice S_m will be delivered to the $\{(m+l)\%N\}^{th}$ peer container, or namely, $P_{(m+l)\%N}$.*

We call this strategy *chained backup*, because the slice backups are placed in a linked sequence, like a chain. The chained-backup strategy evenly distributes the backups to the peer containers at scale, greatly facilitates the elasticity and failure recovery, which will be further elaborated in a later subsection. Figure 4 exhibits how chained backup works in an operator stage given that the backup factor is set to 1. To guarantee high availability, the backup factor is usually set to 3. A user can also change the backup factor in the configuration file to explore a balance between runtime performance and fault tolerance.

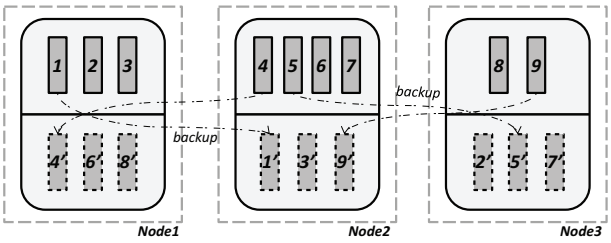

Fig. 4. Chained backup in an operator stage.

ChronoStream also persists output events of each slice periodically, for the purpose of stream replaying when node failure or workload reconfiguration occurs.

Computation progress tracking. Computation slices in ChronoStream need to be reconstructed once failure recovery or dynamic scaling is triggered. Restarting stream processing using dependency tracking preserves computation correctness but suffers from the expensive overhead of cascaded recomputation when the dependent state in the lineage graph is lost from memory [7], [6], [20]. This problem becomes even worse if the missing state in the lineage graph is large or the execution logic is complex. ChronoStream addresses this problem by tracking computation progress for each independent slice. Given a computation slice σ that consumes a number of input streams from the neighboring upstream operator stage(s), we define its computation progress as a vector of the number of consumed events from its input streams, denoted as $\texttt{prog}(\sigma) = \langle i_1, i_2, i_3, ... \rangle$. To record such computation progress, ChronoStream labels each event in a stream with monotonically increasing sequence number in order to identify them uniquely. At the time slice checkpointing is triggered, the progress vector is recorded along with the slice snapshot. Figure 5 illustrates how progress tracking works with two input streams. The computation slice σ takes events from these two streams X and Y in a deterministic manner and continuously records the progress. When the first checkpoint point arrives, the input events X_1, Y_1, X_2, Y_2 have been consumed, with the slice σ updated to σ_1. In this case, the snapshot of σ_1 is persisted, with progress vector $\texttt{prog}(\sigma_1) = \langle 2, 2 \rangle$ attached in its header.

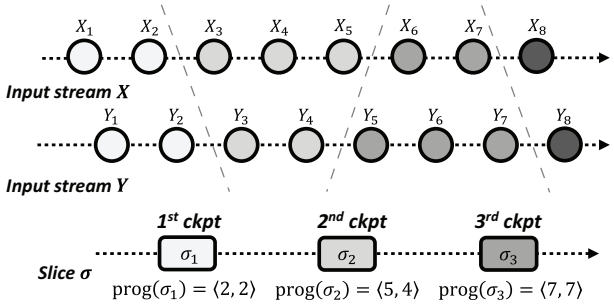

Fig. 5. Computation progress tracking with two input streams.

Progress information is transparent to users and essentially provides data-dependency information to exploit deterministic state reconstruction. This approach cuts off the dependency lineage across multiple linked operator stages, completely eliminating state rollback-and-recompute overhead.

Asynchronous delta checkpointing. Traditional streaming platforms such as Storm and S4 expect system users to provide state persistence semantics, generally requiring periodic synchronous state checkpointing. This approach is unsuitable for big streaming applications in a multi-tenant environment, for two reasons. First, synchronous checkpointing requires expensive locking of the internal state, possibly interrupting ongoing stream processing for a long period. Second, checkpointing the entire computation state leads to expensive network and disk I/O, causing further interference to collocated systems. ChronoStream tackles this problem using an asynchronous delta checkpointing mechanism. In ChronoStream, the lifetime of a computation state is divided into three phases: *normal phase*, *checkpointing phase*, and *merging phase*. In the normal phase, every update to a computation state is directly reflected at the corresponding key-value store and the updated entry is marked with a dirty bit. On triggering the checkpointing phase, ChronoStream scans all the maintained key-value stores and persists the corresponding updated entries to remote storage. The checkpointing phase is nonblocking, i.e., ongoing stream processing remains active and every incoming update is buffered to a temporary data structure. Once the

checkpointing phase is completed, the merging phase starts and all the buffered updates are further integrated into the associated key-value store. Delta checkpointing comes with the overhead of reconstructing the computation states from scratch. ChronoStream periodically merges the deltas at the backup side so as to reduce reconstruction overhead.

C. Horizontal Elasticity

In a multi-tenant computing cluster, a system runtime is expected to dynamically utilize resources and balance workloads across provisioned nodes, without interrupting in-progress computation or affecting collocated tenants. To enable such flexible elasticity, *transparent workload migration* is a critical capability. State migration techniques reported in the database and virtual-machine literature [15], [16], [21] do not fit well in the latency-sensitive stream-processing scenario, as these approaches inevitably incur significant overhead from network I/O and state synchronization [22]. ChronoStream addresses this challenge using a lightweight transactional migration protocol based on stage-level state reconstruction. Figure 6 illustrates the procedure for migrating a computation slice S_{mig} in the operator stage SG_{mig} from the source computing node N_{src} to the destination node N_{dst}.

