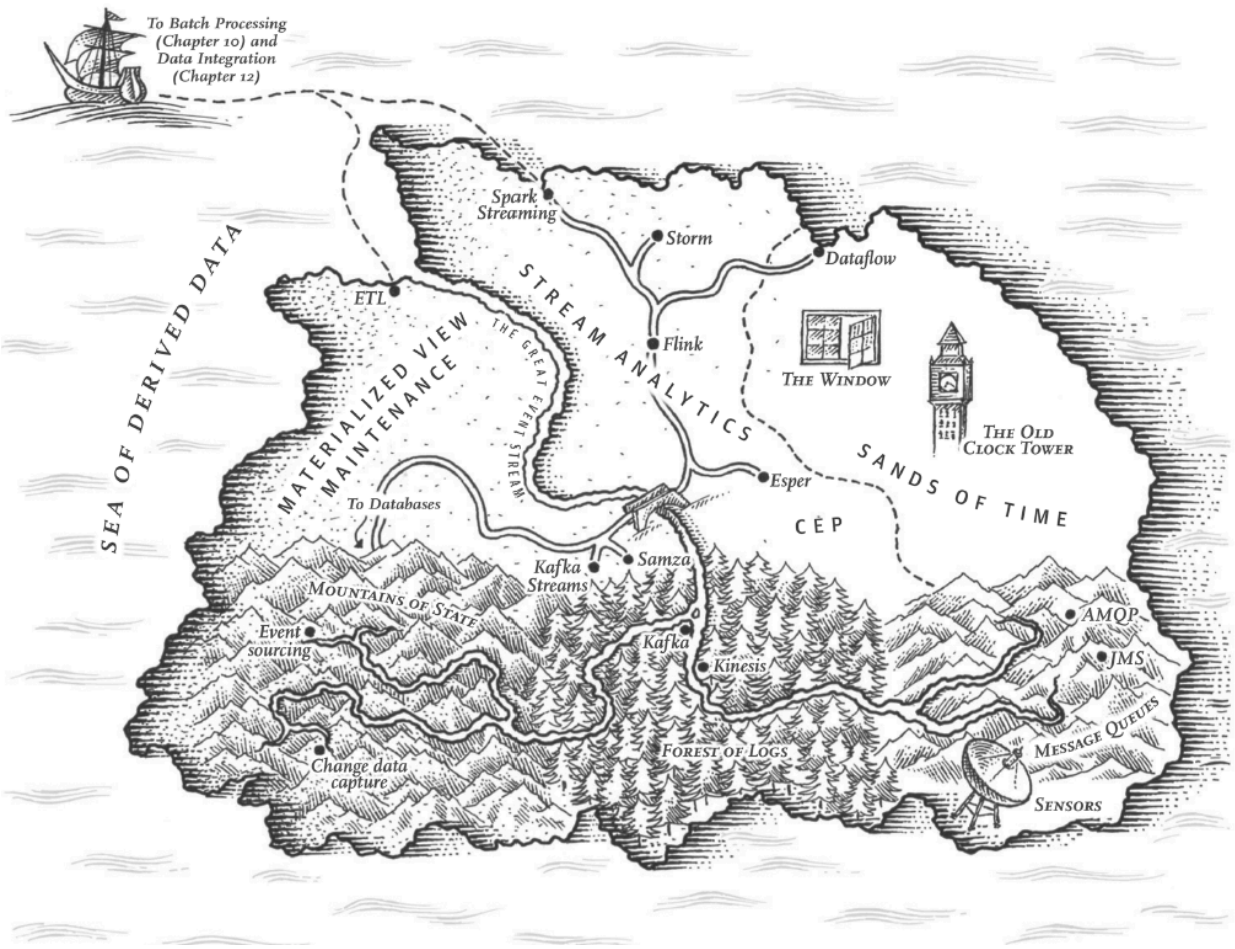


Chapter 11 Streaming Processing

In Chapter 10, one big assumption is that the input is bounded(known and finite size), so the batch process knows when it has finished reading its input. In reality, a lot of data is unbounded because it arrives gradually over time.
We will look at **event streams** as a data management mechanism: **unbounded, incrementally processed** counterpart to the batch data. We will see how streams are represented, stored, transmitted over network, and investigate the relationship between streams and databases, and explore approaches and tools for processing those streams continually, and ways that can be used to build applications.

Transmitting Event Streams

- When the input is a file (**a sequence of bytes**), the first processing step is to parse it into a **sequence of records**. In stream processing, a record is known as an **event**:

A small, self-contained, immutable object containing the details of something that happened at some time. An event contains a timestamp indicating when it happened according to a time-of-day clock.

- **An event may be encoded as text string, JSON, or some binary form.** This encoding allows you to store an event, by appending to a file, inserting into a relational table, or writing to a document database. Or allow you to send the event over the network to another node to process it.
- In streaming terminology, an event is generated once by a **producer / publisher / sender**, then processed by multiple **consumers / subscribers / recipients.** Related events are grouped together into a **topic or stream**.
- **A file or database** is sufficient to connect producers and consumers: a **producer** writes every event it generates to the **datastore**, and each **consumer** periodically **polls** the datastore to check for events that appeared **since it last ran**.
  For processing with low delays, polling can be expensive if the datastore is not designed for this usage, it may be better for **consumers to be notified when new events appear**. Databases traditionally did not support this notification mechanism well.

Messaging Systems

- A common approach for notifying consumers about new events is to use a **messaging system**: a producer sends a message containing the event, then pushed to consumers.
- Messaging system allows multiple producer nodes to send messages to the same topic and multiple consumer nodes to receive messages in a topic.
- What if the producers send messages faster than the consumers process them? **Three options: the system drops messages, buffer messages in a queue, or apply backpressure** (flow control; blocking the producer from sending more messages).
- If messages are buffered in a queue, it is important to understand what happens as that queue grows. Does the system crash if the queue no longer fits in memory, or does it write messages to disk? If so, how does disk access affect the performance of the messaging system?
- What happens if nodes crash or temporarily go offline? Are any messages lost? In databases, durability may require some combination of writing to disk and/or replication which has a cost. If losing messages is OK, you can get higher throughput and lower latency.

Direct messaging from producers to consumers

Many messaging systems use direct network communication between producers and consumers without going via intermediary nodes. The faults they can tolerate are limited.

Message brokers

- Send messages via a **message broker / message queue**: a kind of database that is optimized for handling message streams. It runs as a server, with producers and

consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker.

- **By centralizing the data in the broker, these systems can easily tolerate clients come and go (connect, disconnect, crash), and the question of durability is for the broker instead.** Some message brokers only keep messages in memory, while others (depending on configuration) write them to disk so they won't be lost in case of a broker crash.
  For slow consumers, they allow unbounded queueing (as opposed to dropping messages or backpressure), this can also depend on the configuration.
- Consumers are **asynchronous**: when a producer sends a message, it only waits for the broker to confirm the message has been buffered. The delivery to consumers will happen later.

