Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka

Guozhang Wang
guozhang@confluent.io
Confluent Inc., USA

Lei Chen
lchen576@bloomberg.net
Bloomberg, USA

Ayusman Dikshit
adikshit@expediagroup.com
Expedia Group, India

Jason Gustafson
jason@confluent.io
Confluent Inc., USA

Boyang Chen
boyang@confluent.io
Confluent Inc., USA

Matthias J. Sax
matthias@confluent.io
Confluent Inc., USA

John Roesler
john@confluent.io
Confluent Inc., USA

Sophie Blee-Goldman
sophie@confluent.io
Confluent Inc., USA

Bruno Cadonna
bruno@confluent.io
Confluent Inc., USA

Apurva Mehta
apurva@confluent.io
Confluent Inc., USA

Varun Madan
vmadan@confluent.io
Confluent Inc., USA

Jun Rao
jun@confluent.io
Confluent Inc., USA

ABSTRACT

An increasingly important system requirement for distributed stream processing applications is to provide strong correctness guarantees under unexpected failures and out-of-order data so that its results can be authoritative (not needing complementary batch results). Although existing systems have put a lot of effort into addressing some specific issues, such as consistency and completeness, how to enable users to make flexible and transparent trade-off decisions among correctness, performance, and cost still remains a practical challenge. Specifically, similar mechanisms are usually applied to tackle both consistency and completeness, which can result in unnecessary performance penalties.

We present Apache Kafka's core design for stream processing, which relies on its persistent log architecture as the storage and inter-processor communication layers to achieve correctness guarantees. Kafka Streams, a scalable stream processing client library in Apache Kafka, defines the processing logic as read-process-write cycles in which all processing state updates and result outputs are captured as log appends. Idempotent and transactional write protocols are utilized to guarantee exactly-once semantics. Furthermore, revision-based speculative processing is employed to emit results as soon as possible while handling out-of-order data. We also demonstrate how Kafka Streams behaves in practice with large-scale deployments and performance insights exhibiting its flexible and low-overhead trade-offs.

CCS CONCEPTS

• **Information systems** → **Stream management**.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457556

KEYWORDS

Stream Processing, Semantics

ACM Reference Format:
Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3448016.3457556

1 INTRODUCTION

Stream processing is gaining tremendous attention in the industry as a new programming paradigm to implement real-time data-driven applications. This paradigm is more collaboration friendly to modern organizations that are composed of vertically separated engineering teams responsible for loosely coupled sub-systems. Compared with traditional data-driven applications that centralize their state in a shared database management system, the asynchronous nature of stream processing allows teams to build their sub-systems as *event-driven* applications that communicate with each other via event-message passing. The sub-systems react to those messages with local application state updates without deeply coupled interactions that usually need to be coordinated and synchronized.

One of the biggest challenges for streaming systems is to provide correctness guarantees for data processing in a distributed environment [6, 11, 20]. Although data stream management systems (DSMS) have been an active research topic in the database community for many years, early efforts were focused on producing real-time, perhaps approximate and lossy, results with high scalability and low latency. As a result, until recently stream processing has still been considered as an auxiliary architecture in addition to the more reliable, periodic batch processing jobs [8].

Modern streaming engines are designed to be authoritative and are no longer treated as an approximation of the truth generated by

batch processing systems. This is accomplished by offering strong correctness guarantees along with fault-tolerance. In these engines, however, conventional stream processing wisdom still holds that users will have to make trade-off decisions among performance, correctness, and cost since they are not all achievable at the same time [3]. For example, if users choose to optimize for correctness, they can specify the stream processing paradigm as batch processing (chunking continuous data streams into discrete finite data sets and process them as micro-batches) [19]; if they choose to optimize for performance, they may use an in-memory streaming framework that generates approximate results and may lose data in the face of failures [15, 39]. How to achieve correct streaming results while not sacrificing too much on performance remains a common concern for stream processing developers.

In this paper, we identify two primary properties that contribute to stream processing correctness guarantees, namely *consistency* in face of failures and *completeness* with out-of-order data. Consistency is a guarantee that a stream processing application can recover from failures to a consistent state such that final results will not contain duplicates or lose any data. In other words, the stream processing application delivers results as if the records were processed exactly once without any failures. Completeness is a guarantee that a stream processing application does not generate incomplete partial outputs as final results even when input stream records may arrive out of order. Completeness requires the application to be able to measure processing progress in time and estimate how complete the emitted output results are corresponding to its input streams.

Figure 1 showcases these two correctness challenges in stream processing. It depicts a simple stateful processor with a single input and a single output stream. Processing state is maintained in a store and accessed by this processor for reads and writes. The input stream contains only three records with timestamps 11, 13, and 12 (Figure 1.a). Suppose that after processing the record with timestamp 11 and updating the state but before the processing is acknowledged on the input stream (denoted by the dotted bar), a failure occurs on the processor (Figure 1.b). Upon recovery, the processor would reprocess the record with timestamp 11 from the input stream and hence double update the state (Figure 1.c), causing inconsistent results. Furthermore, suppose that after the first and second records are processed and the results for times 11 and 13 are emitted respectively, an out-of-order record with an earlier timestamp 12 is received, indicating that previously emitted results are actually not complete up to 12 (Figure 1.d).

Today, a common technique employed in production streaming systems to achieve these two properties is state checkpointing along with output delays. While effective in providing correct results, this mechanism would requires users to make hard trade-off decisions on end-to-end streaming latencies.

Apache Kafka [1] is an open-source distributed event streaming platform that addresses these correctness challenges in a different manner by integrating stream processing with persistent logging. The key idea comes from our experience developing and operating stream processing applications such that, by paying a modest cost to persist streaming data, we can implement more flexible mechanisms aiming for both correctness and performance. More specifically,

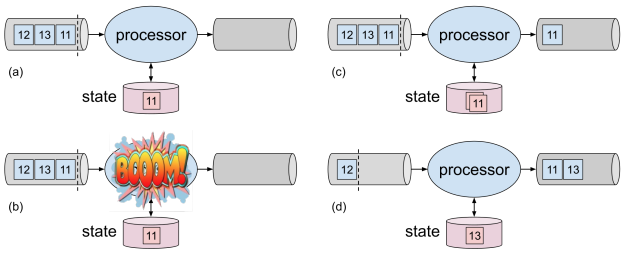

Figure 1: Examples for Streaming Consistency and Completeness Challenges

we designed Kafka to store all continuous data streams as replicated append-only logs. This allows us to simplify the streaming consistency and completeness challenges by using ordered transactional log appends and replays. Apache Kafka contains a stream processing library, Kafka Streams, introduced in version 0.10 (May 2016) to help users build real-time applications with streaming data stored in Kafka. It decouples these two challenges and tackles them with separate approaches: idempotent and transactional writes for consistency, and speculative processing with revision for completeness. Kafka Streams maintains both processing state updates and intermediate shuffled streams as persistent logs in Kafka, and hence does not require state checkpointing and delayed emitting of output results. As shown in some large-scale deployments of Kafka Streams in production, by configuring different values for transaction committing intervals and per-operator lateness tolerance periods, developers using Kafka Streams have more flexible trade-offs between performance and correctness.