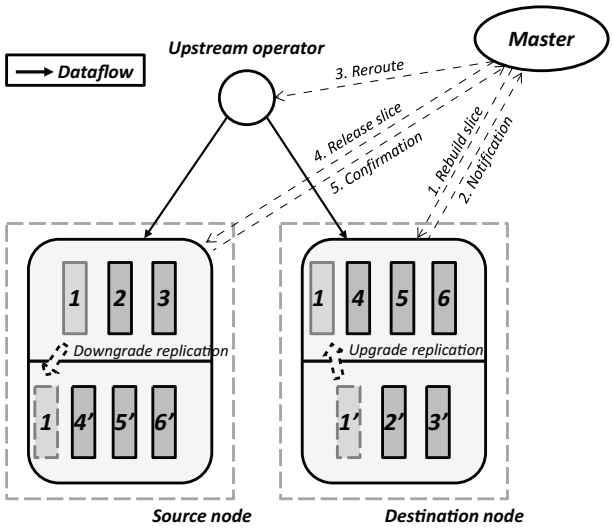

Fig. 6. Transparent workload migration.

Phase 0 (Migration preparation): Workload migration is initiated by spawning a dedicated resource container RC_{dst} of the stage SG_{mig} in node N_{dst}, if the container of SG_{mig} does not exist in N_{dst}. In Figure 6, Slice 1 is to be migrated.

Phase 1 (State rebuilding): The job master sends a request to RC_{dst} to trigger slice reconstruction for S_{mig}. On receiving the instruction, RC_{dst} rebuilds the slice by retrieving the slice backup from the local persistent storage, or from remote peer containers if the corresponding backup does not exist locally. We say that the slice backup is *upgraded* to an active slice, which is ready for stream processing. The computation progress vector included in the slice backup header is extracted, and sent back to the job master to indicate the completion of slice reconstruction. The container RC_{src} continues powering S_{mig} during the slice-rebuilding phase to guarantee zero service interruption. In Figure 6, the backup of Slice 1 at RC_{dst} is upgraded to an active state.

Phase 2 (Dataflow rerouting): On receiving notification, the master immediately sends dataflow rerouting requests to the upstream operator(s) for input re-forwarding. Dependent stream events are reloaded and resent to the destination resource container according to the computation progress vector. Once dataflow is rerouted, the corresponding stream gets consumed in RC_{dst}, in which case slice S_{mig} is actually processed in both RC_{src} and RC_{dst}, forming a *dual mode* step. The downstream operators filter out the duplicated tuples according to the sequence number attached to each event.

Phase 3 (Resource releasing): After rerouting, the master further request resource release from RC_{src}, which keeps processing S_{mig} until no further output tuples can be generated. The whole migration procedure completes once the resource is confirmed as released from RC_{src}. In Figure 6, Slice 1 at RC_{src} is downgraded and becomes inactive.

This transparent workload migration mechanism works best when the slice backup can be directly reloaded from local storage. In this case, the expensive overhead caused by network I/O is fully eliminated, leading to zero computation interruption and negligible performance interference. To achieve this ideal performance, ChronoStream tries with best effort to migrate the computing workload to those nodes where corresponding slice backup resides. The migration protocol also guarantees that the workload from a single resource container can be migrated to multiple peer containers in parallel, if necessary.

D. Vertical Elasticity

Modern cluster negotiators support fine-grained runtime resource allocation. ChronoStream enables such resource management by providing vertical scaling in a straightforward manner. For a resource container in ChronoStream, each assigned OS-level thread is one-to-many mapped to application-level computation slices for stateful computation scheduling. Such a thread-to-slice mapping is recorded in a *thread-control table* inside the configuration state. On receiving vertical scaling request, ChronoStream directly manipulates the thread-control table in order to reschedule the computation. At any time, the computation workload assigned to each thread can be dynamically rearranged to achieve thread-level load balance. Programmer may even encode a customized scheduling mechanism into the resource containers. Figure 7 illustrates how a resource container scales up its core usage.

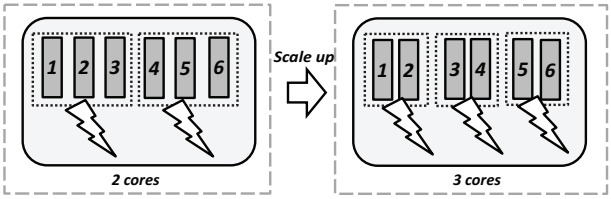

Fig. 7. Dynamic vertical scaling in a resource container.

Dynamic updates to the thread-control table need to be transactional. A traditional readers-writer lock enables such atomic operation, but drastically limits effective concurrency.

ChronoStream minimizes such locking costs by implementing a lightweight *optimistic readers-writer latch*. Instead of acquiring and releasing a lock every time the thread-control table is accessed, each reader thread will hold the latch until a periodically installed synchronization barrier has arrived. This approach minimizes the lock-checking frequency, while preventing the writer-starvation problem. Our experiments further confirm the efficiency of the chosen locking strategy.

E. Fault Tolerance

ChronoStream recovers lost slices in parallel if node failure occurs. Similar to the workload-migration protocol that powers the horizontal elasticity capability, ChronoStream masks node failure through a stage-level state-reconstruction approach. Figure 8 illustrates how failure recovery in ChronoStream works in a single operator stage running on three nodes.

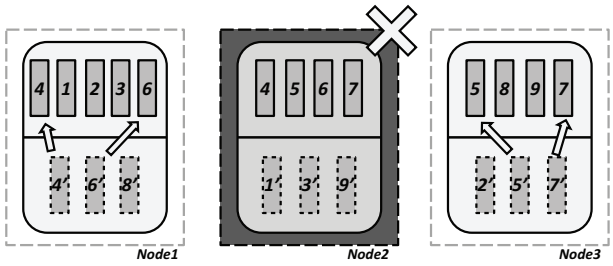

Fig. 8. Parallel failure recovery.

