Applied Database Principles

Store data to make its intended use fast, efficient, easy and correct

Benjamin Bercovitz - Co-founder, Verkada

Introduction

Stanford CS '09 (BS) '10 (MS)
Concentrations: Systems, Databases

...

Co-founder, Verkada (2016)
cloud-managed security cameras, locks, ambient sensors
1700 employees

Why I'm here

1. Introduce a few types of databases
2. Point out some key features and limitations
3. Give a method for choosing your structured data storage

Some databases

1. Transactional SQL database (OLTP)
2. Persistent k/v database
3. In-memory k/v database
4. Analytics SQL database (OLAP)
5. Free text search database

Before you start: Collect some info

1. Data structure definition
2. Total data size
3. Write rate
4. Anticipated read patterns (and rate)
5. Scope of isolation

Running Example: Pirates!

```json
{
    "userId": 12312,
    "orgId": 1098,
    "firstName": "Jack",
    "lastName": "Sparrow",
    "email": "swashbuckler97@example.org",
    "groups": [{
        "name": "senior pirates",
        "access": ["main deck", "treasure chest"]
    }],
    "canOpenTreasure": true
}
```

Data size:	10GB
Reads/sec:	low
Writes/sec:	low
Patterns:	

- lookup by ID
- lookup by orgId
- lookup by group

Isolation:	full dataset
Recency:	immediate

User Record

```json
{
    "doorLockId": 5666,
    "properties": {
        "beepEnabled": true,
        "power-settings": {
            "poe-802.3": "af",
            "power-save-level": 0
        },
        "unlockFor": ["jack", "nathan", "alec"],
        "soundSirenFor": ["grace", "martin"]
    }
}
```

Data size:	100GB
Reads/sec:	high
Writes/sec:	low
Patterns:	
• lookup by ID	
Isolation:	record
Recency:	immediate

Door Lock Record

```json
{
    "cameraId": 2562132314123,
    "orgId": 1098,
    "timestamp": 1705348871,
    "maxSoundLevel": "35 dB",
    "ambientLightLux": 77,
    "objectDetections": {
        "ship",
        "island",
        "mermaid"
    },
}
```

Data size:	40TB
Writes/sec:	high
Isolation:	row

Pattern 1: ID and time range

Reads/sec:	low
Recency:	~immediate

Pattern 2: last value by org ID

Reads/sec:	high
Recency:	immediate

Pattern 3: Daily summary

Reads/sec:	one / day

Camera Status Record

```
{
    "cameraId": "ab815fb6",
    "ts": 1705351366,
    "eyePatch": false,
    "parrot": false,
    "hat": "tri-corner",
    "vest": "six button",
    "beard": "two braids",
    "boots": "knee-high",
    "belt buckle": "enormous",
    "swagger": "comical"
}
```


Data size:	10TB
Writes/sec:	high
Isolation:	row
Patterns: lookup by AND query	
Reads/sec:	high
Recency:	recent

Detected Pirate Record

The dawn of durable storage

At some point, computers got durable storage.

People started to put structured data in there.

It was a mess.

Problems with stored data files

1. What if my program uses a different version of the file format?
2. How do I share my data with other teams?
3. How do I make changes at the same time as others without conflict or corruption?
4. What if my computer crashes?
5. Issues working with pointers - how to reference data without a memory pointer

Introducing, the data base

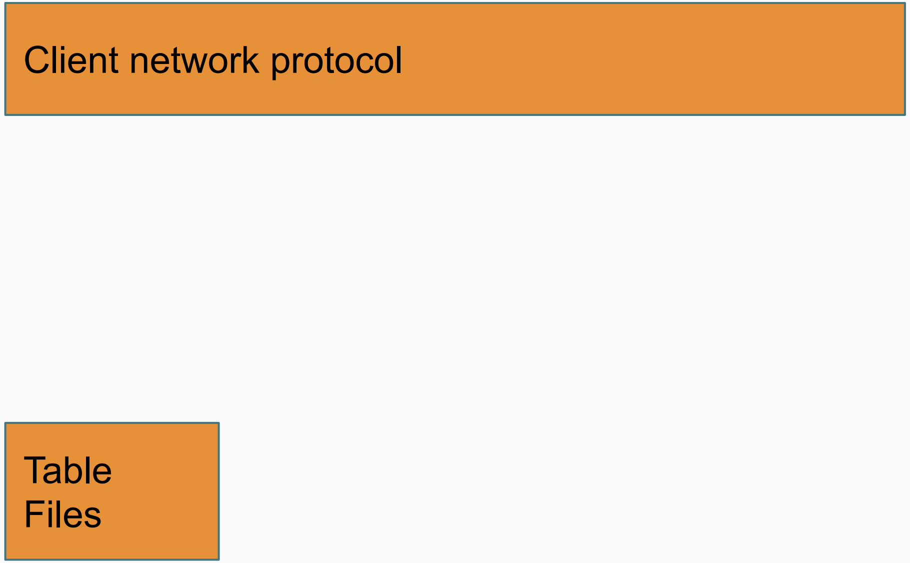
So, they invented an API to put on top of the storage.

