A Study of Incremental Checkpointing in Distributed Stream Processing Systems

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Aristidis Chronarakis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN COMPUTER SYSTEMS

University of Ioannina
2019

Examining Committee:

- **Kostas Magoutis**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Vassilios V. Dimakopoulos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

Dedicated to my family.

Acknowledgements

I would like to thank my advisor Prof. Kostas Magoutis for his guidance and support throughout my studies on the department, from the undergraduate level till the graduate.

Special thanks to Prof. Vassilios Dimakopoulos and Prof. Evaggelia Pitoura for their participation as members of the examination committee.

Finally, I would like to thank my family for the support and my friends for all the good moments we spent.

Table of Contents

List of Figures

Abstract

Aristidis Chronarakis, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, 2019.
A Study of Incremental Checkpointing in Distributed Stream Processing Systems.
Advisor: Kostas Magoutis, Assistant Professor.

Cost-efficient fault-tolerance approaches for distributed stream processing systems rely on state checkpointing to recover continuous queries featuring stateful operators after a crash. Incremental checkpointing reduces the overhead of state checkpointing by continuously logging state updates in an incremental fashion. This thesis conducts an experimental study of incremental checkpointing in a distributed stream processing system, focusing on the performance and recovery-time characteristics as well as tradeoffs in this approach. The experimental analysis is supported by load-generating tools and benchmarks featuring stateful operators such as aggregate and join, developed in the context of this thesis. Experimental results validate the low overhead of incremental checkpointing and expose a recovery-time vs. compaction-cost tradeoff that allows tuning the system to the desired performance-availability operating point.

Εκτεταμενη Περιληψη

Αριστείδης Χροναράκης, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, 2019.
Μελέτη προοδευτικής παραγωγής σημείων ελέγχου στην κατανεμημένη επεξεργασία ροών δεδομένων.
Επιβλέπων: Κωνσταντίνος Μαγκούτης, Επίκουρος Καθηγητής.

Τα συστήματα επεξεργασίας ροών δεδομένων βασίζονται σε τελεστές οι οποίοι υπολογίζουν ενδιάμεσα αποτελέσματα ενός μεγαλύτερου υπολογισμού. Καθώς τα ενδιάμεσα αποτελέσματα μπορεί να συσσωρεύονται επί μεγάλα χρονικά διαστήματα, τα συστήματα επεξεργασίας ροών δεδομένων πρέπει να προσφέρουν ανοχή σε σφάλματα και ανάκαμψη της αποθηκευμένες κατάστασης. Οικονομικά αποδοτικές προσεγγίσεις ανοχής σε σφάλματα σε συστήματα επεξεργασίας ροών δεδομένων βασίζονται στην δημιουργία σημείων ελέγχου της κατάστασης του συνόλου των τελεστών που υλοποιούν την συνεχή επεξεργασία των δεδομένων. Μια κλασική τεχνική βασίζεται στην ιδέα της περιοδικής καταγραφής όλης της κατάστασης των τελεστών, η οποία όμως δεν ενδείκνυται σε περιπτώσεις που πρέπει να καταγράψουμε μεγάλο όγκο κατάστασης. Εναλλακτικά μια τεχνική που έχει προταθεί βασίζεται στη περιοδική δημιούργια σημείων ελέγχου που περιέχουν μόνο τις διαφορές σε σχέση με το προηγούμενο σημείο ελέγχου, με την προϋπόθεση ότι ο σχηματισμός της κατάστασης του συνόλου των τελεστών χρειάζεται το συνδυασμο επιμέρους σημείων ελέγχου. Μια τρίτη τεχνική βασίζεται στη προοδευτική παραγωγή σημείων ελέγχου με συνεχή καταγραφή των αλλαγών κατάστασης. Η τεχνική αυτή έχει αποδειχθεί ότι μπορεί να μειώσει την επιβάρυνση στην απόδοση κατά τη διαδικασία παραγωγής τους. Η παρούσα διπλωματική εργασία διεξάγει μια πειραματική μελέτη της προοδευτικής παραγωγής σημείων ελέγχου στο κατανεμημένο σύστημα επεξεργασίας ροών δεδομένων Apache Samza το οποίο κάνει χρήση ενός τοπικού και ενός απομακρυσμένου μέσου για την αποθήκευση της κατάστασης του συνόλου των τε-

λεστών. Μάλιστα εστιάζουμε στα χαρακτηριστικά απόδοσης σχετικά με το τοπικό μέσο αλλα και στο χρόνο ανάκτησης απο το το απομακρυσμένο μέσο σε περιπτώσεις που το τοπικό μέσο δεν είναι διαθέσιμο. Για να υποστηριχθεί η πειραματική μελέτη αναπτύχθηκαν εργαλεία παραγωγής συνθετικού φόρτου και τυπικές εφαρμογές βασισμένες σε βασικούς τελεστές συσσώρευσης κατάστασης όπως οι aggregate και join. Τα πειραματικά αποτελέσματα υποδεικνύουν χαμηλή επιβάρυνση των προοδευτικών σημείων ελέγχου στην απόδοση του συστήματος και αναδεικνύουν την σχέση μεταξύ του κόστους συμπίεσης της δομής των σημείων ελέγχου και του χρόνου ανάκαμψης, η οποία επιτρέπει ρύθμιση του συστήματος στο επιθυμητό σημείο λειτουργίας-απόδοσης. Η παρούσα διπλωματική εργασία συγκρίνει θεωρητικά την προσέγγιση που υλοποιεί το Apache Samza με μια προγενέστερη προσέγγιση προοδευτικής παραγωγής σημείων ελέγχου, την continuous eventual checkpointing (CEC), αναδεικνύοντας τις ομοιότητες και διαφορές των δύο προσεγγίσεων. Τέλος στα πλαίσια μελλοντικής δουλείας προτείνονται σε θεωρητικό επίπεδο δύο υλοποιήσεις με την πρώτη να έχει να κάνει με έναν πράκτορα ο οποίος θα βελτιστοποιεί τη λειτουργία της συμπίεσης με βάση περιορισμούς του χρήστη όσον αφορά είτε το επιθυμητό μέγεθος της δομής των σημείων ελέγχου είτε τους επιθυμητούς υπολογιστικούς πόρους που είναι διατεθειμένος να διαθέσει για τη συμπίεση. Η δεύτερη προτεινόμενη υλοποίηση έχει να κάνει με την παροχή εγγυήσεων για την επεξεργασία των δεδομένων ακριβώς μια φορά (exactly once) στο Apache Samza κατά την ανάκαμψη από σφάλματα.

Chapter 1

Introduction

This thesis studies incremental state management in a distributed stream processing setting. *State* can be defined as "a sequence of values in time that contain the intermediate results of a desired computation" [3, 4]. The main example is *operator state*, but there are others (e.g., input or output queues). State takes the form of windows, hashmaps, or sets of key-value pairs (tables) maintained by an operator (or each partition of an operator). Previous work proposed using data structures, such as *key-value pairs* to externalize various types of internal operator state [5], an approach that is followed by several contemporary distributed stream-processing systems. Fault tolerance is typically achieved by periodically checkpointing operator state, and recovery performed by loading checkpointed operator state and replaying tuples from upstream queues from a specific point in the input streams and on.

In this thesis we are focusing on *incremental* state checkpointing, a technique known to reduce checkpointing overhead. Rather than periodically performing full checkpoints of the entire state, known to incur significant overhead to stream-processing performance, incremental checkpointing maintains a continuous log of updates to state. We study the Apache Samza distributed stream-processing system, a robust prototype that implements a sequential *changelog* that can be used to recover the

state of a local key-value store reflecting operator state. Samza is a new (and rapidly evolving) system with little published information about its internal operation.

We are especially interested to contrast and compare the Samza approach to an earlier proposal for incremental state checkpointing, *continuous eventual checkpointing* (CEC) [1], and point out their similarities and differences. CEC was showcased for single-node deployments and aggregation operators. Samza provides the opportunity to project CEC principles to multi-node deployments and to a richer set of stateful operators.

In this thesis, we develop streaming applications and use them for benchmarking and evaluation purposes, taking into account recent work in the field [6]. We also describe a stream-processing application that we designed for a real-world use case featuring the joining of parallel indoor-localization and application-click streams for understanding the interaction of museum visitors with their environment.

Our evaluation metrics for Samza application execution focus on the overhead of incremental checkpointing in streaming performance and of recovery times under different streaming benchmarks under different compaction policies for the changelog.

Finally, we contribute a novel design for achieving *exactly-once* semantics in Samza (a system that currently supports *at-least once* semantics) through propagation of input-tuple information and by leveraging transactional support when logging state updates at the level of its underlying log management system (Apache Kafka) and the idea of an agent that performs automatic tuning of log compaction based on appropriate control parameters.

1.1 Objectives

This thesis has the following objectives

- Gain insight into how the Apache Samza distributed stream processing system performs state-management operations, with special focus on incremental state checkpointing
- Develop streaming analytics involving stateful operators (aggregate and join) and use them to stress-test and benchmark these distributed deployments
- Evaluate performance and reliability aspects of Samza in distributed deploy-

ments

- Contrast and compare Samza to an earlier incremental checkpointing approach, CEC

1.2 Structure of this dissertation

This dissertation consists of 6 chapters. Chapter 2 provides background to distributed stream processing, incremental checkpointing, and Apache Samza. Chapter 3 describes our insight into the Apache Samza internal operations regarding state management and incremental checkpointing, gained through extensive code instrumentation and experimentation. It also describes the stateful streaming applications developed in the context of this thesis and used for stress testing and benchmarking Samza in distributed deployments. Chapter 4 outlines the evaluation of Samza's incremental checkpointing support, the impact of various parameters on performance, and the achieved recovery time. Chapter 5 discusses related work. Chapter 6 summarizes our conclusions and discusses future work.

Chapter 2

Background

2.1 General concepts

Data streams are possibly unbounded append-only sequences of data items (also called tuples) composed of attribute values based on a schema [7]. Figure 2.1 outlines an example where input tuples conform to a schema featuring three fields: time, item ID, item price. Stream processing is performed in a graph of *operators*, each of which can be classified as either *stateless* or *stateful*. An example of a stateless operator is the *filter operator* (φ in Figure 2.1, defined to select all tuples where (item ID=X)). An example of a stateful operator is the *aggregate operator* (Σ in Figure 2.1, defined to compute the average price of each item ID over last hour). Section 2.4.6 provides a comprehensive list of the built-in operators supported by the Samza distributed stream processing system studied in this thesis. An important concept for accumulating state in a stateful operator is the a *window* of tuples, typically defined to store incoming tuples satisfying a certain predicate, e.g., based on the value of a

tuple key field. Windows can be defined either based on a time specification (e.g., window open for a certain amount of time), a count specification (e.g., window open until a certain amount of tuples enters it), or other specification (e.g., session windows in Samza, which close after a certain amount of inactivity). Certain parts of the state, such as output queues or operator state can be stored reliably in durable local or remote storage, as shown in Figure 2.1. The advantage of using local storage is higher performance compared to remote storage. The advantage of remote storage is the ability to fail independently from the node maintaining the state being persisted. The approach of using a remote store (Figure 2.2) is nowadays extended with a combination of local and remote store (Figure 2.3) to combine the advantages of both approaches. An objective of this thesis is the study a specific state checkpointing technique based on incrementally storing state updates to achieve fault tolerance.

