On the impact of log compaction on incrementally checkpointing stateful stream-processing operators

Aris Chronarakis[‡], Antonis Papaioannou[*†] and Kostas Magoutis[*†]
[*]Institute of Computer Science, Foundation for Research and Technology – Hellas, Heraklion 70013, Greece
[†]Computer Science Department, University of Crete, Heraklion 70013, Greece
[‡]Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece

***Abstract*—Incremental checkpointing (IC) is a fault-tolerance technique used in several stateful distributed stream processing systems. It relies on continuously logging state updates to a remote storage service and periodically compacting the update-log via a background process. We highlight a tradeoff between the intensity of compaction of the IC update-log (and the associated resource overhead) and its impact on recovery time in such systems. We also highlight the control parameters that can be used to adjust this tradeoff in the Apache Samza stream processing system, and demonstrate this tradeoff experimentally.**

***Index Terms*—distributed stream processing, incremental checkpointing, high availability**

I. Introduction

Achieving high availability in distributed stream-processing systems is important due to the stringent uptime demands of Internet services such as Google [1], Tweeter [2], and LinkedIn [3]. Streaming applications often use stateful operators, such as window and join, as part of continuous queries that may accumulate large amounts of state in different forms over time. One way to represent such state is as key-value pairs [4], stored in memory, local disk, and/or remote disk to provide different performance and reliability characteristics.

Approaches to high availability in stream-processing systems [5] include *active standby* (where operator state is actively replicated in another node's memory), *passive standby* (where operator state is replicated in backup storage), and *upstream backup* where recovery is achieved simply by replaying tuples from upstream operators and/or queues, which log tuples until explicitly receiving acknowledgment to drop them. Active standby is expensive in terms of memory requirements, while upstream backup results in high recovery time when reconstructing state requires replay of a large number of tuples.

An instance of the passive backup approach is *checkpoint-rollback* [6], which periodically constructs a checkpoint of each operator's state and stores it in durable storage. The constructed checkpoint could include the entire state, or just the "diff" from the previous checkpoint (*incremental checkpointing*). Periodic full-state checkpointing often relies on complex techniques such as copy-on-write [7] to avoiding freezing the operator during checkpoint production. Incremental checkpointing advances over full-state checkpointing by continously writing incremental state updates, spreading checkpointing overhead over time. On failure, the operator loads its latest checkpoint (all incremental "diffs" that constitute a full image of operator state must be loaded from disk up to the most recent full checkpoint) and then replays messages from the input stream, from the last tuple that has contributed to the last checkpoint at the time of the failure and on. Since a long sequence of incremental checkpoints (which in effect form a log) increase recovery time, traditional approaches bound recovery time by periodically producing a full checkpoint, throwing away previous incremental checkpoints.

Several incremental checkpointing approaches in use in distributed stream processing systems today [3], [8] differ from the traditional approach in that instead of periodically taking a full checkpoint, they apply periodic compaction on the evolving sequence (log) of incremental checkpoints. In this way, the log of incremental checkpoints is continuously maintained by removing deleted and overwritten items, rather than periodically thrown away and replaced by a new full checkpoint. Storing operator state in a local (embedded) key-value store while also asynchronously saving the log of incremental checkpoints to a remote disk combines performance (reading/writing a local store) and reliability aspects (recoving from a local store when possible, or from remote disk in the general case) and is followed in several systems today [3], [8].

While the overhead of periodic full checkpointing in transaction processing systems is generally well understood, the impact of periodic compaction operations on incremental checkpoints of stateful stream-processing operators has not been studied in the past. Frequent compaction would reduce the amount of state that must be loaded from the log of incremental checkpoints, reducing recovery time, however it would also impact performance on the node that performs the compaction. Our main contribution in this paper is in highlighting this tradeoff in a real implementation (Apache Samza) and in demonstrating the practical effects of tuning relevant parameters, the interaction of which might not be immediately predictable. The presented experiments are aimed to motivate future research on exploring mechanisms for automatic tuning of the explored parameters, towards a methodology by which incremental checkpointing can be tuned to a specific operating point of (resource use, recovery time), opting for low recovery time or low resource impact or a level in between.