- This was called a data base. (yes, two words. also, data were plural)
- Define what the data are logically (instead of how the bytes look on disk)
- Express which data you want to retrieve (instead of writing a program).

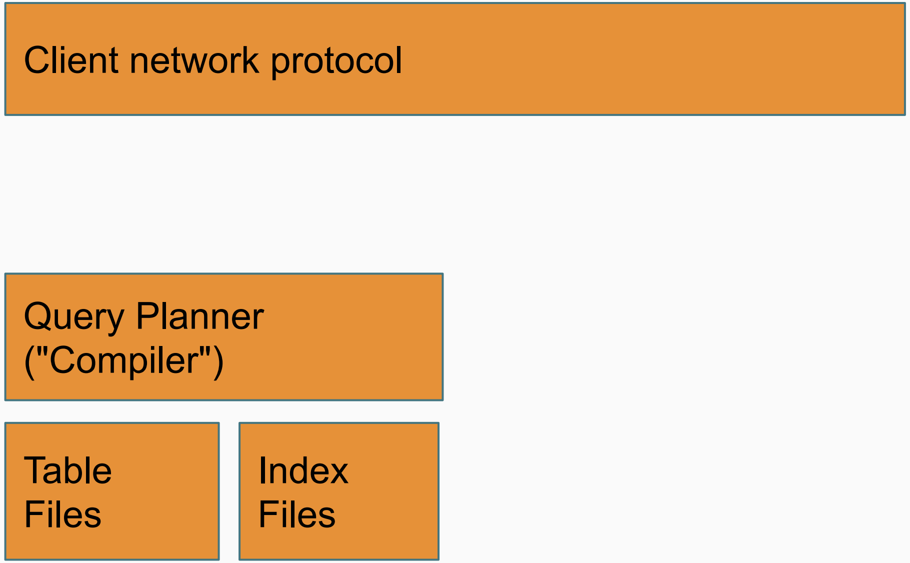
Stick figure database evolution

Table
Files

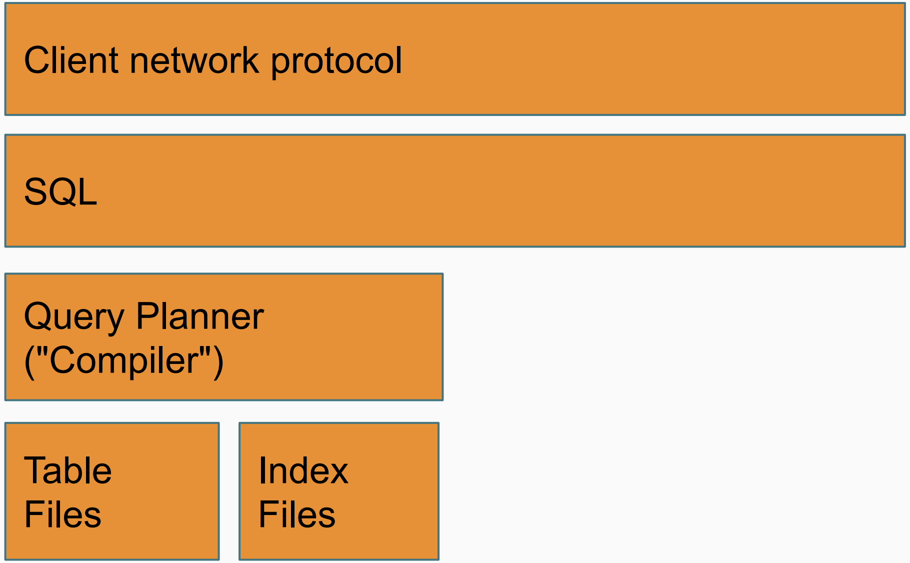
Stick figure database evolution



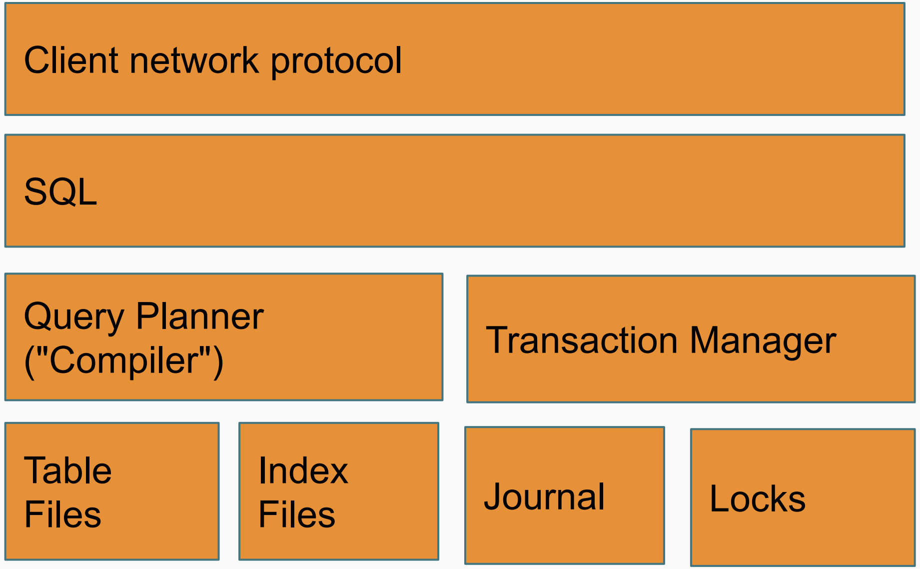
Stick figure database evolution



Stick figure database evolution



Stick figure database evolution



Transactional SQL database

Defining Features

SQL - Query expression language

ACID

Single processor and memory, disk files not shared

SQL

"Sequel"

SELECT ... FROM ... JOIN ... WHERE ...