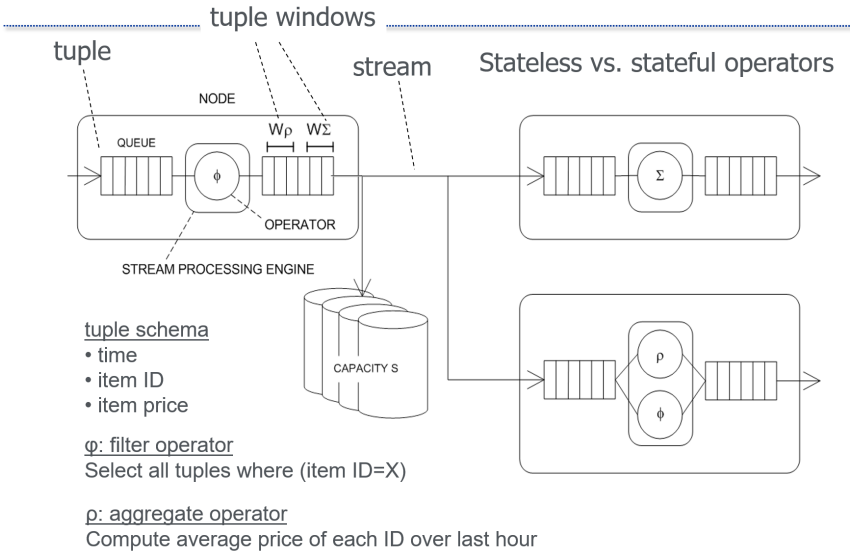

Figure 2.1: General concepts

The first generation of distributed stream processing systems emerged off of research projects in the Stanford STREAM [8] and Aurora/Borealis [9, 10] projects. The stringent demands of Internet companies such as LinkedIn, Facebook, Twitter, etc. gave rise to a new generation of scalable and highly available stream processing systems such as Apache Storm[1], Apache Kafka[2] Streams, Apache Flink[3], Apache

[1]https://storm.apache.org/
[2]https://kafka.apache.org/
[3]https://flink.apache.org/

Spark Streaming[4], Apache Samza[5]. In the following sections we discuss fault-tolerance in stream processing systems through the checkpoint-rollback methodology (Section 2.2), the related (incremental) continuous eventual checkpointing (CEC) methodology (Section 2.3), an overview of Apache Samza (Section 2.4), the system we use to perform our experimental study in this thesis, and finally (Section 2.5) an overview of benchmarking approaches for stream processing systems.

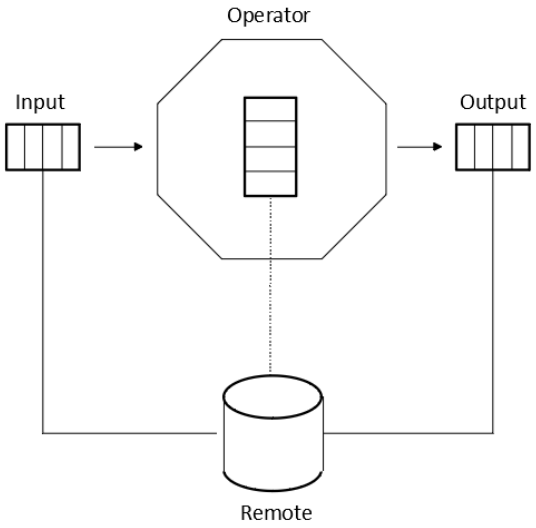

Figure 2.2: Checkpoint on a remote store

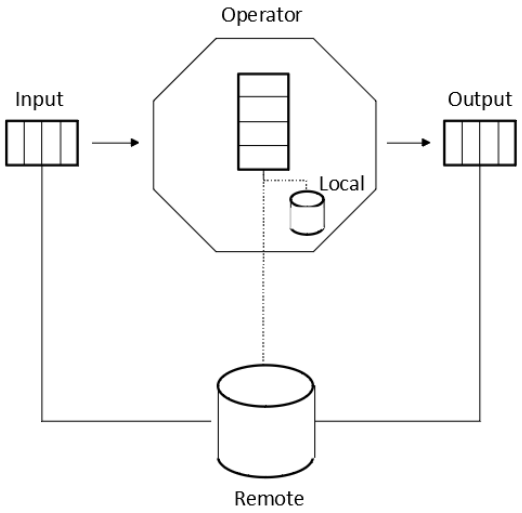

Figure 2.3: Checkpoint on both local and remote stores.

[4]https://spark.apache.org/streaming
[5]https://samza.apache.org/

2.2 Checkpoint-rollback methodology

Approaches to fault-tolerance [11, 12] 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. Passive standby and upstream backup are rollback-recovery approaches [12]. 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 the checkpoint-rollback (CRB) methodology described here. In the following discussion we assume a distributed stream processing setup such as described in Figure 2.2, where operators receive tuples in their input queues, process those tuples possiby accumulating state, and preserve results in their output queues until receiving an acknowledgment from all downstream nodes.

CRB is a common approach used on many fault-tolerant systems. In a stream-processing setting, CRB periodically constructs a checkpoint of each operator's state and stores it in durable storage. The constructed checkpoint could include the entire state (full checkpointing) or the "diff" (delta) from the previous checkpoint for the same state (incremental periodic checkpointing). On a failure, the operator first loads its latest checkpoint and then replays messages from the input stream past the last tuple that has contributed to the latest checkpoint at the time of the failure. The time difference between the checkpoints affects the number of input messages that must be replayed after recovering the latest checkpoint. Full periodic checkpointing involves high overhead in preparing a full checkpoint. To implement CRB efficiently, avoiding freezing the operator during checkpoint production and saving, the use of *copy-on-write* [13] or adaptive checkpointing [14] variants have emerged. However, another approach to reduce checkpointing overhead is *incremental periodic checkpointing*, which computes and stores state deltas (or diffs). Incremental continuous checkpointing goes even further by storing all updates in a continuous log as an evolving checkpoint.

2.3 Continuous eventual checkpointing (CEC)

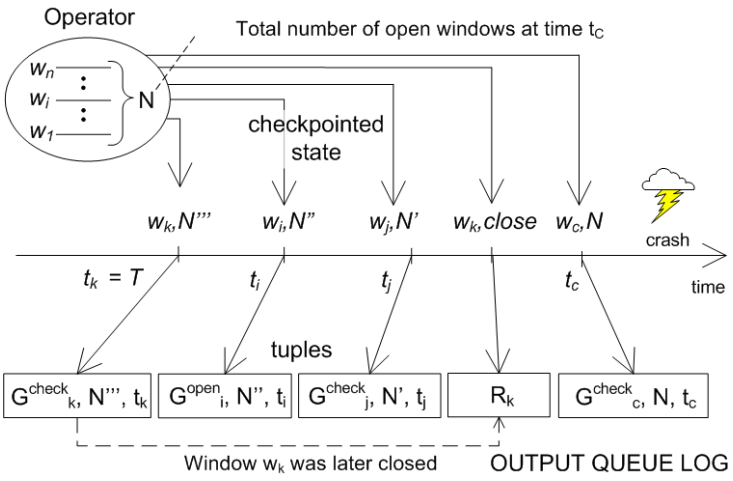

Figure 2.4: CEC operation [1]

Aiming to reduce the checkpointing overhead in CRB, Sebepou and Magoutis [1] proposed *continuous eventual checkpointing* (CEC) [1], an incremental checkpointing approach for fault tolerance on stateful stream operators. The key idea in CEC is to consider a checkpoint as a continuously growing entity. In CEC, operator state is split into parts that can be checkpointed independently. For example when we have a window operator with multiple windows as in Figure 2.4, state state of each window at some point in time is considered partial state that can be checkpointined independently through control tuples (G_k^{check}, N, t) where G^{check} is the state of window w_k, N is the number of open windows, and t is the timestamp of the last input tuple that affected the state of the window. While the production of a G_k^{check} is a matter of policy, the opening of a window is always logged using a G_k^{open} control tuple.

A nice feature of CEC is that partial checkpoints can be sequentially logged along with regular output tuples (such as the tuple R_k in Figure 2.4, rendering previous checkpoints of that window obsolete). After a crash, a recovery process reconstructs the operator state by going over the output stream in reverse and loading in memory the last footprint of all active windows, their number inferred from the last successfully persisted control tuple (G_k^{check} or G_k^{open}) logged. Recovery is then followed by replay of tuples starting at the timestamp of the oldest window checkpoint. Tuple replay affects only windows whose checkpoint time is earlier than the tuple being replayed.

The incremental checkpoint approach pursued in CEC reduces the long checkpoint capture time necessary when performing a full or partial (but large) checkpoints in

traditional CBR. Recovery time can be adjusted depending on how frequently partial checkpoints are performed, highlighting a trade-off between overhead and recovery speed. CEC was evaluated [1] using a continuous query with an aggregate operator in the context of the Borealis distributed stream processing system, validating the low overhead of the mechanism using either long or short duration windows.

2.4 Apache Samza

Apache Samza [15] is a widely used distributed data processing framework characterized by its flexibility and scalability. Samza supports both bounded and unbounded data sources, supporting batch and stream processing with the same data processing model (termed *lamda-less*). Streaming applications process data from one or more data sources in the form of a stream and create output data for one or more data sinks. Key concepts in Samza, including streams, applications, tasks, containers, as well as state and fault-tolerance properties, are described below.

Figure 2.5: Stream application Overview

2.4.1 Streams

Samza can plug into a variety of data sources. It can read one or more stream(s) from a data source if the latter fulfills a set of requirements. First, the source should be able to shard streams in a number of partitions. Each partition must maintain its own order for messages, which cannot be changed, and each message must have its own identifier (offset) indicating the position of the message in its partition. There is no requirement for global ordering of messages across partitions.A stream should be able to receive messages from many producers and serve multiple simultaneous

consumers. Currently Samza is integrated with a variety of consumers such as Kafka, Microsoft Azure, AWS Kinesis, Hadoop Filesystem; and producers, such as Kafka, Microsoft Azure, Hadoop Filesystem, and Elasticsearch.

Figure 2.6: Samza's supported stream

2.4.2 Applications, Tasks, Containers

For distribution and scalability, Samza breaks each application into a number of tasks that can be deployed in different machines. Each task handles a different partition (thus there is not need for global order on the messages from all partitions) and the number of all the tasks is the same with the number of the partitions of the input stream. Thus tasks could be considered as the logical unit of parellelism on Samza. All the tasks that consist the application are being handled by Samza's containers. Each container could possibly hanlde many tasks. Except from the containers that handle the tasks there is a special container (Job Coordinator) responsible for monitor containers' health and also responsible for the assignment of tasks across the available machines on starting and on failure conditions. On distributed deployments most of the time a cluster manager is used for management of these containers like Apache YARN. Each container till is alive has an event loop for checking for input messages and informing about them the tasks, for communicating with producers about the output of handled messages, for the commit procedure etc. Each task periodically (can be configured how often) does a "commit". This procedure writes durably the checkpoints for the latest processed message of the input stream and flush the task's store.

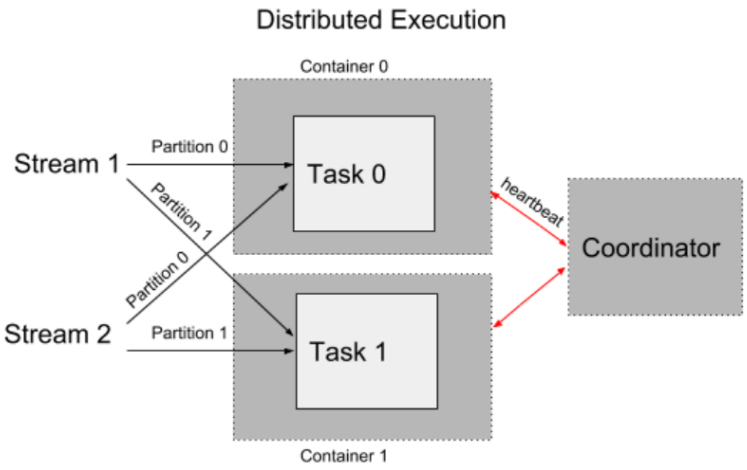

Figure 2.7: Samza's container Overview

2.4.3 State

Samza supports stateless and stateful operators. There are many options for keeping the state. Two common approaches are in-memory checkpointing and the use of an external means, such as a key-value store or a remote database. The in memory checkpointing option has the weakness of the limited memory resources along with the need for checkpointing (full or diff from previous), which make this option unsuitable for large production use cases. The use of an external database or key-value store has the weakness that we should overcome challenges related with perfomance, scalability, isolation, fault-tolerance of the means etc. Samza's developers has realized that in order to provide scalability and good throughput rates their implementation should be based on a local storage solution.

