Relational (SQL) Databases

Structured Query Language (SQL)

Its a "query" language - a way of specifying what to fetch from the DB.
SQL is not a DB type - it's just a query language.
SQL is the de-factor query language for Relational DBs.

When people say SQL, they mean Relational DB.
Even in Relational DBs, you can choose other querying languages - example: GraphQL.

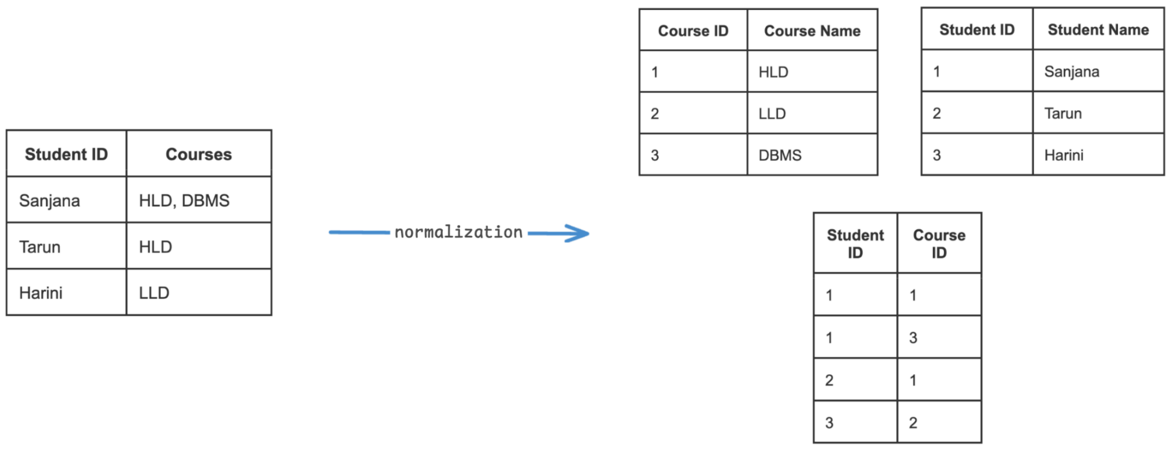
Strengths of RDBMS

All these strengths apply at low scale

Low Scale: "queries/second, amount of data" is small enough to be handled by a single server.

Data is stored in tables - tables can have relations with each other.

Normalization



Student ID	Courses
Sanjana	HLD, DBMS
Tarun	HLD
Harini	LLD

Course ID	Course Name
1	HLD
2	LLD
3	DBMS

Student ID	Student Name
1	Sanjana
2	Tarun
3	Harini

Student ID	Course ID
1	1
1	3
2	1
3	2

It is recommended to model data in a normalized manner.
Normalization prevents anomalies & reduces redundancy.

Strong Schema

Well Defined

1. You know exactly what tables exist
2. You know exactly what columns exist in each table
3. You know exactly the data-type of each column
    a. Allows you to allocate fixed amount of space for each row on the disk

Static

1. The schema does not change for different rows.
2. Changing the schema is difficult.
    a. It is possible to add/remove columns from an existing table - however that requires a complete table re-write
        i. extremely slow (table migration)
        ii. requires the service to be down
            1. they're ways around this (rolling migration)

Enforced

If a row violates the table's schema, then the DB will let you know. It won't let the write succeed.

ACID Transactions

Extremely powerful & good to have.

Atomicity

All or nothing: Each transaction is either executed completely (all rows) or not at all. There are no partial states.

Consistency

ACID consistency =/= CAP consistency (wait, what !!?)

CAP Consistency: *no stale reads.*
ACID Consistency: *db constraints are enforced*

1. schema: columns & data-types
2. null / unique / foreign key constraints
3. triggers
4. every transaction should leave the database in a consistent state

Isolation

Multiple transactions that are running simultaneously don't mess with each other.
There's different isolation levels. (homework: read up on this)

Durability

Any transaction that has been executed will be stored in non-volatile storage (HDD/SSD) and not just the volatile RAM.
Note: Durability does NOT protect you against HDD failures - only replication does.

Other Strengths

1. Very powerful querying capabilities
    a. you can perform joins
    b. filtering by column values
    c. aggregate calculations
    d. grouping
    e. nested queries
    f. recursive queries (*CTEs*)
2. Relational DBs are extremely mature

Weaknesses of RDBMS

When the scale becomes large, all the strengths become weaknesses!

Large Scale: data or requests is too large to fit on a single server

Normalization

On the frontend, we (almost) always need to show denormalized data (data about a lot of entities)
Therefore, to display the data, we need to perform joins.
For example, to display the page (https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array) you will need to join around 30-40 tables

1. If you have lots of tables, then joining all of them grinds to a halt
2. What if the data is too large to fit on a single server
    a. joins have to be executed across servers!
        i. impossibly slow
        ii. too much n/w overhead

Strong Schema

What if the data itself is inherently unstructured?

Q: How to model product listings for Amazon in a SQL DB?