Message brokers compared to databases

Differences between message brokers and databases:

- Databases keep data until it is explicitly deleted, whereas most message brokers automatically delete a message when it has been delivered to consumers. Message brokers are **not for long-term data storage**.
- Message brokers assume their working set is small, the **queues are short.**
- Databases support secondary indexes and various ways of searching, message brokers support **subscribing to a subset of topics matching some pattern**.
- When querying a database, the result is typically based on a point-in-time snapshot of the data; if data changes by another client, the first client won't know. Message brokers do not support arbitrary queries, but **notify clients when data changes**.

Multiple consumers

When multiple consumers read messages in the same topic, two main patterns of messaging:

- **Load balancing**
  Each message is delivered to **one** consumer, consumers share the work of processing the messages in the topic. The broker assigns messages to consumers arbitrarily.
  This pattern is useful when the **messages are expensive to process**, and can add consumers to parallelize the processing.
- **Fan-out**
  Each message is delivered to **all** consumers. Allows several independent consumers to the same broadcast of messages without affecting each other.

The two patterns can be combined: two groups of consumers subscribe to a topic, each group receives all messages, within each group only one of the nodes receives each message.

Acknowledgments and redelivery

Consumers may crash, a broker may deliver a message to a consumer but the consumer never or partially processes it. To ensure message not lost, message brokers use **acknowledgments**:

a client tells the broker when it finished processing a message so that the broker can remove it from the queue.
It happens if a message was processed by a consumer, but **acknowledgement was lost in the network**. Handling this requires an **atomic commit protocol**.

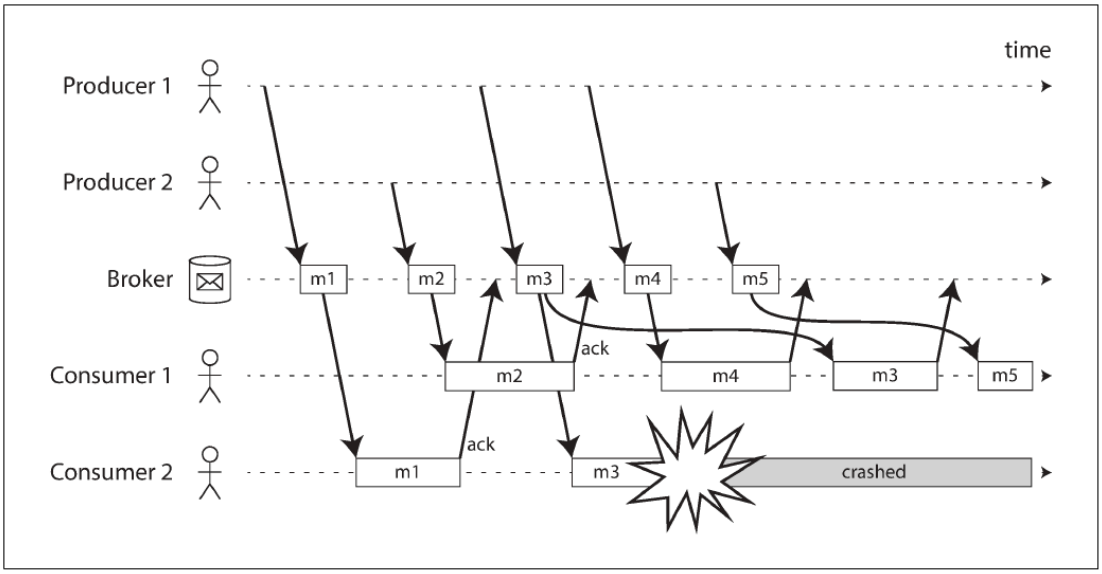


Figure 11-2. Consumer 2 crashes while processing m3, so it is redelivered to consumer 1 at a later time.

The combination of load balancing and redelivery leads to messages being reordered. To avoid this, use a **separate queue per consumer (not load balancing feature)**.
Message reordering is not a problem if messages are independent of each other, but it is important if they are causal dependent.

Partitioned Logs

- Sending a packet over a network or making a request to a network service is a **transient operation**.
- This difference has a big impact on how derived data is created:
  A key feature of batch processes is that you can run them repeatedly, experimenting with the processing steps, without damaging the input (since the input is read-only).
  Message systems are built around a transient messaging mindset, receiving a message is destructive if the acknowledgement makes it to be deleted, you cannot run the same consumer to get the same result. If you add a new consumer, it only starts receiving messages after it is registered.
- The idea behind log-based message brokers: combining the durable storage of databases with the low-latency notification facilities of messaging.

Using logs for message storage

- A log is simply an **append-only sequence of records** on disk. We discussed logs in the context of log-structured storage engines and write-ahead logs, and in the context of replication.
- Use this to **implement a message broker**: a producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially. If a consumer reaches the end of the log, it waits for notification of new messages.
- The log can be partitioned to **scale to higher throughput** in multiple machines. **Each partition is a separate log** to be read and written independently from other partitions. A **topic** can be **a group of partitions** that carry **messages of the same type**.
- Within each partition, the broker assigns a **monotonically increasing sequence number**, or **offset**, to every message. The messages within a partition are **totally ordered**. **No ordering guarantee across different partitions**.

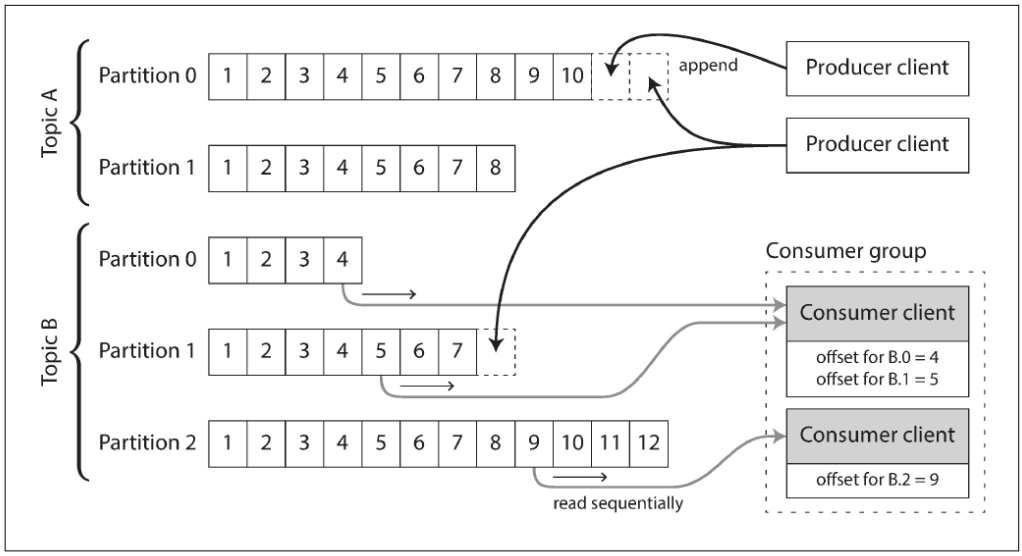


Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

- **Apache Kafka, Amazon Kinesis Streams, Twitter's DistributedLog** are **log-based message brokers. Google Cloud Pub/Sub** is architecturally similar but exposes a JMS-style API rather than a log abstraction.
- Even though they write all messages to disk, they can achieve throughput of **millions of messages per second** by **partitioning across multiple machines**, and **fault tolerance by replicating messages**.

