1 MapReduce

The MapReduce method is designed by google to simplify large-scale distributed data processing. key features:

- Parallelizes computations across many CPUs
- Automates data distribution and result aggregation
- Eliminates locks by restricting data interaction modes
- Provides a generic interface to hide distribution complexities

The two main problems with large data processing are scalability and reliability (hardware failures are common).

The master-worker model is employed and works like this:

- Master: Initiates computation, creates tasks, launches workers, and collects results
- Worker: Processes tasks, signals completion
- Scalability: Uses multiple work queues and work stealing to avoid bottlenecks
- Map function: Input is key,value pairs. Output is intermediate key value pairs
- Reduce function: Input is key value pairs grouped by key. Output is merged result.
- The total algorithm works like Map -¿ shuffle/sort -¿ reduce

The algorithm has fault tolerance, where it reexecutes failed or missing tasks. If the master fails it restarts the task, but this is very rare. It mitigates slow workers by duplicating tasks near phase completion.

The main bottleneck of the system is that the reduce phase has to wait for the map phase to complete. The reduce phase must be commutative and associative, and must have a neutral element (i.e. 0 for summing)

2 Databases

There are several models for the connection between clients and databases:

- Client-Server model: Clients connect directly to a central database server
- Connection pooling: Reusing connections to reduce overhead (JDBC/ODBC)
- Middle-tier Architecture: Application server manages connections and load balancing

- Shared vs. Dedicated servers: Trade-offs between resource isolation and scalability

The storage of data within a database can be expressed in a logical structure and a physical structure. The logical structure consists o:

- Table space: Contains segments, such as tables/indexes
- Segments: Composed of extents: contiguous data blocks:
- Data blocks are the smallest storage unit, i.e. 2KB

The physical structure consists of data files mapped to OS blocks. As best practices, tablespaces for tables/indexes are separated to optimize file sizing.

Clusters also have different important infrastructure concepts:

- High availability: Consists of Active-Active nodes, where multiple instances access shared storage. There is an automatic switch to standby nodes in case of crashes.
- Single point of failure is mitigated via redundancy
- Shared-Nothing architecture: Nodes operate independently

The data structure in databases works as follows:

- Tables: Rows and columns stored in segments/extents
- Indexes: Stored on a B-tree: a balanced tree for efficient lookups
- High water mark is used to track used and free space. Free space can be reclaimed with the SHRINK operation

Indexes speeds up queries but it requires more maintenance.

Partitioning improves performance and manageability by splitting large tables. Common types are range partitioning, list partitioning(by discrete values), and hash partitioning. You can also make composites of this.

Sharding spilts data across independent databases/nodes. There is horizontal sharding, where you have the same database schema on both nodes but with different data, or there is vertical sharding where different nodes store different types of data. Sharding is better for linear scalability and fault isolation, but backups become more complex and cross-shard queries aswell.

The CAP theorem states there is always a trade off between consistency (all nodes get the same data), availability (every request gets a response) and partition tolerance (System works despite network failures).

NoSQL databases are a special type of database. The two most common are Cassandra and MongoDB. Cassandras properties are:

- Ring Architecture: Peer-to-peer nodes; no master.
- Write Path: Memtable $\rightarrow$ SSTable $\rightarrow$ Disk (lazy writes)

- Read Path: Checks memtable/SSTable first; prioritizes local/rack data.
- Replication: Tunable consistency (e.g., quorum reads/writes).
- Virtual Nodes (vnodes): Automatic data distribution (default: 256/node).

MongoDBs characteristics are:

- Document Model: BSON/JSON-like documents with dynamic schemas.
- Atomicity: Single-document transactions.
- Replication: Primary-secondary nodes with automatic failover.
- Change Streams: Real-time data change notifications.

SQL and NoSQL usually get compared via ACID vs BASE:

- ACID (SQL): Atomicity, Consistency, Isolation, Durability.
  - Atomicity: All operations in a transaction succeed or fail together.
  - Consistency: Data remains valid per defined rules.
  - Isolation: Concurrent transactions do not interfere (e.g., row/table locks).
  - Durability: Committed data survives failures.

  There are exclusive locks for write operations and share locks for read operations. Deadlock occurs when transactions wait cyclically for resources; resolved via rollback.
- BASE (NoSQL): Basic Availability, Soft state, Eventual consistency.
  - Basic Availability: System remains operational during partial failures.
  - Soft State: Replicas may temporarily diverge.
  - Eventual Consistency: Data converges to consistency over time (e.g., Cassandra's tunable consistency).

3 Infrastructure as Code

The goals from IaC are:

- Reproducibility: Create identical environments consistently.
- Ephemerality & Immutability: Treat infrastructure as disposable; replace rather than modify.
- Transparency: Version-controlled, auditable configurations.
- Automation: Integrate infrastructure into CI/CD pipelines.

- Scalability: Manage large-scale deployments efficiently.

