DB Scaling

Arnav Gupta

December 7, 2024

Contents

1 DB Recap

DBs provide persistent storage and coordination in a large-scale system (also biggest scalability challenges).

Disk 10 billion times larger than registers but 10 million times larger delay.

Goal is to work with registers, caches, and RAM, rather than SSD and disk.

All computer storage is limited to reads/writes.

Magnetic disks: read/write head can read charges on a tiny portion of the magnetic disk

RAM (memory): memory and flash chips store lots of data, but only few bytes can be transferred at once since only couple hundred electrical connections at the edge

SSD (flash) similar to RAM but with fewer electrical connection

1.1 RAID: Redundant Array of Independent Disks

Disks have limits on capacity (12 TB), throughput (150 MB/s), and likelihood of failure.

RAID uses multiple disks:

- increase capacity through multiple disks
- increase throughput by accessing data in parallel on multiple disks
- reduce impact of failure by storing data redundantly on multiple disks

Disk interface is just array of sectors, so easy to crease **logical/virtual disk** made of sectors from multiple physical disks.

Unlimited capacity but computer has:

- limited compute power
- limited IO bandwidth

Single machine design scales well, so used by relational DBs.

Programming languages deal with storage directly (except SQL):

- programmer writes code to move data from memory to disk
- disk use must be purposeful

Normal way of accessing persistent storage is to pass data in and out of files using open/read/write functions. DBs let programmer use persistent storage without considering files and with advanced performance optimizations, using some query language.

2 Relational DB

Store data in multiple tables with predefined schema (set of columns). Rows are added over time and can refer to other rows through foreign keys.

Many tables allows:

- many users to refer to shared region data without repetition or inconsistency
- a user to have an arbitrary number of positions

A relational DB allows many-to-one and many-to-many relationships while keeping columns finite and clearly defined.

DB **schema** defined tables, columns, primary and foreign keys.

Relational DB benefits:

- scalability: work with data larger than RAM
- persistence: keep data after program finishes
- indexing: efficiently sort and search along various dimensions
- concurrency: multiple users/applications can read/write
- analysis: SQL query language is concise yet powerful
- integrity: restrict data type, duplicate entries, and transactions
- deduplication: save space, keep common data consistent
- security: different users can access specific data

Using file system does not give analysis, integrity, and deduplication, and is not as good for indexing (just filename/path) or concurrency (no transactions).

3 DB Indexes

Sorting data can be helpful for search (binary search).

Sorting is not a complete fix since:

- can't sort in multiple dimensions
- insert new data without shifting (requires rewrites)
- does not take advantage of storage hierarchy (access disk in every step since index spread over full data set)

DB indexes use tree or hashtable, giving logarithmic speed, while allowing quick insert/delete.

Indexes are defined when table is created (primary key unique for each row). Must be able to quickly check that new value does not already exist, so index on primary key.

Certain queries are too slow. Without proper indexes, DBMS examines every row, so adding 1+ indexes dramatically speeds up query.

```
CREATE INDEX index_name ON table_name ( column_name );
```

Multiple indexes allows finding rows based on multiple criteria.

DBMS must translate query into table lookups, trying the most efficient choices (indexes help). Can analyze query execution plan with `EXPLAIN` keyword.

Index columns if used in `WHERE` conditions, `JOIN` conditions, in `MIN` or `MAX` aggregation function, or a foreign key refers to it.

Indexes consume storage space and must be updated when data modified, so don't add unless necessary.

4 DB Scaling

Load balancer does much less work per request than DB.

DB clones are hard due to coordination.

Query planners optimize order of table accesses and index use. RAM used to store more important data and indexes. Responses can be cached and replayed if data unchanged.

To avoid DB bottleneck:

- avoid unnecessary queries (cache data in frontend)
- buy fast machine with RAM for caching
- use fastest possible disks (SSDs, RAID)
- use **read replicas** or **sharding**

4.1 Read Replicas

Most DB traffic is reads. **Replicas** have full copy of all data and can handle reads.

All writes must go to **primary** server, with data changes pushed to read replicas.

Replicas can be behind primary, so sensitive reads should use primary.

Too many replicas would make data push process a bottleneck in the primary.

Read replicas not infinitely scalable, primary is central bottleneck. Primary must send each replica copies of each write.

Optimal number of replicas directly proportional to the ratio of reads and writes, $\approx$10 in typical app.

Multi-level replication can extend read-scalability, with reads going to bottom level replicas and middle-level replicas pushing to children.

Adding more replication levels gives width but adds more delay between write at primary and data availability at read replicas.

Use read-replicas through load balancer in front of read-replicas.

With read-replicas, writes are not scalable, capacity is not scalable, and primary is a single point of failure.

5tandby primary can help to take over if main primary fail (app switches if main primary stops responding).

Using multiple primaries gives same performance bottleneck and could introduce data inconsistency for concurrent operations.