To summarize, in this paper we present that by tackling the consistency and completeness properties separately with a small tribute to cost, Kafka Streams can help stream processing developers to achieve both correctness and performance in their applications. Its design is underpinned by the following technical contributions:

- We discuss the two correctness challenges in stream processing, namely consistency in the face of failures and completeness with out-of-order data. We survey state of the art solutions for these challenges and their implicit latency trade-offs.
- We introduce the Kafka Streams client library of Apache Kafka, which leverages Kafka's core architecture as a persistent, immutable commit log to support stream processing capabilities. It tackles streaming consistency by transforming all computational results as log appends with idempotent and transactional protocols, and handles out-of-order data with a fine-grained speculative approach.
- We demonstrate how Kafka Streams has been used in large-scale streaming application deployments in production environments that rely heavily on the correctness guarantees, and we describe their performance metrics and insights.

The remainder of this paper is organized as follows: Section 2 identifies two key challenges to support distributed stream processing correctness guarantees. Section 3 gives an overview of the core design principles of Apache Kafka and its streaming library, Kafka Streams. Section 4 describes how Kafka Streams implements its

exactly-once semantics via idempotent and transactional log write protocols to support consistency with failure recovery. Section 5 illustrates the revision-based speculative processing mechanism employed by Kafka Streams to handle out-of-order data and reason about streaming completeness. Finally, Section 6 presents existing large-scale deployments of Kafka Streams and discusses their performance insights related to its consistency configurations. Section 7 summarizes related work, followed by our conclusions and future work in Section 8.

2 STREAMING CORRECTNESS CHALLENGES

2.1 Fault Tolerance and Consistency

Consistency has for long been an open research issue in stream processing partially due to the lack of a formal specification of the problem itself. In this paper, we primarily focus on consistency guarantees in the face of a failure. Most streaming systems should be able to produce correct results during failure-free executions, but completely masking a failure is quite hard. In distributed systems, fault tolerance is the capability to continue system operations in spite of failures. Additionally, the delivered service should be as if no failures ever happened. When it comes to stream processing, people usually refer to this guarantee as *exactly-once* semantics. In recent years, this term has lent itself to several different interpretations and for clarity of presentation we define exactly-once as the following: for each record from the input data stream, its processing results will be *reflected exactly once* even under failures. Results are reflected in two ways: result records in the output data streams, but also internal state updates in stateful stream processing operators.

A streaming system must be tolerant to a number of failure scenarios which may even occur at the same time in practice:

The storage engine can fail: If a stream processing system does not guarantee that all of its running state is persisted—e. g. if it stores all state in main memory or if it asynchronously flushes the state to persistent storage engines—then upon failures some or all of its state may be lost. In this case, the stream processing system would have to redo computations from the beginning, hoping the processed input data streams have not vanished yet. During the recomputation period, new streaming data would not be processed, reducing the system's availability. What is worse, if the recomputation cannot generate exactly the same state as before the failure due to reasons such as an approximate restart point or nondeterministic logic, then queries before and after the failure may return inconsistent results.

The stream processor can fail: When a stream processor fails over to a new host, it must be able to recover exactly the same state and resume processing at the exact point where it left off. However, the processor may crash after successfully processing a record and persisting a state update but before acknowledging the reception of that record. Hence, when it resumes from a failure event, the same record could be received and processed again, causing not only duplicated outgoing records but also incorrect state with double updates. Even worse, when a processor temporarily loses connectivity to others in a distributed environment, it may be deemed as failed permanently and a new instance may be started up to replace it while the disconnected processor continues to work on its own. In this case, we can have duplicated processors fetching and processing the same data streams, producing duplicate outputs. We call this the problem of *zombie instances*.

The inter-processor RPC can fail: In a distributed environment, stream processors communicate with each other and propagate processing results through message passing. The durability of those sent messages usually depends on the sender receiving an acknowledgement (*ack*) from the receiver. The failure to receive that ack does not necessarily mean that the message did not reach the other side: for example, it is possible that the receiving processor successfully gets the message, processes its associated records and updates its states, but fails to send an ack back to the sender. It is also possible that the receiving processor does not fail at all but a jitter in the network delays the ack back to the sender, exceeding the sender's ack timeout. Since the sender cannot know the exact reason for the lack of acknowledgements, in practice it is forced to assume the message was not received and processed successfully and hence needs to retry. In this case, the same record may be sent multiple times between processors, causing the record to be processed more than once in the streaming system.

Today, most distributed streaming systems rely on checkpointing mechanisms to tolerate failures: during normal execution, the system periodically checkpoints its processing state, which can then act as snapshots to which the system falls back in case of a failure. To align multiple state checkpoints throughout the processing pipeline, these systems usually inject punctuations into data streams as synchronization barriers [31]. As a result, the completion of the checkpoints is not only determined by the amount of data to checkpoint, but also on the speed at which punctuations flow through the application. If there is *backpressure* in data processing, checkpoint efficiency would also be impacted. In addition, the checkpointing mechanism alone is not sufficient to accomplish the desired semantics: since processing results may have already been emitted before a failure, after state is rolled back to a prior checkpoint, re-processing the input records may cause duplicated results in the output streams. In the literature, this is usually termed as the *output commit problem* [34]. There are several known solutions to resolve this problem, such as assigning unique ids that contain lineage information of streaming records, or keeping track of the received records' unique logical timestamps for de-duplication [12, 27, 32]. However, because the inter-operator communication channels are usually memoryless, such de-duplication approaches have to depend on the downstream operators themselves to detect and discard duplicated inputs based on locally maintained lineage information.

2.2 Out-of-order Handling and Completeness

Streaming systems receive and process input data streams continuously in a certain order to provide semantically correct results. The order of the input records typically represents the logical precedence when the input stream records are generated, as indicated by their timestamps. However, in practice, the order of data stream records may be disturbed, where some records may appear in a data stream with smaller timestamps after other records with larger timestamps. The most common external factor that results in *out-of-order* data streams are clock skewness and network delays. Moreover, stream processors that read multiple input streams and merge

```
builder.stream("pageview-events")
  .filter((key,view) -> view.period >= 30000)
  .map((key,view) -> new KeyValue(view.category,view))
  .groupByKey()
  .windowedBy(TimeWindows.of(5000))
  .count()
  .toStream().to("pageview-windowed-counts")
```

Figure 2: Example in Kafka Streams DSL

them into a single output stream might also shuffle the order in which records are transmitted and introduce artificial disorder.

If no out-of-order data exists in the input data streams, guaranteeing complete final results is straightforward: whenever an input record with timestamp $t + 1$ is received, we can infer that all the events up to timestamp t have been completed and their results reflected in the emitted outputs. However, if out-of-order data does exist, a streaming processor may emit partial results from incomplete input streams. Reasoning about completeness and managing disorder are fundamental considerations for streaming systems. Checkpoint-based approaches tend to block emitting results until the system is confident that all events up to a certain point in time are complete. To make this determination, they either rely on external signals or internal indicators injected into the input streams. Similarly, micro-batching techniques [36, 40] break down unbounded streams into batches of bounded data in which each batch represents a window of the unbounded stream. When the batch of input records are considered as complete, its processing will then be triggered synchronously and the updated states be committed to external storage.

3 STREAM PROCESSING IN APACHE KAFKA

Apache Kafka has been used in many of the largest companies across industries as the backbone for data pipelines, streaming analytics, data integration, and mission-critical applications. Data streams stored in Kafka are organized in *topics*, and each topic can be divided into one or more *partitions*. Each partition is maintained as an immutable sequence of records, i. e. a log. Partitions can be continuously appended by producer clients and continuously read by consumer clients [25].

3.1 Partitions and Timestamped Records

The partitioning mechanism allows for the horizontal scalability of Kafka where log partitions are hosted on different Kafka brokers. The actual processing of records is done by the consumer clients. Consumer clients can subscribe to one or more topic partitions and read the records from those partition logs in append order. In addition, multiple consumer clients with a common subscription can form a *consumer group*. Kafka assigns the subscribed topic partitions to the members of the consumer group such that each topic partition is consumed by a single member at any time. Kafka consumer groups handle task assignment, rebalancing due to membership changes, and durable progress tracking, providing a solid distributed systems foundation upon which stream processing applications may be built.