ACID

Atomic - operations either fully happen or fully don't

Consistent - only one truth and data constraints are always satisfied

Isolated - concurrent operations of two clients do not run into each other

Durable - the data cannot be lost after an operation finishes

Indexes

Typically:

- Manually defined per table, can be modified later
- Automatically updated and kept consistent
- Key is some combination of columns (order matters)
- Stick figure: B-tree where values are the row IDs on the table
- Helps with query speed - more options for the Query Planner
- Adds storage cost and lowers writes per second

Query Planner

SQL -> Mathematical expression -> Candidate plans (programs)

Keeps statistics on tables and indexes for cost estimation

Most critical - table row count and index selectivity

Query Planner

"Joins/subqueries are always slow" - It's a LIE!

1. Your intuition of what the DB is planning is frequently wrong.

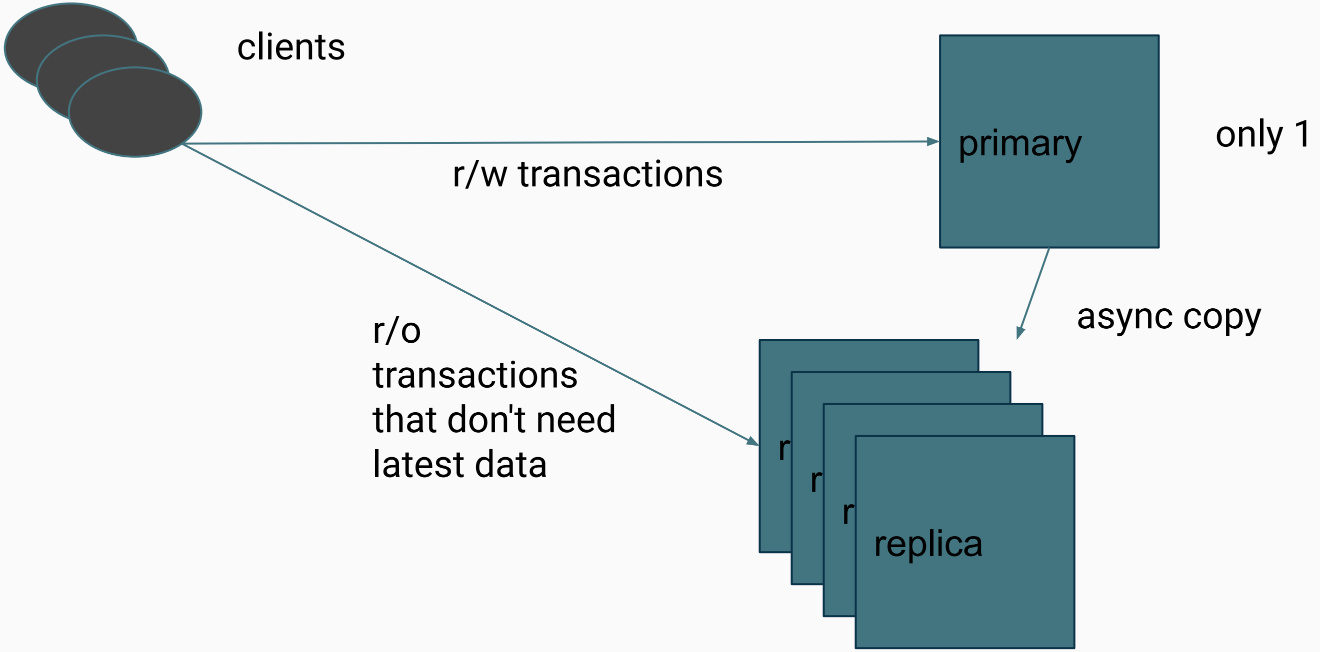
2. DB query plans can change suddenly - even with a single row added

Transaction Manager

- Automatically prevents data modification conflicts and inconsistencies
- Can roll back even a long series of updates if a conflict or failure occurs

- Tip: Default isolation level (READ COMMITTED) can include updated values from other concurrent transactions after they commit. This can have very unintuitive effects. For complicated transactions, use SERIALIZABLE by default
- Tip: Isolation level on reads also matters

Replication



At Verkada: AWS Aurora for PostgreSQL

- Introduced in 2017
- Heavily optimized/hacked for AWS infrastructure
- Disk-based, but all data should fit in RAM for best performance and scalability
- Automatic quick failover for primary node failure
- Up to 15 read replicas
- Query plans can be pinned to keep query performance from changing

How to use a SQL database

1. Decompose all data structures into unnested tables (BCNF)
2. Insert / Update / Query using SQL
3. Sit back and let the database do its magic
4. Profit

You don't need to know which queries future people will use!