Amazon has 10 million+ products across 10,000+ categories.
Every category has a different attribute set

1. **T-shirts:** Brand, Price, *Color, Fabric, Neck-shape, Sleeve length*
2. **Laptops:** Brand, Price, *Screen Size, RAM, CPU, GPU, OS*

3. **Notebook:** Brand, Price, *Page thickness, Number of Pages, Ruled*

Giant table with All Columns

products: id, name, brand, price, *Color, Fabric, Neck-shape, Sleeve length, Screen Size, RAM, CPU, GPU, OS, Page thickness, Number of Pages, Ruled, ...*

Too many columns (10,000 categories, 10 unique cols per category => 100,000 columns)

- every row will only store data for 10 columns - rest of the columns will be nulls
- SQL pre-allocates data for the entire row (all columns included)
    - even for the columns where the value is null
- you're effectively wasting 99.99% of the disk space

Multiple tables

products: id, name, brand, price
tshirts: product_id, *Color, Fabric, Neck-shape, Sleeve length*
laptops: product_id, *Screen Size, RAM, CPU, GPU, OS*
notebooks: product_id, *Page thickness, Number of Pages, Ruled*

1. if you want to fetch data like "find top 10 most expensive products" then you will have to join across 10,000 tables.
2. SQL is not designed for this scale
    a. SQL can handle lots of rows and terabytes of data (few tables, lots of data in those tables)
    b. but SQL cannot handle lots of "schema" (many tables)

Attribute List

products: id, name, brand, price
product_attributes: product_id, attribute_name, attribute_value

product_id	**attribute_name**	**attribute_value**
1	RAM	16GB
1	CPU	i9 14400k
1	Screen Size	17"
1	Fabric	cotton
2	Neck Type	Rounded
2	Sleeve Length	full
2	Fabric	cotton

1. **Schema enforcement is lost:** laptops can now have fabric
    a. you can still enforce attributes on the application layer
    b. but the DB has lost the ability to enforce a schema (you can assign any attribute to any entity)
2. **Fetching data about a single entry is too expensive:** if you need to show the details of the laptop - you have to fetch all attributes of the laptop, and then join them into an array

So we see that since the data was inherently "semi-structured" SQL was not a good choice. At low scale (only 2-3 different product categories) SQL could've worked.

ACID Transactions

When the data is large (high scale), then you need sharding - because you can't store all the data in a single server.

Sharding nullifies ACID

SQL dbs provide ACID guarantees only within a single server.
Because it is easy to do that

1. inside a single server, you can acquire locks easily (OS, hardware facilitates locks/semaphores)
2. inside a single server, you can share memory (RAM, HDD) - so no n/w overhead to communicate across multiple transactions - no consistency (stale read) issues when two threads are reading the data from the same location in the RAM
3. inside a single server, you know whether the write has succeed or failed
4. inside a single server, writes mostly always succeed

If you've multiple servers

1. locks is extremely hard - 2PC
2. sharing memory is not possible - keep copies, and that leads to consistency challenges
3. don't know if the writes succeeded - you have to pass acknowledgements
4. networks & servers regularly fail

Providing ACID guarantees across shards is extremely difficult

1. possible
2. but, very slow

NoSQL Databases

What is NoSQL?

NoSQL =/= No SQL (what??)

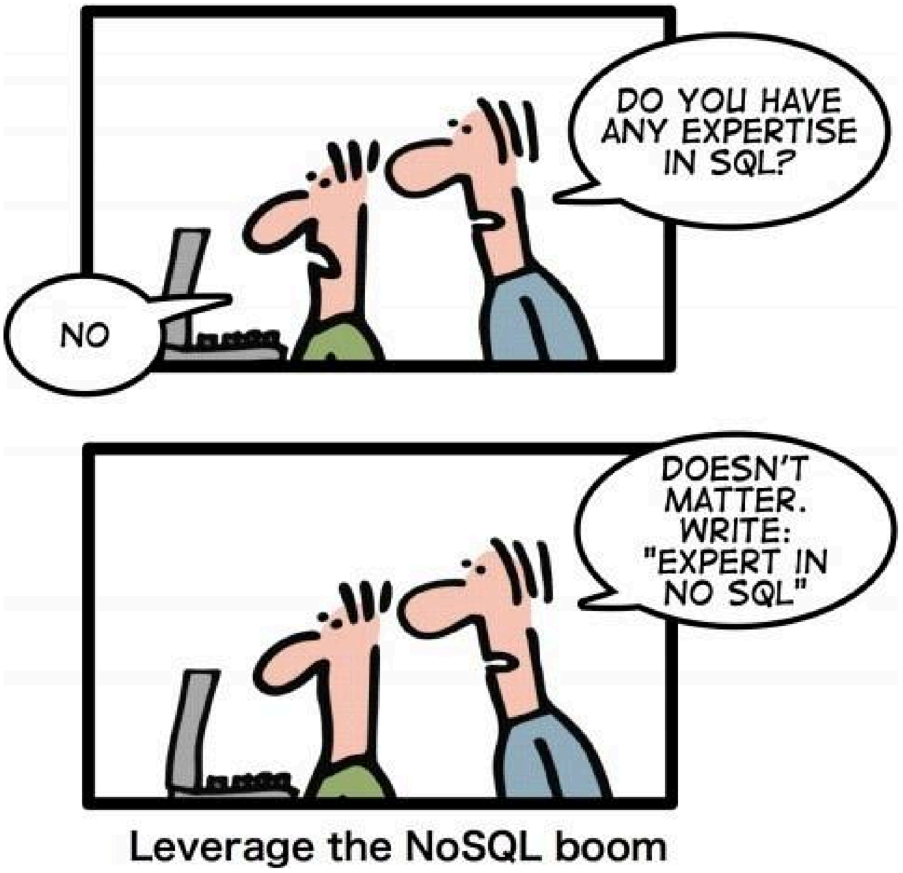
No SQL =/= don't use SQL

NoSQL = **Not Only** SQL - we will still continue using relational databases, but, we will augment their capabilities with additional non-relational databases