II. Background

We provide more details on incremental checkpointing through a more detailed description of the experimental platform (Apache Samza) used in this paper. Samza [3] (Figure 1) implements incremental checkpointing by logging state updates to each window[1]. Samza logs state updates to a remotely hosted pub-sub topic implemented by the Kafka log manager [9] called a *changelog*, and simultaneously writes them to a local embedded key-value store (KVS), RocksDB, for fast local access. Logged (incremental) updates to window state may include each tuple entering a window, when all such tuples are retained in the window (to be emitted as a window pane [1] at window-closing time) or the new value of a quantity being maintained for each window (when using an aggregating function, also known as a FoldLeft function). At regular intervals (by default per minute), Samza flushes (commits) the changelog to Kafka and persists the corresponding input-stream offset. Samza obsoletes information previously written to the changelog whenever it writes a new value (an overwrite) or when closing a window (deleting the window state and all tuples in the corresponding window pane). Deleted information does not get removed from the changelog until compaction, thus it increases recovery time as it needs to be loaded and played to recover local KVS state.

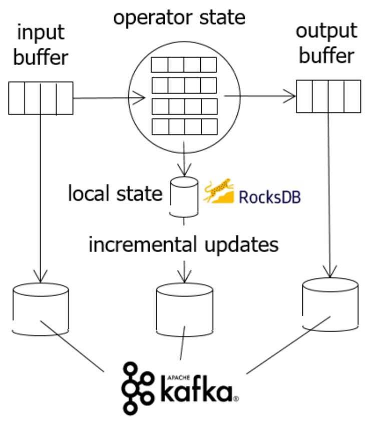

Fig. 1. Incremental checkpointing on local and remote stores (Samza)

In Samza, a window operator supports two options for maintaining accumulated state. In the first option (RetainAll (RA), the default), each emitted result from the operator (window pane) maintains all input tuples that have entered the window. In this option all input tuples must be kept on the store. The internal key used to store window tuples as key-value pairs in this case comprises

- the key of the input tuple (key of the window)
- the window's opening timestamp, and
- a sequence number to identify individual tuples within a window (pane)

There is a second option (FoldLeft function (FLF)), where each tuple contributes to a value (e.g., a counter) maintained in that window, and that value is eventually the emitted result from the operator. In this option, only the counter must be kept on the store, however the procedure to get (read) the previous value in order to update the counter requires flushing the counter to the disk at each read. This is to ensure that what is read by the operator is reliably stored on disk prior to reading it (for consistency after a crash).

[1]A window is a finite chunk of tuples within an infinite stream, used for processing such tuples as a group.

The internal key-value store key used to store the window value in this case is the combination of

- the key of the input tuple (key of the window)
- the window's opening timestamp

Discarding (overwriting or tombstoning) state: We determined that Samza overwrites the single-valued state of a FoldLeft function window with a new version after each update to it (e.g., incrementing a counter). When the FoldLeft function window closes, Samza deletes that state by writing an explicit tombstone (null value for that key). In the case of a regular window pane (where all window tuples are stored through separate keys), closing a window results to deleting each individual key within that window, i.e., writing tombstones (null values) for each individual key of that window pane. Cleaning the Samza *changelog* of deleted state is handled by the Kafka broker(s). The frequency and intensity of compaction is configurable, as described in Section III.

We note that other stream-processing systems implementing incremental checkpointing exhibit similar tradeoffs. Flink, another popular stream-processing system, implements incremental checkpointing by periodically and asynchronously backing up the diff of the set of immutable on-disk structures (termed SSTables) storing RocksDB tables [10] representing operator state, to a remote distributed file system. Additionally, an experimental approach, CEC [11], writes incremental operator-state updates to a parallel file, interspersed with output tuples in the operator's output queue. As in any log where obsolete information piles up over time, these incremental checkpointing systems also involve a form of compaction to reduce the size of the incremental-checkpoints log.

