Introduction to Big Data and Distributed Systems

Storage?

Big Data Report Card - SESFAC

1. Schematized (CSV? Database? Protobuf? Blob? Columnar?)
2. Execution (human-readable? tooling? SQL?)
3. Storage-efficient (binary? database?)
4. Fast (RDBMS? KV? Indexes? Partitioned?)
5. Available (replicated?)
6. Consistent (Paxos? Scuttlebutt?)

Level 1 - CSV on a shared server

1. *D* — it has structure, but no guarantee row-to-row beyond what the producer provided. Commas break easily, as do pipes. No type information, no versioning possible.
2. *C-* — Easy to read and consume, but must exclusively use external tools to parse.
3. *D+* - no compression, no encoding, underlying bytes are dependent on platform
4. *D* - Access patterns are confined to full scans, regardless of how much you actually need.
5. *F* - if it's deleted, the node crashes, retention policies, etc., it's gone.
6. *F* - everyone modifies separately, in place, no access control, no history

Concept 1 — Data formats / IDLs

When we begin to work with some data, we'd like some guarantees about that data. Some guarantees we want: established schema, established types, and preferably some compatibility / evolution rules ("if I add a field to this schema, will readers of the old schema be okay with that?"). Additionally, once you begin to add information on schema and types, you are able to represent data of that schema more concisely. Think of JSON — it's a weird compromise between field names, types, and data, meaning all JSONs carry field names. If you already have that information, you just need to know which bytes correspond to which fields. If you have type information, you can represent those types more compactly ("bit packing"). If you have a lot of records of a type, you can encode them together for even more savings!

Level 2 — Parquet on a shared server

Parquet is a binary storage format designed for analytical tasks, and introduces the idea of columnar data storage.

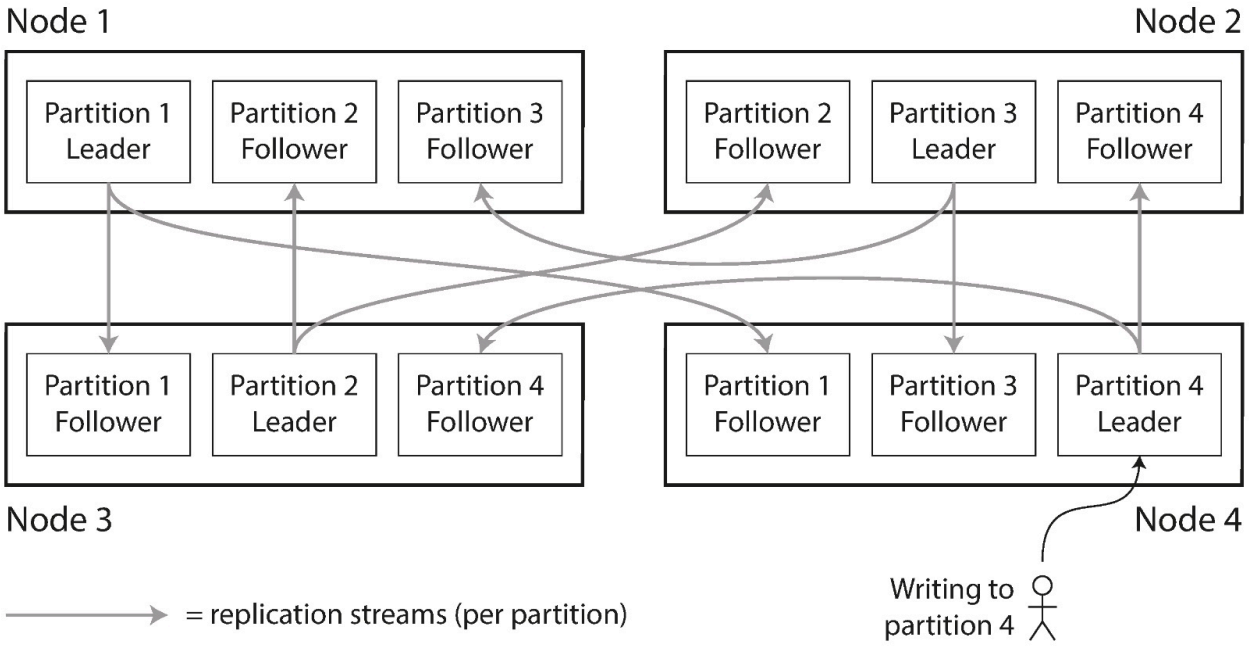
1. *B* — schematized on write, columnar access patterns, no versioning
2. *C+* — CLI tools for specific access, not human readable, still easy to pass around
3. *B+* — Dremel encoding, dictionary encoding, compact binary, compressed, columnar
4. *A-* - metadata for seeks, fast deserializing on read, taking only what's necessary
5. *F* - no change
6. *F* - no change

Concept 2 — Replication and Partitioning

Stated simply:

1. Replication is duplicating your dataset onto multiple independent nodes for both fault tolerance and performance scaling.
2. Partitioning is splitting your dataset horizontally (row-wise) to allow for different nodes to store parts of one complete dataset, mostly for performance scalability.

Replication can be single leader-based, multi-leader-based, and leaderless. Partitioning aims to distribute load through consistent hashing while also avoiding hot spots / keys and allowing for range queries.



Level 3 — Parquet on HDFS

1. *B* — no change
2. *B-* — a little more complex working with a DFS directly, but now we have Hive and MapReduce!
3. *B* — trade space for availability
4. *B+* - trade speed for availability
5. *B+* - file is replicated to nodes; maybe still limited to rack / AZ / region is dangerous?
6. *F* - no change

Concept 3 - Execution and Computation

How do we compute on data? Traditionally, it's with imperative programming languages or with declarative query languages.