SQL dbs have existed for a very long time - even before modern computers came into picture, the theoretical foundations for relational algebra were already laid.
NoSQL dbs are extremely recent

1. first NoSQL db was BigTable by Google
2. NoSQL dbs started getting popular only after 2005
    a. most internet giants did not even exist before 2005

Around 2009/2010 there was a conference about non-relational DBs. The promoters needed a twitter hashtag to go viral - ***#NoSQL***

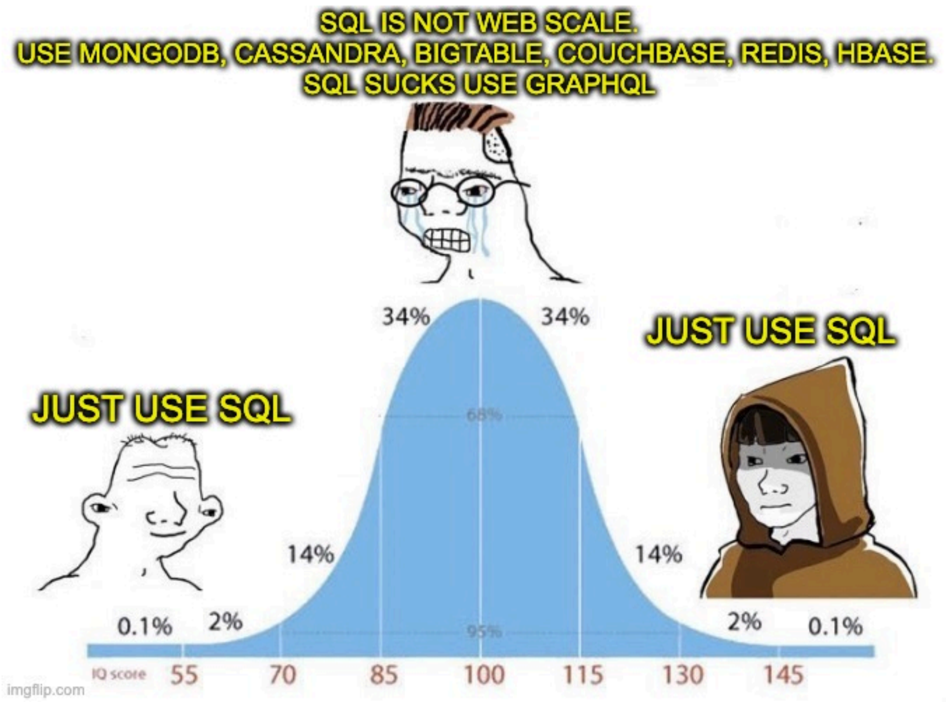

DO YOU HAVE ANY EXPERTISE IN SQL?
NO
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

Don’t jump to NoSQL

Your de-facto choice should ALWAYS be Relational (SQL) databases.
You can use NoSQL - but, only if, you can justify the need for it.

Modern SQL databases *(like postgres, mysql)* can do absolutely everything that any NoSQL can do & even more

- just at a low scale
    - the bar of "low scale" keeps increasing every day
- for up to 1 million userbase, a SQL db will just work flawlessly



Don't be this guy ↓
(caution: abusive language)
https://www.youtube.com/watch?v=b2F-DItXtZs

BASE

https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/

Basically Available

The system as a whole will remain available, even though some services might be unavailable for a small fraction of the users for some time.
High availability

Soft State

ACID transactions provide atomicity (all or nothing)
Soft state: Transactions **can be in a partial state** for some time - they're not all or nothing
(we will see this in the last class of HLD - distributed transactions in microservices - Saga Pattern)

Eventually Consistent

There might be stale reads. But, eventually (if we wait long enough) every write will be reflected across the entire system (all replicas)

Horizontally Scalable

SQL databases require manual sharding - they're don't provide built-in support for sharding (note: most modern SQL dbs have built-in support for replication, but not sharding)

NoSQL db are automatically sharded - they're built with horizontal scaling in mind.
You just have to get a bunch of servers, and install the database on them - and they will figure stuff out themselves.

Denormalization & Replication

SQL databases discourage denormalization & redundancy to remove anomalies.

NoSQL databases realise that

1. in the frontend you're anyway going to show denormalized data
2. to prevent data loss, we're going to have multiple copies of the data anyway

NoSQL dbs encourage storing data in a denormalized manner / semi-structured (sometimes even schemaless) manner.

Weaknesses of NoSQL databases

SQL databases have tons of features

1. joins
2. powerful ways to structure data
3. enforceable constraints
4. triggers
5. ACID guarantees
6. powerful indexing
7. powerful filtering
8. recursive queries
9. *modern features*
    a. full text search

b. semi-structured data (first class json support)
c. spatial indexing (nearest neighbor queries)
d. vector support (KNN queries)
e. denormalized views (table views)

Any feature that you can think of, modern SQL dbs have it.
The only con of SQL is that it is feasible only at low scale.
SQL databases "generalize" - jack of all trades, master of none

NoSQL databases can work at massive scale because they don't support most of the features

1. it is hard to do everything perfectly
2. it is easy to do a few things flawlessly

NoSQL databases "specialize", jack of 1 trade, and master of it

You can shard a SQL db manually

- modern SQL dbs don't have built in support, but there will be popular sharding solutions available for each SQL db
- you can also use "managed" SQL databases like Amazon RDS
    - these support sharding (partitioning) out of the box
    - you just have to configure the sharding key

Choosing a Sharding Key

Sharing key decides how your data gets distributed across various db servers.
It will also determine how the queries need to be routed to fetch the data.