You only need to know WHAT data they will use.

Normalized data

- Basic principle is DRY - every data "fact" is stored only once
- Tables contain fields and are defined up front
- Tables are linked together by unique keys (usually a synthetic ID)
- Basic relationships are: one-many and many-many
- Nesting, stringified fields, blobs, "k/v" tables are **not** normalized

```
"users": {
    "userId": 12312 {uuid_1},
    "orgId": 1098 {uuid_2}, // index on orgId
    "firstName": "Jack",
    "lastName": "Sparrow",
    "email": "swashbuckler97@example.org", // unique index
    "groups": [{
        "name": "senior pirates",
        "access": ["main deck", "treasure chest"]
    }],
    "canOpenTreasure": true
}
"groups": {
    "groupId": {uuid_3},
    "name": "senior pirates",
}

"access_levels": {
    "accessId": {uuid_4},
    "name": "main deck",
}
```

```
"group_access": {
    "groupId": {uuid_3},
    "accessId": {uuid_4},
}

"user_groups": {
    "userId": {uuid_4},
    "groupId": {uuid_1}, // index on groupId
}
```

User Record

```
{
    "userId": 12312,
    "orgId": 1098,
    "firstName": "Jack",
    "lastName": "Sparrow",
    "email": "swashbuckler97@example.org",
    "groups": [{
        "name": "senior pirates",
        "access": ["main deck", "treasure chest"]
    }],
    "canOpenTreasure": true
}
```

Data size:	10GB
Reads/sec:	low
Writes/sec:	low

Patterns:

- lookup by ID
- lookup by orgId
- lookup by group

Isolation:	full dataset
Recency:	immediate

✅

User Record

```json
{
    "doorLockId": 5666,
    "properties": {
        "beepEnabled": true,
        "power-settings": {
            "poe-802.3": "af",
            "power-save-level": 0
        },
        "unlockFor": ["jack", "nathan", "alec"],
        "soundSirenFor": ["grace", "martin"]
    }
}
```

Data size:	100GB
Reads/sec:	high
Writes/sec:	low
Patterns:	
• lookup by ID	
Isolation:	record
Recency:	immediate

✅

```
{
    "cameraId": 2562132314123,
    "orgId": 1098,
    "timestamp": 1705348871,
    "maxSoundLevel": "35 dB",
    "ambientLightLux": 77,
    "objectDetections": {
        "ship",
        "island",
        "mermaid"
    },
}
```

Data size: 40TB
Writes/sec: high
Isolation: row

Pattern 1: ID and time range
Reads/sec: low
Recency: ~immediate

Pattern 2: last value by org ID
Reads/sec: high
Recency: immediate

Pattern 3: Daily summary
Reads/sec: one / day

Camera Status Record

```
{
    "cameraId": "ab815fb6",
    "ts": 1705351366,
    "eyePatch": false,
    "parrot": false,
    "hat": "tri-corner",
    "vest": "six button",
    "beard": "two braids",
    "boots": "knee-high",
    "belt buckle": "enormous",
    "swagger": "comical"
}
```

Data size:	10TB ⚠️
Writes/sec:	high ⚠️
Isolation:	row

⚠️ **Patterns: lookup by AND query**

Reads/sec:	high
Recency:	recent

Detected Pirate Record

Transactional DB - Pros and Cons

- **Capped** writes/s
- **Pretty high** reads/s
- **Capped** total data size
- **Changeable** data layout
- **Flexible** access patterns
- **Easy** to read all the data out
- **Inconsistent** performance
- **Database-wide** atomicity

Cost driver is usually transactions per second

Transactional DB - Summary

Good for:

Metadata with infrequent updates and fits in RAM

Small data that needs complex internal consistency between multiple clients

Ad-hoc queries, flexibility

Ease of use

Bad for:

Machine-generated data series

Logs; anything with unbounded size

Schema-less data, blobs

Operations that require exceptionally high availability

Key-Value Persistent Database

History - web scale broke transactional DBs

- Starting about year 2000 with huge web platforms with dynamic user content becoming common, transactional DBs could not keep up.
- The big web companies were forced away from traditional databases for data that had unbounded growth.
- They each did this in different ways, but basically all by restricting isolation or consistency guarantees and limiting query types.

What broke?