Logs compared to traditional messaging

- The log-based approach supports **fan-out** messaging, because several consumers independently read the log without affecting each other.

- To achieve **load balancing** across **a group of consumers**, instead of assigning individual messages to consumer clients, **the broker assigns entire partitions to nodes in the consumer group.**
- Each client consumes all the messages in the partitions it has been assigned. When a consumer is assigned a log partition, it reads the messages in the partition sequentially with single-thread. This load balancing approach has downsides:
    1. The number of nodes consuming a topic can be at most the number of partitions in that topic.
    2. If a single message is slow to process, it holds up the processing of subsequent messages in that partition.
- **JMS/AMQP style message broker:**
  When messages are **expensive** to process and **parallelize processing** message by message, message ordering is not so important.
- **Log-based message broker:**
  With **high throughput**, each message is **fast** to process and **message ordering** is important.

Consumer offsets

- Consuming a partition sequentially can easily tell which messages were processed: **messages** with an offset less than a **consumer's current offset** have already been processed, and messages with a greater offset have not yet been seen.
  The **broker** no need to track acknowledgments for every message, it only needs to **record the consumer offsets**.
  The reduced bookkeeping overhead and the opportunities for batching and pipelining in this approach help increase the throughput of log-based systems.
- Offset is similar to **log sequence number** in **single-leader database replication**. In database replication, the log sequence number allows a follower to reconnect to a leader after it is disconnected, and resume replication without skipping any writes.
  The same principle is used: **the message broker** behaves like a **leader** database, the **consumer** like a **follower**.
- If a consumer node fails, another node in the consumer group takes over the failed consumer's partitions, it starts consuming messages at the last recorded offset. If the consumer had processed subsequent messages but not yet recorded their offset, those messages will be processed a second time upon restart. We will discuss it later.

Disk space usage

- If you only append to the log, you will eventually run out of disk space. To reclaim disk space, the log is **divided into segments**, old segments are deleted or moved to archive storage.
- If a **slow consumer** cannot keep up with the rate of messages, and it falls far behind that its consumer offset points to a deleted segment, it will miss some messages.
  The log implements a **bounded-size buffer(circular buffer / ring buffer)** that discards old messages when it gets full. But since the buffer is on disk, it can be quite large.

- **Back-of-the-envelope calculation:**
  When writing, a large hard drive has a capacity of 6 TB and a write throughput of 150 MB/s. If writing messages at the fastest rate, it takes about 11 hours to fill the drive. The disk can buffer 11 hours' messages, after which it starts overwriting old messages. This ratio remains the same, even if you use many hard drives. In practice, deployments rarely use the full bandwidth of the disk, the log can keep a buffer of several days' or weeks' messages.
- Regardless of how long you retain messages, the **throughput of a log remains constant**, since every message is written to disk.
  This is in contrast to those messaging systems who keep messages in memory by default and only write them to disk if the queue grows too large: such systems are fast when queues are short and become slower when they start writing to disk, so the throughput depends on the amount of history retained.

When consumers cannot keep up with producers

- We discussed three choices if a consumer cannot keep up with the rate producers send messages: dropping messages, buffering, or applying backpressure. In this taxonomy, the log-based approach is a form of **buffering** with a large but fixed-size buffer.
- If a consumer falls behind more than retained time on disk, it cannot read those messages, the broker effectively drops old messages that go back further than the **size of the buffer** can accommodate.
  You can **monitor** how far a consumer is behind the head of the log, and raise an **alert** if it falls behind significantly. As the **buffer is large**, there is enough time for a human operator to fix the slow consumer and allow it to catch up before it starts missing messages.
- Even if a consumer falls far behind and starts missing messages, **only that consumer is affected**, it does not affect other consumers.
  This has a big **operational advantage**: you can experimentally consume a production log for development, testing, or debugging purposes, without worrying about disrupting production services.
  When a consumer is shut down or crashes, it stops consuming resources, the only thing remains is **its consumer offset**.

Replaying old messages

- With AMQP- and JMS-style message brokers, processing and acknowledging messages is a destructive operation, since it causes the messages to be deleted on the broker.
  On the other hand, in a log-based message broker, consuming messages is like reading from a file: it is a read-only operation that does not change the log.
- The only **side effect** of processing is that the **consumer offset moves forward**. But the offset is under consumer's control, it can be manipulated: e.g., you can start a copy of a consumer with yesterday's offsets and write the output to a different location, in order to reprocess the last day's worth of messages.

- This aspect makes log-based messaging like the **batch processes**, where **derived data** is separated from **input data** through a **repeatable transformation process**. It allows more **experimentation** and easier **recovery from errors and bugs**, making it a good tool for **integrating dataflows** within an organization.

Databases and Streams

- We saw log-based message brokers take ideas from databases and applied them to messaging. We can also take ideas from messaging and streams and apply them to databases.
- An event is a record of something that happened at some point in time. This may be a user action, a sensor reading, or a write to a database. Something written to a database is an event that can be **captured, stored, and processed**. The connection between databases and streams is deeper.
- **Replication log** is a stream of database write events, produced by the leader as it processes transactions. The followers apply that stream of writes to their own copy of the database and catch up with the leader. The events describe the data changes that occurred.
- We discussed the **state machine replication principle** in “Total Order Broadcast”: if every event represents a write to the database, and every replica processes the same events in the same order, then the replicas will end up in the same final state.
- We will discuss problems in heterogeneous data systems and how to solve it by bringing ideas from event streams to databases.

Keeping Systems in Sync

Each system has its own copy of the data, stored in its own representation that is optimized for its own purposes. As the same or related data appears in several different places, they need to be kept in sync with one another.
Dual writes: the application code explicitly writes to each of the systems when data changes. e.g., first write to the database, then update the search index, then invalidate cache entries.

1. Dual writes have serious problems, one is a **race condition**.
   Unless there are concurrency detection mechanisms like the version vectors. You may not notice that concurrent writes occurred, one value will silently overwrite another.