The core features of IaC are:

- Declarative Code: Define what infrastructure should exist (not how to create it).
- Version Control: Track changes using Git (e.g., GitHub, GitLab).
- Continuous Integration/Deployment (CI/CD)

There are thus many different parts and tools to IaC:

- Application-Level: NPM, pip, NuGet (dependency management).
- Containers: Docker (Dockerfile, Docker Compose).
- Orchestration: Kubernetes (Helm, Kustomize, ArgoCD), Rancher, k3s.
- Configuration Management: Ansible, Chef, Puppet.
- Virtual Machines: Vagrant.
- OS-Level: CoreOS, Flatcar, k3OS.
- Infrastructure Provisioning: Terraform, AWS CloudFormation.

There are a couple challenges to IaC:

- Ephemeral vs. Stateful: IaC promotes statelessness, but most apps require state (e.g., databases). Solutions: External storage (S3, databases), backup strategies.
- State management: Terraform tracks state to detect drift (differences between desired and actual infrastructure). Backends: Store state remotely (e.g., Terraform Cloud, GitLab) to enable team collaboration.

4 Big Data Processing

There are a couple of different choices for processing data:

- HDFS/Data Lake: Immutable, append-only storage for bulk CPU-intensive processing.
- Lambda Architecture: Consists of a batch layer (HDFS version to process historical data), a speed layer (Handles real time streams with low latency) and a serving layer (merges results for queries)
- Kappa Architecture: Simplifies Lambda by using a single stream-processing pipeline (e.g., Kafka queues for raw, preprocessed, and analytics data).

There are 2 main types of streaming: Microbatching and True streaming. Microbatching processes streams as discrete RDD batches, which is usually simpler to integrate but has more latency. True streaming processes each result individually.

Declarative infrastructure such as Kubernetes uses operators, which are custom controllers, via custom resource definitions. They use a control loop: observe -¿ analyze -¿ act.

The BSP model can be used to scale iterative algorithms, using the overarching steps: Compute, synchronize, repeat. Vertices thus compute in parallel and exchange messages, and aggregators reduce the network traffic (think of sum/max)

5 Data Processing Architecture Weaknesses and Fail Points

Cassandra:

- Architecture: Distributed, masterless nodes with data partitioned by hash (virtual nodes/vnodes).
- Write process:
  - Data written to commitlog (disk).
  - Sent to responsible node (hash-based).
  - Stored in memtable (memory) → sstable (sorted string table) → disk.
  - Temporary nodes handle writes if primary nodes fail.
- Read Process: Parallel reads across nodes; prioritizes local → rack → data center → remote data.
- Replication: Tunable replication factor (e.g., RF=3 for fault tolerance).
- Can have both strong consistency and eventual consistency. Higher consistency reduces availability (CAP theorem)
- Gossip Protocol: Nodes exchange state information to maintain cluster awareness.

MongoDB:

- Document Model: BSON/JSON-like documents with embedded fields (denormalized) or references (normalized).
- Atomicity: Single-document operations are atomic; multi-document transactions require careful design
- Scalability: Sharding (horizontal partitioning) and replication (high availability).

- Change Streams: Real-time data change subscriptions.

Common database pitfalls:

- Connection Management: Frequent connect/disconnect operations are costly.
- Bind Variables: Avoid hardcoded SQL (security/performance risks); use parameterized queries.
- Bulk Operations: Commit per transaction, not per row.
- Cursor Overuse: Multiple cursors for similar queries waste memory.
- Lock Contention: Poor isolation levels or long transactions block resources.

To have a processing infrastructure be scalable, we occupy ourselves with the following components:

- Firewall/Proxy: Manages traffic and load balancing.
- Processing Sites: Stateless services with local cache databases (e.g., Redis, MongoDB) to reduce latency.
- Metadata Database: Tracks data lineage and state
- Data Warehouse: Centralized storage for analytics.
- Storage Sites: Distributed storage (e.g., S3, HDFS).

The main performance factors are I/O Bottlenecks, caching and DNS/Proxy Servers: Critical for routing and failover.

6 CEPH deep dive

The purpose of CEPH is to have a large-scale, distributed storage system designed to handle massive amounts of data (petabytes to exabytes) with high reliability, scalability, and performance. Its key features are self-healing, thus it automatically recovers from failures. It must be autonomous, thus have no single point of failure, and it must be scalable thus add nodes seamlessly. Its components are the following:

- RADOS (Reliable Autonomic Distributed Object Store): Core storage layer managing data distribution and replication.
- LIBRADOS: Library for direct RADOS access (C/C++, Python, etc.).
- RADOSGW: RESTful gateway compatible with S3/Swift APIs.
- RBD (RADOS Block Device): Distributed block storage for VMs/disks.
- CephFS: POSIX-compliant distributed file system.

CEPH prioritizes consistency over availability over performance.

The CRUSH algorithm is a decentralized, deterministic data placement algorithm. It uses pseudorandomness to ensure a uniform distribution without central coordination. The data placement are based on configurable rules. Its key procedures are:

- TAKE(a): Adds item a to a working list.
- SELECT(n, t): Chooses n items of type t (e.g., disks, cabinets).
- EMIT: Finalizes placement.

The advantages are that it is decentralized and thus does not have a single bottleneck, and that it is self-managing thus has automatic recovery and rebalancing. This rebalancing does trigger overhead due to data movement. And naturally we have to deal with the CAP theorem.