To provide local disk state storage, Samza developers followed a design where each task has its own local state store. The store that is created on the disk for a task, can be managed with different storage engines which strengthen Samza's pluggability and integration features. By default Samza comes with a key-value based store implementation built on RocksDB. This approach has certain benefits. First of all, since each task has its own state store locally, the reads/writes to the store are faster and we have some kind of isolation considering that each task has to deal only with its state store. Also this approach with the collocation of task with a state store makes easier the migration when needed (for failure/recovery or scaling out) since each stateful task needs to has only each state to be able to run on a new machine.

2.4.4 Fault tolerance of stateful applications

The use of the local state store on the disk gives Samza support for fault tolerance under task failures. When a task restarts after a failure, it can be placed on anyone of the available machines. To take advantage of the fact that on conditions like task failure etc., the state store may have survived from the failure, Samza uses a *host-affinity* mechanism [15]. In support of this feature, Samza periodically persists information (on a Kafka stream) about which machine a task is being run on. Local storage however may be lost due to disk failures or may not be available following a crash. To overcome the challenge and offer full fault tolerance support, Samza proposes a mechanism with which the state of a task store can be restored on a different machine after a failure. To achieve this, a special (Kafka) stream called *changelog* is used. Each Task replicates all the changes (writes) of the state store's, on a partition of this changelog stream associated with the specific task. In case of a disk failure where the state is lost or on general machine failure condition, the restarted task on the machine reconstructs its state store using changelog partition information. *Log compaction* is performed by Kafka on the changelog stream to delete records that have been overwritten or deleted, allowing faster state recovery when needed. Finally, the fact that a Kafka stream can be replicated increases durability. Figure 2.8 provides a better understanding of the sate and fault-tolerance mechanism.

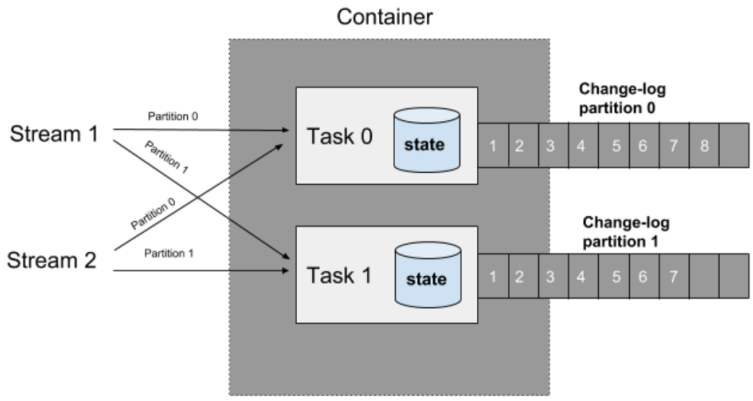

Figure 2.8: State management on Samza

2.4.5 Message (tuple) replay and semantics

When a task is restarted after a failure, besides recovering task state (from RocksDB or the changelog), it should also be possible to recover information about the last

message processed from input streams before the failure so that messages can be replayed from that point on. Samza periodically could persists in Kafka stream the offset of the last processed message for the input streams of a task. This periodic procedure (termed an *incremental checkpoint* in Samza) is performed as part of flushing the state store on the disk in the *task commit* procedure. Commit and state flush are associated and the frequency of task's commit procedure affects the range of input messages that must be replayed after task's restart. Since the last-message-processed and state flushing are not performed atomically, it is possible that messages may be processed more than once, a condition termed *at-least-one* semantics. Figure 2.9 provides a representation of the Samza incremental checkpointing mechanism.

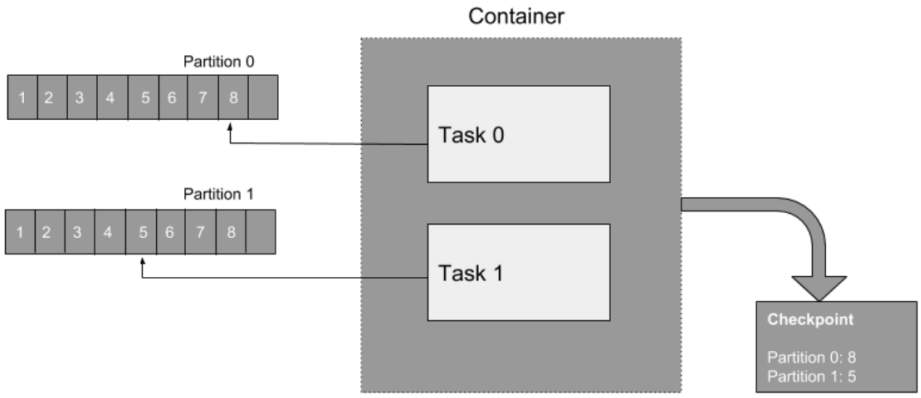

Figure 2.9: Incremental checkpointing

2.4.6 APIs

Samza provides multiple APIs to support application development. It supports two Java APIs, an SQL-related API, the Table API, and an API related with Apache Beam. Both Java-based APIs (High Level, Low Level) are aimed for the development of stream-processing applications. In these two APIs, along with the application logic one has to declare input/output data sources and various configuration parameters. The main difference between the high and the low-level API is on how the processing of the input data is perceived. If it is perceived in term of messages then the Low-Level API should be used since one must describe the logic for each incoming message. If processing is perceived in terms of a stream then the High-Level API should be used. Samza provides built-in support for standard operators and allows the combination of multiple such operators. Application developers can express complicated stream-

processing applications as Directional Acyclic Graphs combining different operators.

The Table API provides easily integration with data sources on which random access by key is supported where the SQL API gives benefit to Samza since someone using this can describe the application logic on SQL format. Beam's goal is to provide the option of running the same developed application with Beam on various processing systems and the provided API comes for the running support of Beam's applications on Samza.

The built-in operators of Samza's High Level API are:

- *Map*, *FlatMap* : Apply a function to transform the data of a stream
- *Filter* : Filter data of an input stream
- *PartitionBy* : Partition input data of a stream based on an attribute (a key) of the data
- *Merge* : Merge two input streams into a single output stream
- *Broadcast* : Broadcast on multiple data processors
- *SendTo*, *Sink* : Send data to output destinations
- *Join* : Join two input data sources based on some common attribute. Although there is a time period associated with the specification of a join, a tuple is emitted each time a result (a match) is found, rather than waiting for that time to expire
- *Window* : Aggregate tuples from an input stream over time. Samza windows can be either tumbling (lasting for a fixed-size time interval, without overlap between different window instances) or sliding window (where there may be overlap between contiguous window instances). An output tuple may be emitted when the window closes, or before that if an *early trigger* condition is defined. The state of a window may comprise all messages (tuples)that enter that window since its opening, or value that accumulates the impact of all such tuples (defined using a FoldLeftFunction)

Overall, Samza features scalability, fault-tolerance, and ease of application development through rich APIs. It has been used in production by many companies such as LinkedIn (where it was originally developed), eBay, Slack, TripAdvisor, and others.

2.5 Benchmarking stream-processing systems

Validating and comparing the various frameworks that support stream processing requires solid benchmarks. In the streaming domain, benchmarks are modeled off of typical streaming applications evaluating various streaming-related metrics. One of the early benchmarks on stream-processing systems, Linear Road [16] simulates a toll system for the motor vehicle expressways of a large metropolitan area and requires maintaining tables of historical data whose update patterns are determined by the values of streaming data. A more recent streaming benchmark is the Yahoo Streaming Benchmark [17] modeling a simple advertising use case, where advertisement events arrive and parsed to a usable format, filtered for the ad view events of interest (removing unneeded fields), adding new fields by joining the event with campaign data stored in a key-value store (Redis). The ad views are finally aggregated by campaign and by time window and stored back into Redis, along with a timestamp to indicate when they are updated. The Yahoo Streaming Benchmark aimed to evaluate and compare Storm, Spark and Flink, using Kafka for data ingest. Later work [6] pointed out that the ingest system and the key-value store are often bottlenecks in this benchmark.

Karimov *et al.* [6] contribute benchmarks for Storm, Flink, and Spark in which data on the operators are not ingested by a broker such as Kafka but instead, created on the fly (likewise, the output data are not stored in Kafka). This design decision has led to a better understanding about the appropriate metrics related with the throughput and latency. It has also led to better accuracy results for the proposed metrics since the streaming framework being tested is separated from the other systems. They have proposed that relative to the latency someone has to separate the latency on processing time and event time latency where the first is the processing time and the second is the processing time plus the time from creation and before being processed. As the throughput they have selected that should be the highest load that can be handled from the operator without a continuously increasing event-time latency.

The load could be handled from benchmark with two different streaming applications, the first one plays the role of an windowed aggregation query on the created data based on a key and the second which plays the role of join query in which created data from two different sources" combined into one output based on the in-

dividual keys. Both of the streaming applications are common processing use cases, used on the real world and the workload that they used on their scenarios for their experiments are also from the real world load.

Chapter 3

Implementation

In this chapter we describe implementation aspects of our work taking a top-down approach. We start by describing in Section 3.1 a set of stateful stream-processing applications that we developed in order to stress distributed deployments of Samza and understand its performance and availability aspects. Since we use Samza as a reference stream processing system, we also want to understand how Samza internally manages operator state and supports recovery in case of failures. Thus in Section 3.2 we describe the results of our empirical (experimental) introspection, through appropriate code instrumentation, of how state management and fault tolerance are implemented internally in Samza. Our in-depth look at the implementation of Samza allows us to compare and contrast it to continuous eventual checkpointing (CEC) in Section 3.3. Finally, in Section 3.4 we briefly describe a real-world application that we developed to support a digital personal tour-guide use case.

3.1 Stateful stream-processing applications

The user-level stateful streaming applications that we developed to support our work, are implemented using the high-level Samza API 2.4.6. These applications cover the main types of stateful operators (aggregate, join) used in streaming applications and are in line with other distributed streaming benchmarks. More specifically, following standard practice in recently proposed benchmarks [6], there are two input streams. The first stream conveys *purchases* and the second one *advertisements*.

Each purchase object on the stream of purchases has 3 attributes:

- the *userId* for the user who makes the purchase
- the *gemPackId* i.e the id for the purchased bundle (gemPack), and finally
- the *price* of this bundle

Each advertisement object on the stream of advertisements (ads) has 2 attributes:

- the *gemPackId* of the gemPack which is being advertised and
- the *userId* which indicates the user who will see this advertisement for this gemPack.

For both of the implementations the built-in stream operators that are used from the Samza's high level API have a set of parameters that can be configured. The full code implementation can be found in Appendix A.

3.1.1 Application 1: Window-based aggregator over an input stream

Using the stream of purchases we implement a windowed aggregation query in order to construct a tumbling window of revenue made from each gemPack. In terms of configuration parameters, the window operator has attributes that indicate when windows close, the window's output tuple structure, etc. Each purchase tuple is a combination of a key (gemPackId) and the full purchase object and thus tuples of the same key are routed to same stream partition and then to the same task and its window operator. If the input tuples have not a key, then they can be explicitly partitioned using the built-in *partitionBy* operator and information from the purchase object. We configure the duration a window remains open. For each incoming purchase on a

window we add up the purchase price to the previous sum for this gemPackId (using a FoldLeftFunction) avoiding the need to store all purchase objects.

When we have a trigger for emitting an output tuple by the window (meaning when the window duration has elapsed), the window emits an output tuple that is mapped to a desired format output tuple using the map operator and then sent to the final output stream from which can be consumed by consumers. Figure 3.1 provides a better understanding of the tumbling window related implementation.

