Chapter 12 The Future of Data Systems

Book Summary - Designing Data-Intensive Applications

Ideas and approaches that can improve the ways we design and build applications.

Data Integration

* Every solution for a problem has pros, cons and trade-offs.
* Appropriate choice of tool depends on circumstances.

Combining Specialised Tools by Derived Data

* A single tool might not solve all the problems so might need combination of multiple. E.g. different datastore for different needs - relational, search indexes, caching, analytics….
  * What one person considers pointless features might be very much important feature for someone else.

Reasoning about Dataflows

* same data is stored in different stores for different access patterns
* You need to be clear about input/output; what's source and destination for the data.

E.g. Database -> CDC -> Search Index
If there is single source of truth for the derived data, it's easier to reconstruct in case of faults.

Derived Data vs Distributed Transactions

Distributed transactions - can be helpful to keep data consistent — generally in sync or atomic operation
Derived data - generally written in asynchronous way

Combining Specialised Tools by Derived Data

* Integration of OLTP db + full-text search index for arbitrary keyword queries
* For simple applications, PostgreSQL full-text indexing feature can be used.
* Search indexes are generally not used as durable system of record.
* One person might consider something as pointless feature but it could be central requirement for someone else.

The limits of total ordering
If system is small, construction of totally ordered event log is relatively easy. Limitations with scale —
* all events should pass via single node which decides ordering. You might need to partition if single machine can't handle the volume.
* If servers are multi-region; network delays can make synchronous coordination inefficient — can lead to undefined event ordering.
* Microservice architecture — two events generated in separate services

Consensus algorithms/total order broadcast work well with single leader node; it becomes difficult in case of multi-nodes

Ordering events to capture causality
* If there is no causal link between events, lack of total order is not an issue.
* Facebook; two users in relationship and broke up
  * User A removed user B from connections and posted something online
  * User B should not see this post.
* If there are separate systems for messaging and friendship; the ordering of events can be lost.
* e.g. message event is processed first and unfriend event later.
* So the notifications can be join between messages + friend list.
  * Logical timestamps can be helpful. The recipients should still have handling for out of order events.
  * Log an event to record state of system and assign unique identifier. This identifier can be used to record causal dependency.
  * Conflict resolution algorithms.

Batch and Stream Processing

* The output of batch/stream processing are derived datasets

Lambda Architecture
1. Record incoming data — append immutable events to dataset
2. Generate read-optimised views from the events
3. This architecture paradigm purposes two systems in parallel
    1. Batch processing systems such as MapReduce (consumes the events and produce corrected version of derived view)
    2. Stream processing system such as Storm (consumes event and produce (approximate) update to the view)

Issues with Lambda Architecture:
* Operational overhead of two systems
* Output from separate systems is combined for user requests — can be hard for complex joinsU

Unifying the Two Systems — need below features
* Ability to replay historical events
* Exactly-once semantics for stream processors
* Tools for windowing by event time

Unbundling Databases

Difference between Unix and databases
* Unix is thin wrapper on hardware resources - files
* High level abstraction, user don't have to worry about implementation details.

Creating an Index
* Take a snapshot of existing data and store them in index in sorted fashion
* Continue on this with any new data added

Meta-database for Everything

A single database can't all solve all the data needs.
* Federated Databases : Unifying Reads — single interface to access all underlying data stores.
* Unbundled Databases : Unifying Writes — synchronising writes across the underlying data stores.

Making Unbundling Work
* Traditional approach to synchronise writes — distributed transactions
  * Every system has different transaction semantics, so distributed transactions could be hard.
  * This might not be a good approach for transactions spreading multiple data stores
* Practical Approach — async event log with idempotent writes
  * Loose coupling between components

Unbundled vs Integrated Systems
* Complexity is high for managing multiple infrastructure (learning curve, operations)
* Single integrated software can be easy to manage.
* If feasible, we can try to use single software for multiple needs as long as it serves the use case. <Don't think of too high scale in beginning, you may never reach there>

Designing Applications Around Dataflow

Separation of Application code and State
* Application code is deployed with tools such as Docker, Kubernetes, etc.
* Database is used as durable storage
* Most web applications are stateless — easy to add/remove servers

Dataflow : Interplay between state change and application code
Application code can respond to state changes — by consuming events

Stream Processors and Services
Example — customer is buying something priced in USD but paying in INR; you need currency conversion logic
* Microservices — query exchange-rate service/or database to get currency rate
* Dataflow approach — subscribe to stream of exchange rate updates and keep record of exchange rate in local database.

* dataflow approach is faster
* Dataflow is more robust, independent of failures of another service

Reads are Events too
Generally write requests are seen as events but you treat reads as well as events
* read event goes to stream processor
* Processor emits the results of read to output stream.

Multi-partition Data Processing
* For queries touching single partition, using streams for reads could be overkill
* Could be useful for complex queries needing multi-partition data

Aiming For Correctness

Databases
Just because database has safety properties such as serialisable transactions; the issue can still happen if there is bug in application (write incorrect data or delete some records)

Exactly-once execution of an operation
* Processing twice is a form of data corruption.
  * Idempotency is one way to overcome this.

Duplicate Suppression
Suppress duplicates
* TCP use sequence numbers to figure out lost/duplicated packets.
* Retransmit lost packets and remove duplicates.
* e.g. database connection - transaction is tied to client connection
  * Network issue at client after sending COMMIT and before getting response from db server
  * Client can’t know if transaction commited/aborted
  * Client can retry but it’s outside of scope of TCP duplicate suppression (because new connection)

* Two-phase commit is solution to 1-1 mapping b/w TCP connection & transaction.
* Is this enough ?
  * No
  * What if connection issue between user and application server
  * No response received by user
  * So user retry
  * So duplication can still happen

Uniquely identifying requests
Consider end-to-end flow of request.
* generate unique identifier such as UUID
* This id is used by all the components involved in serving the request

Applying end-to-end thinking in data systems
You need end-to-end solution to fix the problem;
One component handling it really well don't solve the problem

Doing The Right Thing

Every system is built for a purpose — and every action has consequences (intended and unintended)

*** Predictive Analysis**
Data analysis to predict weather
* **bias and discrimination**
* **Feedback loops**
* **…**

Privacy & Tracking
* **Consent**
* **Privacy and use of data**
* **Data as assets and power**

Subscribe on YouTube — @MsDeepSingh

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