III. Evaluation

Our experimental testbed consists of four servers, each equipped with a Intel® Xeon Bronze® 3106 8-core CPU clocked at 1.70GHz, 16GB DDR4 2666MHz DIMMs, and a 256GB Intel SSD, running Ubuntu Linux 16.04.6 LTS. The nodes are interconnected through a 10Gb/s Dell N4032 switch. The software versions used are Samza version 1.1.0, Kafka version 0.10.1.1, Zookeeper 3.4.3, and Yarn version 2.6.1. A typical Samza/Kafka deployment is shown in Figure 2.

A. Operator-state recovery from changelog

In this experiment we measure the time to reconstruct the local state of an operator in RocksDB from the remote changelog. This is necessary after disk failure, requiring task restart at a new machine (e.g., when the previous host is not available). We used a streaming application featuring a window operator with an input load of 1M, 3M, and 6M tuples (single producer), where each tuple had the same key (and thus routed to a single window). The input stream had one partition

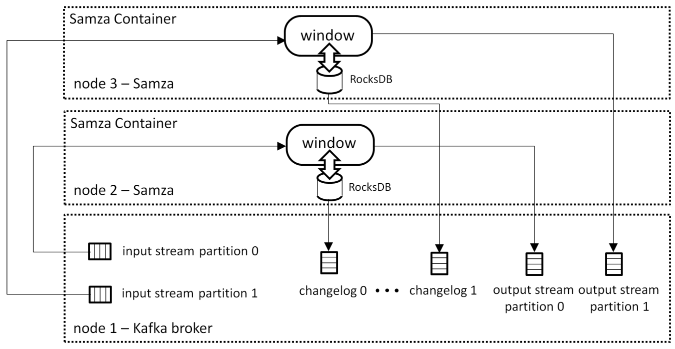

Fig. 2. Typical Samza setup (Kafka broker hosted on node 1)

and thus Samza constructed one task within a single Samza container (on node 2).

We experiment with both options of handling the window state per key, FLF and RA. These two options differ on how tuples are stored (Section II) and on failure handing. With FLF, each update for the same window has the same key on the changelog (overwriting previous versions), whereas in RA each emitted record of window state has a different key. This could play a significant role in restoration since in the first case we need to restore just the latest record of the aggregated value, whereas in the second case, all records that belong to a window state need to be restored. We observed that in the Samza implementation we worked with, log compaction is not enabled by default on the changelog (handled by Kafka broker on node 1).

Normally, after the closing of a window, Samza emits tombstones to the changelog for records contained in the window. In experiments in this section, we use windows with long duration (60 minutes), remaining open for the entire runtime of the experiment (i.e., no closing of windows and emission of tombstones). We experiment with inputs of 1M, 3M, and 6M records with FLF windows (same key for window in changelog records), and similarly with RA windows (different key per changelog record). After the processing of the tuples and forming of the operator state, we kill the container hosting the window and destroy the local store. This has as effect that on restart, the container has no access to its (previously local) state and thus need to restore from the changelog.

We measure the time to restore operator state. More precisely, we measure the time to consume changelog records and rebuild local state in RocksDB, excluding time related to other activities such as creation of threads to consume the changelog, creation of the local store, etc. Our results reported in Table I are averages of 3 runs with negligible standard deviation. Column FLF of Table I depicts results with FLF windows, restoring a changelog of 1-6M records with all records having the same key (Section II). Had log compaction been performed prior to the crash in this case, we would need much less time to restore from the changelog, since compaction would eliminate most records but the last. In the second case (Table I, column RA) all keys on the changelog are distinct.

TABLE I
TIME TO RESTORE WINDOW STATE (IN SECONDS)

Number of tuples \ Window type	FLF	RA
1M	2.6	1.8
3M	8.4	4.4
6M	17.1	9.2

Comparing FLF to RA for the same number of input tuples (the changelog has about the same number of records in both cases), the difference in restore time (8.4s vs. 4.4s for 3M tuples) can be explained by the fact that in FLF we have 3M updates for the same key whereas in RA we have 3M inserts for 3M keys, resulting in different RocksDB per-key costs. The results exhibiting a near linear relationship between restore time and number of input tuples (1-6 million). Restore takes longer for FLF vs. RA windows for the same amount of state in all cases.