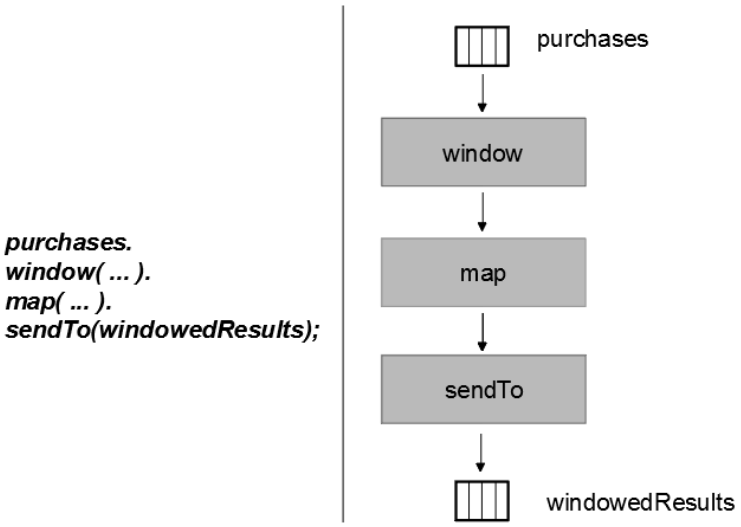

Figure 3.1: Tubling window aggregator

3.1.2 Application 2: Window-based joining of two input streams

Using the advertisements and purchases input streams described above, we joined them on the *combined* (gemPackId, userId) key that both streams have. This join can give an indication of the impact an advertisement campaign has on purchases. In this implementation again the tuples on the purchases and the ads streams are combinations of a key (gemPackId) and (purchase or ad) object and thus there is no need to use the *partitionBy* operator. These two input streams are joined by the Join operator based on the same combined key. On the Join operator we set also the duration of the retention time in which we expect to have a matching. Final, the output tuple from the Join operator, which has the joined result, is being sent to the final output stream for consumption. Figure 3.2 depicts the join-based application implementation over the Samza high-level API.

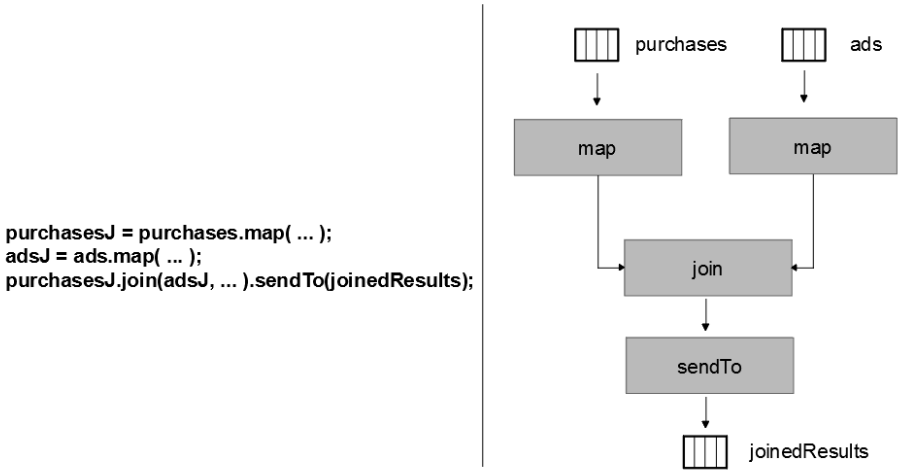

Figure 3.2: The stream-stream join

3.2 Introspection on Samza state management

To support fault tolerance for stateful operators, Samza maintains operator state on local disks to ensure that it can be recovered after failures such as task crash/restart, assuming the failed task can be restarted on the same machine. To ensure that state is recoverable even if this is not possible, Samza introduced the possibility of recovering from a remote persistent mirror of local disk state maintained in Kafka, the *changelog*.

State in Samza is represented using a *key-value store* abstraction. When it is preferrable to store the state on the local disk, Samza uses a RocksDB instance (a storage engine with a key-value interface) to provide fast persistent reads and writes, especially when SSDs are used as backing store.

3.2.1 Maintaining state on disk

When maintaining state on disk, Samza implements a simple cache in front of the disk to benefit from accessing frequent state data without interacting each time with the disk, while also avoiding serialize/deserialize costs. Samza further buffers writes in order to apply them in batches on the underlying store on disk.

Samza achieves these objectives through a store management implementation that consists of multiple stacked layers as seen in Figure 3.3. Storing the state of a window operator starts with a put operation in the Timeseries store (top layer of Figure 3.3)

followed by a put to the Key-value storage engine. Then there is a layer responsible for checking that the state data (key-value pairs) that we want to store are not null (Nullsafe store). After that, data are headed to an LRU cache with configurable size and write-back batch size (Cache store). After the cache there is a layer responsible for data serialization/deserialization (Serialized store), and following that is the Logged store, which performs the writes to the local (RocksDB) and remote (Kafka) stores. Samza disables the RocksDB write-ahead log (WAL) based on the assumption that logging in the Kafka changelog renders the RocksDB WAL redundant [18]. Thus writes are pushed to the local disk device as soon as a write buffer fills up. A write buffer is by default 32MB and RocksDB provisions three such buffers for concurrent use and I/O.

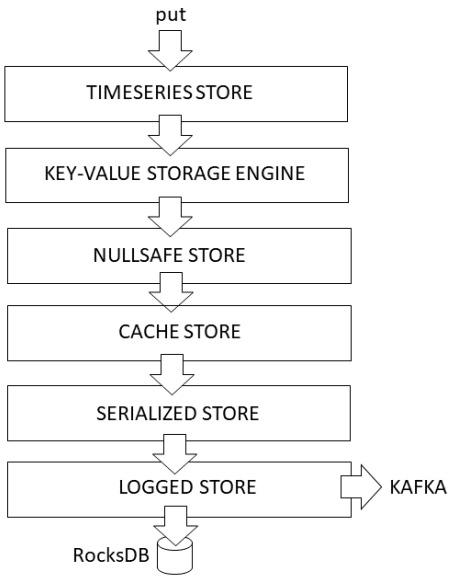

Figure 3.3: Stacked storage layers

An important policy is to decide when cached key-value pairs representing state data are being written to the disk and to the changelog for fault-tolerance. Through code instrumentation and extensive experimentation we determined the following behavior: Each time that some state data are flushed on the disk, they are simultaneously written to the changelog (see Logged store in Figure 3.3). Cached state data are written to the disk and the changelog in three cases:

- When their size exceed the batch size (default: 500 keys)
- When the task periodically performs the commit procedure (explained in Section 2.4.5), part of which flushes cached state to disk, and

- When stream application logic requests this. In this case, we examine how stateful operators, such as window and join, could be affected:

3.2.1.1 Window

A window operator supports two options for maintaining accumulated state. To describe them we use the running example of a stream with purchases and an application that wants to group together the purchase objects of the same user in a separate window.

In the first option (the default), each emitted result from the operator (window pane) must *maintain all purchase objects* needed to make the count, as indicated by the trigger type. In this option all purchase objects must be kept on the store[1].

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

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

In the second option (FoldLeftFunction, see Section 3.1.1), each purchase for a specific user can increment a counter in that window, and this counter 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 still *requires flushing the counter to the disk*. 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)[2].

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

- the key of the purchase object
- the window's opening timestamp

An emission from an early trigger (Section 2.4.6) contains the object difference from the previous emission of the same window or all objects since the opening of

[1] Use of batching in this case is likely to result to less frequent but larger writes to disk

[2] Use of batching should not have an effect in this case

the window. In case of a *discarding window*, Samza discards old messages from the store after the trigger, or else (in an *accumulating window*) it retains older tuples.

3.2.1.1.1 Discarding (overwriting or tombstoning) state

Through experimental observation, we determined that Samza overwrites the single-valued state of a FoldLeftFunction window with a new version after each update to it (e.g., incrementing a counter). When the FoldLeftFunction 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.

3.2.1.2 Join

A join of two (or more) streams identifies related events in two different streams close in time, and combines them into a single output event. The user can set a maximum period of time over which related events in different streams can be spread out. To perform a join between streams, a streaming job needs to buffer events for the time window considered. The join operator buffers events in Samza's state store (using the TTL feature, described further in evaluation Section 4.6), which supports buffering more messages than can fit in memory, and also automatically erases events when the stated TTL expires. The efficiency of multi-stream joins has been studied by Zhuang *et al*. [19]. We summarize this work in Section 5.2.

3.3 Comparison between Samza and CEC on fault-tolerance

The Samza changelog approach [15] features similarities and differences compared to CEC [1] (Section 2.3). Recall that in CEC, partial checkpoints of individual window state (similar to the single-value accumulated state maintained per-window by Samza's `LeftFoldFunction` feature) in an operator are continuously being appended (durably) in an operator's output log, interspersed with standard operator output. In CEC (Figure 3.4), recovery typically has to read q tuples (the *extent*) representing parts of operator state from the output queue, and then replay u tuples from the

input queue to fully bring operator state to a consistent state[3].

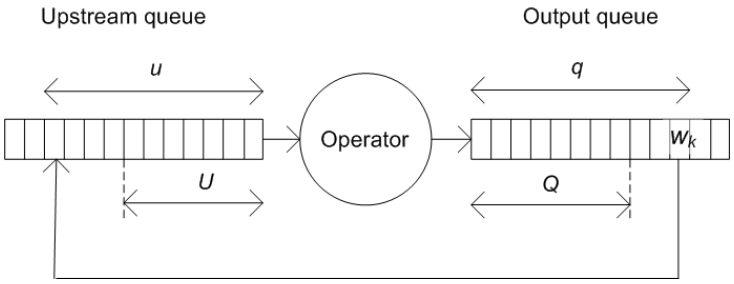

Figure 3.4: CEC extent size (q tuples) [1]

3.3.1 Similarities

The first similarity is that both approaches serialize state updates in a log (output stream in CEC, changelog in Samza) and can restore operator state from there. After a failure, both approaches reconstruct the state by reading the log and replaying a certain range of input tuples (those whose influence on the operator state has not yet been recorded durably in the log). In both cases, the log may contain entries that have been deleted through later activity, such as overwrites by more recent versions of state, or explicit deletes, such as when throwing away the state accumulated in the context of a window, after closing it. Both approaches may attempt to reduce the state being maintained (size of changelog in Samza, extent size in CEC), though the approaches to achieve this differ in each case as described in the following section.

Another similarity is that in both approaches each operator maintains its own log and recontructs state from it. In particular, Samza maintains a separate changelog topic per task.

3.3.2 Differences

A difference between Samza and CEC is that Samza additionally stores state in a local key-value store (an embedded instance of RocksDB), whereas the (remote) log is the sole checkpoint store used in CEC. Another difference is that in Samza, each operator update potentially leads to a changelog append, whereas in CEC the frequency of producing partial checkpoints may be configured to balance its costs and benefits.

[3]U, Q refer to control targets in previous work [1] and are not of interest in this thesis.

Another important difference between CEC and Samza is the approach they follow to keep checkpoint size small:

- CEC reduces the size of the extent (q in Figure 3.4) by performing more frequent partial checkpoints: As the extent is laid out sequentially up to the oldest partial window state (w_k in Figure 3.4, last of the N windows that must be restored), CEC can reduce the size of the extent by performing a newer checkpoint of w_k (the number of active/open windows is also atomically stored). A CEC extent may still contain obsolete tuples (e.g. overwritten state), that should be skipped when loading the checkpoint into memory.
- Samza, on the other hand, separates checkpoint tuples from output tuples and stores the former in a changelog. Samza may occassionaly perform log-compaction (supported by Kafka) on the changelog, reducing the amount of overwritten or deleted state (Figure 3.5) it goes over during recovery.

