NOV 4, 2021

READ TIME: 12 MIN

4 Key Design Principles and Guarantees of Streaming Databases

Technology > Use Cases

Guozhang Wang
Software Engineer

Classic relational database management systems (RDBMS) distribute and organize data in a relatively static storage layer. When queries are requested, they compute on the stored data and then return results in the query responses. Streaming database systems, on the other hand, bring infinite and rapidly changing data in motion to the compute layer, by having continuous long-lived queries that keep executing on newly arrived data whenever it becomes available. Although data stream management systems (DSMS) entered the software industry several decades ago, they have primarily been focused on providing approximate, one-pass methodologies—such as data compression, data synopses, and data sampling—in order to handle large scale data streams in real time. As a result, such systems have usually been viewed as an auxiliary implementation, which one would use in addition to periodic, batch-oriented jobs that accumulate and chunk input data streams into finite and static data sets and then process them in offline mode. The attentive reader may recall the term Lambda architecture—a bygone pattern that we nowadays avoid.

We at Confluent believe the new generation of streaming database systems should no longer be satisfied with trading approximate, lossy query results for high scalability and

scale data streams while guaranteeing that the original data ordering is preserved.
2. Common data computations can be parallelized and processed independently with light coordination—or even no coordination—between parallel processors and still generate correct results. (This is called data parallelism and function monotonicity in the literature.) As a result, a streaming database should be distributed in nature, and should be able to automatically recover from failures of any of its distributed instances.

More specifically, we think that the following four principles are must-haves in the design of a streaming database.

This blog post is part two of a series of Readings in Streaming Database Systems. Check out the other posts in this series:

- Overview: Readings in Streaming Database Systems
- Part 1: The Future of SQL: Databases Meet Stream Processing
- Part 3: How Do You Change a Never-Ending Query?

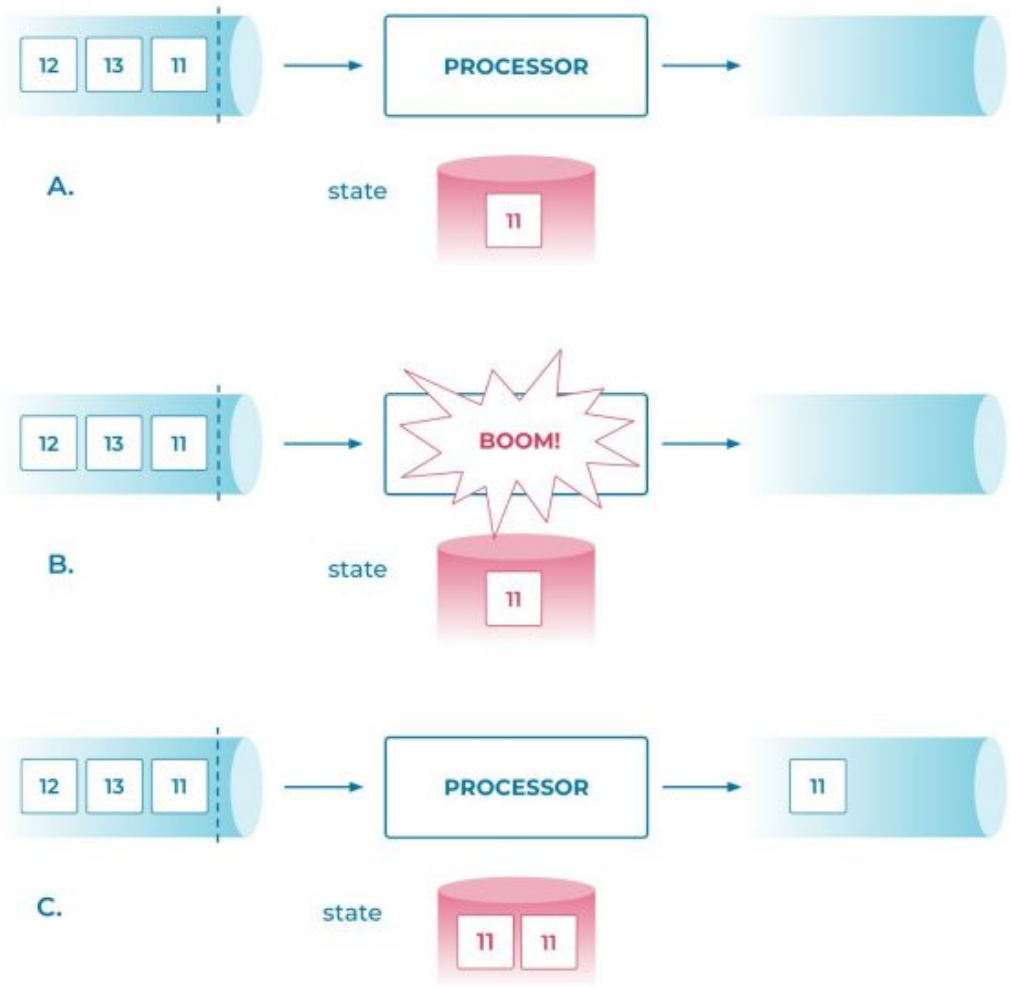
The streaming database auto-recovers from failures (principle 1)