Records stored in the topic partitions are *key-value pairs*, and each record has an assigned incremental *offset* when appended to the partition log to uniquely identify the record's position in the log. Kafka records within a partition log are fetched in the same order they were appended, called the *offset order*. However, as mentioned in Section 2.2, in practice this offset order does not necessarily reflect the logical order of the streaming events those records represent. To tackle this issue, each record also has an embedded *timestamp* field. Producers can set the record timestamp to represent *event time*. Processing systems may use the timestamp field to properly handle out-of-order data, in which records that are appended later actually have smaller timestamps than some others that are appended earlier in the log [33].

3.2 Streams DSL and Operator Topology

Kafka Streams is a Java library included in Apache Kafka that allows users to build real-time stateful stream processing applications. The library contains a high-level DSL for users to specify their streaming logic that continuously reads data streams from *source* Kafka topics with consumer clients, transforms the input streams into new streams and evolving tables, and finally pipes the result streams back to *sink* Kafka topics. It is also used as the underlying parallel runtime of *ksqlDB* [24], an event streaming database built to work with streaming data in Apache Kafka. ksqlDB takes input data streams stored in Kafka and applies continuous queries that derive new data streams or materialized views such as continuous aggregates over windows. Those continuous queries submitted to ksqlDB are compiled and executed as Kafka Streams applications that run indefinitely until terminated.

Figure 2 illustrates an example application written in the Kafka Streams DSL. The application first reads in an event stream from Kafka topic `pageview-events`, filters out those pageview events whose period is less than 30 seconds, and then creates a windowed count of the number of pageviews per category every 5 seconds. The windowed count is considered an evolving table whose aggregate results are continuously updated and new update records are appended as a changelog stream. The changelog stream is written back to another Kafka topic `pageview-windowed-counts`.

Kafka Streams translates the application logic in Figure 2 to a *topology* composed of connected data transformation *operators*. A topology is sub-divided into sub-topologies where each sub-topology consists of consecutive operators between which no data shuffling through network is required. For example, the `filter` and the `map` operator in the example application belong to the same sub-topology. No data shuffling is required between them since the filter only removes records from the data stream and does not change the partitioning key of the records. In contrast, the `map` and the `count` aggregation operator do not belong to the same sub-topology. This is because `map` may change the partitioning key of the records and the key-based `count` requires all records with the same key to be contained in one partition, therefore data shuffling based on the new key is required between the operators. Operators within a sub-topology are effectively *fused* together as upstream operators can directly pass data to down stream operators within the sub-topology without incurring any network overhead [20].

When the processing logic requires reshuffling input streams for data localities, such as the key-based `count` operator in our example, Kafka Streams routes the data through a *repartition topic*. Upstream sub-topologies produce to the repartition topic as a sink

and determine which partition to send to based on the record key. Downstream sub-topologies consume from the repartition topic as a source. Once downstream sub-topologies have processed some records in offset order, they can request Kafka to delete these records from the repartition topics. These repartition topics serve as linearized, durable, fault-tolerant channels of communication between sub-topologies. The infinite capacity of repartition topics relieves Kafka Streams of the need to handle streaming backpressures. When upstream sub-topologies temporarily process faster than downstream sub-topologies, the excess data is simply buffered persistently in the repartition topic.

Stateful operators within a sub-topology are associated with state stores. Users can either configure to use the built-in persistent or in-memory state stores from Kafka Streams, or they can provide custom implementations from the supplied state store APIs. By default, writes to the states stores are also replicated to Kafka as *changelog topics*. Kafka brokers hosting these changelog topics will remove records for which another record was appended with the same key but a higher offset, effectively compacting the log. Both the repartition and changelog Kafka topics are considered internal data streams and abstracted away from the users.

3.3 Data-Parallelism and Tasks

A sub-topology is executed as one or more *tasks*; one task per source topic partition. A task represents the smallest parallel unit of work in Kafka Streams. Tasks execute independently from each other irrespective of whether they execute the same sub-topology for another partition or a different sub-topology. Within a task, input records from the source topic partition are processed in-order by traversing the sub-topology, updating the state stores and generating output records to the changelog and sink topic partitions. Repartition topics may also be configured with different numbers of partitions than the external input topics so that different sub-topologies may have a different number of tasks.

A Kafka Streams application can be deployed on multiple computing nodes as *instances* of the same application. When new instances of the application are launched or existing ones shutdown or crash, tasks will be re-distributed across instances automatically to balance the workload. Since both the input and output records of a task are persisted in Kafka logs, tasks can be independently paused, resumed, and migrated between instances. If a task with stateful operators needs to migrate to a new instance, an exact copy of the state is restored by replaying the corresponding changelog topics. Note that a single instance can potentially host and execute multiple tasks, and Kafka Streams tries to achieve both workload balance among instances and task stickiness to minimize the amount of state migration required when deciding how tasks should be re-distributed among instances.

A sketch topology generated from the above Streams DSL example is shown in Figure 3 (with some trivial operators omitted). It is composed of two sub-topologies connected by a repartition topic. The repartition topic is needed because the `map` operator changes the key of the records and the key-based `count` requires all records with the same key in the same partition to achieve data locality. The first sub-topology contains the `filter`, `map`, and `groupByKey` that represents the repartitioning operation. The second sub-topology

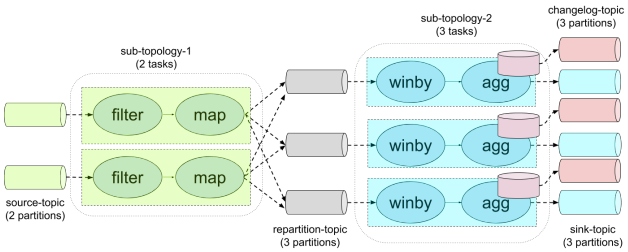

Figure 3: Generated Processing Topology and Parallel Tasks from the Kafka Streams DSL Code Example in Figure 2

contains the `windowBy` and `aggregation` operators where the former assigns each input record read from the repartition topic to its associated window(s) and the latter aggregates the input record values into the count for each window. Assuming the source topic `pageview-events` has two partitions, while the internal repartition topic and the sink topic `pageview-windowed-counts` have three partitions each, these two sub-topologies will have two and three tasks respectively. The changelog topic of the aggregation state store would also have three partitions, as determined by the number of tasks of the second sub-topology. Tasks may be executed independently and in parallel on different instances. Optimizations are also applied in Kafka Streams when generating the topology. In our example, the changelog topic of the state stores associated with the aggregation operator can potentially piggy-back on the sink topic since there is no further transformations between the aggregation operator and the sink operator sending the outgoing changelog data stream. For the sake of our running example demonstration in this section though, we skip such optimization and still display two separate Kafka topics.

Kafka Streams uses embedded producer clients to send output records to Kafka brokers. If a produce request gets timed out, the producer would retry assuming those records were not received by the brokers. However as mentioned in Section 2.1 it is possible that the broker did receive the records and append to the logs, but the acknowledgements get delayed over the wire. In this case retries would result in duplicated appends. In addition, during normal processing Kafka Streams would periodically commit its current processing position on the source topics after it has flushed the outgoing records to sink topics as well as the updates to the state stores. But if a failure happens in between these operations, then upon failover the same source topic records could be refetched after the processing state are restored from the changelogs. In that case, we will reprocess these records while the processing results have already been persisted to the sink topics and state stores, causing *at-least-once* stream processing scenarios. In the next section, we will explain how Kafka solves the challenge to recover from failures and provide consistency guarantees.