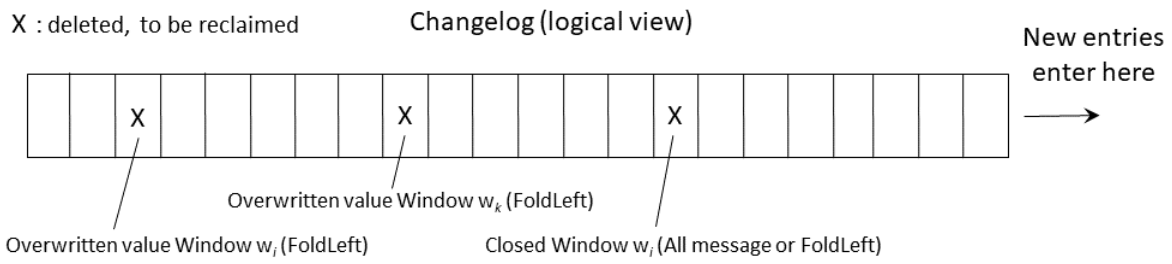

Figure 3.5: Samza changelog contents

3.3.3 Kafka compaction details

Samza is faced with a tradeoff in how aggressively to compact changelogs. Changelogs are partitioned (since there is a separate changelog partition for each partition task of an operator) and each partition has a number of segments, which contain records with keys and values. Each log has a *head*, which the not-yet-compacted portion of the log containing new appended records, and a *tail*, which is the part of the log that gets compacted (records there maintain their original offset, but may be removed if a tombstone or a more recent version of their key has been written to the log).

The Kafka broker compacts logs through a configurable number of cleaner threads, whose purpose is to search for the log with the highest *dirty ratio* (ratio of log head over log tail, that is percentage of new uncompacted over previously compacted records)

and, if that exceeds a configurable threshold, clean the log. Otherwise, cleaner threads sleep for a configurable period of time. The higher the number of threads, the lower the dirty-ratio threshold, and the shorter the sleep time, the more aggresive is the compaction policy. The tradeoff is between compaction-cost overhead and time to restore operator state.

In Section 4.5 of our evaluation, we demonstrate the impact of these Kafka configuration parameters in adjusting changelog size, and through it, recovery time, and point out that Samza could utilize these knobs in achieving explicity recovery-time targets, similar to how this was achieved in CEC through adjusting the frequency of writing window state to the log [1].

3.4 A real-world use case

Another contribution of this thesis is the design and implementation of a real-world use case. Samza is used as part of a bigger platform for personalized touring, Proxitour. The role of Samza is to detect events of interest, and feed them back to the ProxiTour platform for intelligent context-sensitive interaction with users. The real-world use case is based on the common business logic pageview-adclick pattern with which someone could correlate user clicks on advertisements which are represented on the adclick stream, with the users' views on pages that contain these advertisements from the pageview stream. Proxitour's suggestions to visitors for touring (pointers and directions on the space) play the role of the pageviews, where visitors' movements on the space after the view of the suggestion, play the role of the adclicks. The value of this use case is that could monitor the impact of the suggestions on the users, feed follow-on real-time interaction or feed a machine learning related sub-system. The implementation of the use-case is based on the stream-stream join operator of Samza's high level api, in which we join the two input streams to generate output tuples with combined information from the two streams.

Chapter 4

Evaluation

In this chapter we describe the evaluation of incremental checkpointing in Samza, focusing first on the impact of state handling on performance with an SSD backing store, for the stateful window operator using FoldLeftFunction (FLF) and retainALL (RA) type windows. We also evaluate the impact of changelog size on restore time, as well as the impact of compaction policy to changelog size and CPU requirements of maintaining a bound on size, for FLF and RA window types.

4.1 Experimental setup

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.

4.2 Applications and deployment topologies

Figure 4.1 depicts the physical layout of the aggregation benchmark based on the stateful window operator, mapping Samza elements to 3 physical nodes.

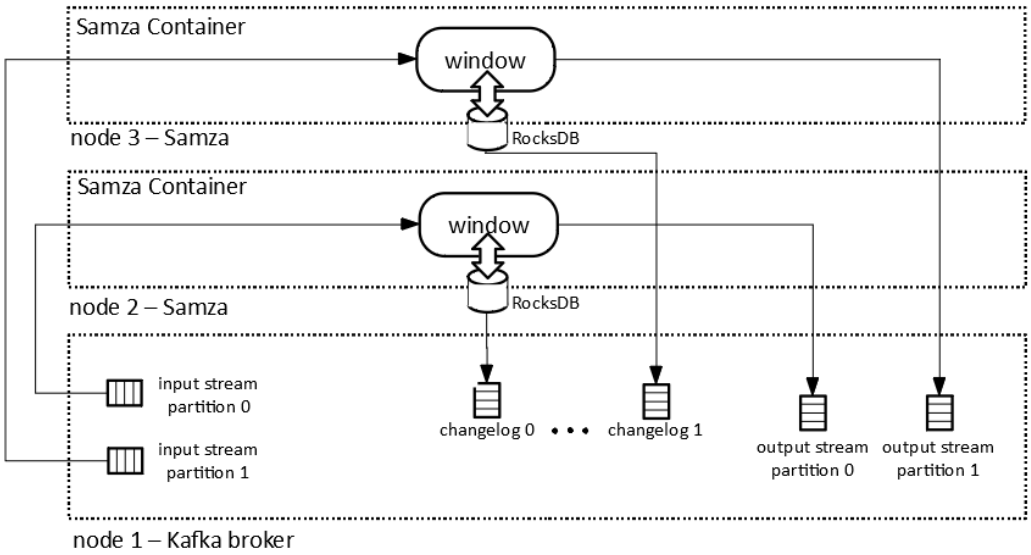

Figure 4.1: Experimental setup for aggregation benchmark

Figure 4.2 depicts the physical layout of the stream-stream join benchmark and mapping to 3 physical nodes. These deployments are meant to depict general deployment principles; some of our experiments involve different deployments.

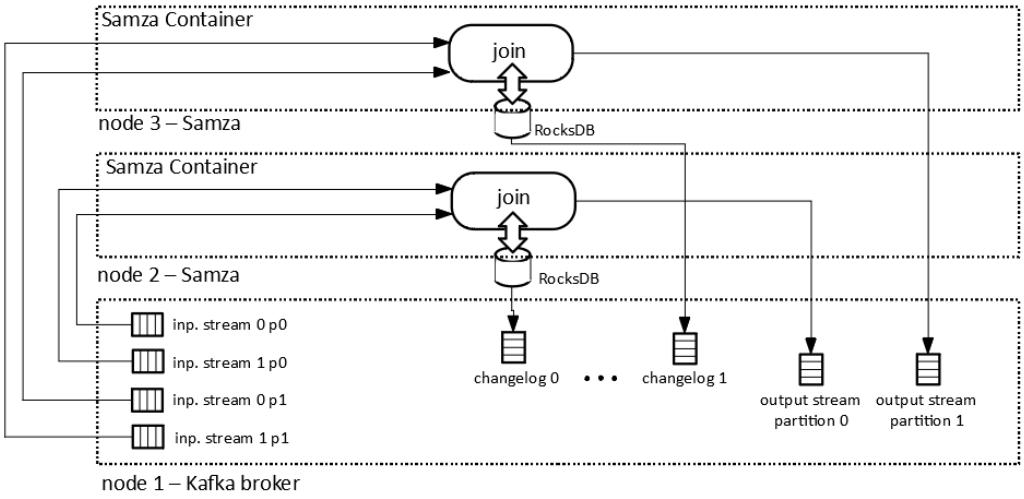

Figure 4.2: Experimental setup for stream-stream join benchmark

The software versions used are Samza version 1.1.0 (latest stable version at the time of this work), Kafka version 0.10.1.1, Zookeeper 3.4.3, and Yarn version 2.6.1.

4.3 Incremental checkpointing I/O activity on a local store

In this experiment we observe write I/O activity to the SSD backing store and CPU usage during experiments[1]. We start with the window aggregation application (Section 3.1.1) for both FLF and RA window operator options. The input stream had two partitions and thus Samza constructed two tasks. Each of these tasks belonged to a different Samza container and these two containers were located on the same machine (node02). We fed the application with a moderate load of a single source emitting a total of 3 million tuples or a heavier workload of 3 concurrent sources emitting a total of 9 million tuples. Each tuple has a key whose value is drawn uniformly at random from a set of 10 distinct keys. Window duration is one second.

We load our streaming application on Samza and monitor I/O activity at the backing store for 1200 seconds (without load). After about 180 seconds we start the feed of the topics from which our streaming application consumes tuples. The feeding lasts about 8 minutes. From the end of processing of the tuples until the end of the 20-minute period we monitor only the baseline I/O activity, which is mainly the result

[1]The changelog is enabled in these experiments. Given that I/O is duplicated to both the local store and the changelog, we expect similar I/O activity towards the changelog too.

of periodic commits.

Figures 4.3 and 4.4 depict write I/O activity (write throughput in KB/sec) at the backing store for the single source and the 3 concurrent sources, for an FLF type window. Figures 4.5 and Figure 4.6 depict write I/O activity for RA type windows. Write activity exhibits clear periodicity with spikes during flushes (commits) by the window operators, increasing in size with heavier load and when using RA-type windows, which produce more writes with tombstones when windows close. The total number of bytes written to the backing store in the case of 1 producer was about 4.7MB for FLF and 28.6MB for RA. For 3 producers, it was about 5.6MB for FLF and 86.4MB for RA.

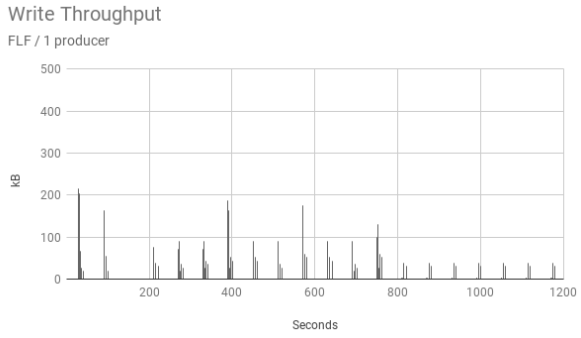

Figure 4.3: FLF window, 1 producer

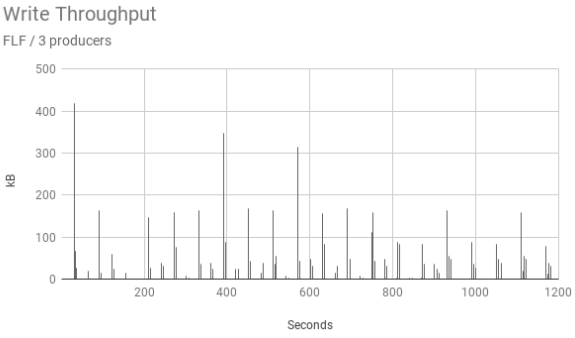

Figure 4.4: FLF window, 3 producers

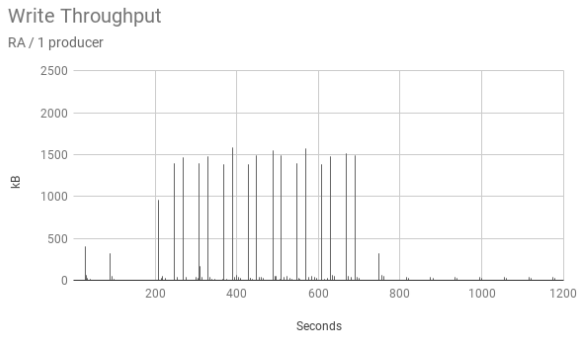

Figure 4.5: RA window, 1 producer

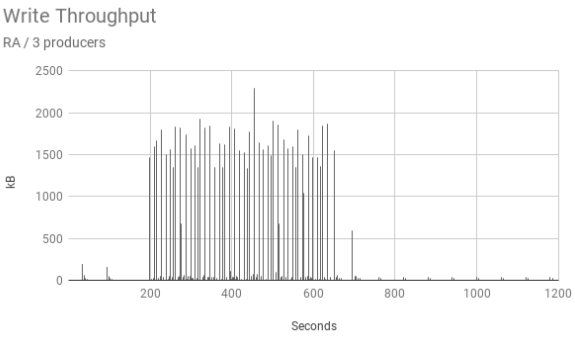

Figure 4.6: RA window, 3 producers

Table 4.1 depicts CPU busy on the machine that hosts the application using FLF and RA type windows, with 1 and 3 concurrent sources respectively. CPU needs increase with the higher load as expected. Final RA-type windows seem to consume less CPU in comparison with the FLF-type.