Overall, our results indicate that restore time can differ based on changelog size and type of window, demonstrating that it is important to configure log compaction on the changelog. The following section evaluates the impact of specific compaction settings on changelog size and on resource requirements.

B. *Changelog compaction policies: Log size vs. CPU*

Given the direct relationship between changelog size and recovery time, we are interested to evaluate policies for limiting changelog size and their cost. The experiments in this section measure the impact of Kafka log compaction configuration parameters on changelog size and associated CPU usage on the node responsible for log compaction (Kafka broker hosted on node 1).

We perform experiments on windows of small duration (1 second) with 3M input tuples in which tuples had a key chosen uniformly at random from a set of 10 distinct keys. Frequently closing windows result in frequent emission of tombstone tuples on the changelog, one tombstone for each different key. We perform experiments for both FLF and RA window types. The input stream had a single partition and thus Samza constructed one task within a Samza container.

In terms of configuration parameters, we keep the number of compaction threads fixed at two, and vary the dirty-ratio threshold (min.cleanable.dirty.ratio) and time at which the currently active log segment closes, becoming available for compaction (log.segment.ms). The min.cleanable.dirty.ratio parameter indicates the minimum ratio of dirty log to total log for a log to be eligible for compaction, in other words it indicates how frequently log-compaction threads will try to compact the logs of a topic. If there is no log with a dirty ratio over that threshold, the compaction threads sleep for a specific time (default=15,000ms). The log.segment.ms parameter indicates how often Kafka creates a new segment for the log, at which point it becomess active (the segment that receives new records is not eligible for compaction).