4.2 Sharding

For horizontal scaling of writes and capacity:

- **functional partitioning**: create DBs to store different categories/types of data
  - limits queries joining rows in tables in different DB
  - only few functional partitions possible (not scalable)
- **data partitioning**

With **sharding**, divide data universe into disjoint subsets called **shards**.

Sharding key determines assignment of rows to shards.

Relational DBs don't support sharding natively, so must be done at application level.

Sharding benefits:

- capacity scales
- data consistent
- if sharding key chosen carefully
  - balanced data
  - many queries involve only 1 or few shards, so no central bottleneck

Sharding downsides

- cannot use plain SQL
- queries must be adapted for sharding
- if key chosen poorly, shard load is imbalanced, by capacity or traffic
- some queries involve all shards, so capacity is limited by single machine's speed

For N nodes, total request rate R, node capacity C:

- if each request send to one node: $R_{max} = NC$
- if each request sent to constant k nodes: $R_{max} = NC/k = \mathcal{O}(NC)$
- if each request sent to all nodes: $R_{max} = C$

5 NoSQL

Sharding does not work well if all nodes involved (scaling limited). NoSQL databases solve problem by **denormalizing** data, so duplicated data isolates queries to one node.

Normalized relational DB has no duplication of data. Foreign keys point to shared data. To optimally partition rows into shards, can solve balanced graph partitioning problem.

Balanced Graph Partitioning model for DB sharding:

- nodes represent DB rows, edges represent FKs
- partition nodes, such that
  - total edges between partitions is minimized (need to fetch data from another shard is minimized)
  - nodes per partition is roughly balanced (balanced shards)

Solving graph problem is NP-complete. Model's cost is too simplistic (does not account for frequency).

Even with optimal partitioning, still have FKs between partitions.

Data affects solution quality since:

- random interconnections hurt
- nodes with high degree (many edges) hurt
- structured, independent relations are easy
- nodes with single edge are easy

By removing FKs and JOINs with NoSQL, no more references, so **denormalized** data allows for easy partitioning, since there are just copies of references data.

NoSQL DBs are key-value stores using a single column and a single table (usually).

A single column is required since no references (unknown schema) and unknown positions. Some NoSQL DBs allow multiple columns, but each row can have different columns.

Some NoSQL DBs allow multiple tables, but since rows can have any format, no need to.

5.1 Hash Table

Hash: algorithm that takes value and returns pseudo-random value derived from it (constant but unpredictable)

Same input always gives same output. Input length can vary but output has fixed length.

Hash Table: stores key-value pairs

Hash the key to determine address where value is stored. If address already full (collision), use next open slot.

Hash table is alternative to search tree (find data in single step), but bad with ranges.

In a **distributed hash table**:

- each cluster node responsible for a range of hash values
- each client gets list of nodes and range assigned to each
- when querying for key's value, client computes key hash to determine which node to query for data

NoSQL DBs use distributed hash table. Interface is *get* a value for a key or *put* a value for a key.

Each operation only affects the node storing that key, so very scalable.

For distributed, shared-nothing architecture:

1. create **cluster** of computers connected to each other
2. each node in the cluster stores a fraction of the data set

NoSQL Downsides:

- just one indexed column (key) since index built with hash-based partitioning
- denormalized data is duplicated (wastes space and must be edited everywhere)

- references possible but following reference requires another query, probably to another node (no constraint checking)

Normalized data would not perform well in a NoSQL DB. References not enforced by schema, so can become broken.

6 Distributed DB Consistency

To find data, client must have list of all nodes, and **hash ranges** assigned to each node. Sharing the node/range info is a **distributed consensus** problem.

6.1 Shared-Nothing Architecture

Each request handled by one node (no bottlenecks). Both throughput and capacity directly proportional to number of nodes.

Distributed Hash Tables can scale to 1000s of nodes.

Many nodes means high chance of node failure, so must replicate data to avoid loss. Can create overlap in hash ranges covered by nodes.

Replicating data introduces possibility of inconsistency.

6.2 Consistency

CAP Theorem: a distributed system cannot achieve all 3 of the following:

- **Consistency**: reads always get the most recent write
- **Availability**: every request received a non-error response
- **Partition Tolerance**: arbitrary number of messages between nodes can be dropped (or delayed)

When distributed DB nodes are out-of-sync, must either accept consistent responses or wait for nodes to resynchronize.

To build distributed DB where every request immediately gets a globally correct response, need a network that is 100% reliable and has no delay.

CAP theorem gives tradeoff between consistency and delay. Inconsistency more important than delay (only centralized DB can give both).

Distributed NoSQL DB designs give different options for handling consistency/delay tradeoff.