Table 4.1: CPU busy on the n

	1 producer	3 producers
FLF	15%	22%
RA	4%	10%

4.4 Operator-state recovery from changelog

In this experiment we measure the time to reconstruct the local state of an operator in RocksDB from the changelog. This is necessary after disk or container failure requiring task restart at a new machine, in other words it is not possible to restart the task on the machine it was executing before the failure. We used the window aggregation application and an input load of 3M tuples (single producer), where each tuple had the same key (and thus routed to a single window). The input stream had only one partition and thus Samza constructed one task within a single Samza container on machine node02.

We experiment with both options of handing the window state per key, FLF and RA. These two options differ on how tuples are stored (Section 3.2.1.1) and on failure handing. With FLF, each update on the changelog for the same window has the same key (overwriting previous versions), whereas in RA each emitted record of window state has a different key. This could play a significant role on restoration since in the first case we need to restore the latest record of the aggregated value, but in the second case, all records that belong to a window state need to be restored. It is worth mentioning our observation that in the Samza implementation we worked with, compaction is not enabled by default on the changelog (container on node02, Kafka broker on node01).

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 an input of 3M records with FLF windows (same key for window in changelog records), and 3M records with RA windows (different key per changelog record). After the processing of the tuples and forming of the operator state, we kill the con-

tainer hosting the window and disable access to 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 using a timer at the restore code path within the Samza implementation. We measure precisely 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 Figures 4.7- 4.9 are averages of 3 runs with negligible standard deviation.

Figure 4.7 (left bar, FLF) depicts results with FLF windows, restoring a changelog of 3M records with all records having the same key (Section 3.2.1.1). 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 use case (right bar in Figure 4.7, RA) all keys on the changelog are distinct.

As the changelog has about the same number of records (3M) in both cases, we should need the same time to restore state. The difference (8.41sec vs. 4.42sec) 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.

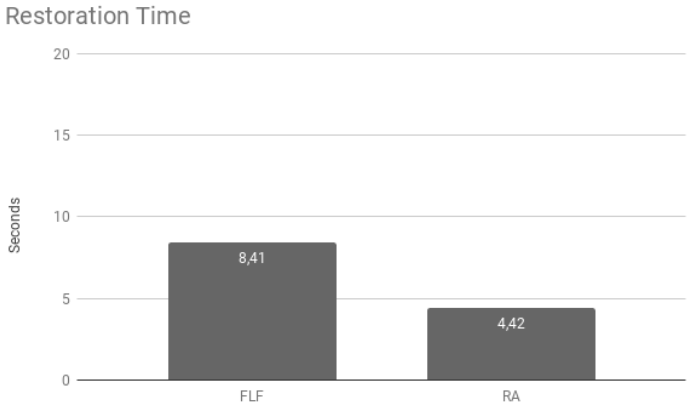

Figure 4.7: Time needed to restore local state (3M input tuples)

We performed additional experiments with lower and high number of tuples as input (1 million and 6 million), with the results exhibiting a near linear relationship between restore time and number of input tuples. Figures 4.8 and 4.9 are for FLF and RA windows respectively. Consistent with our previous measurements, restore

takes longer for FLF vs. RA windows for the same amount of state.

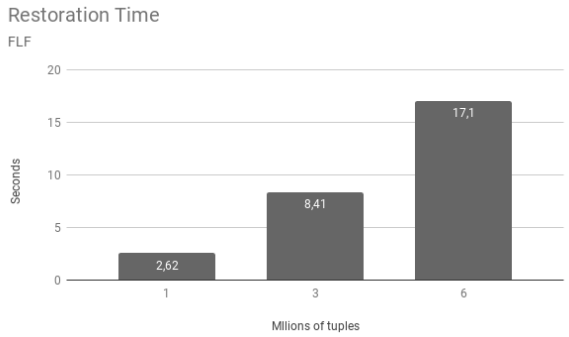

Figure 4.8: Time needed to restore local state for increasing state size (FLF)

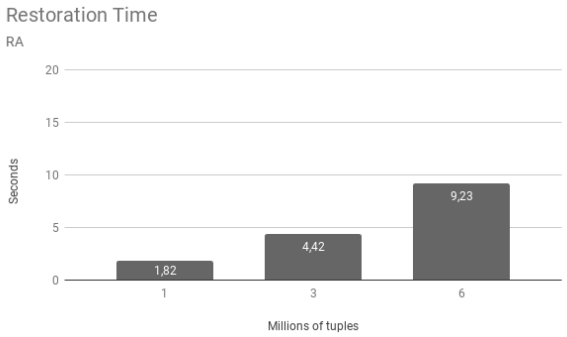

Figure 4.9: Time needed to restore local state for increasing state size (RA)

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.

4.5 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's log compaction configuration parameters on changelog size and associated CPU usage on the node which is responsible for log compaction (the Kafka broker is hosted on node01).

We perform experiments on windows with 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 residing on node02.

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 indicates how frequent the log compaction's threads will try to compact topic's logs. If there is no log with dirty ratio over than threshold then, 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 a new segment will be active (segment that receives new records and not eligible for compaction).

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 (node01). 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). 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 us to fewer compactions but to compactions with a lot of compacted records. 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 retaintombstone markers (deletes) for log compacted topics, is fixed at 100ms.

Starting with the study of FLF windows, Figures 4.10, 4.11 and 4.12 depict the evolution of the changelog topic partition size (in KB) without compaction, using Agressive settings (dirty ratio: 0.01, segment.ms=100ms), and Relaxed settings (dirty ratio: 0.66, segment.ms=1000ms), 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. CPU usage on the node that hosts the Kafka broker is about 18% busy without compaction, about 20% busy with the Relaxed policy, and about 42% busy 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% more) 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).

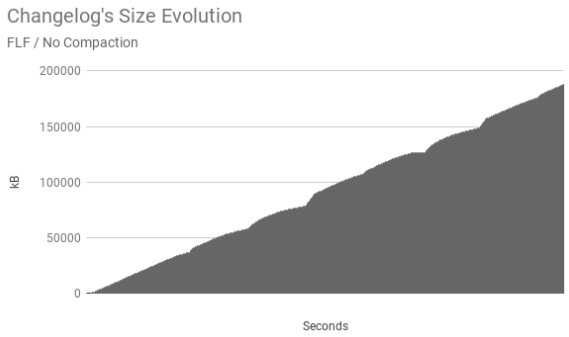

Figure 4.10: FLF, No Compaction

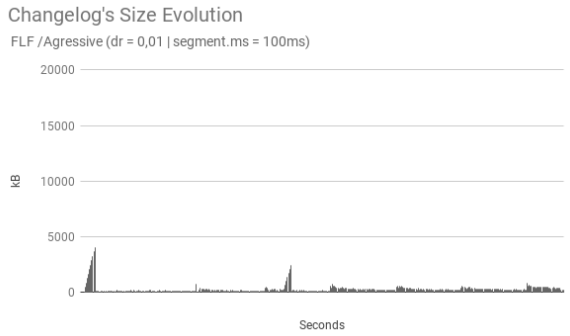

Figure 4.11: FLF, Aggressive settings

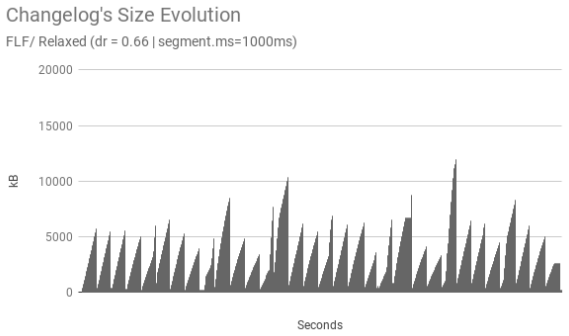

Figure 4.12: FLF, Relaxed settings

Moving to the study of RA type windows, Figures 4.13, 4.14 and 4.15 depict the evolution of the changelog topic partition size (in KB) without compaction, and with the Aggressive and Relaxed 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. As we increase the dirty ratio we get bigger 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 when a compaction take place, the size is falling back under 200MB. Without log compaction, changelog size reaches 600MB the end of the experiment.

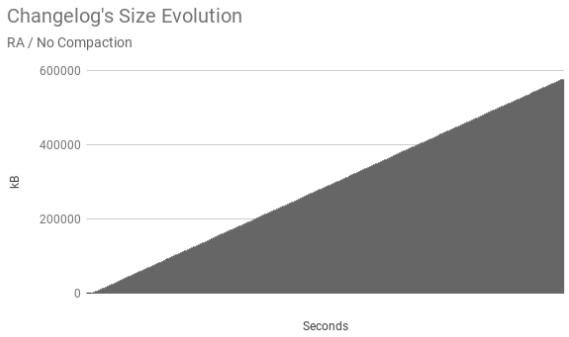

Figure 4.13: RA, No Compaction

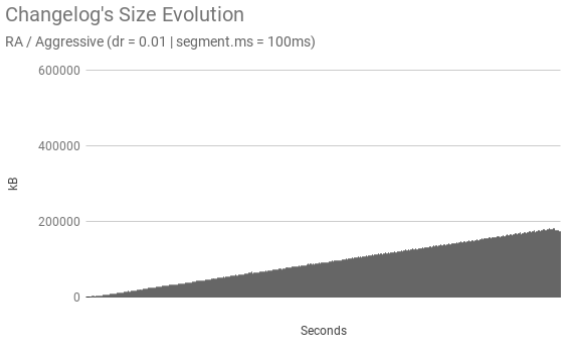

Figure 4.14: RA, Aggressive settings

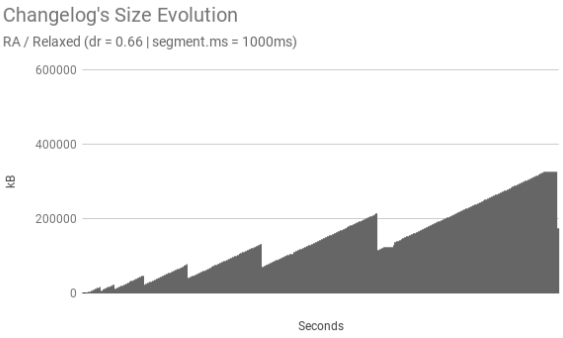

Figure 4.15: RA, Relaxed settings

Figure 4.16 depicts CPU utilization on the node hosting the Kafka broker for all experiments. The CPU on the node hosting the Kafka broker was about 44% (busy) with Aggressive compaction, about 20% (busy) for Relaxed compaction, and about 18% (busy) without compaction, indicating that again more work is being done with the Aggressive compaction policy. The results also indicate that spending a little more CPU (about 2%) we keep a smaller changelog in contrast with the the no-activation of the log compaction approach. 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 have to be consumed for RA windows to achieve the same level of changelog size. 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.