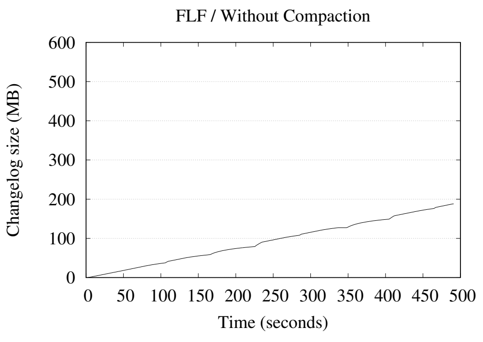

Fig. 3. FLF, No Compaction

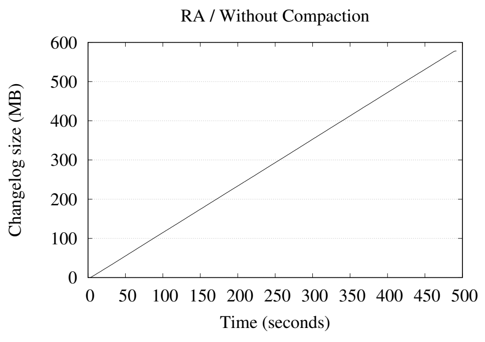

Fig. 4. RA, No Compaction

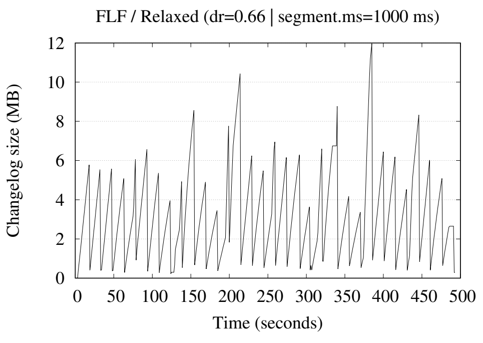

Fig. 5. FLF, Relaxed settings

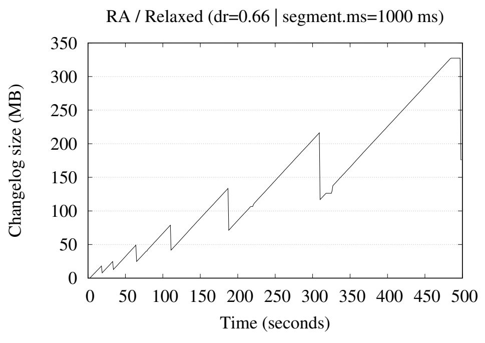

Fig. 6. RA, Relaxed settings

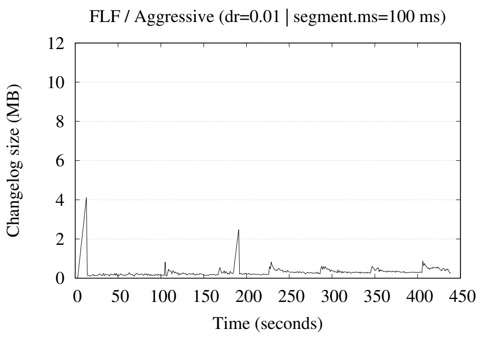

Fig. 7. FLF, Aggressive settings

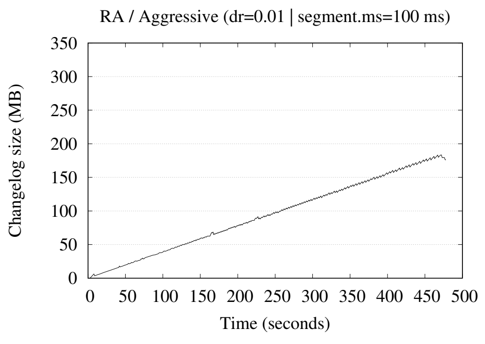

Fig. 8. RA, Aggressive settings

We perform experiments without the activation of log compaction to observe the evolution of changelog size over time and the baseline CPU usage (without log compaction) on the machine otherwise responsible for compaction (node 1). We also performed experiments with the log compaction enabled, more specifically for segment.ms=100ms and segment.ms=1000ms, and for 3 different dirty ratio thresholds (0.01, 0.33, 0.66). These settings are empirically decided to demonstrate different compaction intensity levels in our deployments; they are not otherwise considered special or exhaustive. The higher the segment.ms the longer the active segment will remain locked, and the more uncompacted data will be available to be compacted when discharged from being active. A higher dirty ratio will lead to fewer, heavier compactions. The deletion.retention.ms parameter (at the level of a topic, or log.cleaner.delete.retention.ms at the level of server), the amount of time to retain tombstone markers (deletes) for log compacted topics, is fixed at 100ms.

Starting with the study of FLF windows, Figures 3, 5, and 7 depict the evolution of the changelog topic partition size (in MB) without compaction, using Relaxed settings (dirty ratio: 0.66, segment.ms=1000ms), and Aggressive settings (dirty ratio: 0.01, segment.ms=100ms), respectively. A lower segment.ms makes dirty data available for compaction more frequently, allowing cleaning to proceed, keeping changelog size low. We observe that the Aggressive approach keeps changelog size low most of the time. With the Relaxed approach the maximum changelog size occasionally (but not frequently) reaches 10MB. Without any compaction, changelog size reaches 200MB at the end of experiment (Figure 3). CPU usage on the node that hosts the Kafka broker is about 18% without compaction, about 20% with the Relaxed policy, and about 42% with the Agressive policy. The results show that the Relaxed policy manages to exercise some control over the size of changelog with low overhead (about 2% over the baseline CPU usage without log compaction). However, restore time can still be significant when the changelog reaches large sizes (in the order of 5-10MB), making a case for the Aggressive policy when short recovery times are required.

Moving to the study of RA type windows, Figures 4, 6, and 8 depict the evolution of the changelog topic partition size (in MB) without compaction, and with the Relaxed and Aggressive settings respectively. With RA type windows and with frequently closing windows, we have more tombstones (null-value records, as many as the tuples that entered the window pane) than on the FLF, resulting in a longer changelog. Increasing the dirty ratio results to a larger size for the changelog with much more data to be compacted. We observe that the Aggressive policy manages to keep the size below 200MB for the entire experiment. With Relaxed settings, changelog size exceeds this limit but after a compaction, the size temporarily falls back. Without log compaction, changelog size reaches 600MB the end of the experiment (Figure 4).

