Designing Data-Intensive Applications

Chapter 11: Streaming Processing

Introduction

- Last chapter we talked about batch processing, which requires finite sized inputs, but in reality many data sources are unbounded and produce data continuously
- Batch processes can run daily, maybe hourly, but some use cases require more frequent updates, and that's where stream processing comes in
- This chapter introduces streaming, talks about how databases have taken a page from streaming, and then covers how stream data is processed

Transmitting Event Streams

- Polling for changes would be too expensive at high frequencies, so we have systems built to handle streaming data
- Each group of related events is called a topic or a stream

Messaging Systems

- A publish/subscribe model allows multiple producers and/or consumers
- Two questions differentiate messaging implementations:
    - What happens if consumers can't process messages as fast as they are produced: drop, buffer, or apply backpressure
    - If nodes go down, can messages be lost: yes or no
- Direct messaging from producers to consumers is one kind of implementation
    - Options include UDP multicast, TCP or IP multicast, straight UDP, or even HTTP or RPC calls
    - Generally require application to be aware of message failure and implement any fault tolerance support

Message brokers

- Most common is a separate server that runs a message broker or message queue
- Producers and consumers connect as clients
- Queueing means delivery to consumers is asynchronous (a return message can notify the producer if it wants to wait for acknowledgement)
- Some message brokers participate in two-phase commit, but the nature of message queues where delivered messages are typically deletes is different from permanent databases that can be queried
- Multiple consumers can be:
    - Load balancing - dividing the work of processing messages
    - Fan out - independent consumers each doing their own thing
- Acknowledgements and redelivery - beware that the following can happen:
    - A message that wasn't acknowledged actually was processed, so redelivery to another consumer can cause it to be processed twice

    - With load balancing, redelivery can cause messages to be reordered for processing

Partitioned Logs

- Log-based message brokers combine lightweight message delivery with durable log-based durable storage. This includes Apache Kafka and Amazon Kinesis.
- In this structure, producers append to a log and consumers read the log sequentially
- For scalability, the log is usually partitioned. It can also be replicated for fault tolerance.
- Load balancing consumers is usually handled by assigning consumers to partitions. If you want to balance individual messages, a traditional JMS/AMPQ style message broker is easier to implement.
- Consumer offsets - instead of individual acknowledgements, the broker only needs to (periodically) track the message offset for each consumer
- Because everything is written to disk, disk space usage needs to be considered. Many systems implement a circular buffer with a fixed maximum amount of storage.
- When consumers cannot keep up with producers - typically message loss is possible, although admins can be warned before this happens. A side benefit is that consumers which are shut down don't cause runaway large message queues.
- Replaying old messages - is possible. The consumer offset can be intentionally set backward to an earlier value.

Databases and Streams

- A replication log is like a stream of writes, so databases can borrow from streaming to help address heterogeneous systems

Keeping Systems in Sync

- Most of the time multiple systems, such as the OLTP system and data warehouse, and they all need to be kept in sync
- Dual writes by clients can have race conditions, and ensuring both writes commit or abort is the atomic commit issue

Change Data Capture (CDC)

- Instead of treating database replication as a proprietary internal implementation detail, change data capture looks to expose all changes in an externally visible form that can be replicated by other systems
- CDC can provide changes immediately as a stream, and they can be applied by another system, such as a search index, continually
- A log-based message broker can preserve the order of messages, providing the right kind of delivery
- Basically, one database becomes the single leader, and the others act as followers
- Having snapshots to start from is a lot faster for initial synchronization than replaying every change since the beginning of time. Some CDC systems incorporate snapshots, but sometimes you have to do it manually.