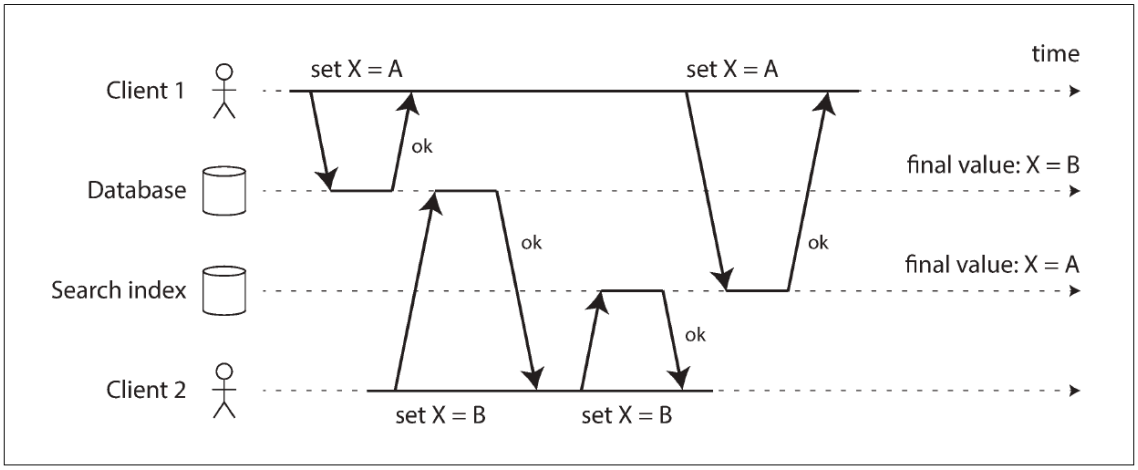


Figure 11-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

2. Dual writes also have a **fault-tolerance problem**: one of the writes fails while the other succeeds. Two systems becoming inconsistent with each other. This is an atomic commit problem which is expensive to solve.

If only one replicated database with a single leader, then the leader determines the order of writes, so the state machine replication approach works. However, the database may have a leader and the search index may have a leader, so conflicts can occur. Is it possible to make the search index a follower of the database?

Change Data Capture

- **Change data capture (CDC): the process of observing all data changes written to a database and extracting them in a form in which they can be replicated to other systems.**
- CDC is interesting if changes are made available as a stream, immediately as they are written. You can capture the changes in a database and continually apply the changes to a search index. The search index and other derived data systems are consumers of the change stream.

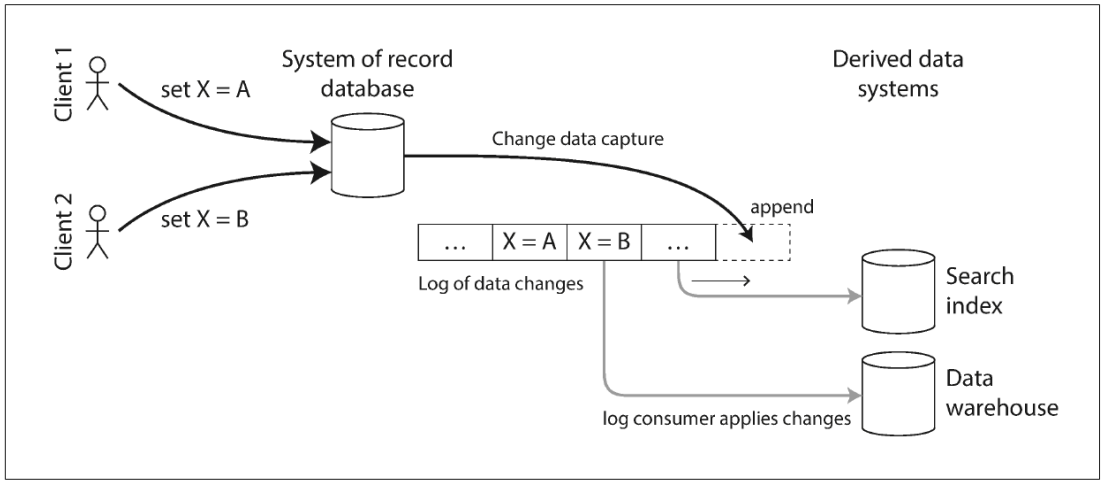


Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

Implementing change data capture

- Log consumers are **derived data systems**, the data stored in search index and warehouse is another view of data in the system of record.
- **Change data capture is a mechanism for ensuring changes made to the system of record are reflected in derived data systems so that the derived systems have an accurate copy of the data.**
- CDC makes **one database the leader (the one captures the changes)**, and turns the others into followers. A **log-based message broker** is suited for transporting the change events from the source database, since it keeps the ordering of messages
- **Database triggers** can be used to implement change data capture by **registering triggers** that observe all changes to data tables and add corresponding entries to a **changelog table**. But they have significant **performance** overheads.
- **Parsing the replication log** is more robust, although the need to **handle schema changes**. PostgreSQL uses an API that decodes the write-ahead log.
- Like message brokers, change data capture is **asynchronous**: the system of record database does not wait for the change applied to consumers before committing.
- This design has **operational advantage:** adding a slow consumer does not affect the system of record too much, but it has all the issues of replication lag.

Initial snapshot

- If you have the log of all changes to a database, you can reconstruct the entire state of the database by replaying the log. But this requires too much disk space, and replaying takes too long, the log needs to be truncated.
- Building a new full-text index requires a full copy of the entire database. If you don’t have the entire log history, you start with a consistent snapshot.

- The snapshot of the database must have a known position or offset in the change log, so you know at which point to start applying changes after the snapshot has been processed.

Log compaction

- If you only keep a small amount of log history, you need to process the snapshot every time when adding a new derived data system. Log compaction provides an alternative.
- We discussed log compaction in “Hash Indexes” in log-structured storage engines: **The storage engine periodically looks for log records with the same key, throws away duplicates, and keeps only the most recent update for each key. This compaction and merging process runs in the background.**
- In a log-structured storage engine, an update with a special null value (tombstone) indicates a key was deleted, and causes it to be removed during log compaction. If no deletion, only the latest value is retained.
  The same idea works in log-based message brokers and change data capture. **It’s sufficient to keep the most recent write for a particular key.**
- To rebuild a derived data system like a search index, start a new **consumer** from **offset 0** of the **log-compacted topic**, and scan all messages in the log. The log is guaranteed to contain the **most recent value for every key** in the database.
  Can obtain a full copy of the database contents without having to take another snapshot of the CDC source database.
- This log compaction feature is supported by Apache Kafka. It allows the message broker to be used for **durable storage**, not just for **transient messaging**.

API support for change streams

Databases support change streams as a first-class interface. Examples:

1. Allows queries to subscribe to notifications when the results of a query change.
2. Provide **data synchronization** based on a **change feed**, also made available to applications.
3. **Subscribe** to data changes and update the **user interface**.
4. Allows **transactions to export data from a database in the form of a stream**. The database represents an output stream in the relational data model as a **table** into which transactions can insert tuples, but which cannot be queried.
   The stream then consists of the log of tuples that committed transactions have written to this special table, in the order they were committed. External consumers asynchronously consume this log to update derived data systems.

Kafka Connect integrates **change data capture** tools for a wide range of **database** systems with **Kafka**. Once the stream of change events is in Kafka, it is used to update derived data systems like search indexes, also feed into stream processing systems.

Event Sourcing

Similarly to CDC, event sourcing stores all changes to the application state as a log of change events. The difference is event sourcing applies it at a different level of abstraction:

- In CDC, the application uses the database in a **mutable** way. The log of changes is extracted from the database at a low level, ensuring the **order** of writes avoiding the **race condition**. The application writing to the database no need to be aware CDC is occurring.
- In event sourcing, the application logic is built on the basis of **immutable** events, the event store is **append-only**. **Events reflect things happening at the application level**, rather than low-level state changes.

