Chapter 11 Stream Processing

Book Summary - Designing Data-Intensive Applications

In batch processing systems, the input is bounded.
But the data could gradually arrive over time — unbounded data.

Stream — data is made available incrementally over time. e.g. stdin and stdout of Unix

Transmitting Event Streams

Input - sequence of records or events
* Event is a immutable object containing details of something happened at some of time.
* e.g. order is placed on e-commerce website
* Event is generated by one producer (or publisher) and can be processed by multiple consumers (or subscribers)
* Related events are grouped together into topic or stream

Messaging Systems

* Operates on publish/subscribe model
* Multiple producer nodes can send messages to a topic and allows multiple consumers nodes to receives messages in a topic.

What happens if producers send messages faster than consumers can send them?
* Drop messages
* Buffer messages
* Apply back pressure — blocking sender from sending more messages

Direct messaging from producers to consumers
* Without any intermediary nodes
* UDP multicast in financial industry. e.g. stream stock market feeds
* Web hooks — consumers expose a endpoint and producers can push messages directly.

Issues —
* assumption producers and consumers are online; or application logic should be aware to handle faults
* Consume if offline? It will miss messages

Message Brokers
* message broker or message queue
* Runs a server; producers and consumers connect as clients
* Producers write message to broker and consumers read them from broker
* Durability is handled by broker

Comparison with Database
* database keep data until explicitly deleted
* Data search capabilities such as indexing

Multiple Consumers — multiple consumers read message from same topic
* Load Balancing — each message delivered to one of the consumers, so consumers can share the work.
* Fan-out — each message delivered to all consumers.
* These two patterns can be combined as well

Ack & redelivery
* consumer crash - messages were delivered by consumer but never processed by it.
* Consumer should send a ack that message is consumed
* If broker is unsure of message delivery, it retries on it's end
* If message ordering is important (messages can be reordered in case of retries/redelivery), recommendation is to keep separate consumer per queue.

Partitioned Logs
* Message brokers are built around transient messaging mindset — delete after delivered to consumers
* Databases/Files — delete on user request
* If messages delivered, they can't be recovered — in AQMP/JMS-style messaging

Can we have — durable storage + messaging notification facility???

Using Logs for message storage
Log — append only sequence of records on disk
* this approach is used to implement message broker
* Producer send message by appending to end of log
* Consumer receive it by reading log sequentially
* Similar to unix tail -f

For more scale —
* the log can be partitioned across multiple machines
* Topic can be defined as group of partitions to carry messages of same type

* Each partition, broker assigns monotonically increasing number, (offset).
* Messages within partition are totally ordered
* No ordering guarantee across different partitions
* So , all messages that need to ordered consistently, route them to single partition

Example — Kafka, Amazon Kinesis Streams

Logs compared to traditional messaging
* In general, the broker can assign entire partition to nodes in the consumer group
* Consumer reads messages in partition sequentially, in single-threaded manner.

Downsides —
* number of nodes sharing the work can be almost the number of log partitions in the topic
* If single message is slow to process, it also blocks others

If the downsides overweigh over pros of log based message brokers, use JMS/AMQP

Consumer Offsets
* All messages with offset less than consumer's current offset has been processed
* Broker doesn't need ack for each msg
* Just periodically record consumer offsets
* This helps in increasing throughput

Disk Space Usage
* Keep on appending to log — will run out of disk space
* Log is divided into segments
* Old segments are deleted (or moved to archive store)

Databases and Streams

* The event can also be any operation on the databases — writing to a database. This event can be captured, stored and processed.
* Replication log is stream of database write events, produced by leader

Keeping Systems in Sync

Same data can be stored at multiple places in different forms for different use cases. such as OLTP as source of truth, search index for search capabilities, cache for frequent queries, etc.

* Full database dumps
* Dual writes — application is responsible to write to multiple data sources
  * Concurrency issues
  * One write fails and other succeeds — probable solution, Atomic Commit & Two-Phase Commit

Change Data Capture

Process of observing all data changes in database — dedicated video in System Design playlist

Database — > Parse DB log — > Publish event
e.g. debezium by parsing MySQL binlog

Log Compaction

* Let's say the broker only keeps data for last 14 days. So if a new data source is added built on top of primary database, we need to build new data source on top of some snapshot
* Log Compaction — storage engine periodically looks for log records with same key; removes duplicates; and keep most recent update of key.
* For CDC system, if every change as primary key; and every new update replaces previous value of key; then it's sufficient to keep most recent write.
* So you can rebuild new data source; by reading log-compacted topic from offset zero without taking snapshot.

Event Sourcing

* Stores all changes to application state as a log of change events (similar to CDC)
* But the idea is applied at different level of abstraction
  * CDC — database is used in mutable way; updating or deleting records at will
  * Database don't have to be aware about CDC, it is independent operation
  * Event sourcing, application logic is built on basis of immutable events
  * Event store is append-only
  * Updates/deletes are discouraged or prohibited

Deriving current state from event log

CDC — the entry in database represents current state
Event Sourcing — You need to look all the event history associated with specific entry and then compute current state

* This can be done based on some application logic
* Recomputation again and again can take time
* You can take snapshot or cache is somehow faster response time.
* e.g. series of debit and credit events for your bank account to compute available balance.