Q: Can a sharding key be composite?

Yes. Totally allowed.

Primary Key vs Sharding Key

Primary Key: Uniquely identifies a item in the data *(eg. row in a table)*
Sharding Key: Just tells you how to distribute the data

it is common to use the same key as both the primary & sharding key - but not always.
A lot of times, you will have to choose a different sharding key as well.

If you want to read/write some data, which key should you use?

You need to use both!
Sharding key will tell you which server to go to (routing).

Primary key will tell you which entry to touch inside that server.

What constitutes a "Good" Sharding Key

When choosing a sharding key, we should assume that different keys will end up in different servers. Note that a single server can house data for multiple keys.
Basically, if you have 2 items, and the value of the sharding key is different for those two items, then those two items will (most likely) end up in different servers.

1. Equal data distribution

All key values should be equally likely.
If some values are more likely than others - that will lead to "hot shards" *(a shard that is overwhelmed with data/requests)*

Age: ages 0-5 are less likely. Ages 50+ are less likely. Ages 15-30 are most likely.
Sharding based on age will be a bad idea, as it will lead to poor load distribution.

User id: user is unique for each user. Suppose you want to shard posts by the user id, then of course, there will be some users that have made many many posts and some users that have made 0 posts.
But since each server has thousands of users, overall, the data distribution across the servers will be pretty even.
Note: *this is not always true: eg, "celebrity problem"*

Q: if we're sharding by user_id, does that mean that each user gets a dedicated server?

No. A single server will house multiple users (because multiple values of the key can hash to the same server)
If Post_13 is by User_1 and Post_20 is by User_2, then most likely, these posts will end up in different servers.

2. High Cardinality

Cardinality = set of possible values

Age: {0, 123} has 124 possibilities max.
This means that if we have 125 servers, then 1 server will not get any value.
Basically, we're limited to max 124 servers. We cannot scale any further than that.

User_Id: {64 bit value} has 16 quintillion possibilities
There's no limit to the number of servers we can scale to. 10 million servers? No issue.
Note: we're not saying that we will have 16 quintillion servers. A single server will house millions of users.

3. Part of every read/write request

Because if the sharding key is not part of the request, then how will you route the request in the first place? You will have to go to every shard - fan-out (bad)

4. No fan-outs

Most frequent queries should have to hit only 1 (or at most 2) shards.
No frequent query should lead to a fan-out request.

Note: rare queries are okay to fan-out

5. Immutable

The value of the sharding key should not change for any row.
Because if we change the value of sharding key, then we will have to re-shuffle the data.

Age: what happens when the user's B'day comes? We will have to move the user to a different server because the age has changed.

Examples

Banking System

- Users can have active bank accounts across cities.
- Most Frequent operations
    - Balance Query *`(account_id)`*
    - Fetch Transaction History *`(account_id, date_range)`*
    - Fetch list of accounts of a user *`(user_id)`*
    - Create new transactions *`(sender_account_id, reciever_account_id, amount)`*

~~location~~

A user can have accounts across multiple cities. If we shard by location, then, we will need to store the user's data across multiple shards.
If we fetch list of accounts of users, we will have to make a fan-out request.
Poor load distribution (large cities will have much more data than small cities)
Not part of request

~~branch_id~~

(same issues as location)
+ low cardinality

account_id

ideal sharding key here.
This means that 1 account's data will be housed entirely within a single server *(note: 1 server still houses thousands of accounts)*

- **balance enquiry:** hit only 1 shard *(the account's shard)*
- **transaction history:** hit only 1 shard (all the account's transactions will be there)
- **list of accounts of the user:** ***This one is tricky.*** *User to account_id mapping needs to be stored separately in another store.*
- **transaction:** this transaction is data about both the sender and the receiver
    - each transaction has to be stored in 2 shards (sender's shard + receiver's shard)
    - this is NOT a fan-out. Hitting 2 shards is okay.
    - Done in 2 phases. Phase 1, debit from sender. Phase 2, credit to recipient.

~~transaction_id~~

Is not even available for any of our queries.
Transaction id is generated in the response. It's not part of the request query.

~~account_balance~~

Subject to change. Every time the user adds/removes money, we will have to move their data.
Not part of the request.
Poor load distribution.

~~timestamp~~

NEVER choose timestamp.
- unequal load distribution (recent days are hot, future days have 0 data)
- subject to change (updated at timestamp can be changed)
- not part of the request

Types of NoSQL Databases

Different types of NoSQL databases specialize for specific features & use-cases.

Be careful about the features you see being claimed by popular NoSQL databases.

Key-Value

Simplest type of NoSQL database.
Think of just as just a giant hashmap distributed across servers

Examples

- Redis (in-memory + optional disk persistence)
- Memcached (in-memory)
- DynamoDB (disk persistence)
- ...

How is data stored

Keys & Values (hashmap)
Both the key & the value are just plain strings (the database doesn't know & doesn't care about what is contained inside those)

Violated by most modern key-value stores.