- Log compaction is beneficial for keeping down the size of logs, especially if you're going to persist everything
- Newer DBs are supporting CDC functionality, instead of it being a bolt-on afterthought

Event Sourcing

- Event sourcing is similar to CDC in that all changes are captured, but it is involved application design, and events are at a higher level
- The event store can only be appended -- updates or deletes are discouraged or forbidden
- Event sourcing records logical, immutable actions. It usually won't record entire records, like CDC does, so full history may be needed to get the complete current state of a record, and log compaction may not be possible
- Note that all events start as commands, but once all validation is completed, it is now an event and is immutable

State, Streams, and Immutability

- The events that change data in a database are an immutable history. You can think of the contents of the database as a cache of the latest values in the log.
- Advantages of immutable events
    - The book has an example that accounting uses and append-only ledger. Immutability can also help debugging and provide a richer history for analytics.
    - Deriving several views from the same event log - having an explicit process for translating event log entries to the database makes logic explicit, facilitating multiple views. Separating how data is written and read can offer a lot of flexibility.
    - Concurrency control - one big negative with CDC and event sourcing is that event log consumers are usually updated asynchronously, so a read after a write may be stale. On the flip side, good self-contained event design may eliminate the need for multi-object transactions.
- Limitations of immutability
    - Workloads with lots of updates and deletes may be hard. Fragmentation and how well compaction and garbage collection perform may be critical.
    - Note also that certain circumstances require deletion, such as privacy rules around someone closing their account

Processing Streams

- Input streams can be processed to create new output streams
- Mapping and filtering work similarly to batch processing, as does partitioning and parallelization

Uses of Stream Processing

- Monitoring and alerting - credit card fraud detection, stock market price changes, etc.
- Complex event processing - usually a high level declarative language is used to define criteria. Unlike databases, the query persists over time, and the data comes and goes.

- Stream analytics - usually about metrics, not events, e.g. number of comments per minute.
    - Note that probabilistic algorithms are often used because they are much cheaper/faster, but are not required
- Maintaining materialized views - can keep another copy of data, as has been discussed
- Search on streams - while CEP often looks at combinations of events, sometimes you just want complex search on individual items, such as monitoring news articles for topics of interest
- Message passing and RPC - streams provide fault tolerance in a way that RPC doesn't. Can have a stream for calls and another stream for returns, and these can scale to multiple nodes.

Reasoning About Time

- Since message delays are unbounded, usually differentiate event time from time message was received/processed
- Knowing when you're ready - if you want to process all messages using a window between 1:00 and 1:01, how do you know when you have them all?
    - You can set a timeout and drop stragglers that arrive late (and you can monitor how often you get stragglers)
    - You can output a correction, voiding prior output
- Whose clock are you using, anyway?
    - Because of all the issues with clocks, some systems track three times:
        - The time when the event occurred, per the producer's clock
        - The time when the message was sent, per the producer's clock
        - The time when the message was received, per the broker's clock
    - With fairly short network delays, you can estimate how far apart the producer's clock is by calculating the difference between the last two times
- Types of windows
    - Tumbling window - adjacent time slots
    - Hopping window - overlapping time slots
    - Sliding window - fixed time slot starting with the oldest event in buffer
    - Session window - e.g. per user events until a period of inactivity

Stream Joins

- Stream-stream join (window join) - typically state is maintained (e.g. a hash index) to join events from two different streams, such as searches and clicks mentioned in the book
- Stream-table join (stream enrichment) - typically a local copy of the table is kept using change data capture, such as for a user table
- Table-table join (materialized view maintenance) - the concept of streams updating a materialized view of a join between two tables. Changes to either stream need to inform potentially multiple rows from the other table in the join output.
- Time dependence of joins - if ordering between two streams is undetermined, joins can be nondeterministic. Using history on a slowly changing dimension can solve determinism, but means you can't do log compaction.

Fault Tolerance

- Microbatching and checkpointing - write state to durable storage periodically
- Atomic commit revisited - can’t easily support full XA across heterogeneous systems, but can provide some support for atomic commit internally between streams
- Idempotence - events that are the same if repeated (or smart enough not to repeat even when you try) can be retried without risk
- Rebuilding state after a failure - because state is required, must periodically persist it to local or remote storage for fault-tolerance

Summary

- Streams can be transported through direct messaging, message brokers, or event logs
    - AMQP/JMS-style message broker - consumers acknowledge individual messages once processed, then they are deleted
    - Log-based message broker - consumers read log partitions, and brokers maintain messages on disk
- Streams are a powerful way of integrating all kinds of systems
- Time windowing strategies and three kinds of joins are used in stream processing