As is depicted in the figure, the resource container running on Node 2 periodically backs up its active slices, namely S_4, S_5, S_6, S_7, to its peer containers in Node 1 and Node 3. Once node failure is detected, the job master immediately sends requests to Node 1 and Node 3 for slice reconstruction. As the peer container in Node 1 holds the backup of S_4 and S_6, these two slices will be retrieved and activated directly in Node 1. Similarly, S_5 and S_7 will be directly reconstructed in Node 3. Computation determinism is guaranteed by the progress dependency attached in the header of each slice backup. This parallel failure-recovery protocol minimizes the computing downtime when the active backups in a single resource container can be spread over to as many peer containers as possible. The chained-backup mechanism adopted in ChronoStream guarantees this condition and ensures any computation interruption caused by node failure can be restored within a short period.

F. Computation Slice Granularity

In ChronoStream, the computation slice granularity in each operator stage is determined at compilation time and is kept constant throughout the system runtime. Setting the slice granularity statically in a reasonable manner is not easy. ChronoStream configures the slice granularity with the purpose of providing elasticity *at best effort*. Given a parameter array $< N_i^1, N_i^2, N_i^3, ..., N_i^m >$ denoting the maximum number of cores that can be utilized by the operator stage SG_i in each of the m provisionable nodes, SG_i aggressively partitions its computation states to N_i^{max} slices, where N_i^{max} is equal to $N_i^1 + N_i^2 + N_i^3 + ... + N_i^m$. To achieve the highest elasticity degree, the operator stage SG_i should scale to m nodes and utilize all the N_i^{max} available cores, in which case each OS-level thread is one-to-one mapped to the upper-level computation slices, obtaining the best system performance. ChronoStream also allows the system user to manually configure the slice granularity according to specific application scenarios.

G. Comparison

ChronoStream differs from other stream-processing systems in supporting flexible elasticity and high availability for big stateful applications, especially in multi-tenant cluster environment. Table I compares several stream-processing platforms with the focus on state management strategy, elasticity, and fault tolerance. D-Stream manages its internal state using an immutable abstraction called RDD. While D-Stream also supports parallel recovery, its rollback-recompute-based fault-tolerance strategy pays a high cost when an operator's execution logic is complex and its internal state is large. In SEEP & SDG, state management is made explicit to the system users, who must perform workload redistribution themselves. In addition, state-repartition-based mechanisms for dynamic scaling can incur significantly high overhead. TimeStream enables elasticity by sub-DAG reconstruction, which can cause long computation downtime. In comparison, ChronoStream provides large-state management at the system level and supports fine-grained two-dimensional elasticity, without affecting system performance or requiring user interaction.

IV. Implementation

We implemented ChronoStream from scratch using 20,000 lines of C++ code, excluding the test suites and third-party libraries. This section describes the system runtime of ChronoStream and presents some optimization techniques that further improve system performance.

Distributed runtime. The ChronoStream runtime is built with the principle of seamlessly integrating with a next-generation cluster negotiator such as Mesos and Yarn. Figure 9 depicts the distributed runtime overview of ChronoStream. In this system framework, a *framework tracker* is deployed in a dedicated node to manage resources granted to each streaming job. A node service is running on each computing node in the cluster. On receiving a job submission, the framework tracker communicates with the cluster negotiator to apply for computing resources. Once resources are granted, the framework tracker distributes the user-submitted job to each provisioned node and starts monitoring the resource allocation status. A job master is generated to spawn resource containers, track computing progress, and periodically send garbage-collection instructions for each container. The master also helps in balancing resource allocation at runtime.

Extended state library. ChronoStream utilizes a key-value store for computation-slice maintenance. The performance of this generalized model may be limited when supporting certain specialized structures such as sliding windows. To improve performance, we also implemented an internal state library on top of the key-value store to support a broader range of structures, including queues, array lists, and sliding windows.

Asynchronous output persistence. ChronoStream persists output events to replicated storage. Synchronously persisting tuples brings high disk-access cost. ChronoStream supports output buffering to "cache" recently generated tuples and periodically dumps them into persistent storage in batches.

System	State management	State update	Horizontal elasticity	Vertical elasticity	Fault tolerance
Storm & S4	Not applicable	Not applicable	Stateless reconstruct	Stateless reconstruct	Recompute
Samza	External databases	Fine-grained	Not supported	Not supported	Sync. checkpoint
D-Stream	RDD	Immutable	Not supported	Thread rescheduling	Recompute
SEEP & SDG	User-aware	Fine-grained	State repartition&migration	State repartition	Async. checkpoint
TimeStream	User-transparent	Fine-grained	Sub-DAG reconstruct	Sub-DAG reconstruct	Dependent recompute
ChronoStream	*User-transparent*	*Fine-grained*	*Fine-grained reconstruct*	*Thread rescheduling*	*Async. delta checkpoint*

TABLE I. A COMPARISON AMONGST SEVERAL DISTRIBUTED STREAM PROCESSING SYSTEMS.

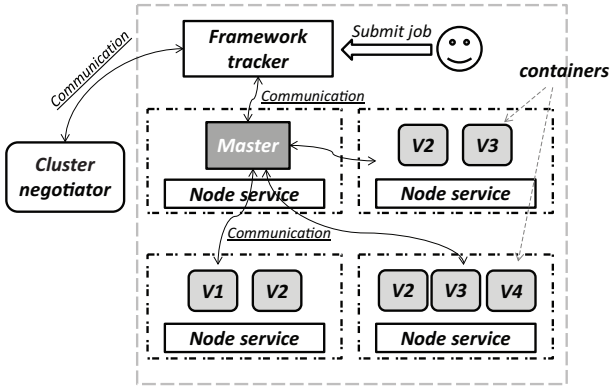

Fig. 9. Distributed runtime overview of ChronoStream.

Replication buffering. ChronoStream also supports replication buffering. On receiving slice replica from remote peer containers, a resource container delays the replication persisting procedure and temporarily stores the replica in memory. Once dynamic scaling or failure recovery occurs, the replica can be directly utilized without incurring disk I/O.