- Redis
  - Key => string
  - Value => Redis supports multiple datastructures (arrays, json, sets, sorted sets, bloom filter, custom data structures)

Key (string)	**Value** (string)
`"contest:13:page:10"`	`"{ ["rank" 130, "..."], [...], [...] }"`
`"contest:13:winner"`	`1361`

Strengths

- Extremely simple.
- Because they're typically in-memory, they're ridiculously fast!

Queries

- get(key) => value
- set(key, value) => ack/failure

Weaknesses

- No complex queries (joins / filtering)
- No search
- No indexing

- No relations

Sharding Key = Key

Automatically sharded by the hash(key)

Primary Key = Key

When to use

- **Cache**
- Storing very simple data (key-value) that needs to be queried extremely frequently (user preferences, rules, rate-limiter bucket counts)

In Redis any string has a limitation of max 500 MB.
Q: Does this mean that storing 500MB of data per entry is a good idea?
Absolutely NO!
To use key-value database, your keys <≈ 100 bytes, values <≈ 10 KB

Redis (Mandatory Reading)

1. Try online: https://onecompiler.com/redis
2. Quick start: https://redis.io/learn/howtos/quick-start
3. Eviction policies & Cluster mode: https://docs.google.com/document/d/1k4nzubvtX_yLctUT4VWK8ZJt4KCcOEdRJdxQgWCaiU8/

Redis (Optional Reading)

1. Tutorial: https://redis.io/university/
2. Docs: https://redis.io/docs/latest/

Document

Storing unstructured / semi-structured data.
Think of this as a collection of **json/jsonb** files distributed across multiple servers.

Examples

MongoDB, ElasticSearch, Couchbase, CockroachDB, ...

How is data stored

```
{
    __doc_id:   uuidv4
    product_id: int
    name: string
    type: string            (t-shirt)
    brand: string
    color: string

    neck_type: byte
    sleeve_length: byte
    image_url: string
}
{
    __doc_id:   uuidv4
    product_id: int
    name: string
    type: string            (laptop)
    brand: string
    color: string

    ram: {
        size: integer
        technology: string      (ddr4/ddr5/..)
        cas_latency: string
    }
    cpu: string
    image_url: [string, string, string]
}
```

Every document in mongodb has a unique document id.
Any document can have any set of attributes.
`__doc_id` is automatically created on the client side when the document is being inserted

Strengths

- can store semi-structured / unstructured data
- powerful query & search capabilities via indexes
    - you can put an index on any top level attribute (not-nested)
    - caution: indexes are maintained locally - there's no global index
        - example: suppose we've amazon's product listing, & we've an index on the attribute "brand"
          then, the query `mongoClient.find({brand: "dell"})` will be a fan-out read! This query will go to every shard (broadcast query) and then within each shard, the shard will use the index on "brand" to return the relevant documents
          `mongoClient.find({brand: "dell", type: "laptop"})`
          `will not be fan-out`

- provide **full-text-search**

Queries

`mongoClient.find( {__doc_id: "..."} )` will fetch the document with the given document_id
`mongoClient.find( {brand: "dell"} )` will fetch the documents where the value of the attribute "brand" is equal to "dell"
- if there's an index on the column, it will use this index, otherwise, it will go search through all documents.

Primary Key = `__doc_id`

Sharding Key = ?

The default sharding key (if nothing else is configured) is `__doc_id`
But any top-level attribute (or composite of top-level attributes) can be configured as the sharding key.

For example: for the amazon products listing we could set the sharding key as product_type (laptop/tshirt/...)

Weaknesses

- No relations & joins
- No global indexes
- (typically) no ACID transactions
  - MongoDB provides ACID transactions, even across shards
  - caution: ACID within the same shard is fast, but ACID across shards is slow

When to use

- Unstructured / Semi-structured data
- Full-text search
- example: amazon product listing / social media posts / user notes / ..

Documents should <≈ 10MB
MongoDB has a cap of 16MB for their documents

MongoDB

1. Try online: https://mongoplayground.net/
2. Tutorial: https://www.mongodb.com/docs/manual/tutorial/getting-started/
3. Docs: https://www.mongodb.com/docs/manual/

8:57 => 9.05

Column Family / Wide Column

The data is still tabular in format (just like relational databases)
However

1. Data is stored in wide-column format (as opposed to row-wide of SQL)
   a. very fast aggregate queries
2. No joins
   a. data is tabular, but no relations b/w tables

Timeseries DBs are a subcategory of wide-columns DBs

Examples

Cassandra (popular), BigTable (first), ScyllaDB, **HBase**, ...

How is data stored

Every column family database stores data in a very different manner!

Strengths

1. Highly performant for analytics (aggregate queries that span a few columns but many rows)
2. Provide easy pagination (time based)
3. Extremely fast writes!
    a. Not as fast as key-value, but still, much faster than others
    b. writes are fast because they use LSM trees

Weaknesses

(same as previous - no joins, no relations, ...)