In distributed systems, failures are inevitable: the inter-process RPC can fail, the data storage engine can fail, the query processing engine can fail. And for streaming databases, queries are typically long-lived—they may even run forever. This means that when failures occur, streaming databases need to automatically recover the query from where the failure occurred, and then resume processing instead of restarting the query from scratch. This could result in processing anywhere from seconds to hours to years of historical data. In addition, many streaming queries come along with some state (think about aggregations that keep the running aggregates) that could be partially or even completely lost during a failure. Therefore, a streaming database should be able to restore its state in order to resume queries during a failover procedure.

The streaming database guarantees exactly-once semantics (principle 2)

Being able to auto-recover from a failure and resume the query processing is not sufficient. While most streaming platforms are able to produce correct results during failure-free executions, a streaming database should be able to completely **mask a failure**: its continuous query results should appear as if no failures have ever happened. Such a correctness guarantee is usually referred to as exactly-once processing semantics (EOS). In recent years, this term has lent itself to several different interpretations, and for clarity of presentation in this post we define exactly-once as the following: for each input record of a streaming database, the continuous query processing result will be reflected exactly once, even under failures. Here, the query results could be reflected in both the output data streams and any updates to the

maintained processing states.



The figure above demonstrates these two principles for streaming correctness. It depicts a simple stateful query processor with a single input and a single output stream. Processing state is maintained in a state store and accessed by the processor for reads and writes. The input stream contains only three records, with timestamps 11, 13, and 12 (see a). Let's see what happens during a failure: suppose that *after* processing the record with timestamp 11 and updating the state, but *before* the processing is acknowledged back to the input stream (denoted by the dotted bar), the processor crashes (b). Upon recovery, it would falsely re-process the same input record with timestamp 11 and hence update the state twice (c), causing incorrect results.

The streaming database transparently handles out-of-order records (principle 3)