Event sourcing is a powerful technique for **data modeling**: from an application point of view it is better to record the user's actions as immutable events, rather than recording the effect of actions on a mutable database.
Event sourcing makes it easier to **evolve applications over time**, helps with **debugging** by making it easier to understand against application bugs.
The event sourcing approach allows the **new** side effect to be **chained** off the existing event.
Event sourcing is similar to the **chronicle data model**, also similar to the **fact table** in star schema.

Deriving current state from the event log

Users generally expect to see the current state of a system, not the history of modifications. E.g., shopping website users expect to see contents in cart, not append-only list of changes they made.
Applications using event sourcing take the log of events and transform it into application state for showing to a user, it should be deterministic so you can run it again and derive the same application state from the event log.
Like change data capture, replaying the event log allows you to **reconstruct the current state** of the system. However, **log compaction** needs to be handled differently:

- A CDC event for update contains the entire new version of the record; the current value for a primary key is determined by the **most recent event** for that primary key. Log compaction can discard previous events for the same key.
- With event sourcing, events are modeled at a **higher level**: an event expresses the **intent of a user action**, not the state update as a result of the action.
  Later events do not override prior events, you need a **full history** of events to reconstruct the final state. Log compaction cannot be the same.

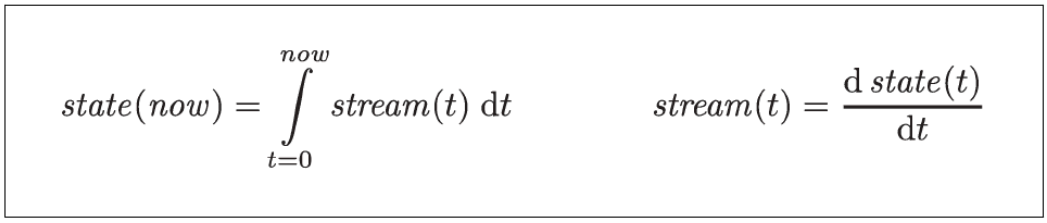
Applications typically store **snapshots** of the **current state** derived from the log of events, no need to repeatedly reprocess the full log.
However, this is a **performance optimization to speed up reads and recovery from crashes**; the **intention** is that the system store all raw events forever and reprocess the full event log whenever required.

Commands and events

- The event sourcing philosophy **distinguish between events and commands**:
  A user request is initially a command, the application first validates that it can execute the command. If the command is accepted, it becomes an event, which is durable and immutable.
- A consumer of the event stream is not allowed to reject an event: by the time the consumer sees the event, it is already an immutable part of the log, and may have already been seen by other consumers.
  Validation of a command needs to be **synchronously** before it becomes an event.
  Use a **serializable transaction atomically** validates the command and publishes the event.
- The user request to **reserve a seat** can split to **two events**: first a **tentative reservation**, and then a separate **confirmation event** once the reservation has been validated.(Implementing linearizable storage using total order broadcast)
  This split allows the **validation** to take place in an **asynchronous** process.

State, Streams, and Immutability

- The immutability makes batch processing, event sourcing, and change data capture powerful.
- Databases store the current state of the application, which representation is the best to serve queries. How do databases support updating, deleting, inserting fits immutability?
- The key idea is that mutable state and an append-only log of immutable events do not contradict each other. The changelog represents the evolution of state over time.



$$state(now) = \int_{t=0}^{now} stream(t)\,\mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\,state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

- Store changelog durably makes the state reproducible.
  If the log of events is your system of record, and any mutable state is derived data, it is easier to reason about the flow of data through a system.
- **Log compaction is a bridge between the log and database state: it retains the latest version of each record, and discards overwritten versions.**

Advantages of immutable events

- Accountants have been using immutability in financial bookkeeping. When a transaction occurs, it is recorded in an append-only ledger, profit and loss or the balance sheet, are derived from the transactions in the ledger by adding them up.

- If you accidentally deploy buggy code that writes bad data to a database, recovery is much harder if the code overwrites data. With an **append-only log of immutable events**, it is easier to **diagnose** and **recover** from the problem.
- Immutable events **capture more information** than just the current state.

Deriving several views from the same event log

- By separating mutable state from the immutable event log, you can derive different read-oriented representations. This is like having multiple consumers of a stream. Kafka Connect sinks export data from Kafka to various databases and indexes.
- Having explicit translation steps from an event log to a database makes it easier to evolve your application over time: a new feature with data in a new way, you can build a read-optimized view for the new feature, no need to modify existing systems.
  Running old and new systems side by side is easier than performing a complicated schema migration in an existing system. Once the old system is no longer needed, shut it down and reclaim its resources.
- Storing data is easy if there is no need for query or access. Otherwise you gain flexibility by separating the form in data written from the form read, and allow several read views. This idea is **command query responsibility segregation (CQRS).**
- The traditional approach to database and schema design is based on data written in the same form as to be queried. Debates about normalization and denormalization become irrelevant if you translate data from a **write-optimized event log** to **read-optimized application state**: It is reasonable to **denormalize** data in the **read-optimized views**.
- We discussed **Twitter's home timelines**, a **cache** of recently written tweets by the people a particular user is following (like a mailbox).
  Example of a **read-optimized state**: home timelines are highly denormalized, since tweets are duplicated in all timelines the people following you. However, the **fan-out service** keeps this duplicated state **in sync** with new tweets and new following relationships, the duplication is manageable.

Concurrency control

- **Downside** of event sourcing and change data capture: the consumers of the event log are **asynchronous**. A user makes a write to the log, then reads from a log-derived view and finds their write is not reflected in the read view.
- **One solution** is to perform the updates of the read view **synchronously** with the event log. This requires a transaction of writes into an atomic unit. Either keep them in the same system, or a distributed transaction across the different systems.
- **Deriving the current state from an event log simplifies concurrency control.**
  Multi-object transactions require data change in different places. With **event sourcing**, you can design an event as a **self-contained description of a user action**. The user action requires only a single write in one place, appending events to the log, easy to make atomic.
- If the **event log** and the **application state** are **partitioned in the same way**, a **single-threaded** log consumer **needs no concurrency control for writes.**

The log defines a serial order of events in a partition, removing the non-determinism of concurrency. If an event touches multiple state partitions, a bit more work is required Chapter 12.

Limitations of immutability

Many systems rely on immutability. Databases use immutable data structures or multi-version data to support snapshots. Version control systems like Git also use immutable data to preserve version history. Limitations of immutability:

Performance reasons:

- For workloads with **high rate of updates and deletes** on a **small dataset**, the immutable history grows large, fragmentation may be an issue, and the performance of compaction and garbage collection is crucial for operational robustness.