When to use

1. Analytics
2. High write throughput
    a. sensor data (gps coordinates / IOT sensor data / ..)
3. Paginated queries / time based queries
    a. fetch the location history of user in the last month

Cassandra (Optional)

1. Try online: https://jbcodeforce.github.io/db-play/
2. Introduction: https://cassandra.apache.org/_/cassandra-basics.html
3. Case Studies: https://cassandra.apache.org/_/case-studies.html
4. Architecture - Overview:
https://cassandra.apache.org/doc/stable/cassandra/architecture/overview.html
5. Architecture - Guarantees:
https://cassandra.apache.org/doc/stable/cassandra/architecture/guarantees.html
6. Data Modeling: https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html

Large File / Object

Examples

- S3
- Google Cloud Storage
- Git Large File Storage (Git LFS)
- Hadoop Distributed File System (HDFS)

How is data stored

flat files directly on the disk (files are chunked & distributed across servers)

Strengths

- These files can be extremely large (100TB file)
- can stream the data

Weaknesses

- No search
- No relations
- Difficult to modify files (can append, can replace, but not modify)
- reads & writes are slow

When to use

- any sort of user generated multimedia (pdf, images, videos, audios, csv, zip, ...)
- HTML/CSS/JS (if they're static)
- any large files (>10KB) that are mostly static

Others

Graph

Famous, because people have a weird attraction to graphs
But rare in practice!

- Facebook friendship / Linked follower / Twitter follower => Social connection graphs
- none of these companies stores the relationships in a graph db => all of them use SQL to store the relations
    - they use a graphDB for the search queries (as a cache) in front of the SQL db

Good when your queries require "path-finding"

1. Recommender systems (Amazon / Netflix)
2. Shortest route b/w two places (Uber / Google Maps)

Vector

More popular due to AI
Provide fast K-Nearest Neighbor queries - extremely useful for searching over an embedding space

Object Oriented

Table inheritance (postgres)

.... every kid and their grandmother have their own NoSQL database type.

Multimodal

All modern database are multi-modal - they provide multiple features.

Choosing the right Database

Twitter-HashTag

Requirements

- Store the most popular & most recent tweets for each hashtag
- Paginated queries (first 20 tweets, next 20 tweets, ..)
- Very large volume of tweet writes

~~SQL~~

scale is too large!
For popular hashtags like #USelections2025, #Diwali, ... a single server won't be able to store all the tweets (even for 1 hashtag)

~~Key-Value~~

Key? `"#Diwali2025 : popular"` or `"#Diwali2025: recent"`
Value? all the top-100 popular/recent tweets for the hashtag, or all the tweets for the hashtag

- Value is way too large (#Diwali will have 100 millions of tweets => multiple GBs of data)
- pagination will not be possible
- inserting a tweet will require re-writing the "recent" key value which will be costly

~~Document DB~~

Same reasons as key-value

Column Family

HBase for example
All our requirements match exactly with the strengths of Column Family DBs.

Live scores of Sports/Matches

Requirements

- Given a recent event or match, you have to show the ongoing score information

~~SQL~~

is the data too large? No - data will easily fit in a single server!

is the data relational? Maybe?

- match
- players
- overs
- individual balls

is the data relations? Is any user asking *"show me all the matches where Sachin got bowled out when Dhoni was balling"* (this query requires joins & relations)
Data is NOT relational!
We don't need transactions!
On frontend, we will show de-normalized data!

Key-Value DB

key: match_id
value: json object that contains all the information we want to show on the page (10kb)

Isn't the value changing constantly?
Cricket match => 6-7 hours, 240 balls
Number of updates/second? `0.011 updates / second`
A new ball is played every 1.5 mins
Key-value dbs can easily handle 100,000 updates/sec in a single server!

Cab location in Uber

Requirements

- show current location of a cab
    - **key-value:** key (driver_id)   value (latitude, longitude)
- show historical location of a cab by date or recency
    - **column family**
        - location data is GPS sensor data
        - high write throughput
        - pagination
        - queries by time

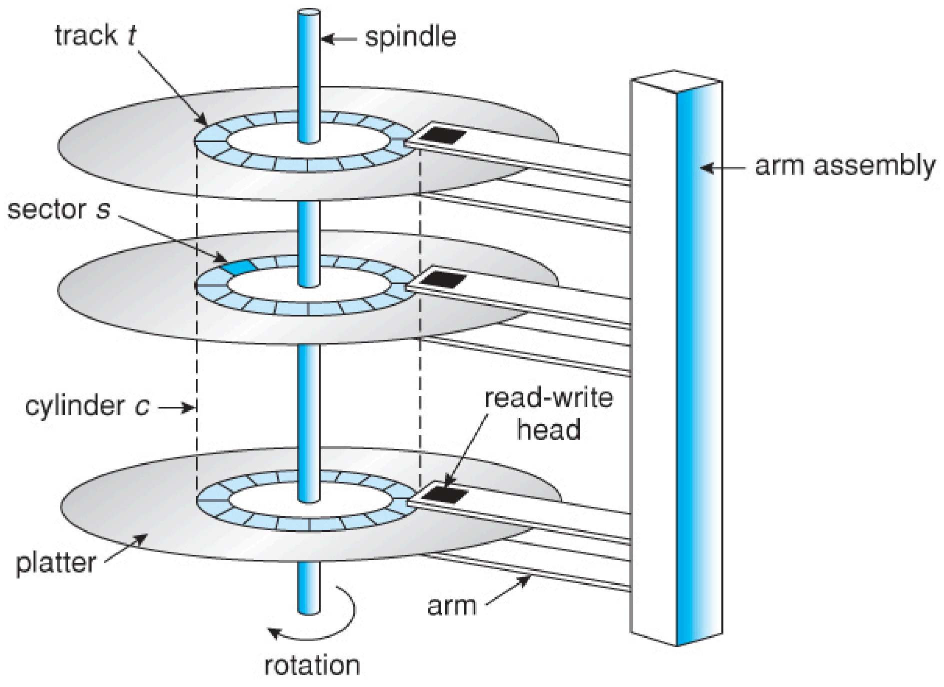
~~SQL~~

Uber has ~5 million drivers, say each driver's phone app sends the new GPS coordinates every 10 seconds.
Number or writes / second? **500,000**
Can a single SQL server handle 500,000 writes / second? Absolute no way!