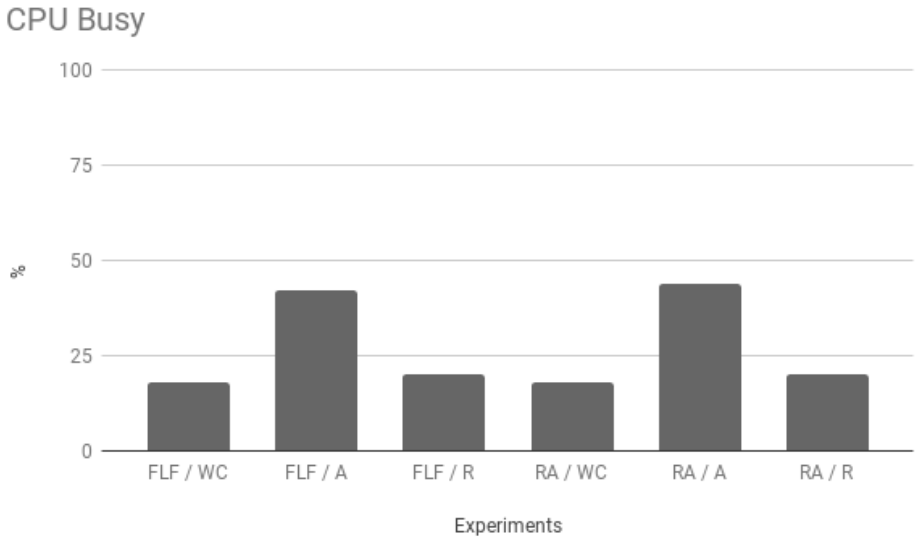

Figure 4.16: CPU busy on the node hosting the Kafka broker

Given suitable tuning of the configuration parameters controlling compaction, we can achieve specific recovery-time goals by taking advantage of understanding (through experiments such as those described in this section) of the relationship between compaction parameters, changelog size, and recovery time.

4.6 Join-based application

State management for the join-based application differs from the aggregate-based application. The join operator specifies a TTL (time to live) during which tuples to be joined are retained in the operator. That TTL maps to underlying capabilities of specific stores, namely the TTL feature of RocksDB, after which records are automatically erased.

We observe that with a TTL of 3min and a checkpoint interval of 1min, tuples that enter the join operator are written to RocksDB and the changelogs at each commit (only the latest version of each key). We do not observe any tombstones (null values), which is explained by the fact that with a TTL, deletes are implicitly performed by the stores.

The implementation we used was throwing a warning that the "changelog is not supported for TTL based stores", thus we decided to not further pursue this case.

4.7 Summary of results

Recovery in an incremental checkpointing system leveraging a local store (such as Samza) is optimized for the case where a failed container/task restarts at the same node where it was previously hosted (host affinity). A slower recovery path is followed when the container/task cannot restart at the same node, and thus local-store state has to be reconstructed from the changelog. In this thesis we determined that recovery time depends on the size of the changelog, as well as on the type of operations the latter carries (FLF vs. RA). To ensure that changelogs do not grow indefinitely leading to very long recovery times, compaction should be configured with appropriate settings to enforce a certain bound on changelog size (and thus restore time) without significantly impacting the CPU of the node(s) hosting the Kafka broker. A restore-time vs. overhead tradeoff (such as investigated in CEC [1]) can also be exploited here, based on the results of this thesis.

Chapter 5

Related Work

The purpose of this chapter is to discuss related work in the field of fault-tolerant stream processing. Some key approaches, such as checkpoint roll-backward and Continuous Eventual Checkpointing were discussed as background material in Sections 2.2 and 2.3 respectively.

5.1 Asynchronous barrier snapshotting

The checkpointing mechanism of Apache Flink builds on the notion of distributed consistent snapshots [20] to achieve exactly-once-processing guarantees. Flink takes a snapshot of the state of operators, including the current position of the input streams at regular intervals. Flink injects *barriers* (control records) into the input streams to separate the stream to tuples whose impact will be recorded in the current checkpoint and those that will be checkpointed in the next checkpoint. Eventually all operators will register a checkpoint of their state, making up a global checkpoint.

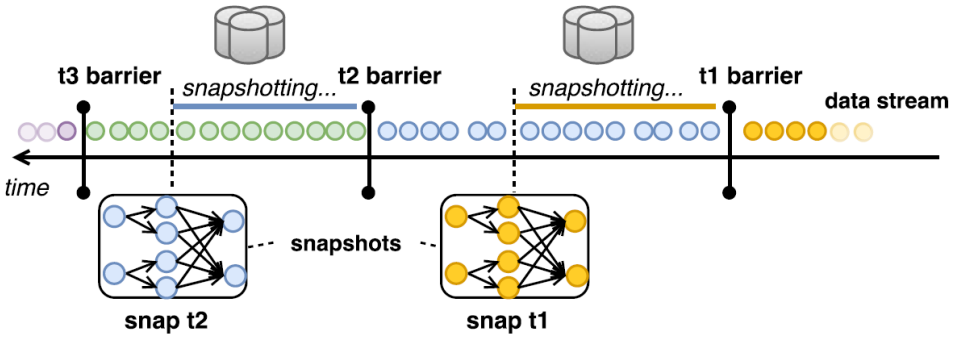

Figure 5.1: Flink's asynchronous barrier snapshotting [2]

Figure 5.1 provides a representation of Flink's asynchronous barrier snapshotting [2]. Snapshot t2 contains all operator states that are the result of consuming all records before t2 barrier. ABS bears certain similarities resemblances to the Chandy-Lamport algorithm for asynchronous distributed snapshots [20], without however the need to checkpoint in-flight records. Recovery from failures reverts all operator states to their respective states taken from the last successful snapshot and restarts the input streams starting from the latest barrier for which there is a snapshot. The maximum amount of tuples replayed after a crash is limited to the amount of input records between two consecutive snapshots. Flink supports incremental checkpointing by using RocksDB as a local store, and asynchronously transferring RocksDB LSM-tree SSTables (immutable parts of the LSM tree) to a remote DFS.

5.2 Multi-stream joining

One of the most common data flow on stream processing as it has previously referred is the join. The use of a join could be a necessity on Big Data handing and handling data from IoT infrastructure in order to combine two input streams into one. Apart from combining two join streams maybe maybe we will need to combine more than two. A real use case scenario is when we have a room and a set for IoT Sensors in order to measure temperature, humidity and pressure every minute and we want to join the individuals streams into one with the the full information about room's conditions on every minute. This is called Multi-Stream Joining and [19] has contributed to this field by making a performance comparison between the two join models using Apache Samza. The first join model is the All-In-One (AIO) and the second the Step-

By-Step (SBS). On the AIO model Samza joins all the streams in one step on the same container with the premise of course that all the corresponding messages from the streams have arrived, for this purpose holds messages on memory buffers and scans if the join could be made. On the SBS model, Samza joins the streams on multiple cascading steps (N) using N containers and on each step we need to hold memory buffers for the subset of the input streams of this step. It is easily understood that in order to have the join result all corresponding messages from of the input streams of this step must to have been arrived.

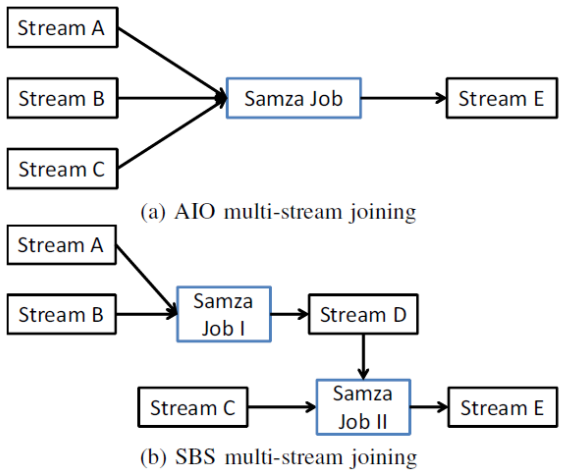

Figure 5.2: Multijoin options

Figure 5.2 presents a graphic representation of the two models. Concerning the comparison between the two models, AIO has easier deployment and administration since we have only one container, with SBS we have the advantage of partial joining results after each step and both of them have similar development complexity. The joining latency i.e the time difference between the time that the first message from one of the input streams arrives and the time in which the desired join result is made, is smaller on the AIO model. Also the AIO model compared with SBS model has a higher CPU usage and smaller (only one) memory footprint due to the fact that needs only one container. The memory footprint includes memory space for the memory buffers which hold the arrived messages from the input streams. In addition they claimed that the joining throughput for the two models is a metric which is derived from deployment's capacity and thus is related with other metrics such as the CPU and memory footprint. Thus they concluded that the best option each time is determined by our needs.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis conducts an investigation of state management and in particular, incremental checkpointing in the context of the Samza stream processing system. Our investigation is driven by an introspection of internal Samza mechanisms and by experimental work. To support our investigation, we develop streaming analytics involving stateful operators (aggregate and join), taking into account modern best practices in benchmarking stream-processing systems, and use them to stress-test and benchmark these distributed deployments. Our experimental investigation evaluates reliability and performance aspects of Samza in distributed deployments. Our findings highlight that changelog-driven recovery is sensitive to the size of the changelog, which in turn depends on log compaction / cleaning policies. Thus, depending on a system's priorities (performance or recovery time), a system can be tuned appropriately. Another contribution of this thesis is a comparison of concepts of incremental checkpointing in Samza to an earlier approach, continuous eventual checkpointing (CEC), highlighting their similarities and differences.

6.2 Future work

6.2.1 An agent for tuning log compaction

Based on the conclusion that a stream-processing system can tuned log compaction depending on specific priorities (optimize for high performance or for rapid recovery), there can be an agent that optimizes log compaction by adjusting parameters that control compaction strategy, and by monitoring changelog size and CPU utilization. Such an agent can take as parameteres specific goals for CPU usage and/or changelog size, which would affect restore time. The agent could automatically select at runtime which is the best possible strategy for the log compaction (in other words, select how aggressive log compaction must be), respecting input (goal) parameters.

6.2.2 Exactly-once processing guarantees

Apache Samza currently supports at-least-once processing. This is because of the way it performs checkpoints: At a specific period of time (when a task ”commits”, by default per 60 seconds), Samza durably records the current offset (last processed tuple) for the input streams of each task into a specific Kafka topic and then asks operators to take also a local checkpoint (flush state into their RocksDB local stores). When there is a failure and a task needs to be restarted, it loads the state from either local or remote store and start consuming tuples from the last checkpointed offset. As there may be a difference between the last checkpointed offset tuple and the offset of the last tuple that contributed to a change in operator state before the failure. This may lead us to record twice the effect of some tuples in operator state after the restarting.

Exactly-once processing can be achieved by atomically storing both the offset and all operator states on the changelog. Since both the input streams and changelog streams are stored in Kafka, it is plausible that exactly-once guarantees can be achieved by writing the Kafka offset along with the last changelog append operations in a single transaction. It would be interesting to investigate whether such an approach has benefits over the distributed snapshot approach supported by Flink (Section 5.1).

Bibliography

[1] 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*, pp. 145–156, 2011.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[3] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st ed., 2004.

[4] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, pp. 847–872, Dec. 2018.

[5] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 725–736, ACM, 2013.

[6] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 1507–1518, 2018.

[7] M. Garofalakis, J. Gehrke, and R. Rastogi, *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Berlin, Heidelberg: Springer-Verlag, 2007.

[8] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation ina data stream management system," Technical Report 2002-41, Stanford InfoLab, 2002.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *CIDR*, (Asilomar, CA, USA), 2005.

[10] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik, *The Aurora and Borealis Stream Processing Engines*, pp. 337–359. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.

[11] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, (Washington, DC, USA), pp. 779–790, IEEE Computer Society, 2005.

[12] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, pp. 375–408, 9 2002.

[13] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endow.*, vol. 1, pp. 574–585, Aug. 2008.

[14] J. Zhou, C. Zhang, H. Tang, J. Wu, and T. Yang, "Programming support and adaptive checkpointing for high-throughput data services with log-based recovery," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pp. 91–100, 2010.

[15] 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.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pp. 480–491, VLDB Endowment, 2004.

[17] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Los Alamitos, CA, USA), pp. 1789–1792, IEEE Computer Society, may 2016.

[18] "SAMZA-543: Disable WAL in RocksDB KV store." `https://jira.apache.org/jira/browse/SAMZA-543`. Accessed: 2019-07-03.

[19] Z. Zhuang, T. Feng, Y. Pan, H. Ramachandra, and B. Sridharan, "Effective multi-stream joining in apache samza framework.," in *BigData Congress* (C. Pu, G. C. Fox, and E. Damiani, eds.), pp. 267–274, IEEE Computer Society, 2016.