Single processor, memory and disk on primary node -> vertically scalable

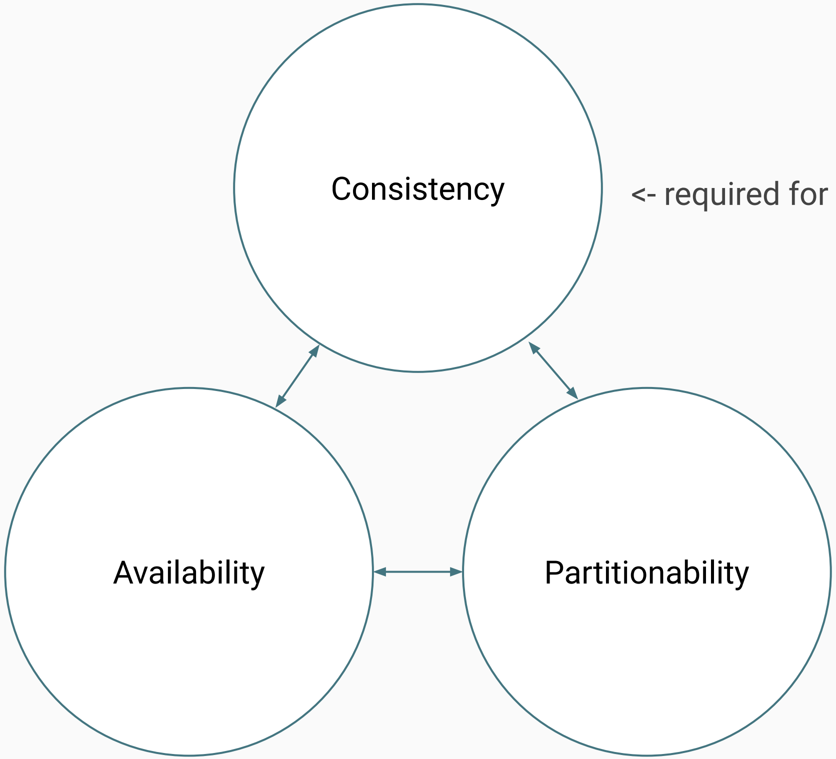
Sometimes the biggest is not big enough - 224 cores and 24TB RAM too small?

Side note: introduction of SSDs really saved this technology from its other bottleneck which was I/Os per second on HDD disk (10,000x difference). But it still has a limit

Why you can't just fix it with more primaries

CAP Theorem:

Can only choose two!



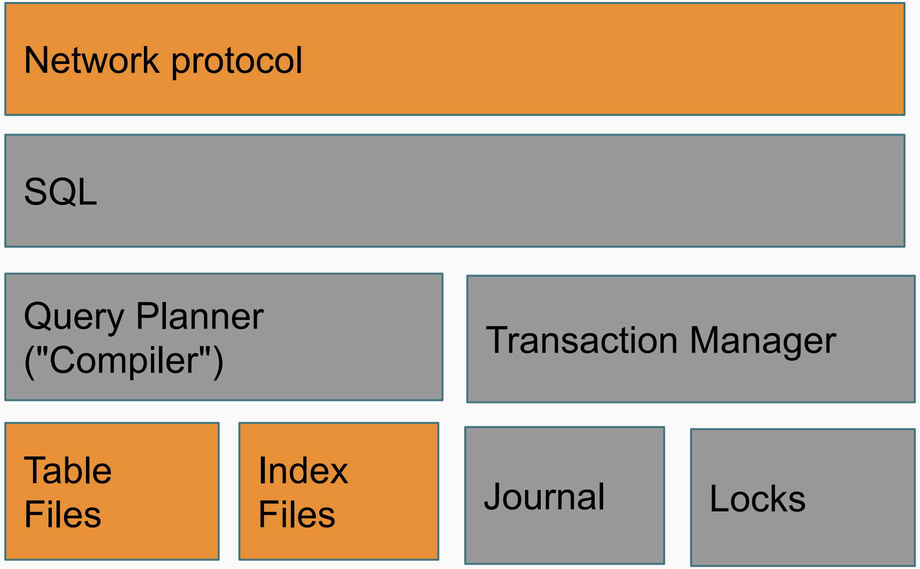
<- required for a lot of the DB features

<- required for distributed systems

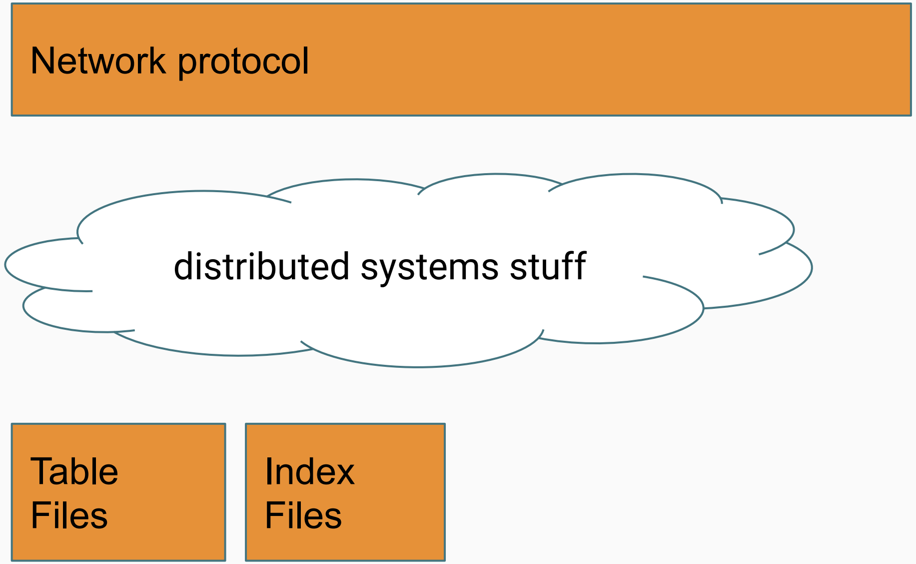
What about a "sharded" Transactional DB?