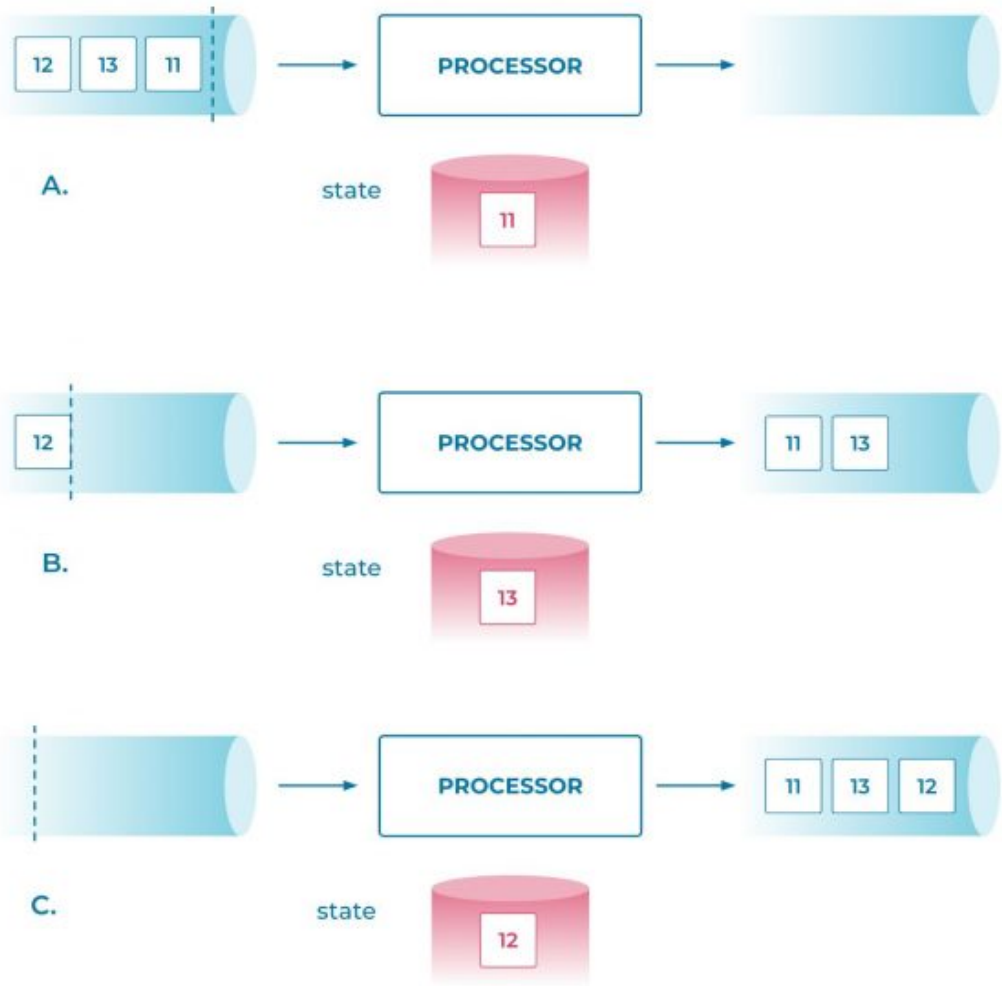
Another important design principle for the correctness of streaming databases is the ability to handle **out-of-order data** in streams. Why is that important? Because the real world is not very orderly—and not just because of thermodynamics! In our technical context, we can observe that the ordering of records in a data stream can and will be disturbed, such that some actually older records (with smaller creation timestamps) appear in the stream after newer records (with larger creation timestamps). In other

words, the general case in practice is that data streams are not guaranteed to be ordered.

The most common causes of out-of-order data include:

- Clock skew (e.g., due to misconfigured NTP settings)
- Network delays or outages (e.g., a mobile device losing connectivity on airplane take-off)

A streaming database is supposed to deliver continuous query results. But, if it does not properly and automatically handle out-of-order records in its input data streams, then the results would be incorrect. Let's continue with our previous example to look at the impact of out-of-order records (a): 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 (b). If that happens, then we would realize that previously emitted results are actually not complete up to time 12 (c). In addition, updating the state with this out-of-order record may also cause incorrect results.



To handle out-of-order records, a streaming database must be able to reason about "stream completeness": it must either wait to emit result records until it can make sure that there would be no out-of-order data earlier than a certain timestamp arriving in the

future (in the example above, that means wait to emit a result for timestamp 13 until it received input with timestamp 12, and be able to tell that there are no other records from the input stream which are earlier than 13), or it must be able to revise the already-emitted result records when out-of-order data is received afterwards (we will talk about this in a subsequent section).