V. EXPERIMENTS

We designed and conducted a series of experiments to evaluate the performance of ChronoStream. We pay special attention to scalability, elasticity, checkpointing, and failure recovery. We further report the possible overhead introduced by computation slice scheduling. We compared our system with two open-source stream-processing systems: Storm [4] and Spark Stream [6]. Our experiments were performed on a 20 node in-house cluster running CentOS 5.5. Each node is configured with 16 cores and 24 GB of memory.

A. Applications

For the experimental study, we ported two online event-analytics applications to our systems.

Top-K frequent words. This application monitors a text stream and reports the top-K most frequently used words over a sliding stream window. We feed the query with Wikipedia article abstract streams containing millions of entries from 100 categories, where the average length of each entry is 474 bytes. The query DAG comprises three components: a source operator, a set of processing operators, and a sink operator. The source operator divides the input stream into multiple substreams according to the categories and delivers them to the processing operators. Each processing operator filters out the stop words in the texts and performs a word count over a sliding window. The results are further ranked by the frequency and the top-K events are sent to the final sink operator.

Tweet-similarity search. This application explores the most similar tweet pairs based on posting location over a sliding window. Like the previous application, three elements comprise the query DAG. A source operator partitions the incoming tweet stream by user location and feeds the substreams to a set of processing operators. For each input event, each processing operator scans the contents of a sliding window of recent tweets and searches for the most similar tweet according to the number of shared words. The most similar paired tweets are sent to a sink operator periodically.

In both applications, a dedicated Kafka server is spawned to continuously load the input datasets from local disk and feed an input stream to the stream-processing system. Each incoming tuple is attached with a sequence-id field to indicate the stream progress. In all these experiments, we implement counting-based sliding window to precisely control the computation-state size.

B. Scalability

We implemented the two streaming applications in ChronoStream, Storm, and Spark Stream, and compared their performance on scalability. For fairness, we turned off checkpointing in this set of experiments, since Storm does not intrinsically support checkpoint-based recovery.

The first experiment evaluates the execution time for consuming every 100,000 tuples in the top-K frequent words application. We fix the computation-state volume at 12 GB, and scale the runtime system to 20 nodes, each utilizing one core. For ChronoStream, we place only one computation slice in each resource container to avoid potential overhead caused by slice scheduling. As shown in Figure 10, the execution time of the three systems drops near-linearly as the number of computing nodes increases. When scaling to 20 nodes, ChronoStream reaches a peak processing rate of 193K events per second. Storm performs two times slower than ChronoStream, since JVM-based systems usually suffer from overhead caused by string manipulation and garbage collection. Spark Stream, to our surprise, does not perform well in this experiment. By consulting the Spark mailing list, we found that the problem is due to disk access during data shuffling phase[4]. The performance can therefore be improved by utilizing a fast storage medium such as SSD or an in-memory filesystem. ChronoStream and Storm, on the other hand, do not require such hardware-related optimization, as all the computations are in memory.

[4]https://www.mail-archive.com/user@spark.apache.org/msg05433.html

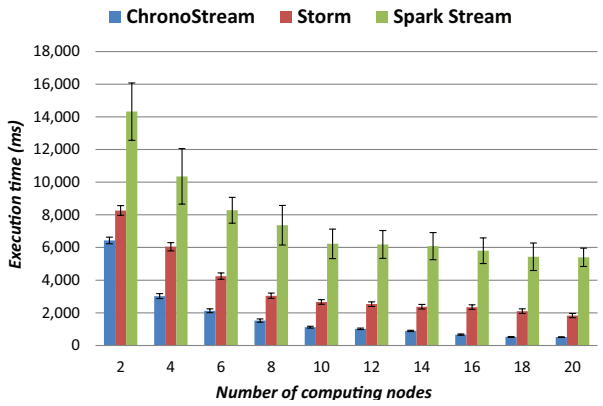

Fig. 10. Execution time for consuming 100,000 continuous events.

The second experiment measures system throughput by bounding end-to-end latency below one second. Figure 11 captures the maximum throughput each system can sustain on the top-K frequent words application with 12 GB of computation state. Similar to what we have observed in the previous experiment, ChronoStream scales out linearly in the cluster, reaching the maximum throughput of over 80 MB per second. In comparison, Storm and Spark Stream also expand to 20 nodes, but achieve comparatively lower throughput, due to the same reasons as above.

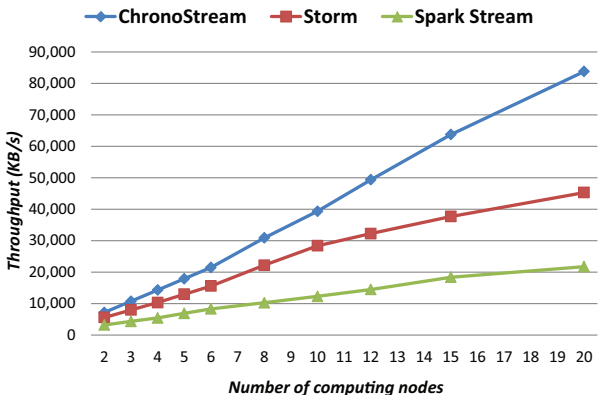

Fig. 11. Throughput under sub-second latency bound.

The following two experiments examine how state volume influences the performance of ChronoStream.

Figure 12 illustrates the execution time of ChronoStream for processing 100,000 continuous events in the top-K frequent words application with varied computation-state volumes. The application is deployed on 4 computing nodes, with the workload distributed in a balanced manner. As is shown in the figure, the execution time grows smoothly as the computation state enlarges from 5 GB to 40 GB. The increase is primarily caused by cache misses and memory paging. This type of application is essentially *scale independent* [23], because theoretically the computation complexity remains stable with the increase in state volume, and the RAM capacity in the computing nodes tends to become the system bottleneck.