- This is OK if all of your tables are partitionable by a single ID (e.g. a customer ID) and your maximum partition size is bounded.
- If you are thinking of doing this, most likely your data is already suited to persistent K/V because it has a natural partition key.
- It has many of the drawbacks of both k/v and transactional DBs combined, plus is currently more difficult to manage operationally. If you shard, you need to be able to split or reassign shards to scale.
- Maybe there is a product / project out there to address the manual parts...

Start ripping stuff out



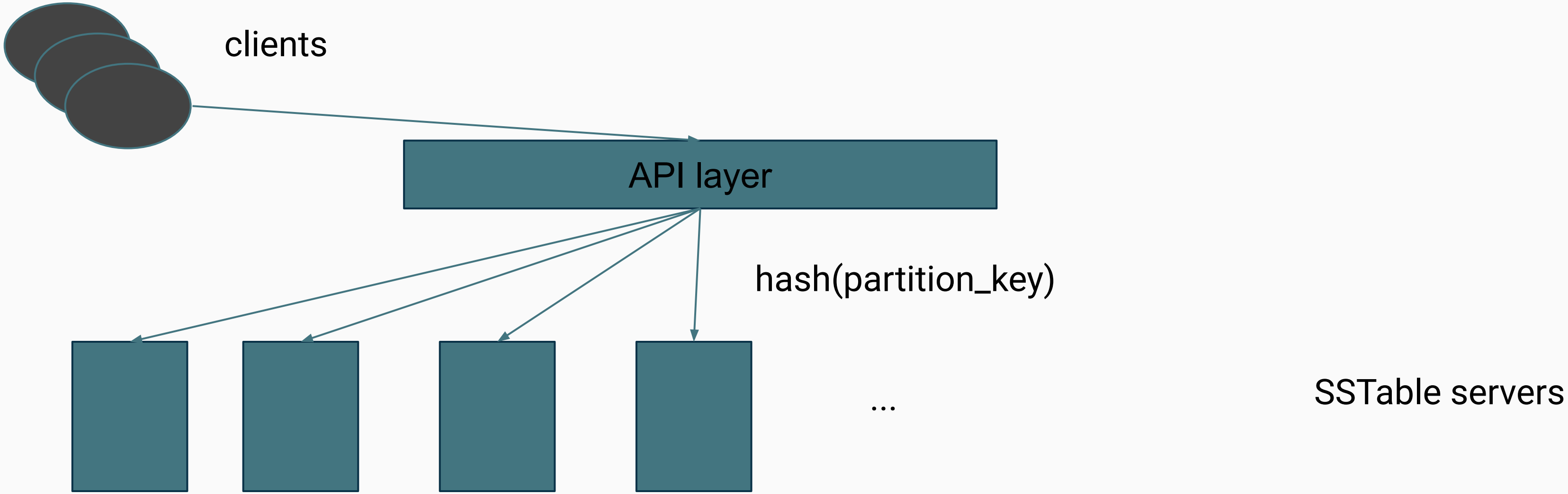
Start ripping stuff out



Typical k/v database Properties

- It's basically the storage API on the *inside* of a SQL database
- Very limited operations (but they are all fast and highly available)
- Typically, can read stale data after a write (eventually consistent)
- Granular atomicity - row level
- Scaling is basically unlimited and redundancy is handled for you
- Can still suffer from "hot keys"
- Each row can have different columns ... but should it?

Approximate System Architecture



Automatic shard splitting and reassignment
Automatic failure handling
Consistency ... eventually

Our selection: DynamoDB

- Automatic high availability and replication (3 replicas)
- Super fast (single digit ms)
- Unlimited size, unlimited scale
- Key = partition (hash) key + <optional sort (b-tree) key>
- Operations:
  - **Put**, **Get**, **Query** (Sorted index scan), **Scan** (Table scan, very slow)
- Can set a TTL field for automatic deletion of time-ordered data

Columns can have container types

- List
- Set
- Map

Atomic modification operations are supported so you don't need to read-modify-write to append an element to a list, for example.

Hash+Range Compound Primary Key

- Tables are defined with a unique primary key consisting of
  - a partition key (which is hashed to determine what nodes contain the data)
  - a sorted range key (to pinpoint the row uniquely in that partition)
- This enables a very efficient sorted index scan of everything on a single partition key - API is called **Query()**

Row-level Conditional Operations

- One powerful tool is Conditional Put
- Returns error if the condition fails, otherwise update is applied
- "Send the alert only if not previously alerted"

obvious way: (race condition)

```
x = GET(user); if x.alerted==false then x.alerted=true; PUT(user); alert()
```