Client-centric consistency properties:

- **monotonic reads**: if client reads the value of x, later reads of x by the same client will return the same value or a more recently written value
  - can fail from reads from different nodes during incomplete write
  - prevent problem by making client connect to same node for every request or delay second request
- **read your writes**: if client writes value to x, later reads of x by the same client always returns same value or a more recently written value
  - can fail if system allows write on one node and read from another
  - prevent problem by making client connect to same node for every request or delay second request
- **monotonic writes**: if client writes twice to x, first write must happen before second
  - second write can occur on a node before first write arrives
  - does not matter unless writes are cumulative
  - prevent problem by making client connect to same node for every request or delay second request

To achieve consistency:

1. make client send all requests to one replica node
   (a) simple solution, but consistency problems arise when node fails
2. make client wait until read or write is synchronized across whole system
   (a) simplest approach is for client to send request to all nodes and wait

For monotonic read, wait for all nodes to return same value. For read your writes, wait for all nodes to acknowledge write.

Quorum: min percentage of committee needed to act

Wait for quorum of acknowledgment of consistent data before considering read/write completed Prevent progress until replicas have certain degree of consistency.

Read/write quorum of majority gives balanced read/write performance.

Send request to all nodes but wait for quorum of responses.

In practice, there can be many nodes, but a constant number of replicas for each data key. Quorum only applies to replica nodes.

A distributed system is **linearizable** if the partial ordering of distributed actions is preserved:

- distributed actors each know the order of their own actions
- certain knowledge must never be contradicted by the distributed system
- creates a partial ordering of all events in the distributed system

Every observed serialization of the parallel activity must be agreeable to individual actors. Observations will vary across the system.

In addition to NoSQL DBs, distributed file systems can use DHTs. Path is key, value is file contents.

7 Choosing a DB

7.1 Relational DB

Relational DB is most common, traditional solution. Use transactions for steps to be completed atomically together.

7.2 NoSQL DB

Transactions less common on NoSQL DBs, since slow. Transactions less necessary since a single key stores a lot of related data that can be modified at once.

Transaction can be implemented by locking the keys involved:

1. lock keys involved (lock prevents reads/writes)
    (a) replicas must agree to lock

   (b) multiple competing lock requests may occur in parallel, but once must be chosen, so multiple rounds of communication may be needed to agree

2. execute transaction on all replaces, and wait for all to confirm
3. unlock key involved (let reads/writes proceed)

To implement distributed lock:

- requires atomic conditional write operation
  - many NoSQL DBs support something like this
- if scalability not a concern
  - store transactional data in a SQL DB
  - use a SQL DB to implement lock used to control access in NoSQL

7.3 Column-Oriented Relational DB

Read-replicas and sharding help for online transaction processing.

Online Analytics Processing (OLAP) involves few large queries:

- analytics queries with scanning tables, not using indexes
- must be parallelized over many nodes
- workload is mostly reads, with occasional importing of new data

Column-oriented DBs optimized for SQL analytics workloads.

7.4 Semi-Structured Stores

MongoDB stores JSON objects. Best for changing and keeping history of JSON documents.

MongoDB details:

- built-in sharding
- master/slave replication
- JS expressions as queries
- runs arbitrary JS functions
- performance over features

- best for dynamic queries, preferring indexes over functions, good performance on big DB, and if documents change frequently and keeping history

ElasticSearch also stores JSON documents, but indexes every word in the document and to handle advanced queries. Uses inverted index with DHT. ElasticSearch is advanced search. Best for objects with flexible fields (plain text) and need search by all words in the document or need to construct complex search queries.

Cassandra rows have key mapped to value, which is group of columns mapped to values. Column names are indexed within the row. Row key is the hashing key that determines on which nodes the row is stored. Cassandra stores huge datasets. Best for when requirement is to store data so huge that it doesn't fit on the server, but still want a familiar interface to it.

DynamoDB is 2D key-value store, with **partition key** (hashed to find partition), **sort key** (allows efficient range queries withing partition key), **primary key** (from partition and sort keys together), and **attributes** (key-value pairs stored under primary key).

7.5 Distributed Cache

All data stored in RAM for high performance. Redis understands many types of data values.

Originally developed to reduce load on relational DBs through caching.

Same basic key-value abstraction as NoSQL distributed DBs. Store data across many nodes.

Have same data consistency issues as NoSQL DBs.

Optimized to do everything in memory (except storage on disk).

In cache, items expire, speed is primary goal, and capacity is limited.

As opposed to CDN, distributed cache caches commonly used data that contributes to responses.

Redis is blazing fast storage. Best used for rapidly changing data with a foreseeable DB size and for caching data that can be rebuilt from another data store.

7.6 Networked File Systems and Cloud

Networked file system provides regular filesystem interface to applications by mounting remote drive. Not too useful today, but may be necessary if app is built to work directly with local file system. Modern apps should instead interact with cloud-based storage services.

Cloud object store (S3) is a flexible general-purpose file store for cloud apps, managed by a cloud provider (unlimited capacity). Provides a network API for accessing files (like REST), so app accesses files like remote DB.

Hadoop File System is good for using Hadoop/Spark for distributed processing, where data too big to move for analytics. Allows data to reside on same machines where computation happens, making processing efficient. Designed to work well with Hadoop distributed processing tools.