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, Feb. 1985.

Appendix A

Code

```
from kafka import KafkaProducer
import random
import numpy as np

INPUT_TOPIC="window-purchases"

# produce json messages
producer = KafkaProducer(bootstrap_servers=['localhost:9092'])
selection = np.random.uniform(1,10,3000000)
for ID in selection:
  ID=int(round(ID))
  purchase_tuple='{"userID": "user'+str(ID)+'", "gemPackID": "gemPack'+str(
    ID)+'", "price":'+str(ID)+'}'
  producer.send(INPUT_TOPIC, key=("gemPackID"+str(ID)), value=
    purchase_tuple)

# block until all async messages are sent
producer.flush()
# configure multiple retries
producer = KafkaProducer(retries=10)
```

Listing A.1: producer.python

```java
package samza.benchmark;
import org.codehaus.jackson.annotate.JsonProperty;

public class Purchase {

  public final String userID;
  public final String gemPackID;
  public final Integer price;

  public Purchase(@JsonProperty("userID") String userID, @JsonProperty("
   gemPackID") String gemPackID, @JsonProperty("price") Integer price) {
    this.userID = userID;
    this.gemPackID = gemPackID;
    this.price = price;
  }
}
```

Listing A.2: Purchase.java

```java
package samza.benchmark;
import org.codehaus.jackson.annotate.JsonProperty;

public class Ad {

  public final String userID;
  public final String gemPackID;

  public Ad(@JsonProperty("userID") String userID, @JsonProperty("gemPackID
   ") String gemPackID) {
    this.userID = userID;
    this.gemPackID = gemPackID;
  }
}
```

Listing A.3: Ad.java

```
package samza.benchmark;
import org.apache.samza.system.SystemStreamMetadata;
import org.apache.samza.application.StreamApplication;
import org.apache.samza.application.descriptors.StreamApplicationDescriptor
    ;
import org.apache.samza.operators.KV;
import org.apache.samza.operators.MessageStream;
import org.apache.samza.operators.OutputStream;
import org.apache.samza.operators.windows.Windows;
import org.apache.samza.system.kafka.descriptors.KafkaInputDescriptor;
import org.apache.samza.system.kafka.descriptors.KafkaOutputDescriptor;
import org.apache.samza.system.kafka.descriptors.KafkaSystemDescriptor;
import org.apache.samza.serializers.IntegerSerde;
import org.apache.samza.serializers.JsonSerdeV2;
import org.apache.samza.serializers.KVSerde;
import org.apache.samza.serializers.StringSerde;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableMap;
import java.time.Duration;
import java.util.List;
import java.util.Map;
import samza.benchmark.Purchase;

public class Window implements StreamApplication {

  @Override
  public void describe(StreamApplicationDescriptor appDescriptor) {
    KafkaSystemDescriptor kafkaSystemDescriptor = new KafkaSystemDescriptor
    ("kafka").withConsumerZkConnect(ImmutableList.of("node01.proxitour:2181"
    )).withProducerBootstrapServers(ImmutableList.of("node01.proxitour:9092"
    )).withDefaultStreamConfigs(ImmutableMap.of("replication.factor","1"));

    KafkaInputDescriptor <KV<String,Purchase>> purchasesDescriptor =
    kafkaSystemDescriptor.getInputDescriptor("window-purchases", KVSerde.of
    (new StringSerde(), new JsonSerdeV2<>(Purchase.class)));
    KafkaOutputDescriptor <Windowed> windowedResultsDescriptor =
    kafkaSystemDescriptor.getOutputDescriptor("window-purchases-output", new
     JsonSerdeV2<>(Windowed.class));

    appDescriptor.withDefaultSystem(kafkaSystemDescriptor);
```

```
33      MessageStream <KV<String, Purchase>> purchases = appDescriptor.
    getInputStream(purchasesDescriptor);
34      OutputStream <Windowed> windowedResults = appDescriptor.getOutputStream
    (windowedResultsDescriptor);
35
36      //FLF
37      purchases.window(Windows.keyedTumblingWindow(kp -> kp.key, Duration.
    ofSeconds(1), () -> 0, (m, prevRevenue) -> prevRevenue + m.getValue().
    price, new StringSerde(), new IntegerSerde()), "revenue").map(windowPane
     -> {return  new Windowed(windowPane.getKey().getKey(), windowPane.
    getMessage());}).sendTo(windowedResults);
38      //RA
39      //purchases.window(Windows.keyedTumblingWindow(kp -> kp.key, Duration.
    ofSeconds(1), new StringSerde(), KVSerde.of(new StringSerde(), new
    JsonSerdeV2<>(Purchase.class))), "revenue").map(windowPane -> {return
    new Windowed(windowPane.getKey().getKey(), windowPane.getMessage().size
    ());}).sendTo(windowedResults);
40
41    }
42
43    static class Windowed {
44      public String gemPackID;
45      public Integer revenue;
46
47      public Windowed(String gemPackID, Integer revenue) {
48        this.gemPackID = gemPackID;
49        this.revenue = revenue;
50      }
51    }
52  }
```

Listing A.4: Window.java is our streaming application for the window aggregation

```
# Application / Job
app.class=samza.benchmark.Window
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=windower
job.container.count=2
job.logged.store.base.dir=/media/localdisk/aris

# YARN
yarn.package.path=http://node01.proxitour:8000/target/${project.artifactId}-${pom.version}-dist.tar.gz
cluster-manager.container.memory.mb=3072
cluster-manager.container.cpu.cores=2

# Metrics
metrics.reporters=snapshot
metrics.reporter.snapshot.class=org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory
metrics.reporter.snapshot.stream=kafka.metrics
metrics.reporter.snapshot.blacklist=^(?!.\*?(?:SamzaContainerMetrics)).\*$
metrics.reporter.snapshot.interval=1
serializers.registry.metrics.class=org.apache.samza.serializers.MetricsSnapshotSerdeFactory
systems.kafka.streams.metrics.samza.msg.serde=metrics
```

Listing A.5: window.properties

```
1  package samza.benchmark;
2  import org.apache.samza.application.StreamApplication;
3  import org.apache.samza.application.descriptors.StreamApplicationDescriptor
       ;
4  import org.apache.samza.operators.KV;
5  import org.apache.samza.operators.MessageStream;
6  import org.apache.samza.operators.OutputStream;
7  import org.apache.samza.operators.functions.JoinFunction;
8  import org.apache.samza.system.kafka.descriptors.KafkaInputDescriptor;
9  import org.apache.samza.system.kafka.descriptors.KafkaOutputDescriptor;
10 import org.apache.samza.system.kafka.descriptors.KafkaSystemDescriptor;
11 import org.apache.samza.serializers.JsonSerdeV2;
12 import org.apache.samza.serializers.KVSerde;
13 import org.apache.samza.serializers.StringSerde;
14 import com.google.common.collect.ImmutableList;
15 import com.google.common.collect.ImmutableMap;
16 import java.time.Duration;
17 import java.util.List;
18 import java.util.Map;
19 import java.io.Serializable;
20 import samza.benchmark.Purchase;
21 import samza.benchmark.Ad;
22
23 public class Join implements StreamApplication, Serializable{
24
25   @Override
26   public void describe(StreamApplicationDescriptor appDescriptor) {
27     KafkaSystemDescriptor kafkaSystemDescriptor = new KafkaSystemDescriptor
       ("kafka").withConsumerZkConnect(ImmutableList.of("node01.proxitour:2181"
       )).withProducerBootstrapServers(ImmutableList.of("node01.proxitour:9092"
       )).withDefaultStreamConfigs(ImmutableMap.of("replication.factor","1"));
28
29     KafkaInputDescriptor <KV<String,Purchase>> purchasesDescriptor =
       kafkaSystemDescriptor.getInputDescriptor("join-purchases", KVSerde.of(
       new StringSerde(), new JsonSerdeV2<>(Purchase.class)));
30     KafkaInputDescriptor <KV<String,Ad>> adsDescriptor =
       kafkaSystemDescriptor.getInputDescriptor("join-ads", KVSerde.of(new
       StringSerde(), new JsonSerdeV2<>(Ad.class)));
31     KafkaOutputDescriptor <Joined> joinedResultsDescriptor =
       kafkaSystemDescriptor.getOutputDescriptor("joined-purchases-ads", new
```

```
    JsonSerdeV2<>(Joined.class));
32
33    appDescriptor.withDefaultSystem(kafkaSystemDescriptor);
34    MessageStream <KV<String,Purchase>> purchases = appDescriptor.
      getInputStream(purchasesDescriptor);
35    MessageStream <KV<String,Ad>> ads = appDescriptor.getInputStream(
      adsDescriptor);
36    OutputStream <Joined> joinedResults = appDescriptor.getOutputStream(
      joinedResultsDescriptor);
37
38    JoinFunction<String, Purchase, Ad, Joined> joinedFunction = new
      JoinFunction<String, Purchase, Ad, Joined>() {
39      @Override
40      public Joined apply(Purchase purchase, Ad ad) {
41        return new Joined(purchase.userID, purchase.gemPackID, purchase.
        price, ad.userID);
42      }
43
44      @Override
45      public String getFirstKey(Purchase purchase) {
46        return purchase.userID + purchase.gemPackID;
47      }
48
49      @Override
50      public String getSecondKey(Ad ad) {
51        return ad.userID + ad.gemPackID;
52      }
53    };
54
55    MessageStream <Purchase> purchasesJ = purchases.map(KV::getValue);
56    MessageStream <Ad> adsJ = ads.map(KV::getValue);
57
58    purchasesJ.join(adsJ, joinedFunction, new StringSerde(), new
      JsonSerdeV2<>(Purchase.class), new JsonSerdeV2<>(Ad.class), Duration.
      ofMinutes(3), "join").sendTo(joinedResults);
59  }
60
61  static class Joined {
62    public String userID;
63    public String gemPackID;
64    public Integer price;
```

```
65      public String userID2;
66
67      public Joined(String userID, String gemPackID, Integer price, String
        userID2) {
68          this.userID = userID;
69          this.gemPackID = gemPackID;
70          this.price = price;
71          this.userID2 = userID2;
72      }
73    }
74  }
```

Listing A.6: Join.java is our streaming application for the stream-stream join

```
# Application / Job
app.class=samza.benchmark.Join
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=joiner
job.container.count=2
job.logged.store.base.dir=/media/localdisk/aris

# YARN
yarn.package.path=http://node01.proxitour:8000/target/${project.artifactId}-${pom.version}-dist.tar.gz
cluster-manager.container.memory.mb=3072
cluster-manager.container.cpu.cores=2

# Metrics
metrics.reporters=snapshot
metrics.reporter.snapshot.class=org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory
metrics.reporter.snapshot.stream=kafka.metrics
metrics.reporter.snapshot.blacklist=^(?!.\*?(?:SamzaContainerMetrics)).\*$
metrics.reporter.snapshot.interval=60
serializers.registry.metrics.class=org.apache.samza.serializers.MetricsSnapshotSerdeFactory
systems.kafka.streams.metrics.samza.msg.serde=metrics
```

Listing A.7: join.properties

Author's Publications

A. Chronarakis, S. Gkouskos, K. Kalampokis, G. Papaioannou, X. Agalliadou, I. Chaldeakis, K. Magoutis, "ProxiTour: A Smart Platform for Personalized Touring", in **Proceedings of the 13th Symposium and Summer School On Service-Oriented Computing (SummerSOC'19)**, June 17 – 23, 2019, Hersonissos, Greece

Short Biography

Aris Chronarakis is a M.Sc. graduate student at the Department of Computer Science and Engineering (CSE) of the University of Ioannina, Greece. Aris received his B.Sc. degree from the CSE Department in 2017. He is a member of the Distributed Systems Group there since 2016. His research interests revolve around distributed systems and scalable fault-tolerant stream-processing systems and applications.