Issues —
* if there are so many updates, computing current state is challenging task
* Sometimes data deletion is a requirement so immutability don't serve the purpose. e.g. deletion of data for GDPR compliance

Commands and Events
* Request arrives to the system, it is a command
* System evaluates the request
* If the system decides to process the request, it becomes an event — durable and immutable
* E.g. booking a ticket - command
  * Check if seat is available
  * If yes, generate an event; which can further trigger other systems such as invoicing

Processing Streams

Process the stream once you receive it.
* write to database, cache, search index, etc
* Push events to users e.g. push notifications
* Process one or more input streams to produce one or more output streams; called derived streams
  * The program handling this transformation — operator or a job

Uses of Stream Processing
* Fraud detection system
* Trading systems — price changes in financial market

Complex Event Processing
* short — CEP
* Describe pattern of events that should be detected based on some regex or filtering logic
* If match found, system emits a complex event — with details of event patterns that was detected
* e.g. SQLstream, Samza

Stream Analytics
Aggregations over large number of events
e.g. comparing current statistics to previous time intervals ; such as TPS or latency over 5 minutes interval

Apache Storm, Spark Streaming, Flink, Kafka streams, etc.

Maintaining Materialised views
* Deriving the alternative view onto some dataset that you can query it efficiently
* e.g. search index built on top of database

Reasoning About Time
* Use time windows; such as average over last five minutes

Event Time vs Processing Time
Processing messages may delay —
* Queueing
* Network faults
* Performance issue in message broker; etc

Confusing event time and processing time can lead to bad data. E.g. consumer went down for sometime, once it comes back, it processes the backlog of messages
* Measuring rate based on processing time — there is spike in requests
* Measuring rate based on event time - normal/steady

Which clock?
Assigning timestamps to events is difficult task if events are buffered at several points in the system

For capturing user events based on device clock, the user might have deliberately setup wrong time on device; so we can't trust it.
Solution? Use server time when event was received.
But what if the the server receives event after some time (minutes/hours/days) because the device was offline.
Not a full proof solution.

Solution — log three timestamps
* #1 Event occurring time as per device clock
* #2 Event sent time to server as per device clock
* #3 Event received by server; as per server clock

Offset = #3 - #2. [assuming negligible network latency]
Event occurrence = #1 + Offset

Types of Windows
* Tumbling window
  * Fixed length, every event belong to exactly one window

* Hopping window
  * Fixed length
  * Allows windows to overlap in order provide some smoothing.
* Sliding window
  * Contains all events that occur within some interval of each other.
* Session window
  * No fixed duration
  * Defined by grouping together all events of same user that occurred closely together
  * Window ends; if user is inactive for sometime.

Stream Joins

Join datasets by key

Stream-Stream join (window join)

1. Detect recent trends in searched-for URLs
2. System store query and result returned for every search request.
3. Further, if something for search result is clicked, another event is logged.
4. Join — combine click and search query event ; key = session id

Implementation —

1. Stream processor maintains state. e.g all events in last one hour; indexed by session id
2. For new search/click event, added to the index
3. Stream processor checks other index for events with same session id
4. For matching event; system emit an event saying search result was clicked.

Stream-table join (stream enrichment)

User activity events
Enriching the user activity events from user profile database

1. You can query db for every new user activity event
2. Or keep a copy of database locally to stream processor. — avoid network call

For local copy;
The database can keep on changing as stream is long running
So the local copy can subscribe to database changes — CDC

Table-table join (materialised view maintenance)
For twitter user's home timeline — expensive to iterate over all people user is following ; find their tweets and merge

So build timeline cache; per user-inbox
* user u publishes new tweet; added to timeline of every user who is following u
* For tweet deletion, it is removed from all users' inbox
* User u1 starts following u2; recent tweets by u2 are added to u1's inbox
* Reverse for unfollow action

Implementation —
1. Stream of events of tweets, follow relationships
2. Database with set of followers for each user

Basically joining two tables — tweets and follows

Fault Tolerance
How stream processors can tolerate fault?

Microbatching and checkpointing

* Break stream into small blocks
* Block like miniature batch process; say one second
* Called microbatching
* Used in Spark Streaming

Another approach
* periodically generate rolling checkpoints of state; and write to durable storage
* Used in Apache Flink
* For any crashes, it can restart from recent checkpoint

Atomic commit revisited
Outputs and side-effects of processing an event take effect if an only if the processing is successful

Idempotence
Perform operation multiple times; it has same effect as it is processed once.

Subscribe on YouTube — @MsDeepSingh

Read my O’Reilly book “System Design on AWS”