A typical SQL db server can handle 10 - 100 writes / second
If the writes are very simple => max 1000 writes/second

How Storage Works

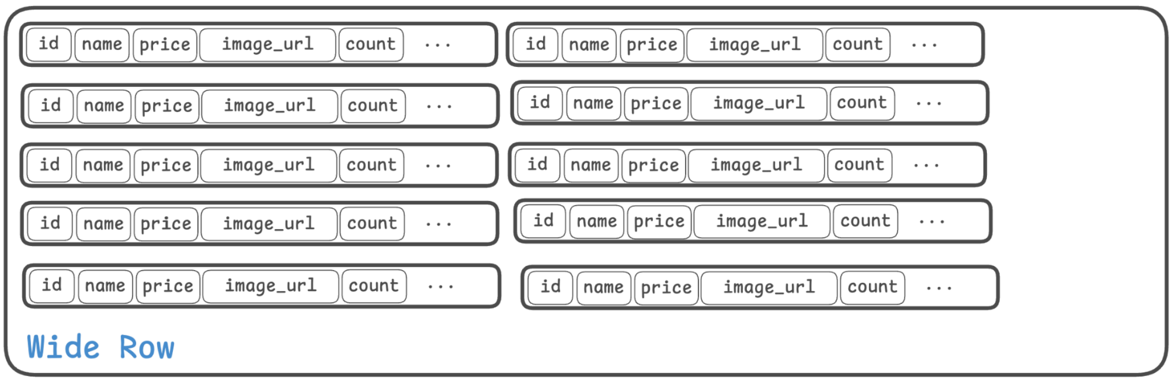
1. **Magnetic:** https://www.youtube.com/watch?v=wtdnatmVdIg
2. **Solid State:** https://www.youtube.com/watch?v=5Mh3o886qpg
3. **RAM:** https://www.youtube.com/watch?v=7J7X7aZvMXQ



Sequential vs Random Access

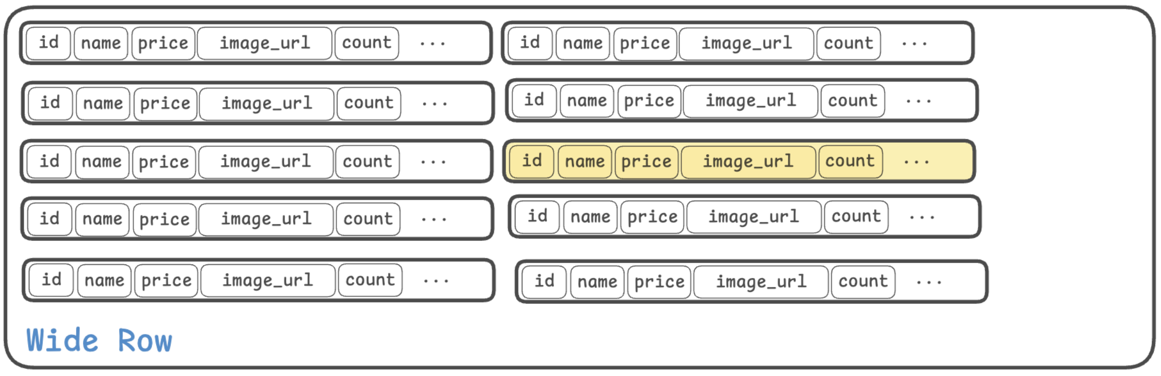
Sequential access is up to 10,000x faster than random access!
True irrespective of the storage technology (HDD, SSD, RAM, L3 cache, Brain)

Wide-Row (SQL db)



Query 1 (fast)

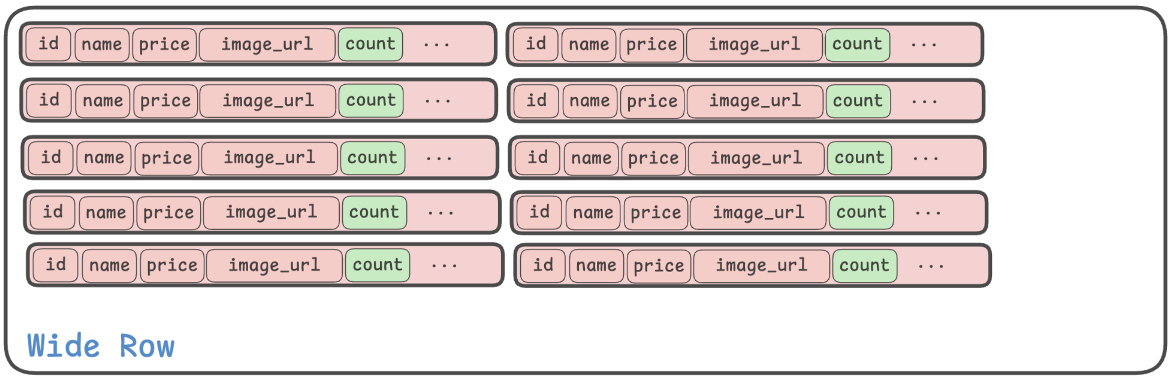
```
select * from product_listing where product_id = 123
```