The streaming database provides consistent query results (principle 4)

Compared with traditional data-driven applications, which centralize all their state in a shared database management system and use stored procedures or triggers to synchronously derive new information from the raw data, modern applications that build on streaming database systems rely on the asynchronous nature of stream processing to allow collaborations between loosely coupled subsystems and sub-teams. However, this means that there's usually a delay from the time when the source dataset is updated to the time when the corresponding query result is available in the derived dataset. Furthermore, when multiple queries are concatenated together (i.e. one query's output is another query's input), then at any given time, some derived query state may reflect the source dataset update while others do not; i.e. they would not be **consistent**. Streaming databases should strive to tackle this challenge and provide query consistency to application developers.

A persistent log-based approach

So far we've discussed four key principles in streaming database designs, and now we will talk about the approach that ksqlDB employs to tackle these principles. At its core, ksqlDB decouples the mechanisms that handle exactly-once consistency from the ones which achieve completeness without out-of-order data, and provides users with flexible performance and correctness trade-off customizations.

As mentioned above, a key observation from this approach is that persistence is, and will continue to be, inexpensive. We've seen that this trend holds even as storage devices move from hard disk drives to solid state drives, where sequential data access continues to perform comparably to main memory access. Based on that observation, we built ksqlDB, a log-based streaming database developed on top of Apache Kafka, as part of Confluent Cloud. ksqlDB's internal plumbing is heavily integrated with Kafka topics as its source-of-truth storage layer. In addition, since Kafka topics are organized as append-only logs that are naturally ordered based on the append offsets, they produce a fully linearized history of computations and data updates.

Write atomicity

As a persistent log-based storage layer, Kafka's own log replication mechanism guarantees that record appends are durable and highly available. In addition, since version 0.11.0.0, Kafka has provided an idempotent and transactional log write protocol

to make sure that within a transaction, log appends across multiple Kafka topics are all successful or the transaction fails.. As a result, write atomicity can be achieved at ksqlDB's storage layer. Unlike traditional two-phase commit protocols that require the data to be written twice—once for the log and another for the data—this mechanism only requires writing data once in the log, and leverages the append ordering in its fully linearized history to determine which records should be considered as committed or aborted. (For more details about the transactional log append implementations within Kafka, see the blog post Transactions in Apache Kafka).

Exactly-once semantics

Guaranteeing write atomicity at the storage layer (Kafka) alone is not sufficient in a streaming database. At the computational layer atop the storage, we also need to ensure that when an error occurs, we can resume from a consistent processing state after failover.

To provide this property, ksqlDB leverages Kafka Streams, a stateful stream processing library for Kafka, as its distributed runtime for executing long-lived, continuous queries. In ksqlDB, these are called "persistent queries". Persistent queries read input data streams and derive new data streams or materialized views, such as continuous aggregates over windows (we will talk more about persistent queries, along with other types of queries, in a later blog post in this series). In practice, a query's output streams can be read in turn by another query as its input, forming an upstream-downstream pipeline. Such queries submitted to ksqlDB are compiled and executed as Kafka Streams applications that run indefinitely until terminated, and a single Kafka Streams application can execute on multiple distributed instances, which process input data streams as Kafka topic partitions in parallel. In addition, Kafka Streams instances can keep local state stores for stateful processing logic, such as running aggregates.

The persistent query processing within an instance executes a cyclic read-process-write operation for each input record: 1) fetch the record from the input streams, 2) process the record, updating the corresponding state if necessary, and 3) emit the output record(s) as the processing result to output streams. After that, the instance can commit the position of the processed record to Kafka brokers, indicating that the record has completed processing, and then continue to the next record, and the cycle would repeat.

The key idea to preserve processing consistency is to back up the state store updates in separate Kafka topics as the stores' changelogs. Changelogs are similar to the write-ahead-logs in traditional databases, but in ksqlDB they are also replicated inside Kafka, and hence are highly available. Therefore, we can reduce the complexity of recovering a failed stream processing state by replaying the corresponding changelogs. All of the operations within the read-process-write cycles can be translated as record appends to certain logs—committing on input stream positions as appending to a specific offset topic, updating states as appending to the changelog topics, and emitting output as

appending to output topics. We can thus rely on the write atomicity provided by Kafka to make sure that processing can always resume from a consistent state upon failure recovery. More specifically, as long as we can reset the input streams' position and restore the processing state by replaying the changelog in a read-committed mode, they are guaranteed to be aligned up to the last successful transaction before the failure.