The execution time for processing 50,000 events in the tweet-similarity search application is shown in Figure 13.

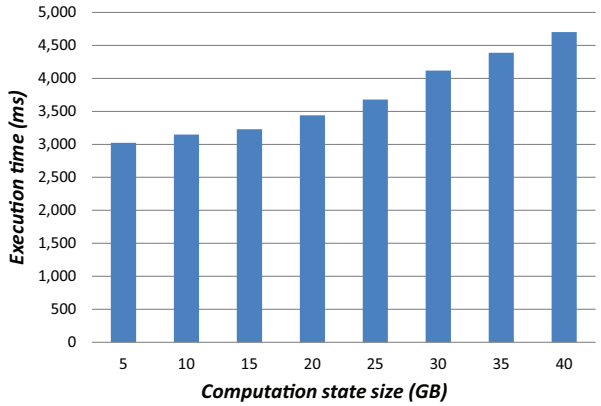

Fig. 12. Processing 100,000 events in top-K frequent words application.

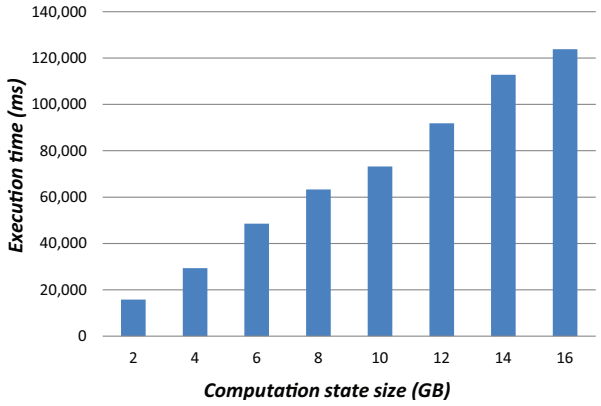

Fig. 13. Processing 50,000 events in tweet-similarity search application.

We deployed the ChronoStream runtime on 20 computing nodes. With the computation state growing from 2 GB to 16 GB, the corresponding execution time rises correspondingly from around 15 seconds to more than 120 seconds. In this type of applications, the computation complexity increases linearly as the program scales. Such applications are known as *scale dependent*, and CPU throughput usually becomes the bottleneck that hampers the execution performance.

C. Horizontal Elasticity

We investigate the efficiency of workload migration mechanism in ChronoStream to validate whether the system can transparently support horizontal elasticity. We compare the slice-reconstruction approach proposed in ChronoStream with state-migration approach, which is widely adopted in many elastic systems [15], [16], [21], [8]. For fairness, the state-migration approach is also implemented in ChronoStream.

To examine the effect of horizontal scaling to in-progress stream computation, we migrate the streaming workload between two computing nodes and measure the resulting execution delay, defined as the time duration from the point when the migration instruction is issued to the point when the next output tuple is generated. As shown in Figure 14, the state-migration mechanism leads to increasingly high latency as the migrated workload grows from 1 GB to 8 GB. Such delay is mainly caused by network I/O and state consistency cost, which is directly related to the migrated state capacity.

In contrast, by eliminating all synchronization and I/O-related overhead, the lightweight slice-reconstruction mechanism enables ChronoStream to generate outputs continuously, with tolerable service delay.

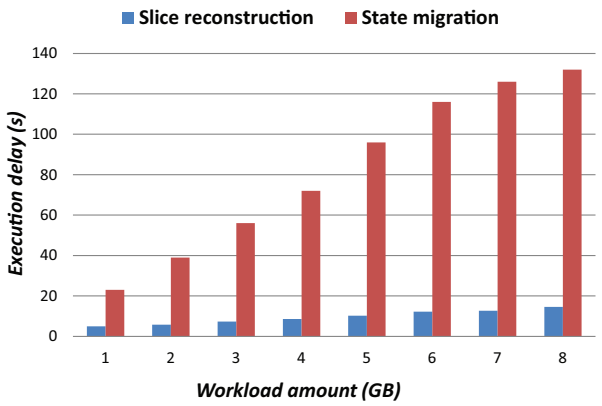

Fig. 14. Execution delay caused by horizontal scaling.

Next we validate how different horizontal scaling mechanisms affect the collocated tenants. We deploy ChronoStream and a main-memory database, namely H-Store [24], on the same 2 computing nodes, and monitor the database throughput variation caused by ChronoStream's workload migration. The H-Store instance continuously processes the TPC-C benchmark with 10 single-threaded partitions[5], each loading 1 GB of warehouse data. We spawn 20 client host threads for issuing query requests from external nodes. For ChronoStream, we migrate 5 GB of data between 2 nodes using the slice-reconstruction approach and state-migration approach. Figure 15 shows the result. Under the interference of the state-migration mechanism, the database throughput drops down to 1,638 transactions per second on average during the workload-transfer stage. In comparison, slice-reconstruction approach relocates its workload smoothly, without causing any remarkable effect on the collocated database service.

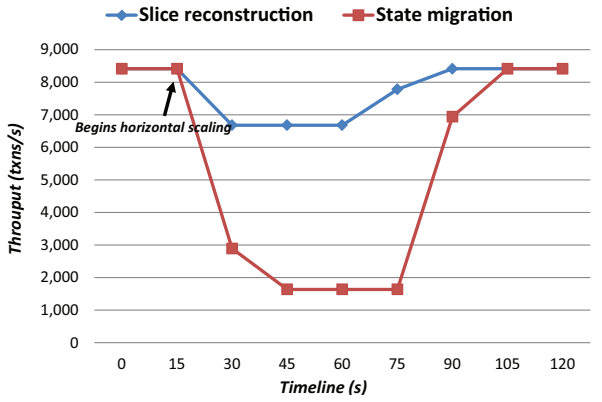

Fig. 15. Horizontal scaling affect on a collocated database system.