better way:

```
...; PUT(user, alerted == false); alert()
```

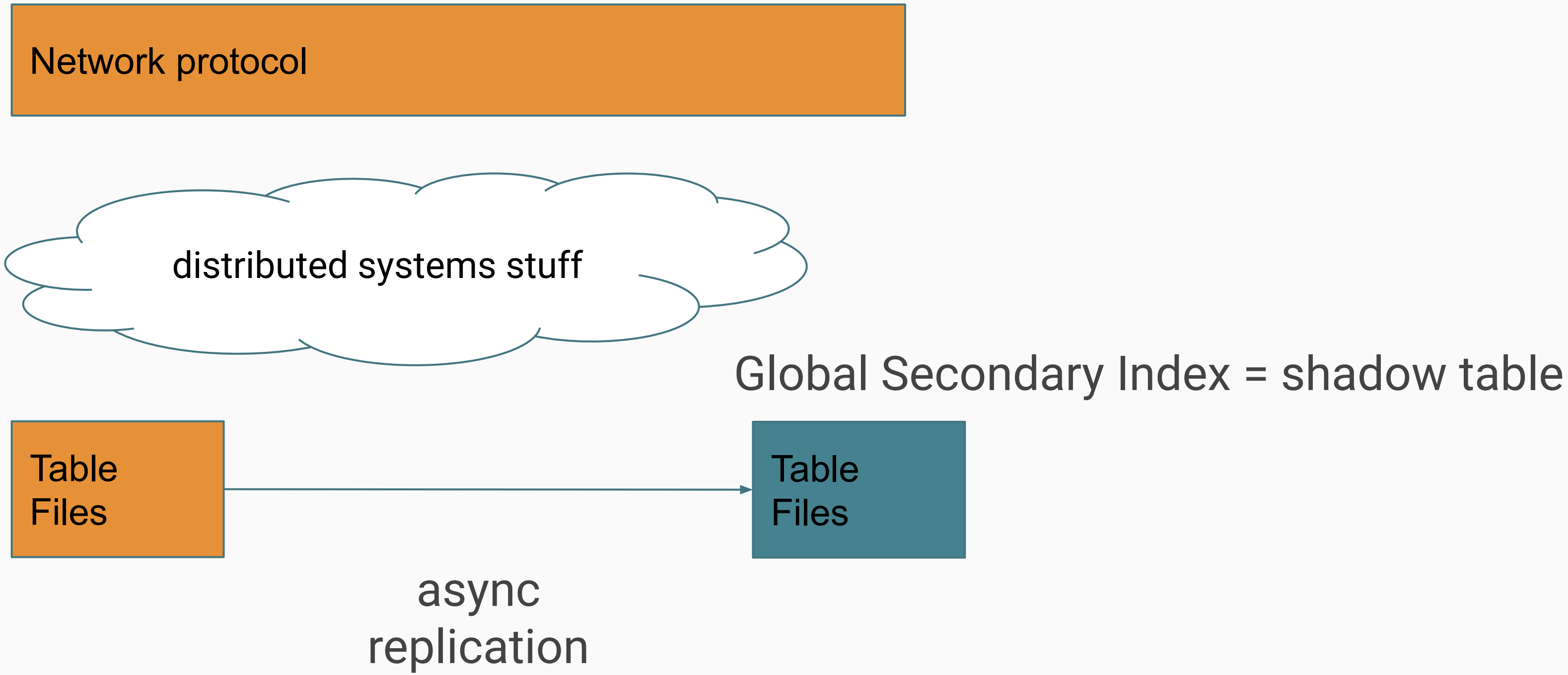
Selectable CP vs AP

- GetItem comes in two flavors: regular and consistent
- Regular is highly available, but the value might not always be the latest value if there are outages
- Consistent is read-after-write consistent, but might timeout if there are outages

Consistent costs twice as much, can you guess why?

Global Secondary Indexes

Need another access pattern? Select another partition and sort key!



Note: Global means "not restricted by the original partition key"

How to use a persistent k/v database

1. List out all the operations you want to support (reads and writes)
2. Figure out the keys required to support them
3. Put / Get items by key
4. Sit back and let the k/v DB scale to the moon
5. Profit

You don't need to know which data future people will read or write

You only need to know the access patterns!

```
{
    "userId": 12312,
    "orgId": 1098,
    "firstName": "Jack",
    "lastName": "Sparrow",
    "email": "swashbuckler97@example.org",
    "groups": [{
        "name": "senior pirates",
        "access": ["main deck", "treasure chest"]
    }],
    "canOpenTreasure": true
}
```

Data size:	10GB
Reads/sec:	low
Writes/sec:	low
Patterns:	

- lookup by ID
- lookup by orgId ⚠️
- lookup by group ⚠️

Isolation:	full dataset ❌
Recency:	immediate ⚠️

```
{
    "doorLockId": ~~5666~~ {uuid},
    ~~"properties": {~~
    "beepEnabled": true,
    "power-settings": {
        "poe-802.3": "af",
        "power-save-level": 0
    },
    "unlockFor": [
        "~~jack~~" {uuid}, "~~nathan~~" {uuid}, "~~alec~~" {uuid}
    ],
    "soundSirenFor": [
        "~~grace~~" {uuid}, "~~martin~~" {uuid}
    ]
}
```

Data size:	100GB
Reads/sec:	high
Writes/sec:	low
Patterns:	

- lookup by ID

Isolation:	record
Recency:	immediate

✅

```
{
    "cameraId": ~~256213~~ {uuid},   // partition key
    ~~"orgId": 1098,~~
    "timestamp": 1705348871,   // sort key
    "maxSoundLevel": ~~"35 dB"~~ 35,
    "ambientLightLux": 77,
    "objectDetections": {
        "ship",
        "island",
        "mermaid"
    },
}
```