Processing completeness

In addition to the consistency guarantee, ksqlDB's design also leverages the log architecture to handle out-of-order data for processing completeness. Here, a simple approach would be to couple processing completeness together with consistency handling, by deferring the transaction commit until all data up to a certain point have been processed. Again, as described above, such approaches require some coordination between the upstream and downstream queries in order to reason about input data completeness and to avoid emitting records in between.

Rather than deferring output emission until completeness, we have chosen a different, optimistic approach in ksqlDB's design. ksqlDB does not try to prevent incompleteness through coordinated blocking, but instead emits output to the persistent Kafka topics early whenever possible, and "refines" the emitted partial results downstream when out-of-order data does occur. This is, again, based on the fact that the upstream-downstream communication channel is built on the persistent Kafka logs: because the logs are highly available, they can always be replayed and reprocessed in the same order whenever this is needed by the system, e.g. in case of instance migrations or failovers of ksqlDB servers. Therefore, as long as the downstream query's processing logic is monotonic, i.e. it can produce the same final outputs for any non-deterministic ordering of input streams, its upstream queries do not need to prevent records from being emitted to the intermediate Kafka topics.

For example, if the upstream query's output type is a time-evolving table, then records emitted later on this linearized log are treated as revisions to the previously emitted records, and hence can still be processed to compensate for the earlier partial result's effects. This design principle can also be viewed as the duality of time-evolving table entities and their corresponding changelog streams within ksqlDB.

Beyond eventual consistency

Observant readers might have already noticed that by only relying on continuous revisions to compensate for early emitted partial results, ksqlDB's query results are eventually consistent: since the persistent query results are by nature derived asynchronously from the source data streams, they may not reflect a view consistent with the source data streams at a given time. For example, if a persistent query in ksqlDB generates one or more materialized views that keep various running aggregates of the input stream events, a pull query on these materialized views may not return results reflecting all of the events that have been inserted into the input stream so far,

and the query results from different views may not align on the same snapshot of the input stream, either.

In traditional database systems, a common approach to tackle this and provide stronger consistency guarantees is to "block on write": that is, to combine both the source table update and any materialized view updates in a transaction, so that the upsert on the input data would not return until all of the computations used to update the derived results have been completed. In ksqlDB, we use a different approach: we "block on read", so that persistent queries would not delay any updates on the input streams they read from, and instead, only pull queries on the generated results that require stronger consistency guarantees would potentially be delayed. Again, this leverages the fact that ksqlDB's storage, Kafka logs, presents a fully linearized history of computations, from which we can easily reason about the versions of various materialized views from continuous updates.

From a user's perspective, ksqlDB would allow them to reason about:

1. **Staleness:** How much is my streaming application, which queries on the derived data inside ksqlDB, currently lagging behind the source data streams?
2. **Completeness:** Are all records inserted into the source data streams earlier than a given time completely reflected in my ksqlDB query results?
3. **Consistency:** Are my queries issued on a group of derived data streams or tables reflecting the same snapshot of the ksqlDB state? For example, if two materialized views are derived from the same raw data streams, then a single update in the raw data stream should be reflected in either all or none of the materialized views when I query them separately. Also, if I query the same derived data set multiple times, is it guaranteed that the query results will never "go backwards"—i.e. will a later query result only reflect more updates, not fewer, regardless of failure recoveries?

What's next

In this blog post, we summarized a few challenging design principles for modern streaming databases that act as a source of truth for stream data management and query processing systems, and we presented ksqlDB's persistent log-based approach to following these principles. We will continue this series to discuss query evolution in ksqlDB and what a fully mature streaming database would be capable of in the next post —stay tuned!

If you're interested in using ksqlDB, get started today via the standalone distribution or with Confluent, and join the community to ask a question and find new resources.