D. Vertical Elasticity

In this subsection, we study the effectiveness of vertical elasticity in ChronoStream. We measure the benefit of our thread-rescheduling mechanism in comparison with the state-repartition approach, which synchronously reallocates the workload to each thread. The state-repartition approach is also implemented in ChronoStream. We installed our system in two nodes, then increased the number of cores utilized in each node from 2 to 3, to see how dynamic vertical scaling affects the ongoing stream-processing task. For thread-rescheduling mechanism, we set the number of slices maintained in each resource container to 6. Figure 16 captures the execution delay introduced by vertical scaling. As the state capacity expands from 1 GB to 8 GB, the state-repartition approach incurs longer execution delays, up to 54 seconds, while thread-rescheduling maintains latency below 5 seconds. The difference in execution delay arises in part because the state-repartition approach synchronously performs state transformation operations, temporarily blocking all the processing tasks.

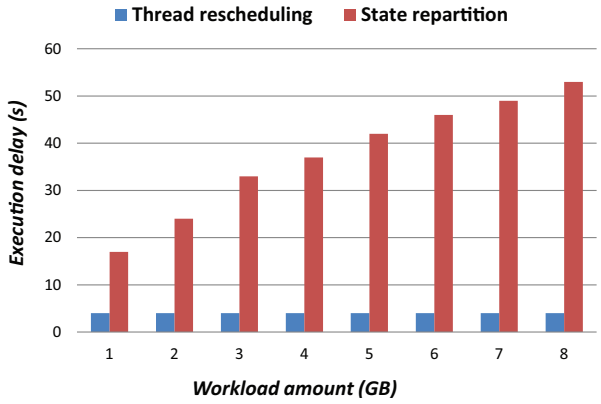

Fig. 16. Execution delay caused by vertical scaling.

We further investigate the possible performance effect on collocated tenants caused by vertical scaling. We deployed the target streaming system and H-Store on 2 computing nodes, and monitored the database performance variation caused by dynamic vertical scaling of the streaming system. For H-Store, we used the same configuration and spawned 20 client host threads as in the previous experiments. For ChronoStream, the CPU allocation in each node is dynamically increased from 2 to 3 cores. From Figure 17, we observe that the thread-rescheduling approach enables vertical elasticity with negligible affect on the collocated systems. In contrast, state repartition can cause significant performance interference.

E. Checkpointing

The following set of experiments investigates how different checkpointing strategies influence the system performance and cluster network bandwidth. We compared the asynchronous delta checkpointing mechanism with synchronous checkpointing approach, which is widely adopted in the traditional stream processing systems [9], [8].

We illustrate the impact of checkpointing strategy on the computation performance in a single node. The system runtime checkpoints the computation states every 60 seconds. We measured the average and maximum execution time for processing 100,000 tuples in the top-K frequent words application. Figure 18 captures the results with the state capacity growing from 128 MB to 4 GB. For synchronous checkpointing, the

[5]https://github.com/apavlo/h-store/issues/159.

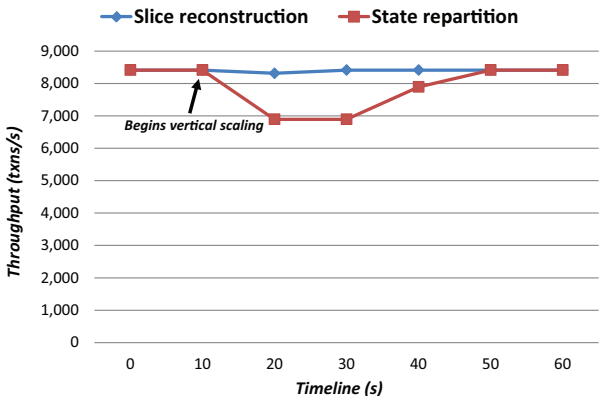

Fig. 17. Vertical scaling affect on a collocated database system.

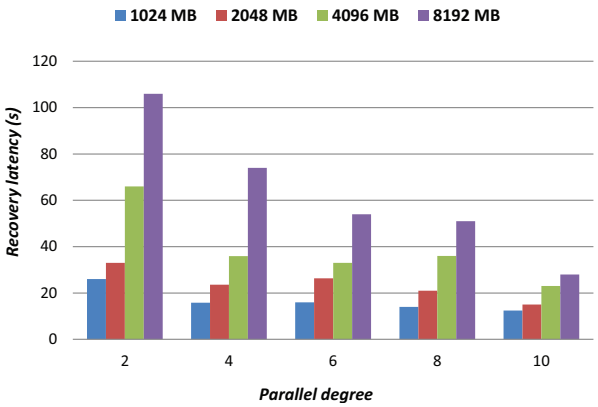

Fig. 19. Recovery latency with different failed state capacity.

ongoing stream computation is significantly affected during the checkpointing phase: the maximum execution time rises to 23,512 and 48,902 milliseconds for checkpointing 2 GB and 4 GB internal states respectively. In contrast, asynchronous delta checkpointing has little effect, only incurring around 4-seconds extra delay when the state capacity expands to 4 GB.

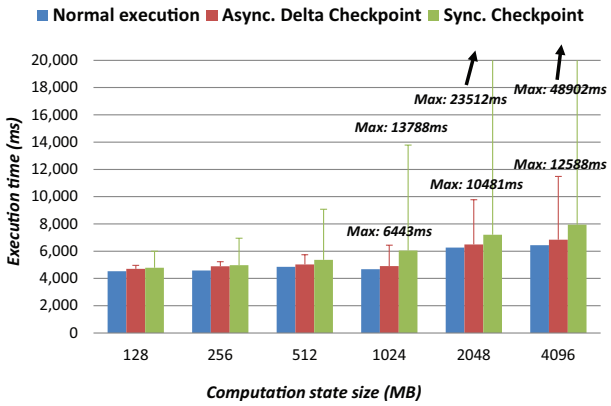

Fig. 18. Average and maximum execution time under the affect of computation-state checkpointing on a single node.