4 EXACTLY-ONCE IN KAFKA STREAMS

A key design principle behind Kafka's architecture is to persist streaming records on the brokers, making them durable and immutable. This design is based on the assumption that persistence is, and will continue to be, not expensive. Past years have shown that

this trend holds even as storage devices move from hard disk drives to solid state drives. Durability in Kafka is achieved via replication, such that every record written to a topic partition is persisted and replicated on n different broker machines (n is configurable). For each partition, one broker is elected to host the *leader replica*, which takes writes from the producer clients. Records appended to the leader replica's local log are then fetched and replicated to the logs of the other $n-1$ brokers that host *follower replicas*. Each Kafka broker may host multiple leader replicas of different topic partitions. When a broker fails, each leader replica it hosted will have a new leader elected from among that partition's follower replicas. As a result, Kafka can tolerate $n-1$ broker failures, meaning that a partition is available as long as at least one broker replica is available [38]. The data replication protocol of Kafka guarantees that once a record has been appended successfully to the leader replica, it will be replicated to all available replicas.

As a persistent storage layer, Kafka eliminates communication dependencies between streaming tasks: since records sent to Kafka are durably buffered, upstream tasks do not have any backpressure issues when downstream tasks are temporarily slow at processing. Persistent buffering reduces the complexity of recovering a failed stream processing instance and makes task failover and resumption relatively lightweight. In addition, since all records are stored durably, changelog topics inside Kafka Streams can be considered as the source-of-truth for state management. State stores, on the other hand, become disposable materialized views of the changelogs since they can always be restored by replaying the corresponding changelog streams. As a result, we can reduce the fault tolerant state management and output commit problems to a contract regarding which input records are deemed as processed and which output records are visible under the various failure scenarios. In Kafka this contract is described as idempotent and transactional record appends.

4.1 Idempotent Writes Per Partition

An idempotent operation can be performed many times without causing a different effect than only being performed once. When it comes to stream processing in Kafka, an idempotent producer client can send the same record multiple times to a partition log and only one append would happen on that log. For that single partition, idempotent producer send operations can remove the possibility of duplicated records due to various failure scenarios such as ack time-outs due to network jitters.

Idempotent producers work in a way similar to TCP: each record sent by the producer will be associated with a monotonically increasing sequence number. This sequence number combined with the producer client's unique identifier can be used by the brokers to de-duplicate record appends within that producer's lifetime. Unlike TCP, which provides guarantees only within a transient in-memory connection, sequence numbers are persisted along with the records to the replicated log. When multiple records are being appended to the log as a batch, we only need to encode one sequence number as the one for the first record as other sequence numbers can be inferred monotonically. On the broker side, latest sequence numbers per-producer are cached and persisted when the brokers are shutting down. Thus, if a Kafka broker fails, any other broker that takes over its hosted topic partitions as the new leader replica will be able to re-populate its sequence number cache by looking at the local logs, and then be able to determine whether a producer send operation is a duplicate.

4.2 Transactional Writes Across Partitions

As mentioned in Section 3, stream processing in Kafka follows a read-process-write cycle: an embedded consumer reads records from the source topics, processing logic defined as a Kafka Streams sub-topology transforms those records and modifies the states, and an embedded producer writes the output streaming records to the sink and changelog topics. Finally, record offsets on the source topics can be committed, indicating the completion of processing.

Exactly-once stream processing in this case requires the ability to execute a read-process-write operation exactly one time even under failures. In other words, all actions within this operation should be done atomically: 1) output records should be appended to the sink topics and became readable for downstream consumers, 2) processing state stores should be updated, 3) input record offsets on the source topics should be committed. Among those actions, state updates are replicated with appends to the underlying changelog topics which, upon failure, can be replayed to roll back the local state. In addition, offset commits in Kafka are translated internally as appends to an internal Kafka topic as well. Therefore, all these actions can be translated as record appends to some Kafka logs. Atomicity can be achieved by making these appends to different Kafka logs in a transactional manner: either all appends succeed and become visible to consumers or none do.

Comparing with traditional two-phase commit protocols that require the data to be written twice (once for the log and another for the data), Kafka's transactional protocol only requires writing data once in the log, and leverages on the append offset ordering to avoid exposing aborted data to the clients. In addition, Kafka avoids single points of failure by making the transaction coordinator highly-available with replicated transactional logs. Details about how transactional writes work in Kafka are described in the subsequent sections.

4.2.1 Transaction Coordinator and Transaction Log. When transactional writes are enabled on a Kafka producer, that producer is assigned a unique identifier called the *transactional id*. This id is used to identify the same transactional producer instance across restarts. An incremental *epoch* is associated with the transactional id to distinguish between multiple lifetimes of the producer when it restarts due to failover or client migrations.

As shown in Figure 4, within each Kafka broker there is a module called the *transaction coordinator* that manages metadata of all the transactional producers assigned to it. There is also a transaction log maintained as another internal Kafka topic, and each coordinator owns some subset of the partitions in the transaction log. The transaction coordinator keeps the metadata of each transaction it owns in memory, and also writes any updates of the metadata to the transaction log (Figure 4.a). It is worth noting that the transaction log only stores the latest metadata of a transaction and not the actual records sent within the transaction. The transaction's metadata include its current state, like `Ongoing`, `PrepareCommit`, and `Completed`, as well as the topic partitions that are registered with

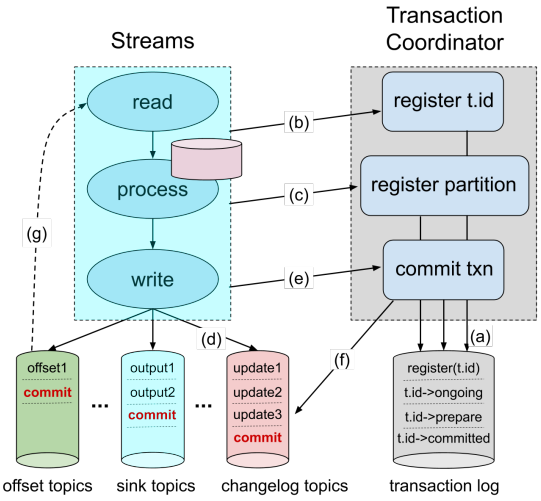

Figure 4: Kafka Transactions Work Flow

this transaction. The transaction coordinator is the only component to read and write the transaction log.

As described at the beginning of Section 4, when a broker hosting one or more transaction log partitions fails, new leader replicas for those transaction log partitions will be elected. Those replicas act as the new transaction coordinators owning those partitions, and rebuild an in-memory collection of the current transactions by replaying the metadata update records from the transaction logs. In this way, we leverage Kafka's own replication protocol to ensure that the transaction coordinators are highly available and that all transaction state is updated durably. Every producer's transactional id is mapped to a specific partition of the transaction log through a hashing function. This means any transactional producer with a given transactional id would only talk to one transaction coordinator across its lifetimes.

After startup, the first operation of a transactional producer is to explicitly register its transactional id with the transaction coordinator (Figure 4.b). When it does so, the coordinator checks for any open transactions with the given transactional id and completes them based on their current states: only if the state is already in `PrepareCommit` it would roll forward the transaction, otherwise it would abort the transaction. It also increments the epoch associated with the transactional id before acknowledging the producer's registration. Once the epoch is bumped and persisted on the transaction log, any producers with the same transactional id and an older epoch are considered zombies and hence fenced off, i. e. future transactional writes from those producers would be rejected. After getting the registration acknowlegement from the transaction coordinator, the producer client completes the initialization process and is ready to start sending records in transactions.