Administrative reasons:

- In case data needs to be deleted. E.gPrivacy regulations require deleting personal information after they close their account, data protection legislation requires removing erroneous information, or an accidental leak of sensitive information may need to be contained.
- In these cases, you need to rewrite history and pretend the data was never written. Deleting data is surprisingly hard, since copies can live in many places: storage engines, filesystems, and SSDs, and backups are often immutable to prevent deletion or corruption.

Processing Streams

We talked about:

- **Where streams come from: user activity events, sensors, and writes to databases.**
- **How streams are transported: direct messaging, message brokers, event logs**.

Three options to process streams:

1. Take the data in the events and **write it to a database, cache, search index, or other storage system**, from where it can then be queried by other clients.
2. **Push the events to users** in some way, e.g. send email alerts or push notifications, or stream the events to a real-time visualized dashboard.
3. **Process one/more input streams to produce one/more output streams**. Streams go through a pipeline with processing stages before they end up at an output (1 or 2).

We will discuss option 3: processing streams to produce other derived streams.

A piece of code that processes streams like this is **an operator** or **a job**.

- **Pattern of dataflow:** A stream processor consumes input streams in a **read-only** fashion and writes its output to a different location in an **append-only** fashion.
- **Patterns of partition and parallelization**: similar to MapReduce and dataflow engines in Chapter 10.
- **Basic mapping operations like transforming and filtering records** also work the same.

The difference to batch jobs: a stream never ends.

- Sorting does not make sense with an unbounded dataset, sort-merge joins cannot be used.
- Fault-tolerance mechanisms must change: a failed batch job can be restarted, but a stream job has been running for years. Restarting from the beginning after a crash is not an option.

Uses of Stream Processing

Stream processing is used for monitoring purposes, an organization wants to be alerted if certain things happen. For example:

- Fraud detection systems determine if the usage patterns of a credit card have unexpectedly changed, and block the card if it is likely to have been stolen.
- Trading systems examine price changes in a financial market and execute trades according to rules.
- Manufacturing systems monitor the status of machines in a factory, and quickly identify the problem from a malfunction.
- Military and intelligence systems track the activities of a potential aggressor, and raise the alarm for signs of an attack.

These applications require sophisticated pattern matching and correlations.
We will briefly compare and contrast some other applications.

Complex event processing

- **Complex event processing (CEP)** is an approach developed for **analyzing event streams**, especially for applications that require **searching for certain event patterns**.
- Similarly to regular expression allows you to search for patterns of characters in a string, CEP allows you to specify rules to search for certain patterns of events in a stream.
- CEP systems often use a high-level declarative query language like SQL, or a graphical user interface, to describe the patterns of events to be detected.
- Queries are submitted to a processing engine, which consumes the input streams, **maintains a state machine** that **performs the required matching**.
  When a match is found, the **engine emits** an **event** with the details of the event pattern.
- The relationship between queries and data is **reversed** compared to normal databases.
  A database stores data persistently and queries as transient.
  In CEP engines, queries are stored long-term, and events from the input streams flow past them in search of a query that matches an event pattern.

Stream analytics

Another area of stream processing is for **analytics on streams.**
The boundary between CEP and stream analytics is blurry. Analytics is less interested in searching events but more to **aggregations and statistical metrics over a large number of events.** For example:

1. Measuring the rate of some type of event (how often per time interval)
2. Calculating the average of a value over some time period

3. Comparing current statistics to previous time intervals (to detect trends or alert on metrics that are high or low compared to the same time last week)

Such statistics are computed over **fixed time intervals,** known as a **window**, e.g., average number of queries per second to a service over the last 5 minutes, and 99th percentile response time during that period.

Averaging over a few minutes smoothes out irrelevant fluctuations from one second to the next, while still giving you a timely picture of any changes in traffic pattern.

Stream analytics systems sometimes use **probabilistic algorithms,** such as **Bloom filters**, HyperLogLog for cardinality estimation, and percentile estimation algorithms.

Probabilistic algorithms produce approximate results, with the **advantage** of requiring significantly **less memory** in the stream processor. Probabilistic algorithms are merely an **optimization**.

Maintaining materialized views

- We saw a stream of changes to a database can keep derived data systems(caches, search indexes, data warehouses) up to date with a source database.
  They are specific cases of **maintaining materialized views:** derive an alternative view to some dataset for efficient query, and update that view when data changes.
- In **event sourcing, application state** is maintained by applying a log of events. The application state is also a kind of materialized view.
  But it is not sufficient to consider events only within a time window, building the materialized view may require all events over an arbitrary time period, and obsolete events that may be discarded by log compaction. You need a window to go back to the beginning of time.
- Any stream processor can be used for **materialized view maintenance**. **Kafka Streams** support this usage, building upon Kafka's support for **log compaction**.

Search on streams

- There is a need to **search for individual events** based on complex criteria such as **full-text search queries**.
  Media monitoring services subscribe to feeds of news articles and broadcasts from media outlets, and search for news mentioning companies, products, or topics of interest.
  On some websites, **users of real estate websites can be notified** when a new property matching their search criteria appears on the market.
- **This is done by formulating a search query in advance, and matching the stream of news items against this query.**
  The **percolator** feature of **Elasticsearch** is one option for implementing this stream search.
- Conventional search engines index the documents and run queries over the index.
  When searching a stream, the queries are stored, the documents run past the queries.
  To **optimize** the process, **index both queries and documents**, narrowing down the **set of queries that may match**.

Message passing and RPC

We discussed message-passing systems as an alternative to RPC, as a mechanism for services to communicate, used in the actor model. Although they are based on messages and events, they are not stream processors:

- Actor frameworks are a mechanism to manage concurrency and distributed execution of communicating modules.
  Stream processing is a **data management technique**.
- Communication between actors is ephemeral and one-to-one.
  Event logs are **durable** and **multi-subscriber**.
- Actors can communicate in arbitrary ways (including cyclic request/response patterns).
  Stream processors are in **acyclic pipelines** where every stream is the output of one particular job, and derived from a defined set of input streams.

There is some crossover between RPC-like systems and stream processing. It is possible to process streams using actor frameworks. However, frameworks do not guarantee message delivery when crashes, it is not fault-tolerant unless implement additional retry logic.

Reasoning About Time

- Stream processors often need to deal with time, especially for analytics purposes.
- Using timestamps in the events allows the processing to be deterministic: running the same process again on the same input yields the same result.
- Many stream processing frameworks use local system clocks on the processing machine to determine windowing. The advantage is being simple, if the delay between event creation and event processing is short. However, it breaks down if there's significant processing lag.

Event time versus processing time

- Reasons for processing delay: queueing, network faults, a performance issue leading to contention in the message broker or processor, restart of stream consumer, or reprocessing of past events.
- Message delays lead to unpredictable message ordering.
- Confusing event time and processing time leads to bad data.