Figure 9 summarizes CPU utilization for the node hosting the Kafka broker in all experiments. The CPU (used for regular log management as well as for log compaction) was about 44% with Aggressive compaction, about 20% for Relaxed compaction, and about 18% without compaction, highlighting the increased resource requirements of the Aggressive compaction policy. The results also indicate that spending a little more CPU (about 2%) results to a smaller changelog in contrast with the case of disabled compaction. A general observation is that changelog size is generally higher for RA compared to FLF for the same compaction settings, due to the more tuples being produced in the former case. This indicates that more CPU would normally have to be consumed to achieve the same level of changelog size for RA windows (compared to FLF windows). In our experiments we use the same compaction settings for each policy in the FLF and RA cases, thus it is not surprising that we do not observe a measurable difference in CPU usage between the two (in other words, we would have to set more stringent settings for the same policy for RA windows to achieve the same level of changelog size, as with FLF windows).

Overall, our experiments demonstrate that aggressive compaction settings lead to shorter changelog sizes at the expense of more CPU use at Kafka broker(s). Our experiments also demonstrate that a shorter changelog leads to lower recovery time in restoring operator state, when local recovery is not an option. Given suitable tuning of the configuration parameters controlling compaction, we can achieve specific recovery-time goals by taking advantage of understanding of the relationship between compaction parameters, changelog size, and recovery time (through experiments such as those described in this section). A *global goal* may be dictated by the need to accommodate as many changelog topics as possible within a set of Kafka resources, while achieving the minimum possible recovery time for operator states. The methodology to achieve this would set compaction parameters at their most stringent level feasible without exceeding available (CPU) resources at the broker. A *local goal* could dictate a specific recovery-time objective for a particular task (operator), which would map to appropriate compaction settings to achieve that goal for the specific operator type (FLF, RA). A measurement-based methodology [12] could automate the mapping of goals to control parameters, but a combined approach would be needed to balance local and global goals under resource constraints.

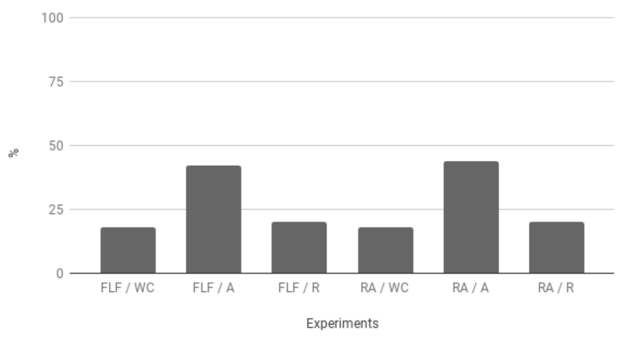

Fig. 9. CPU on node hosting the Kafka broker (A: Aggressive; R: Relaxed; WC: without compaction

IV. Related work

Due to space limitations, we refer the reader to the survey of To *et al.* [13] for background on state management in distributed stream processing systems and on incremental checkpointing techniques in general. There are several incremental checkpointing approaches in use in distributed stream processing systems today [3], [8]. Having discussed Samza [3] and Flink [8] in previous sections, we will relate our work to another previous approach of incremental checkpointing, continuous eventual checkpointing (CEC) [11].

CEC proposed logging partial operator state in the form of *control tuples*, along with regular operator output tuples, in a persistent representation (e.g., a file) of the output queue, as shown in Figure 10. This approach was demonstrated for FLF-type windows and offers the freedom to not capture all updates to the log (since there is always the option to reconstruct some parts of the state through tuple replay at the input). A control tuple atomically records in the log the current number of open windows, the state of a given window, and the timestamp of the input tuple that has triggered this state update. If all open windows are guaranteed to have some record on the log, then recovery is possible by going back in the output log and finding out the latest "footprint" of each open window, then replaying input tuples from the earliest timestamp.