SQL is a `declarative` language. The user describes what they want to occur using a high-level syntax, and the execution engine examines the current context (data being operated on, server load) and chooses some physical operations to match the promises of your SQL description. You *declare* what you want to happen, and the execution engine figures out the best way to do it.

Meanwhile, Spark started as an `imperative` DSL of Scala. The `RDD` API allows for low-level manipulation of distributed, partitioned datasets on commodity hardware - you program exactly what you want to happen. Sure, there are declarative aspects — things like shuffles are taken care of for you. Spark moved more declarative with Spark SQL and the `DataFrame` / `Dataset` API — the Tungsten and Catalyst engines allow for both performance and declarative aspects, respectively.

But where did Spark originally come from? MapReduce!

`MapReduce` was such a powerful win because it abstracted data workflows into a simple lineage maps, sorts and reduces. By allowing the user to perform arbitrary computations on arbitrary data formats on a DFS, users could break out of both declarative languages requiring underlying specialization as well as highly idiosyncratic, proprietary storage formats (duck taped with others via a vaneer of standardization, like ODBC and ANSI).

But MapReduce in and of itself didn't go far enough! It needed in-memory piping between stages (instead of incurring disk IO for every step), it needed more computation models beyond a map-reduce constraint, it needed data APIs that were performant and usable out of the box. In short, MapReduce was like a really good hammer, whereas Spark is a full toolkit.

Level 4 — Spark on Parquet on HDFS

1. *B+* — execution engine makes working with files easier
2. *A-* — we have Hive + MR + all of Spark!
3. *B* — no change
4. *A* — benefits of Parquet + benefits of established, robust execution engine with smart algorithms.
5. *B+* — no change
6. *F* — no change

Level 5 — Table in single-node, unpartitioned table in an enterprise RDBMS

Hold up, you're missing a huge part of the story here. Why introduce a proprietary format and a special execution engine when we could have both at once in databases?! Just put your CSV into a database and you're good to go, right?! Well, sorta.

1. *B-* — highly schematized on write, but in propietary format; type info and usage varies between products, although things like ODBC make it better. Evolution possible with database migrations and tooling, but adhoc.
2. *B-* — only accessible with tooling and credentials, SQL is good but not perfect, relational model has pros and cons, ORM is difficult and clunky. Declarative language
3. *B* — efficient, but proprietary
4. *B-* — can be very fast, but requires careful planning (normalization?) and specialized expertise in your

product. Concurrent access can cause contention and locking. Access patterns are usually limited to table-based scans and seeks, with limited support for massive analytics
5. *C+* — database backups, atomic transactions; no replication to mitigate node failures and no partitioning / sharding
6. *A* — follows ACID, transactions, locking, latches, deadlock management, concurrency, different isolation levels. We get this because we have **one** copy of the data, and we just manage access to that data via locking mechanisms

Level 6 — Table in multi-node, leader-based-replicated, sharded, partitioned table in an enterprise RDBMS

Okay, fine, so let's get some replication for both availability and locality, and let's shard and partition such that we get better concurrent access and speed.

1. *B-* — no change
2. *C+* — no change
3. *B-* — replication is redundant, format is still compact
4. *C+* - same problems as above, plus the possibility of reading from a less ideal replica
5. *B+* — all the above, but now with multiple nodes storing the data; partition / sharding allows for smart R/W distribution
6. *B* — writes are confined to leader and are fine, but reads can suffer replication lag and consistency with read-your-writes scenarios. We've entered the world of **eventual consistency**!

Level 7 — Cassandra

1. *B* — Schematized on write, can be treated as tables.
2. *C+* — only accessible with tooling and credentials, CQL is limited, libraries run the gamut.
3. *B* — Binary SSTable on commodity hardware. Data replication incurred.
4. *B-* — KV access patterns are fast; indexing can be limited and difficult, range scans depend on your partitioning key and partition hashing algorithm. Hot keys are a problem for underlying IO and consistency. Forcing ACID semantics incurs serious performance penalties.
5. *B+* — Partitioned KV with quorum-based replicated writes on commodity hardware. Similar to HDFS!
6. *C+* — Eventual consistency and leaderless replication. Last-write-wins is scary, read repair or 2PC is expensive, pushing resolution to the client is taxing.

Level 100 - Spanner

A- — Paxos state machines for consistency, TrueTime API for total broadcast order, Scuttlebutt for infrastructure; architecture reduces contention incurred by consistency. Still eventually

Streaming data?

What is event sourcing? Given a series of events over time, we should be able to fully reconstruct the state of a system by aggregating down the event set as needed. This pairs nicely with streaming architectures, as events can be produced into messaging systems and then consumed into persistence layers as needed. But streaming architectures present new challenges, foremost among them being — how do I manage all these events, how much state should each event contain, when is a data point complete?

When someone produces some analytics via MapReduce or Spark, they are running a batch job over a set period of time with a finite, complete dataset. Analytics are then couched appropriately as "the known state for this time period". In a streaming world, however, you can make no assumptions on data completeness. Data might be late, or produced at different cadences. It is up to the streaming application to properly aggregate data and only trigger it for release when it is "complete". Other problems become much harder too — how do you join streams? What time windows do you leave for your joined events in two streams to overlap by? What do you do with late data outside of your time windows ("watermarking")?

- ACID vs CAP
- Lambda architecture vs kappa architecture vs CQRS
- Spark demo