State checkpointing also affects network bandwidth in the cluster, and consequently affects the whole cluster ecosystem. Table II shows the network bandwidth variation caused by checkpointing. With the checkpoint interval set to 30 seconds, we monitor network statistics using the *ifstat* tool, which reports network bandwidth numbers at 1-second granularity. To checkpoint 2 GB of internal state, the input and output bandwidths respectively rise to 110,710 KB and 78,346 KB for 8 seconds. In contrast, asynchronous delta checkpointing only causes less-than-4-second bandwidth spike.

F. Failure Recovery

We further evaluate the effectiveness of the parallel failure-recovery strategy adopted by ChronoStream. We carefully analyze the three major factors that may influence parallel failure-recovery performance: failed-computation-state capacity, degree of parallelism, and checkpoint interval.

Figure 19 reports how failed-state capacity and degree of parallelism affect workload-recovery latency. We change the lost computation-state volume from 1 GB to 8 GB and measure the average recovery latency using different numbers of recovery nodes. The checkpoint interval is set to 30 seconds. As Figure 19 shows, doubling the recovery nodes reduces the recovery time by half. When the lost state capacity grows to 8 GB, single node recovery leads to over 100 seconds service downtime, far beyond acceptable range in latency-sensitive applications. Comparatively, parallelized failure recovery with 10 nodes reduces such latency to 27 seconds, which is moderately sustainable in the stateful-computation scenario.

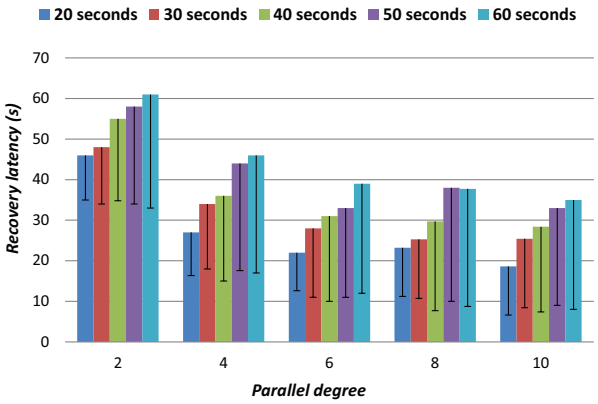

Fig. 20. Recovery latency with different checkpoint interval.

Figure 20 shows the effect of changing the checkpoint frequency and degree of parallelism. The black vertical lines represent the minimum recovery latency that can be achieved. We maintain the failed state capacity at 4 GB. As shown in the results, the average recovery latency increases roughly linearly with checkpoint interval, while the minimum recovery latency remains stable. This pattern is not a coincidence: if the node failure occurs right after the state checkpointing completes, the failure recovery time can be significantly reduced as the recomputation cost is minimized.

G. Slice Scheduling Overhead

The elastic stateful computation model in ChronoStream does not come for free: it may incur slice-scheduling overhead. ChronoStream minimizes such overhead by implementing *optimistic readers-writer latch*, as is described in Section III-D.

	Input network statistics				Output network statistics			
	Normal		Checkpoint		Normal		Checkpoint	
	Bandwidth	Duration	Bandwidth	Duration	Bandwidth	Duration	Bandwidth	Duration
Sync. checkpoint	60,251 KB/s	22 s	110,710 KB/s	8 s	249 KB/s	22 s	78,346 KB/s	8 s
Async. delta checkpoint	60,229 KB/s	28 s	115,313 KB/s	2 s	249 KB/s	26 s	42,251 KB/s	4 s

TABLE II. NETWORK BANDWIDTH CHANGE CAUSED BY CHECKPOINTING. THE CHECKPOINT INTERVAL IS SET TO 30 SECONDS.

To understand the overall impact, we deploy ChronoStream with Top-K frequent words application in a single-core node, increase the number of computation slices from 1 to 10 and measure whether certain performance degradation occurs. As is shown in Figure 21, the slice scheduling is lightweight and no significant deterioration in performance is observed. Even in the worst case, our model only causes 4.69% extra overhead.

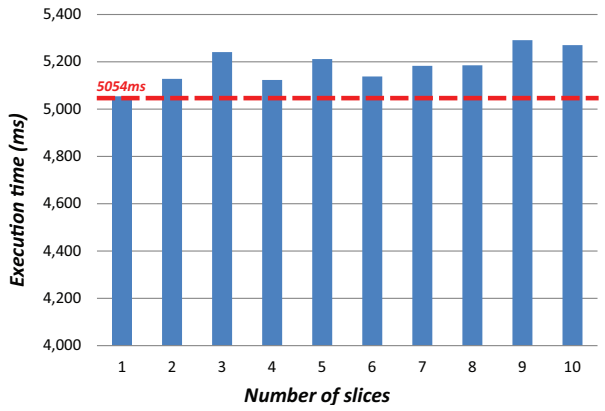

Fig. 21. The overhead of computation slice scheduling.

To conclude, all the experiments above demonstrated the effectiveness of ChronoStream from multiple aspects, including scalability, elasticity, and fault tolerance.

VI. RELATED WORK

ChronoStream exploits the strengths from several different research fields, spanning across system models, elastic computation, in-memory state checkpointing, and failure recovery. We discuss the related works in each of these areas.