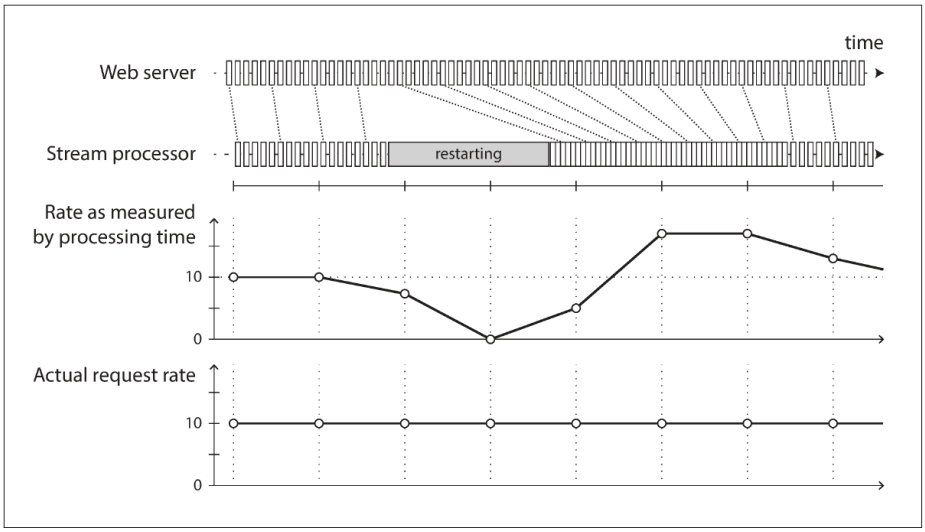


Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

Knowing when you're ready

- A problem of defining windows with event time: not sure when received events for a particular window, or whether there are events still to come.
- You can time out and declare a window ready after not seeing new events for a while, but it could happen that some events were buffered on another machine, delayed due to a network interruption. You need to handle events that arrive after the window.

Two options:

1. Ignore the straggler events, they are probably a small percentage of events. Track the number of dropped events as a metric, alert if dropping a significant amount of data.
2. Publish a correction, an updated value for the window with stragglers included. Retract the previous output.

In some cases, use a special message to indicate, "From now on there will be no more messages with a timestamp earlier than t," which can be used by consumers to trigger windows. However, if several producers on different machines are generating events, each with their own minimum timestamp thresholds, the consumers need to track each producer individually. Adding and removing producers is trickier.

Whose clock are you using, anyway?

- Assigning timestamps to events is more difficult when events can be buffered at several points in the system.
- For example, a mobile app reports events for usage metrics to a server. The app may be used while the device is offline, it buffer events locally on the device and send them to a server when an internet connection is available (may be hours or days later). To consumers of this stream, the events will appear as extremely delayed stragglers.
- The timestamp on the events should be the time user interaction occurred, according to the mobile device's local clock. However, the clock on a user-controlled device cannot be

trusted. The time event received by the server (server's clock) is more accurate, since the server is under your control, but less meaningful in terms of describing the user interaction.

- To adjust for incorrect device clocks, one approach is to **log three timestamps**:
    - The time event occurred, according to the device clock
    - The time event sent to the server, according to the device clock
    - The time event received by the server, according to the server clock
- By subtracting the second timestamp from the third, you can estimate the offset between the device clock and the server clock (assuming the network delay is negligible compared to the required timestamp accuracy). You can then apply that offset to the event timestamp, and thus estimate the true time at which the event actually occurred (assuming the device clock offset did not change between the time the event occurred and the time it was sent to the server).
- Batch processing has the same issues of reasoning about time, but it is more noticeable in stream processing.

Types of windows

Decide how windows over time periods are defined. The window can be used for **aggregations**, e.g. count events, or calculate the average within the window. Several types of windows:

Tumbling window

- Has a fixed length, every event belongs to exactly one window.
- E.g., 1-minute tumbling window, all the events with timestamps between 10:03:00 and 10:03:59 are grouped into one window.
- You can implement a 1-minute tumbling window by taking each event timestamp and rounding it down to determine the window it belongs to.

Hopping window

- Has a fixed length, but allows windows to overlap to provide some smoothing.
- E.g., 5-minute window with a **hop size** of 1 minute contains the events between 10:03:00 and 10:07:59, the next window would cover events between 10:04:00 and 10:08:59, and so on.
- You can implement this by calculating **1-minute tumbling windows**, and then **aggregating** over several adjacent windows.

Sliding window

- Contains events occur within some interval of each other.
- E.g 5-minute sliding window cover events at 10:03:39 and 10:08:12, because they are less than 5 minutes apart (tumbling and hopping windows won't put them in the same window as they use fixed boundaries).
- You can implement it by keeping a buffer of events sorted by time and removing old events when they expire from the window.

Session window

- Has no fixed duration. It is defined by grouping together all events for the same user that occur closely together in time, and the window ends when the user has been inactive for some time.
- Sessionization is a common requirement for website analytics.

Stream Joins

- Three types of joins: stream-stream joins, stream-table joins, table-table joins.

Stream-stream join (window join)

- A search feature on your website, you want to detect recent trends in searched-for URLs.
  Someone types a search query, log event containing the query and the returned results.
  Someone clicks one of the search results, log another event recording the click.
- Calculate the **click-through rate** for **each URL in the search results**, you need to bring together the events for the **search action** and the **click action**, which are connected by having the same **session ID**. Similar analyses in **advertising systems**.
- The click never comes if the user abandons their search. If it comes, the time between the search and click is variable, from a few seconds to days or weeks.
  Due to network delays, the click event may arrive before the search event. You can choose a suitable window for the join,e.g., choose to join a click with a search if it occurs at most one hour apart.
- **To measure search quality**, you need accurate **click-through rates**, you need both the search events and the click events.
- **To implement this join, a stream processor needs to maintain state, e.g., all the events occurred in the last hour, indexed by session ID.**
  Whenever a search event or click event occurs, it is added to the appropriate index, and the stream processor checks **the other index** to see if another event for the same session ID has arrived.
  If matching events, emit an event saying which search result was clicked.
  If the search event expires without seeing a matching click, emit an event saying which search results were not clicked.

Stream-table join (stream enrichment)

Join two datasets: user activity events and database of user profiles. User activity events is a stream.

- Perform join on a **continuous** basis in a **stream processor**: the input is a stream of activity events containing a user ID, the output is a stream of activity events in which the user ID has been augmented with profile information about the user.
- This process is **enriching the activity events with information from the database**.

Stream process looks at one activity event at a time, looks up user ID in the database, and adds the profile information to the activity event.

- The database lookup can be implemented by querying a remote database; however, remote queries are slow and overload the database.
- Another approach is to load a copy of the database into the stream processor so it can be queried locally.
  This is similar to hash joins: the local copy of the database might be an in-memory hash table if it is small enough, or an index on the local disk.

The stream processor's local copy of the database needs to be kept up to date.

- This can be solved by change data capture:
  The stream processor subscribes to a changelog of the user profile database and the stream of activity events. We obtain a join between two streams: activity events and profile updates.

A stream-table join is very similar to a stream-stream join.

- The biggest difference: for the table changelog stream, the join uses a window that reaches back to the "beginning of time" (a conceptually infinite window), with newer versions of records overwriting older ones.