You will read exactly what you need, and you will use all that data

Query 2 (ridiculously slow)

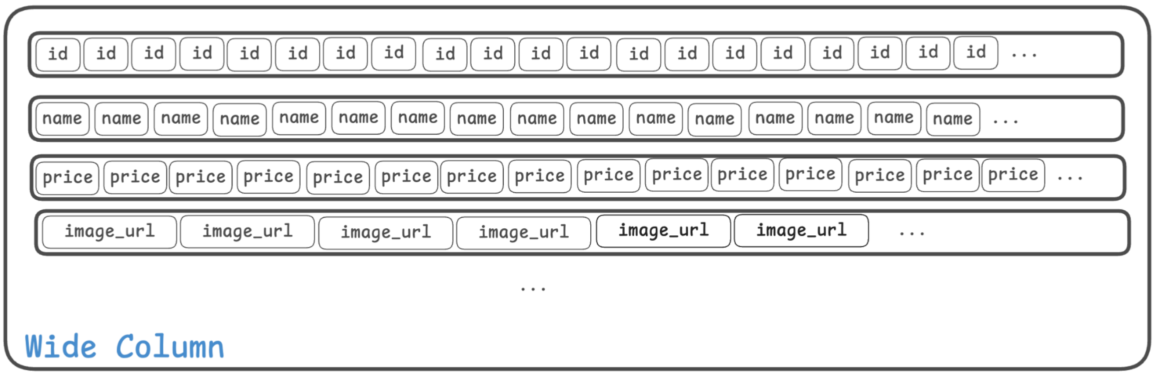
```
select sum(count) from product_listing
```



You will read everything, and discard majority of what you read.
id (4b), name(20b), price(4b), image_url(200b), count(2b) => total size: 230b

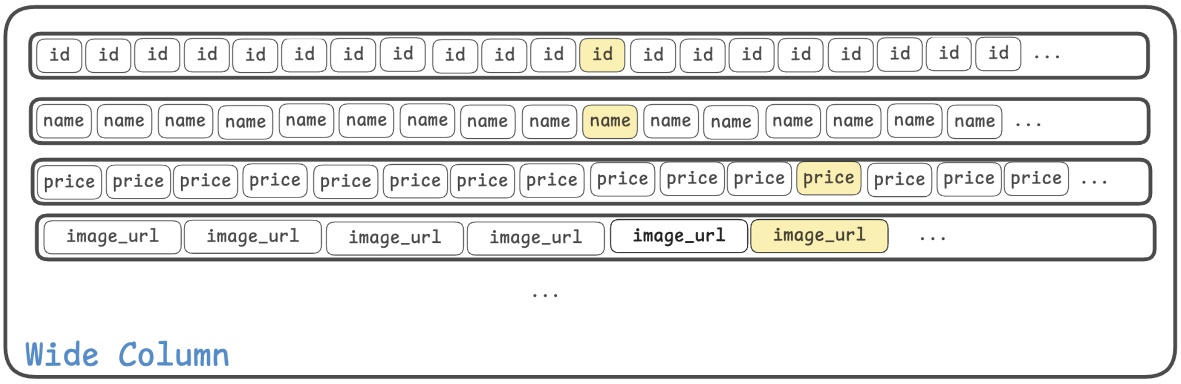
for every 230 bytes read, we're using only 2 bytes => 0.8% utilization

Wide-Column



Query 1 (moderately slow)

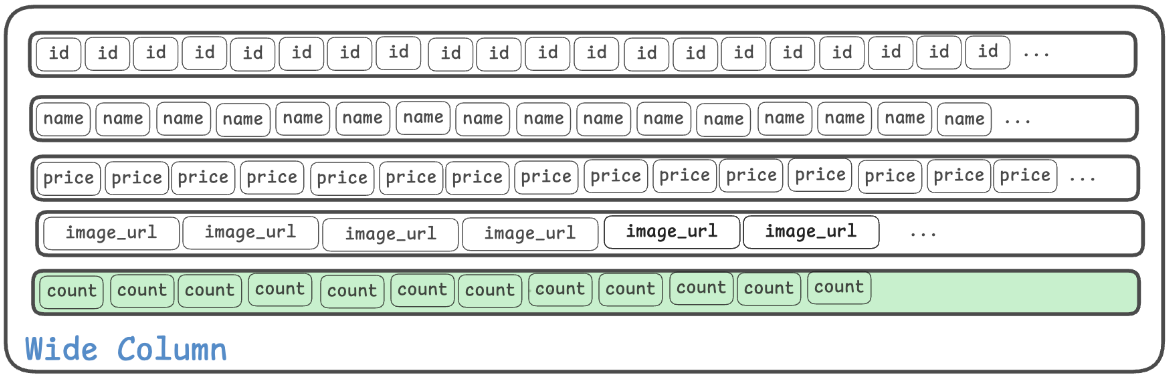
```
select * from product_listing where product_id = 123
```



Not discarding any data, but, we're doing random reads (reading non-sequential data)

Query 2 (ridiculously fast)

```
select sum(count) from product_listing
```



Read exactly what we need, and we will not discard anything (100% utilization)

Just by switching how we store data from row-first to column-first format, we got a speedup of (100 / 0.8) times = 125x

10466.2732183
text: 13 characters => 26 bytes if encoded as utf-8
float64 (binary) => 8 bytes