4.2.2 Two-Phase Transactional Commit. A transactional producer can have at most one ongoing transaction at a given time. Each time the producer is about to send records to a new Kafka topic partition within a transaction, it first registers the partition with the transaction coordinator (Figure 4.c). In Kafka Streams, these partitions can belong to a sink topic, a changelog topic representing a state store, or the internal Kafka offset topic for offset committing. After registering new partitions in a transaction with the coordinator, the producer sends data to the actual data partitions as normal (Figure 4.d).

After one or more records are sent within a transaction, the producer can try to commit the transaction: First, the producer flushes all its writes, awaiting acks for those writes from the Kafka brokers. Then it sends another request to the transaction coordinator to initiate the commit process through a two phase commit protocol (Figure 4.e). In the first phase, the coordinator updates its state to `PrepareCommit` and records its state change in the transaction log. This update is considered the synchronization barrier of the transaction: once the state update is replicated in the transaction log, there is no turning back. The transaction is guaranteed to be committed even if the transaction coordinator crashes immediately after. The coordinator can then begin the second phase asynchronously, where it writes a *transaction commit marker* to each of the transaction's registered partitions. Transaction markers are special records in the Kafka logs indicating all records appended from the specified transactional id before the marker are now committed (Figure 4.f). Just like data records, transaction markers written to Kafka logs are also replicated across brokers. After all the transaction markers have been acked by the partition leaders, the transaction coordinator updates its transaction state to `CompleteCommit`, which allows the producer client to start a new transaction.

On the other hand, if an error happens during processing the producer can also abort the current transaction by sending an abort request to the coordinator. The transaction coordinator itself could also abort an ongoing transaction when the transaction times out. Abortion is handled in a similar way to commit: first the state of the transaction transits to `PrepareAbort`, then *transaction abort markers* are written to that transaction's registered partitions indicating all records appended by the transactional id before this marker should be aborted and may not be returned to consumers. After all markers are acked the transaction state would transit to `CompleteAbort`.

4.2.3 Reading Transactional Records. In Kafka Streams, upstream tasks execute atomic read-process-write cycles and append results to intermediate repartition topics. Downstream tasks then consume from these intermediate topics to continue further processing. The consumer clients of downstream tasks are configured to read only committed data, in which case records appended in a transaction are only delivered to the downstream tasks when their transactions are committed and commit markers are received. If the transaction is aborted with abort markers, its records would not be returned from the consumer clients.

In addition, a task's committed offsets on the source topics are also only reflected when the ongoing transaction is committed. If the current transaction is aborted, the committed offsets would be dropped and the committed offset would effectively roll back to the last committed transaction. As a result, when a Streams task migrates to a new instance, restores its state from the changelog topics and reset its positions on the source topics, the starting positions on the source topics would align with its restored state as well as the output records appended successfully to the sink

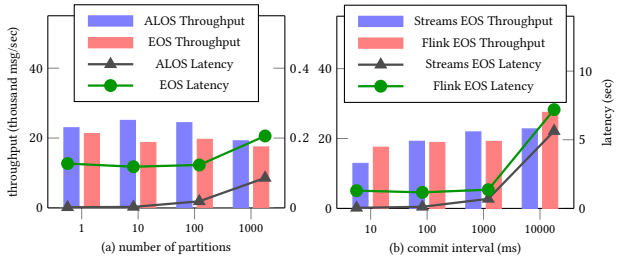

Figure 5: Exactly-Once Impact on Throughput and Latency

topics. This guarantees that no source records get dropped or double processed even upon failures (Figure 4.g).

4.3 Performance Implications

In Kafka Streams, users can switch from at-least-once semantics to exactly-once semantics with a single configuration. The exactly-once configuration enables both idempotence and transactional features. When multiple tasks are assigned to the same Streams instance, their processing can be grouped as a single on-going transaction by the instance's embedded producer, and the producer needs to talk with its transaction coordinator in order to fence zombie writers and manage on-going transactions. With at-least-once configuration, producers within Kafka Streams will not conduct this additional coordination when writing data to Kafka topics.

Figure 5.a illustrates the overhead of exactly-once with different number of transactional partitions. The evaluation was done on a three-node Kafka cluster that hosts two topics: an input topic written by a streaming data generator, and an output topic with varying number of partitions from 1 to 1000. A single-node Kafka Streams application is deployed that reads from the input topic, does a stateful reduce operation that reads from and writes to its local state store, and finally emits results to the output topic. Application's commit interval is set to 100 milliseconds. The output topic is fetched by a consumer configured to read committed data only. The end-to-end latency is calculated per-record based on the record creation time when produced to the input topic, and the consumer reception time for that record's result. All instances were deployed on dedicated `i3.large` AWS EC2 nodes.

As observed in the figure, the throughput degradation of exactly-once semantics (EOS) is relatively small, ranging from about 10 to 20 percent compared with at-least-once semantics (ALOS). The first reason is that idempotence in Kafka producers only requires a few extra numeric fields with each batch of records to be persisted on the log. With a reasonable batch size in practice, these fields add negligible overhead. In addition, the write amplification cost of a transaction—additional RPCs between clients and the transaction coordinator and between the transaction coordinator and other Kafka brokers hosting the partitions involved in the transaction—is constant and independent of the number of records written within the transaction. Although this transaction cost is indeed dependent on the number of output partitions participated in a transaction, the impact is not massive since producers can batch multiple writing partitions in a single registration request. On the other hand, end-to-end latency is heavily dependent on the number of partitions since the transaction markers written per transaction increases linearly with the number of partitions, and the consumer can only fetch the committed records after the marker is received.

The major factor impacting transactional commit throughput and latency is the commit interval: a longer commit interval will result in larger transactions in terms of number of records and hence smaller amortized cost. However, longer commit intervals would also increase end-to-end processing latency because the consumer can only read records from the output topic written by the application's transaction when that transaction has been committed. We illustrate its effect in Figure 5.b, which has the same experimental setup and application logic but fixes the number of partitions to 10 and varies on commit interval from 10 milliseconds to 10 seconds instead. To compare with checkpoing-based approaches, we also evaluate Apache Flink [31] (version 1.12.1) which is a well-known framework that leverages stream barriers to checkpoint state snapshots. Flink's checkpointing mechanism requires a persistent data source that can replay records in case of failures to achieve consistency, and when integrated with Kafka as its data source, its exactly-once implementation also relies on Kafka's transactional protocols. In our experiment, we configure Flink to incrementally checkpoint its local state to an S3 bucket, and vary the checkpoint interval correspondingly as we vary Kafka Stream's commit interval. As shown in the figure, both Kafka Streams and Flink have higher throughput at the cost of longer latency when commit/checkpoint interval increases. Compared with Kafka Streams' finer-grained changelogs though, Flink's checkpointing is per-file based and hence would take longer time when only a small number of keys are updated within the interval. Since the transaction can only be committed after the checkpoint is completed, it would result in longer latency. As we increase the commit/checkpoint interval, more keys would be updated per commit and hence the latency gap becomes smaller.

5 KAFKA STREAMS REVISION PROCESSING

Although Kafka Streams leverages idempotent and transactional writes to achieve exactly-once semantics, it does not rely on the mechanism to handle out-of-order data. In other words, the completion of a transaction with largest event timestamp $t + 1$ in its committed input records does not necessarily indicate that all records up to event time t have been included in the emitted result. Instead, Kafka Streams uses an optimistic approach that assumes all items arrive in the right order for most of its operators and hence does not block emitting results. When out-of-order records do arrive, Kafka Streams updates the computations, invalidating previously emitted results by emitting corrected results.