Stream-processing systems. Several recent papers have proposed possible system models for large-scale stream computation. Comet [2] and CBP [3] adopt a bulk-incremental processing model, which decomposes continuous streaming computation into a series of batch incremental computation tasks that are further processed on the classic MapReduce platform. While these systems take advantage of the scalability and elasticity in MapReduce frameworks, they inevitably result in high latency, of up to several minutes. Storm [4] and S4 [5] are popular open-source stream-processing systems built to provide high-performance message-passing functionalities. While achieving good stream-processing performance, they generally have limited failure-recovery guarantees and do not provide state management from system level. Samza [9] is another open-source streaming system designed with a focus on fault-tolerant state maintenance. However, it requires an external replicated database for state persistence, leading to unpredicted overhead. D-Stream [6] partitions data stream into a sequence of mini-batches that are processed as stateless tasks in parallel. While this discretized stream model unifies batch processing with stream processing, they suffer from expensive cost of batch scheduling and state manipulation, possibly degrading system performance. Photon [10] and MillWheel [25] are two large-scale stream-processing systems built by Google. Although powering different applications, both systems rely on global distributed storage such as BigTable [26] and Spanner [27] to achieve fault tolerance, which may incur high cost of network and disk I/O. TimeStream [7] intrinsically support deterministic stateful computation. We extend its system model and further support transparent elasticity and high availability using an integrated state-management abstraction.

Elasticity. There has been a large body of research on introducing elasticity to distributed systems. Curino et al. [17] explored elastic transactional workloads in a multi-tenant environment. Albatross [16] and Zephyr [15] address the live-migration problem in a shared-storage and a shared-nothing database, respectively. Rajagopalan et al. [28] introduced elasticity to virtual middleboxes using a split/merge approach. EventWave [29] employs an elastic programming model and system runtime for cloud computation.

In the big-data analytics field, Ananthanarayanan et al. [18] proposed a system called Amoeba to support lightweight elasticity in the traditional MapReduce platform. However, their methods cannot be generalized to stream-processing scenarios, since Amoeba takes advantage of the disk-based shuffling phase, which is exclusive to the MapReduce model. SEEP [8], [19], to the best of our knowledge, is the most recently proposed DSPS that supports task reconfiguration at runtime. However, its state-repartition-based strategy inevitably leads to a heavy cost in state migration and maintenance, further degrading ongoing-computation performance and affecting collocated tenants. Meanwhile, explicitly exposing internal states to the programmers makes the system vulnerable to inexperienced state rewriting.

Fault tolerance. Several authors have discussed possible fault-tolerance strategies for DSPS. As a pioneering team in exploring the stream database field, Hwang et al. [30], [31] proposed upstream backup and an active-passitive replication approach to maintain high system availability. These two fundamental approaches balance the tradeoff between resource utilization rate and recovery latency, while ignoring the internal-state maintenance issue. Kown et al. [32] optimized the scheduling mechanism of an underlying distributed file system to better facilitate fault tolerance in stream-processing systems. Meteor Shower [20] fully investigated the cascading failure problem in stream processing, and adopted upstream backup as the fault-tolerance strategy. In D-Stream, a parallel failure-recovery approach is employed that benefits from the lineage tracking of RDDs [33]. Fault tolerance in stateful stream-processing systems is also closely related to that in main-memory database systems. RAMCloud [34] utilizes a high-

speed network to perform fast parallel recovery. H-store [24], [35] adopts logical logging and a copy-on-write approach to database availability. Hyper [36] is a centralized main-memory database that adopts a hardware-assisted approach to achieve fast state checkpointing. ChronoStream differs from these works by carefully modeling application-level internal states and introducing an integrated state-management abstraction for both availability and elasticity.

VII. Conclusion

We have proposed ChronoStream, a distributed system specifically designed for big stream computation in the multi-tenant cloud. ChronoStream treats large internal state as a first-class citizen and provides transparent elasticity support in both vertical and horizontal dimensions. By dividing computation states into a collection of fine-grained slice units, ChronoStream effectively distributes computation states into multiple nodes, and clearly detaches the application-level computation parallelism from OS-level execution concurrency. Our extensive experiments confirmed that ChronoStream can achieve transparent elasticity and high availability without sacrificing system performance or affecting collocated tenants.

VIII. Acknowledgement

The authors would like to thank David Maier, Zhengping Qian, Zhou Zhao, and the anonymous reviewers for their insightful suggestions.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[2] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: Batched stream processing for data intensive distributed computing," in *SOCC*, 2010.

[3] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *SOCC*, 2010.

[4] "https://github.com/nathanmarz/storm."

[5] "http://incubator.apache.org/s4."

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *SOSP*, 2013.

[7] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *EuroSys*, 2013.

[8] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.

[9] "http://samza.incubator.apache.org/."

[10] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *SIGMOD*, 2013.

[11] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *SOCC*, 2010.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, S. S. Hitesh Shah, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*, 2013.

[14] A. D. Konwinski and R. H. Adviser-Katz, "Multi-agent cluster scheduling for scalability and flexibility," 2012.

[15] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.

[16] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," in *VLDB*, 2011.

[17] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *SIGMOD*, 2011.

[18] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica, "True elasticity in multi-tenant data-intensive compute clusters," in *SOCC*, 2012.

[19] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *USENIX ATC*, 2014.

[20] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan, "Meteor shower: A reliable stream processing system for commodity data centers," in *IPDPS*, 2012.

[21] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005.

[22] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, "Cut me some slack: Latency-aware live migration for databases," in *EDBT*, 2012.

[23] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson, "Generalized scale independence through incremental precomputation," in *SIGMOD*, 2013.

[24] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," in *VLDB*, 2008.

[25] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," in *VLDB*, 2013.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006.

[27] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally distributed database," in *OSDI*, 2012.

[28] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *NSDI*, 2013.

[29] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian, "Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications," in *SOCC*, 2013.

[30] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *ICDE*, 2005.

[31] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," in *ICDE*, 2008.

[32] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," in *VLDB*, 2008.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[34] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *SOSP*, 2011.

[35] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking main memory oltp recovery," in *ICDE*, 2014.

[36] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*, 2011.