In CEC, recovery of the operator state has to load a set of q tuples (Figure 11) containing a footprint from each of the operator's open windows at the time of the crash. The amount

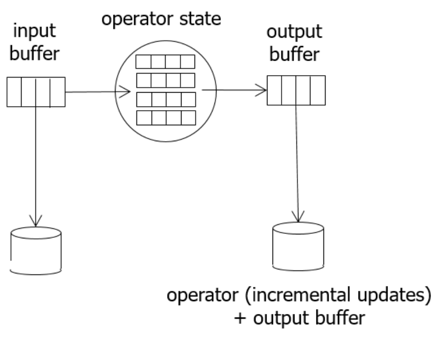

Fig. 10. Continuous eventual checkpointing (CEC)

of tuples to read is determined by the oldest such footprint (w_k in Figure 11). More frequent partial-state logging can reduce the CEC extent (in effect compacting it), thus reducing recovery time, at the expense of a higher overhead [11].

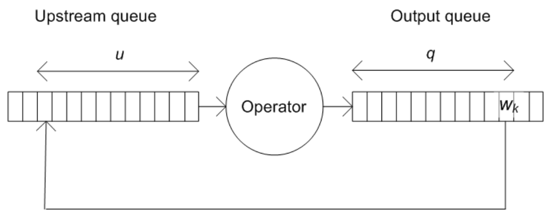

Fig. 11. CEC extent (q) and replay tuples (u) to be read during recovery [11]

Flink relies on a similar scheme, using RocksDB for local recovery when possible, and backing up immutable SSTables of RocksDB to a remote file system. The log-compaction tradeoff described in this paper and applied to Samza as a case study, has also a parallel to this related system: In Flink, obsolete checkpointed state (via explicit deletes or overwrites) is handled by RocksDB, which is scheduling local compaction operations. Restoring operator state from remote uncompacted SSTables means that more information than needed needs to be loaded during state reconstruction, penalizing recovery time. Thus, the investigation performed in this paper has wider implications and can apply to other incremental-checkpointing systems.

V. Conclusions

We note that a tradeoff between checkpointing frequency and recovery time exists even in traditional databases, where frequent checkpointing reduces the size of the log that needs to be replayed. The tradeoff in the case of incremental checkpointing of stateful streaming operators has a different cause, namely the compaction that needs to be performed on a log that may contain overwritten or deleted (tombstoned) tuples, which penalize recovery. We highlighted similarities between other incremental checkpointing approaches, most notably Flink [8] and CEC [11], and pointed out that compaction of an update log is at the core of maintaining size limits and thus bound recovery time in other approaches as well. We believe that establishing relationships (through measurements) between control parameters, log size, and recovery time can be applied to other incremental-checkpointing systems, contributing to controlled systems that can guarantee user goals for recovery time and performance.

Acknowledgment

The authors thankfully acknowledge funding by the EVOLVE H2020 (GA no. 825061) European Commission project and by the Greek Research Technology Development and Innovation Action "RESEARCH - CREATE - INNOVATE", Operational Programme on Competitiveness, Entrepreneurship and Innovation (EπAνEK) 2014-2020 through project ProxiTour (Grant ID: T1EΔK-04819). The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the paper.

References

[1] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[2] S. Kulkarni *et al.*, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, 2015, pp. 239–250.

[3] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Stateful Scalable Stream Processing at LinkedIn," *PVLDB*, vol. 10, no. 12, pp. 1634–1645, 2017.

[4] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management," in *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, NY, USA, 2013, pp. 725–736.

[5] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-Availability Algorithms for Distributed Stream Processing," in *Proc. of 21st International Conference on Data Engineering*, ser. ICDE '05, Washington, DC, USA, 2005, pp. 779–790.

[6] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 9 2002.

[7] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 574–585, Aug. 2008.

[8] S. Richter and C. Ward, "Managing Large State in Apache Flink: An Intro to Incremental Checkpointing," https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html, accessed: 2019-07-03.

[9] J. Kreps, N. Narkhede *et al.*, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, p. 1–7.

[10] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015.

[11] Z. Sebepou and K. Magoutis, "CEC: Continuous eventual checkpointing for data stream processing operators," in *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, 2011, pp. 145–156.

[12] F. Karniavoura and K. Magoutis, "A Measurement-Based Approach to Performance Prediction in NoSQL Systems," in *Proc. of 25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2017, Banff, AB, Canada, September 20-22, 2017*, 2017, pp. 255–262.

[13] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, no. 6, pp. 847–872, Dec. 2018.