Revision-based approaches were first proposed in Borealis [21], and Kafka Streams employs this idea based on the fact that inter-task data transmissions are always via intermediate repartition topics, a persistent, immutable, and ordered communication channel. More specifically, in the Kafka Streams DSL, users can define two types of first-class entities: an append-only record *stream*, and a time-evolving *table* that can also be represented by its changelog stream. The table can be viewed as a materialized view of the changelog stream of successive updates. Various processing operators pre-defined in the DSL can transform between these two entities, for example, a window aggregation operator can take a

record stream as its input and outputs a windowed table that hosts the running aggregates (more details of the Kafka Streams DSL can be found in [22]). While an append-only record stream is semantically a sequence of independent records, a time-evolving table supports amendment semantics from its corresponding changelog stream: new update records can be appended to the stream as revisions to old updates with the same key.

As a result, we can categorize all processing operators regarding how they should handle out-of-order data in the following ways: First of all, stateless operators such as `filter` and `mapValue` are independent of record order by definition. There is no need to impose reordering delays for such order-agnostic operators, where output records can be emitted immediately when new input records are received and processed. Stateful operators such as `aggregation` and `join` would depend on previous received records to determine the current processing results, and hence are order-sensitive. In the Kafka Streams DSL, users can specify a per-operator grace period for those order-sensitive stateful operators. Out-of-order records arriving at these operators no later than the grace period would still be accepted to update the processing states. In addition, Kafka Streams checks on each operator's output entity type to determine if it can emit results speculatively: If the output type is an append-only stream, then we may need to hold on emitting some output records in order to preserve completeness. For example, consider a stream-stream left-join which output a new stream, where the left stream have one input record `a` and right stream have one input record `b` with the same key. If record `b` is delayed and we processed record `a` first trying to execute the join, we would not find any match and hence would emit a join result `(a, null)`. When `b` is later received and processed, we would emit another join result `(a, b)` into the output stream. However, at this time, the first incorrect join result `(a, null)` has already been emitted to the output append-only stream and cannot be revoked anymore. In order to avoid such incorrect results, we need to hold on emitting the join result for record `a` until the grace period has elapsed. Such fine-grained per-operator grace period configurations can be orthogonal to the commit intervals for transaction writes, and add flexibility for users to control completeness delays.

On the other hand, if the output type is a time-evolving table—for example, a table-table left-join which output a new table—then with the same input records and ordering as above, the output sequence `(a, null), (a, b)` would not violate correctness since the second record in the output stream can be viewed as an amendment to the join result that overwrites the first record. Hence, for stateful operators that output a table, we can still emit early records to its corresponding changelog stream speculatively with possible later updates when the joining tables receive out-of-order updates within the grace period. Downstream operators that get the output changelog stream as input can then revise their processing states correspondingly by retracting the effect of old update records and accumulating the effect of new update records. Since the changelog stream generated from the upstream operators is persisted as a Kafka topic, even when downstream operators fail over, this stream can still be replayed to revise their restored processing state.

Figure 6 showcases an example how Kafka Streams handles out-of-order data with eagerly emitted records to downstream tasks. It depicts a single windowed aggregation task from the running

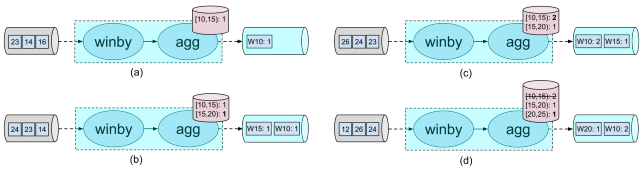

Figure 6: Kafka Streams Revision Processing Highlights

example topology shown in Figure 3. Similar to join operators described earlier, windowed aggregation also need to handle out-of-order data to ensure result completeness. Input records of the task are piped in from the repartition topic partition and output records are sent to the sink topic partitions. The sink topic partitions may be then consumed and processed by other Streams tasks. A state store maintaining the current aggregated value is associated with the `aggregate` operator, which follows the `window-by` operator. The `window-by` operator assigns each input record to a 5-second-length window according to the record's timestamp. The grace period of the aggregation operator is set to 10 seconds. We omit the input record value and assume all their keys are the same and only focus on the record timestamps to simplify the depiction of internal state updates and emitted results.

At the beginning, a record with timestamp 12 is processed, and updated the state store's window $[10, 15)$ with count 1. The result (indexed by the window start time) is emitted to the output stream (Figure 6.a). Then a new record with timestamp 16 arrives in order, the state store updates its count for window $[15, 20)$ to 1 as well, and emits the corresponding record to the output stream (Figure 6.b). After that, an out-of-order record with timestamp 14 is received and accepted since it is still within the grace period. This causes the old window $[10, 15)$ to be updated again, so a revision record with new count value 2 is emitted. This revision record would be sent via the same intermediate Kafka topic partition and hence would be received and processed by the same downstream task that accumulates the previous record with count 1 (Figure 6.c). At the Streams DSL, users would need to provide corresponding implementations for both accumulations and retractions to amend their processing states upon getting revision records. If the downstream operators also output a table—for example, consider a join operator that enriches the aggregated window table and generates a new table—then upon receiving revisions from the upstream, new joined revisions would be further propagated down through the topology. Finally, a new record at timestamp 23 is received. Besides updating the corresponding window and emitting another result, Kafka Streams would also garbage collect the old entry for window $[10, 15)$ since it is now out of the configured grace period (Figure 6.d). Later out-of-order record at timestamp 12 from the input streams would then be discarded. Note that the grace period here only controls how much old state Kafka Streams would need to maintain in order to handle out-of-order data, but does not indicate how long we delay output in order to guarantee completeness.

In practice, emitting every revision downstream may incur high network and CPU cost to do subsequent accumulations and retractions that just offset each other. The Kafka Streams DSL allow users to optionally add `suppress` operators right after these stateful operators in order to buffer their intermediate revisions before

emitting downstream. Multiple revisions of the same key could then be consolidated as a single record when being emitted from the suppress operator.

6 LARGE-SCALE DEPLOYMENTS

Today, Kafka Streams is widely used in various organizations, including Bloomberg, Expedia, LINE, Pinterest, Twitter, Walmart, etc[1]. In this section, we present production insights related to Kafka Streams correctness mechanisms through a couple of use cases from Bloomberg and Expedia.

6.1 Bloomberg Real-time Pricing Platform

MxFlow is a framework developed at Bloomberg to help build both micro-batch and real-time streaming pipelines to process derivatives market data, including options, forwards, futures, and swaps. Data generated by these pipelines is used by numerous pricing modules. Some of the high profile ones get billions of hits every day. The Streaming Flow API of MxFlow is built on top of Kafka Streams, enabling application developers to focus on the implementation of core business logic related to the market data, without worrying about distributed processing details such as I/O, state handling, fault tolerance, etc. This API provides programming abstractions and boilerplate code for both continuous market data stateful stream processing and runtime processing state queries. In addition, the capability to replay market data in streaming jobs is another critical requirement to reproduce and diagnose market data outliers, backtest new algorithms, etc.