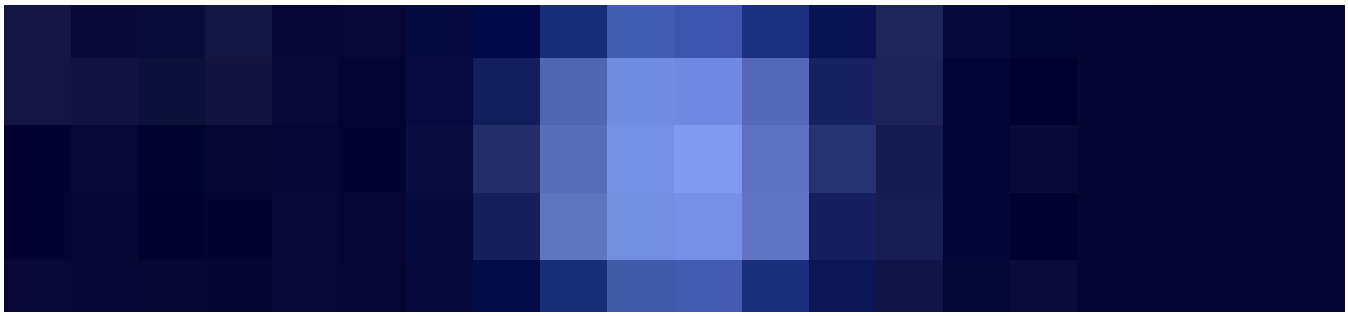
Get Started

Other posts in this series

- Overview: Readings in Streaming Database Systems
- Part 1: The Future of SQL: Databases Meet Stream Processing
- Part 3: How Do You Change a Never-Ending Query?

Guozhang Wang is a PMC member of Apache Kafka, and also a tech lead at Confluent leading the Kafka Streams team. He received his Ph.D. from Cornell University where he worked on scaling data-driven applications. Prior to Confluent, Guozhang was a senior software engineer at LinkedIn, developing and maintaining its backbone streaming infrastructure on Apache Kafka and Apache Samza.

Technology < Use Cases

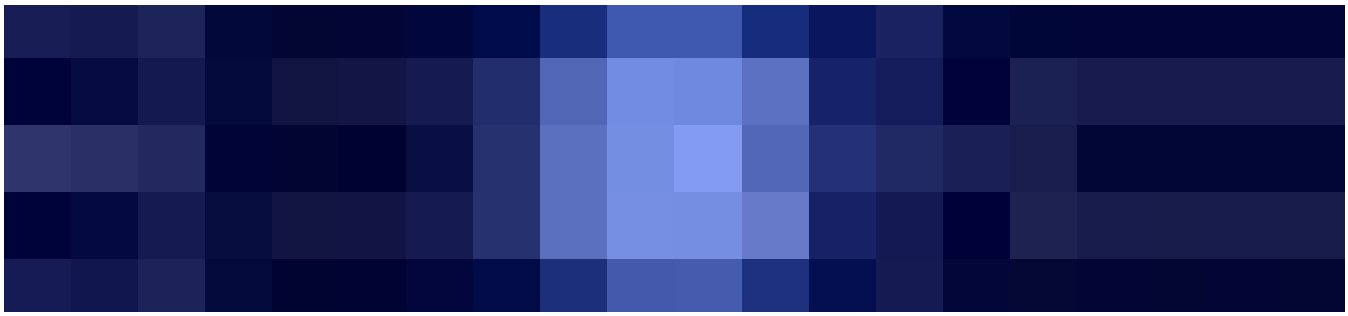


Introducing Versioned State Store in Kafka Streams

AUG 24, 2023

Versioned key-value state stores, introduced to Kafka Streams in 3.5, enhance stateful processing capabilities by allowing users to store multiple record versions per key, rather than only the single latest version per key as is the case for existing key-value...

VICTORIA XIA



Delivery Guarantees and the Ethics of Teleportation

APR 25, 2023

This blog post discusses the two generals problems, how it impacts message delivery guarantees, and how those guarantees would affect a futuristic technology such as teleportation.

WADE WALDRON