Stream Processing

Background

Often, we will have lots of data coming in that we want to process asynchronously in the background. Streaming works on an unbounded set of data, as opposed to just a limited set of files/records.

Messages are created by a producer node, and eventually handled by at least one consumer node.

While these messages can be sent directly from producer to consumer, it is common practice to use a message broker to buffer these messages in a centralized place.

Message Brokers

A type of database that handles streaming, both producers and consumers can connect to it, typically puts messages in a queue if there are many of them.

Some will keep the messages durably, others will delete them after they are successfully consumed.

Two delivery patterns: fan out (send to all consumers) vs. load balancing (one message per consumer).

In Order Messages?

Even though a queue holds the messages in the order they were received, if a consumer crashes/takes a while to handle a message, the broker may have sent the next message to a different consumer which processed it first!

Messages can reliably be delivered in order - but at a performance penalty which we will discuss later.

Types of Message Brokers

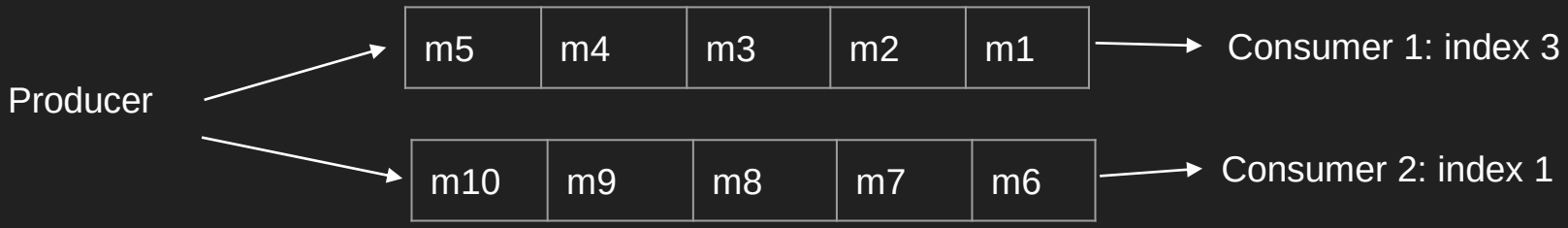
- In memory
  - Can be done on something like a Redis instance, no persistence
  - Messages that are acknowledged by a consumer are deleted
- Log based (on disk)

Log Based message brokers

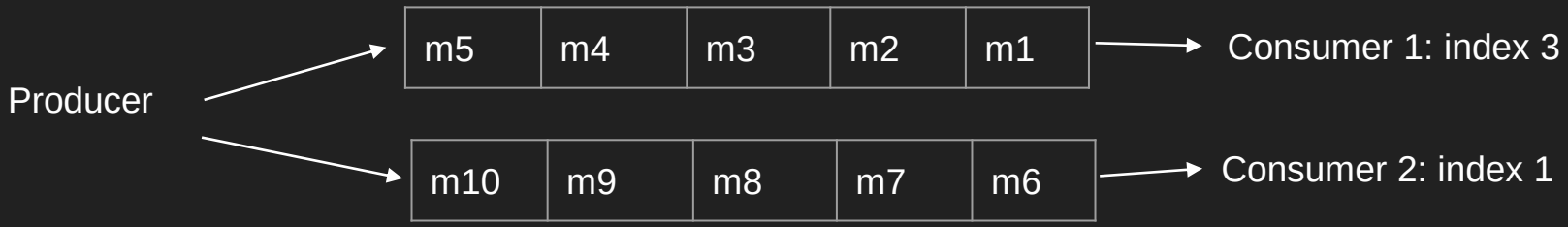
Overview:

- Messages sent to an append only log on disk
- Log can be both partitioned and replicated to improve performance and fault tolerance (partitioning it may mess up order of message delivery)
- One consumer per partition, keeps track of which messages on the log it has already seen to avoid processing a message twice
    - A single hard to process message stops the rest of the partition from proceeding forward!

Log Based message brokers continued

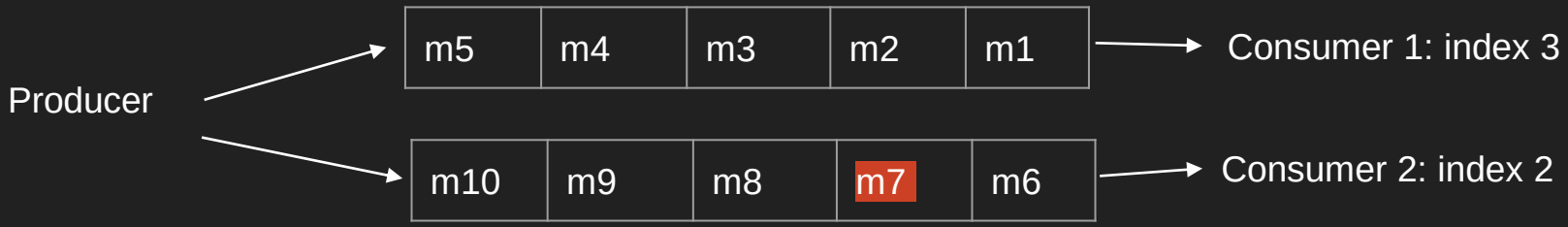


Log Based message brokers continued



Adding a message: decide which partition to send the message to via the use of some function or load balancing method (like consistent hashing) and send it there!