6.1.1 The Streaming Framework. Figure 7 illustrates an overview of the MxFlow Streaming pipeline. Various types of incoming streams are ingested into the pipeline as source Kafka topics. One topic holds the main market data published by external upstream sources such as exchange and direct channels, while other topics contain less frequently updated reference market data published by internal sources. Internally sourced market reference data are piped through Kafka Connect[2], a framework for scalably and reliably streaming data between Kafka and external systems such as databases, key-value stores, and search indexes. The application is composed of a sequence of stateful processing modules: 1) outlier signal detection, 2) dynamic profile-based windowing, and finally 3) weighted aggregation. The aggregated market insight data is streamed into a sink Kafka topic as well, which can then be consumed by the downstream market data hub. The market data hub is used by different internal analytics services such as risk and portfolio modeling, and it is also used by external clients in Wall Street. By using Kafka topics both as the main data ingestion entry as well as the state changelog of Kafka Streams applications, MxFlow Streaming Flow API allows developers to bootstrap streaming jobs in a configuration-driven manner. The built-in fault-tolerance of Kafka Streams allows the streaming application to run 24/7, and the exactly-once processing guarantee makes sure that every market bid and ask will be processed without duplication or loss.

The pipeline also includes a state catalog service that allows users to query current and historical snapshots of the running states from

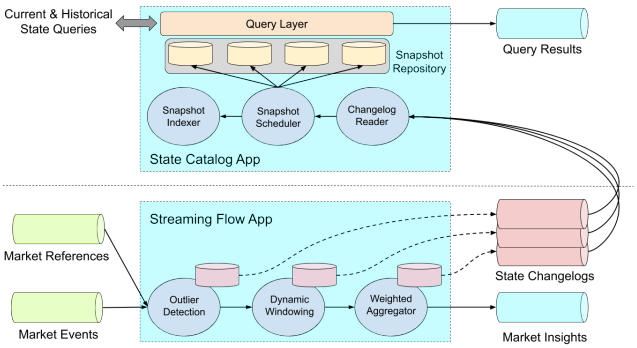

Figure 7: Bloomberg MxFlow Streaming Pipeline Overview

the stateful streaming applications. This service is primarily used for a dashboard UI, as well as for internal troubleshooting and back-testing purposes. It is implemented as another Kafka Streams application that replays the state changelog topics produced by the previous application to regenerate, maintain, and garbage-collect historical state snapshots. New snapshots can be generated by incrementally replaying the changelog topics from existing snapshot positions. Since the changelogs across state stores are appended in atomic transactions, replaying them with a read-committed consumer generates consistent historical snapshots for query serving. Snapshot query results can either be sent back in gRPC responses, or written to Kafka topics asynchronously if requested.

6.1.2 Production Scale and Performance Insights. At the time of writing, Kafka Streams has been deployed on 160 cores with a total of 2 TB RAM inside the MxFlow framework, processing more than one billion market records on the daily basis. All instances are deployed via Kubernetes and provisioned across multiple data centers. The exactly-once processing mode guarantees no data duplication or data loss results from unpredicted pod migration. In order to reduce the serialization cost for state updates, application-level object caches are used in addition to Kafka Streams's own state store caches to achieve sub-millisecond average latency.

Bloomberg conducted extensive measurements to make sure MxFlow can handle market peak hour rates. The testbed launched a total of 32 Kafka Streams threads, with each thread handling 100 topic partitions on average. With Kafka version 2.6 (August 2020), the number of transactional producers, and hence its cumulated overhead when coordinating to write transactional messages, only increases with the total number of Kafka Streams threads regardless of the number of input Kafka topic partitions. With varying streaming loads from 10 to 25 thousand messages per second, they only observe an overhead ranging from 6% to 10% for exactly-once versus the default (at-least-once) configuration.

6.2 Expedia Real-time Conversation Platform

The Conversational Platform (CP) at Expedia serves as a virtual assistant intended to bring real-time, intelligent responses to online inquiries such as booking trips, making changes or cancellations, and asking questions. To fulfill those requests, CP is implemented as a highly complex system composed of various micro-services

[1] https://kafka.apache.org/powered-by/
[2] https://kafka.apache.org/documentation.html#connect

working in unison. For example, there is a set of services to facilitate instructions sent from customers to the virtual assistant, including personally identifiable information redaction, NLP-based localization, and language translation, etc. Each of those services would enrich the data sent to the virtual agent, and the agent itself leverages other services, such as dynamic agent routing and configuration, recommendation and searching services, and customer relationship management (CRM) applications, to handle and process incoming requests.

6.2.1 Conversational Messaging Processing. Kafka is used as the messaging backbone inside CP to integrate all the above microservices to construct a loosely coupled, real-time, asynchronous, highly available system. A conversation is essentially a collection of strictly ordered customer dialogues and noteworthy events over a time period. Each of these messages and events are captured as Kafka log records flowing through the system, where each service within would receive those records in order, process them, and then generate new outgoing records to other services asynchronously. Individual services within the platform can be scaled and upgraded independently as long as their exposed record formats are backward compatible. In addition, all services within CP must process each record once and only once, since otherwise undesirable outcomes such as double payment for a ticket or reservations getting voided could happen. To cope with this requirement, CP includes a stateful event processing library that builds on top of the Kafka Streams API with exactly-once mode enabled to simplify the stream processing implementations. One example stateful streaming application implemented with this library maintains an aggregated view of a conversation, which can then be queried by external processors for operational purposes such as purging all closed conversations from active working queues.

6.2.2 Production Scale and Performance Insights. A typical stream processing application inside CP runs on 5 EC2 nodes with 1.5 GB RAM and 8 threads per node. Stable average throughput per application is about 14 records per second. However, during pandemic peak times, the platform sustained through many orders of magnitude higher traffic, taking much of the load off live agents. For simple data-enrichment services, the commit interval is set at 100 milliseconds such that a single message can traverse through the entire pipeline within sub-second end-to-end latency. For complex conversation-view aggregation services, the commit interval is set at 1500 milliseconds and output suppression caching is also enabled to reduce disk and network I/O.

7 RELATED WORK

One of the most foundational challenges in stream processing technology has been the development and adoption of consistency guarantees. Implementations of such guarantees are significantly impacted by fault tolerance functions, which are applied not only to unexpected failures but also to controlled failures due to parallelism reconfiguration or operator updates [13, 14, 30]. A strict approach in stream processing is to make sure one or more actions (read, update, write) are persisted transactionally. For example, MillWheel uses BigTable to commit a batch of actions that includes the input records, state updates, and generated output records [4]. StreamScope [27] provides a fault tolerance strategy that relies on recomputing state by replaying data from input streams. This is because it relies on the Cosmos distributed file system as the intermediate persistent pipeline, and as a result, sent records can be overridden with a given offset [9]. Another common approach for transactional streaming is a two-phase commit protocol to commit and checkpoint the processing results of a contiguous subset of the input streams. Aligned checkpoints across processing states can also be leveraged to support various control policies such as reconfigurations [28]. The two-phase commit protocol can be either strict [17, 29] or asynchronous based on checkpointing algorithms. For example, the Chandy-Lamport snapshot algorithm is a common asynchronous mechanism adopted by several systems, such as IBM Streams [23] and Apache Flink [31]. In the Chandy-Lamport algorithm, subsets of input streams are separated by punctuation markers, and upon receiving the marker, operators can checkpoint their states independently. However, its checkpoint completion latency is determined by the speed at which each subset of input streams flows through the streaming topology and hence can be impacted by application backpressure in practice. The Kafka Streams fault tolerance mechanism is similar to StreamScope [27] and Apache Samza [2]: it also leverages a persistent communication channel, Kafka topics, to achieve idempotence. But since its log data is immutable, its two-phase transaction commit protocol actually relies on the log's offset ordering to ignore already appended, but later aborted, records. In addition, since state updates in Kafka Streams are translated into changelog stream records that participate in the transaction, it does not require its associated state stores to have any transactional capabilities.