Table-table join (materialized view maintenance)

In the Twitter timeline case, if a user wants to view their home timeline, it is expensive to iterate all the people the user follows, find their recent tweets and merge.
Instead, we want a timeline cache: a kind of per-user "inbox" to which tweets are written as they are sent, so that reading the timeline is a single lookup. Materializing and maintaining this cache requires the following event processing:

• When user u sends a new tweet, it is added to the timeline of every user following u.
• When a user deletes a tweet, it is removed from all users' timelines.
• When user u1 starts following user u2, recent tweets by u2 are added to u1's timeline.
• When user u1 unfollows user u2, tweets by u2 are removed from u1's timeline.

To implement this cache maintenance in a stream processor, you need streams of events for tweets and follow relationships. The stream process maintains a database containing followers of each user, it knows which timelines need to update when a new tweet arrives.

Another way to look at this: **it maintains a materialized view for a query that joins two tables** (tweets and follows), like the following:

```
SELECT follows.follower_id AS timeline_id,
array_agg(tweets.* ORDER BY tweets.timestamp DESC)
FROM tweets
JOIN follows ON follows.followee_id = tweets.sender_id
GROUP BY follows.follower_id
```

The join of the streams corresponds to the join of the tables in the query. **The timelines are a cache of the result of this query**, updated every time the underlying tables change.

Time-dependence of joins

- Three types of joins have in **common**: they require the stream processor to **maintain** some **state** (search and click events, user profiles, or follower list) based on **one input**, and **query** that state on messages from **the other input**.
- **The order of the events that maintain the state is important.**
  In a **partitioned** log, the ordering within a single partition is preserved, but no ordering guarantees across different streams or partitions.

- If events on different streams happen at a similar time, which order are they processed? Example, In stream-table join, if a user updates their profile, which activity events join with the old profile, which are joined with the new profile? Or, if the state changes over time, and you join with some state, what point in time do you use for the join.
  Another example, if you sell things and apply the right tax rate to invoices, depends on the country or state, the type of product, the date of sale. When joining sales to the tax rate table, join the tax rate at the time of the sale.
- If the ordering of events across streams is undetermined, the join is **nondeterministic**: you cannot rerun the same job on the same input and get the same result.
- In data warehouses, this is called **slowly changing dimension (SCD)**. It is addressed by using a **unique identifier** for a particular **version of joined record**: e.g Every time tax rate changes, it is given a new identifier, invoice includes identifier for the tax rate at sale time.
  This makes the **join deterministic**, but the **log compaction** is **not possible**, since all versions of the records in the table need to be retained.

Fault Tolerance

- Batch processing frameworks can tolerate faults: if a task in a MapReduce job fails, it can be started again on another machine, and the output of the failed task is discarded. This transparent retry is because input files are immutable, each task writes its output to a separate file on HDFS, output is visible when a task completes successfully.
  It appears every input record was processed exactly once, the visible effect in the output is as if they had only been processed once. This principle is **exactly-once** semantics.
- Let's consider how stream processors can tolerate faults. Waiting until a task is finished before making its output visible is not an option, because a stream is infinite and so you can never finish processing it.

Microbatching and checkpointing

- One solution is **microbatching:** break the stream into small blocks, treat each block like a miniature batch process. Used in **Spark Streaming**.
- The batch size is **around one second**, the result of a performance compromise: smaller batches incur greater scheduling and coordination overhead, larger batches mean a longer delay before results of the stream processor become visible.
- Implicitly provides a tumbling window equal to the batch size (windowed by processing time, not event timestamps).
  Any jobs requiring larger windows need to explicitly carry over state from one microbatch to the next.
- A variant approach is to periodically generate rolling checkpoints of state and write them to durable storage.
  If a stream operator crashes, restart from the most recent checkpoint and discard output generated between the last checkpoint and the crash.

The checkpoints are triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a window size.

- Microbatching and checkpointing approaches provide the exactly-once semantics as batch processing.
  However, as soon as output leaves the stream processor (e.g, write to a database, send messages to external message broker, or send emails), the framework is no longer able to discard the output of a failed batch.
  Restarting a failed task causes the external side effect to happen twice, and microbatching or checkpointing alone is not sufficient to prevent this.

Atomic commit revisited

- In order to exactly-once processing in the presence of faults, we need to ensure outputs and side effects of processing an event take effect only if the processing is successful. They need to happen atomically. We discussed it in “Exactly-once message processing” in distributed transactions and two-phase commit.
- We discussed the problems in the traditional implementations of distributed transactions like XA. However, in more restricted environments it is possible to implement atomic commit efficiently.
- Unlike XA, these implementations do not attempt to provide transactions across heterogeneous technologies, but keep them internal by managing both state changes and messaging within the stream processing framework.
  The overhead of the transaction protocol can be amortized by processing several input messages within a single transaction.

Idempotence

- To discard partial output of failed tasks and they can be safely retried without taking effect twice. Distributed transactions are one way to achieve this, another way is to rely on idempotence.
- **An idempotent operation is one that you can perform multiple times, and it has the same effect as if you performed it only once.** E.g, setting a key in a key-value store to some fixed value is idempotent, whereas incrementing a counter is not.
- **Even an operation is not idempotent, it can be made idempotent with extra metadata.** E.g, when consuming messages from Kafka, every message has a persistent, monotonically increasing offset. When writing a value to an external database, you can include the offset of the message that triggered the last write with the value. You can tell whether an update has been applied, and avoid performing the same update.
- Relying on idempotence implies several **assumptions**: restarting a failed task must replay the same messages in the same order (log-based message broker does this), the processing must be deterministic, no other node concurrently updates the same value.
- When failing over from one processing node to another, **fencing** may be required to **prevent interference from a node that is thought to be dead but is actually alive.**

Despite these caveats, idempotent operations can be effective to achieve exactly-once semantics with a small overhead.

Rebuilding state after a failure

- Any stream process that requires state (windowed aggregations as counters, averages, histograms; tables and indexes for joins) must ensure this state can be recovered after a failure.
- First option: keep the state in a remote datastore and replicate it, although querying a remote database for each message can be slow.
- Second option: keep state local to the stream processor, and replicate it periodically. When the stream processor recovers from a failure, the new task reads the replicated state, resume processing without data loss.
- E.g, Flink periodically captures snapshots of operator state and writes them to durable storage like HDFS.
  Kafka Streams replicate state changes by sending them to a dedicated Kafka topic with log compaction, similar to change data capture.
  VoltDB replicates state by processing each input message on several nodes.
- In some cases, it is not necessary to replicate the state, because it can be rebuilt from the input streams.
  E.g, if the state consists of aggregations over a short window, it is fast enough to replay the input events corresponding to that window.
  If the state is a local replica of a database, maintained by change data capture, the database can be rebuilt from the log-compacted change stream.

All of the trade-offs depend on the performance characteristics of the underlying infrastructure: in some systems, network delay may be lower than disk access latency, and network bandwidth may be comparable to disk bandwidth. There is no universally ideal trade-off for all situations, and the merits of local versus remote state may also shift as storage and networking technologies evolve.