Log Based message brokers continued



Adding a message: decide which partition to send the message to via the use of some function or load balancing method (like consistent hashing) and send it there!

Reading a message: consumer is alerted of message (via something like long polling or websockets), handles it, and on acknowledgement, broker increases its index - this way if consumer crashes another consumer takes over and knows where to start from.

Log Based vs. In Memory Message Brokers

In Memory:

- Good for when messages take a long time to process and their order does not matter

Log Based:

- Good for when keeping messages around after processing them is useful so that you can potentially replay messages
- More fault tolerant as they will not lose existing messages on crash
- Good for ordering message processing within a log partition

Common Uses of Streams

- Logging and metrics
- Change Data Capture
- Event Sourcing

Logging and Metrics

Often want to be able to aggregate certain events into time windows, such as application logs or certain metrics:

- Challenging because an event can arrive to a stream any amount of time after the window closes, do we add it to the time window or just leave it be?

Putting events in fixed time intervals:

- Tumbling windows (non overlapping intervals of fixed length, e.g. every minute starting at 0 seconds)
- Hopping windows (overlapping intervals of fixed length, e.g. all 5 minute intervals starting at 0 seconds, can aggregate tumbling window metrics to create hopping windows)
- Sliding windows of fixed duration but no set start point can be created by using an in memory buffer of all the events in the window and then removing them once they are outside of the window

Change Data Capture

Writes to a database are then sent to a stream, where they can be consumed by other derived data (such as caches, search indexes, data warehouses) in order to keep them up to date as well.

Assuming these logs are persistent, they can be compacted by only holding the most recent value of these keys, useful in the event that new derived databases are added to our system.

Event Sourcing

Similar to change data capture, but all events of the user are put in a streamed, append-only log, so that you can derive all sources of data from it.

Is more future-proof as it means that you do not have to be stuck with some specific data schema, but can derive many different types of data from it. This comes at the cost that the log cannot be compacted, because everything in it is just an “event”, not a database write.

Examples include: jordan clicks the watch button, jordan is the 40th watcher of Kate Upton’s post, jordan is the 41st watcher of Kate Upton’s post, jordan is the 42nd watcher of Kate Upton’s post.

Stream Joins

Like with batches, there are data associations with streams and we want to reduce the number of network calls made to an actual database. In order to do so, we often have to keep some amount of state local to the stream, representing some other data source. However, we now run the risk that processing certain stream events becomes nondeterministic.

- Stream-stream joins
- Stream-table joins
- Table-table joins

Stream-Stream Joins

Joining two different types of events in a stream (such as a search in the search bar, as well as zero or more clicks that occur from a search). Can keep a local index of both types of events on the message broker, and when a new event comes in, check the other index in order to see if there is a join to be made. If events are only valid for a certain window of time, they can be removed from the index after that amount of time.

Stream-Table Joins

Enriching one stream event with data from a database table, possibly in order to send to another stream. Keep a local copy of the table in the stream broker in order to avoid having to make network calls to the database for each join. Subscribe to the change data capture of the database table in order to make changes to the local copy of the table in accordance with the actual table.

Table-Table Joins

Occasionally we might even have to keep multiple local copies of a table up to date, in order to perform joins on them. In order to do this, we have to subscribe to multiple sources of change data capture. Effectively, the result of the joined streams maintains a cache of the actual SQL query of the two tables.

Fault Tolerance

Want to ensure that each message is processed exactly once (no less, no more):

- Every message processed at least once
  - Occasionally checkpoint stream state to disk to restart from most recent state on crash
  - Can also run microbatches, where you let a few stream events accumulate and then run a batch workload on them
- Messages not processed more than once
  - Can use idempotence, which is some way of keeping track of seen messages (usually via a unique message ID) to ensure that they are not processed multiple times
  - Could also use atomic transactions (see two phase commit), which are slower

Streaming Conclusion

Streaming is becoming incredibly important as companies take in increasing amounts of data which can be processed in the background as opposed to instantly - we will see it pop up frequently in systems design questions.

Recall log based streams optimize for persistence and ordering, whereas in memory streams optimize for speed and are better suited for tasks that take a long time to execute.

Often times, there is a need to enrich or cross reference stream data with other data which we have shown how to do via joins.

Finally, concepts such as event sourcing and change data capture offer promising options for keeping derived data up to date in an eventually consistent way.