Existing techniques for handling out-of-order data in stream processing mainly fall into two categories. In-order processing (IOP) systems manage disorder by fixing an input stream's order. They buffer and possibly reorder records up to a lateness bound and then process the reordered tuples. In this way IOP streaming systems effectively chunk the data stream into finite subsets, and each subset is processed as a micro-batch [10, 12, 18, 20, 40]. Some IOP systems also assume that the order of records within a single buffered batch does not matter, and just process them as an indivisible unit [36]. Out-of-order processing (OOP) systems, on the other hand, do not buffer and delay processing input stream records to enforce ordering. Instead, they rely on implicitly or explicitly provided information to track progress. For example, punctuation records (also called heartbeats in some systems) carrying the low-watermark timestamp are injected into the data streams to indicate the completeness of the stream up to the injected timestamp [4, 5, 7, 16, 26]. For stateful operators such as sliding window aggregations, OOP systems need to bookkeep partial result structures to handle out-of-order data arrivals and recompute aggregations. In recent years, various techniques have been proposed to reduce storage cost as well as latency for these operators [35, 37]. Punctuations are also propagated from upstream operators to downstream ones to keep track of the progress throughout the pipeline. In practice, such punctuations cannot always be provided from external services. Hence, streaming systems need to control the advancement of the punctuation themselves based on their allowed lateness. Kafka Streams can be considered as an OOP streaming system, as it employs revision processing [2, 21] on the subset of operations that are

order-agnostic. Rather than relying on system-level low-watermark timestamps, Kafka Streams allows a granular completeness determination via per-operator grace period configurations. In addition, Kafka Streams addresses the recomputation bookkeeping problem by configuring upstream operators to forward both the prior and the updated results when necessary so that downstream operators can retract the prior result before applying updates.

Another commonly referenced correctness property that can be easily obtained in batch processing while being very hard to achieve in stream processing is called *determinism*. Determinism means that repeated runs of the system on the same data streams always produce the same results. Consistency and completeness are necessary, but not sufficient, guarantees to achieve this property, because it requires all streaming operators to also behave deterministically given multiple input streams, from its calculation logic to the way it chooses the next record to process, regardless of buffering, batching, network delays, and caching dynamics. For example, StreamScope [27] achieves determinism by enforcing deterministic algorithms that choose from incoming records and time-incrementing marker records, as well as forbidding non-deterministic computations. SEEP [13] also assumes deterministic computations without side-effects and a monotonically increasing logical clock to attach scalar timestamps; duplicated records are discarded solely by their timestamps. Kafka Streams does not forbid non-determinism from its DSL, but does make deterministic incoming record choices based on record timestamps. As a result, users can achieve determinism if they enable exactly-once processing mode and do not specify non-deterministic processors.

8 CONCLUSION AND FUTURE WORK

We have presented Apache Kafka's core mechanisms to achieve correctness guarantees for building distributed stream processing applications. Stream processing can be expressed as read-process-write cycles in the Kafka Streams client library, which heavily relies on Kafka's replicated logs as its persistent, immutable, ordered storage and communication layers. The read-process-write cycles in Kafka Streams are translated as record appends to a set of Kafka logs and a two-phase commit protocol is employed to enable idempotent and transactional appends to support exactly-once semantics. A separate speculative approach is applied to the problem of out-of-order data, with revision processing on the subset of operators that are sensitive to order, allowing users to decouple the fundamental trade-off decisions between latency, throughput, and correctness guarantees. We also provided usages of Kafka Streams in large-scale production deployments and discussed their performance insights.

The primary future work is to reduce end-to-end latency by optimistically processing uncommitted input data streams with cascading rollback algorithms in the face of failures. In addition, we would like to enable consistent state query serving across multiple Kafka Streams applications. Finally, we are planning to support live control policies such as auto scaling, application updates and reconfigurations based on aligned transaction boundaries.

REFERENCES

[1] *Apache Kafka*, 2021. Retrieved from https://kafka.apache.org/.
[2] S. A. Noghabi et al. Stateful scalable stream processing at Linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, 2017.
[3] T. Akidau. Open problems in stream processing: A call to action. In *DEBS*, page 4. ACM, 2019.
[4] T. Akidau et al. MillWheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, 2013.
[5] T. Akidau et al. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.
[6] A. Arasu et al. STREAM: the stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.
[7] P. Carbone, A. Katsifodimos, S. Ewen, and et al. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
[8] R. Castro Fernandez et al. Liquid: Unifying nearline and offline big data integration. In *CIDR*, 2015.
[9] R. Chaiken et al. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
[10] B. Chandramouli et al. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.
[11] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
[12] C. D. Cranor et al. Gigascope: A stream database for network applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 647–651. ACM, 2003.
[13] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 725–736. ACM, 2013.
[14] A. Floratou et al. Dhalion: Self-regulating stream processing in Heron. *Proc. VLDB Endow.*, 10(12):1825–1836, 2017.
[15] M. Fu et al. Twitter Heron: Towards extensible streaming engines. In *ICDE*, 2017.
[16] D. Gordon Murray et al. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.
[17] P. Götze and K. Sattler. Snapshot isolation for transactional stream processing. In *EDBT*, pages 650–653, 2019.
[18] B. He et al. Comet: Batched stream processing for data intensive distributed computing. In *SoCC*, pages 63–74. ACM, 2010.
[19] M. Hoffmann et al. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.*, 12(9):1002–1015, 2019.
[20] D. J. Abadi et al. Aurora: A data stream management system. In *Proc. of the ACM SIGMOD Int. Conf. on Mgmt of Data*, page 666. ACM, 2003.
[21] D. J. Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
[22] M. J. Sax et al. Streams and tables: Two sides of the same coin. In *BIRTE*, pages 1:1–1:10. ACM, 2018.
[23] G. Jacques da Silva et al. Consistent Regions: Guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.*, 9(13):1341–1352, 2016.
[24] H. Jafarpour, R. Desai, and D. Guy. KSQL: Streaming SQL engine for Apache Kafka. In *EDBT*, pages 524–533, 2020.
[25] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.
[26] J. Li et al. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288, 2008.
[27] W. Lin et al. StreamScope: Continuous reliable distributed processing of big data streams. In *NSDI*, pages 439–453, 2016.
[28] L. Mai et al. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.*, 11(10):1303–1316, 2018.
[29] J. Meehan et al. S-Store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13):2134–2145, 2015.
[30] Y. Mei et al. Turbine: Facebook's service management platform for stream processing. In *ICDE*, pages 1591–1602, 2020.
[31] C. Paris et al. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017.
[32] Z. Qian et al. TimeStream: Reliable stream computation in the cloud. In *EuroSys*, pages 1–14. ACM, 2013.
[33] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *SOSP*, pages 263–274, 2004.
[34] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.
[35] K. Tangwongsan, M. Hirzel, and S. Schneider. Optimal and general out-of-order sliding-window aggregation. *Proc. VLDB Endow.*, 12(10):1167–1180, 2019.
[36] N. Tatbul et al. Handling shared, mutable state in stream processing with correctness guarantees. *IEEE Data Eng. Bull.*, 38(4):94–104, 2015.
[37] J. Traub et al. Efficient window aggregation with general stream slicing. In *EDBT*, pages 97–108, 2019.
[38] G. Wang et al. Building a replicated logging system with Apache Kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, 2015.
[39] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, and et al. Maximizing determinism in stream processing under latency constraints. In *DEBS*, pages 112–123, 2017.
[40] M. Zaharia, T. Das, H. Li, and et al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.