✅

Data size:	40TB
Writes/sec:	high
Isolation:	row

Pattern 1: ID and time range

Reads/sec:	low
Recency:	~immediate

Pattern 2: last value by org ID ⚠️

Reads/sec:	high
Recency:	immediate ⚠️

Pattern 3: Daily summary ❌

Reads/sec:	one / day

```json
{
    "cameraId": "ab815fb6",
    "ts": 1705351366,
    "eyePatch": false,
    "parrot": false,
    "hat": "tri-corner",
    "vest": "six button",
    "beard": "two braids",
    "boots": "knee-high",
    "belt buckle": "enormous",
    "swagger": "comical"
}
```

❌

Data size:	10TB
Writes/sec:	high
Isolation:	row
Patterns: lookup by AND query	
Reads/sec:	high
Recency:	recent

Detected Pirate Record

K/V Persistent DB - Pros and Cons

- **Unlimited random** writes/s
- **Unlimited random** reads/s
- **Unlimited** total data size
- **Unchangeable** data layout
- **Inflexible** access patterns
- **Difficult** to read all the data at once
- **Consistent** performance
- **Row-level** atomicity
- **Capped** reads/s and writes/s for a single partition key
- **Capped** data size for a single partition key

Cost driver is usually writes per second

K/V Persistent DB - Summary

Good for:

Data with random access patterns

Time-ordered range queries

Availability and performance critical functions

Bad for:

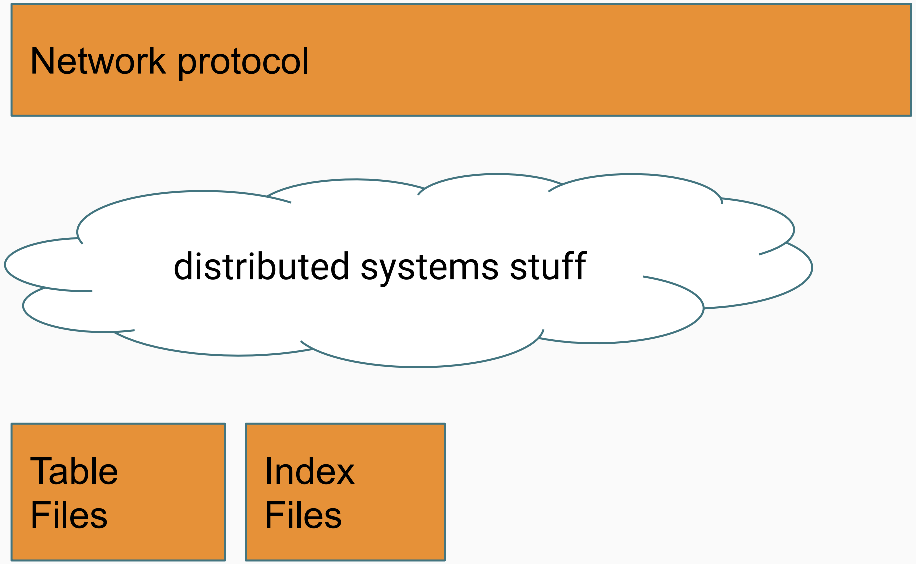
Anything where there is a high write rate for a single key (status updates, org-wide logs)

Blobs (limited record size, cost)

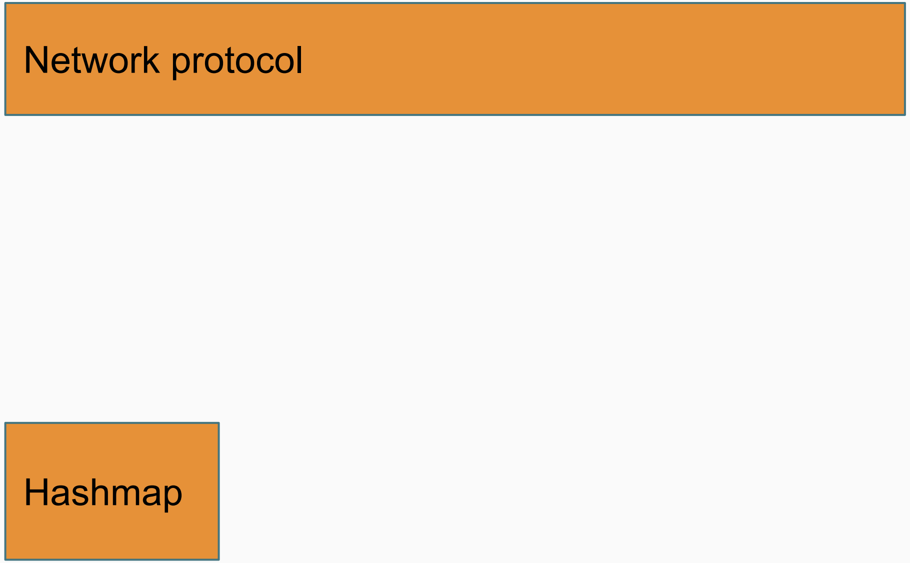
Bulk queries

Key-Value Cache

Rip it all out!

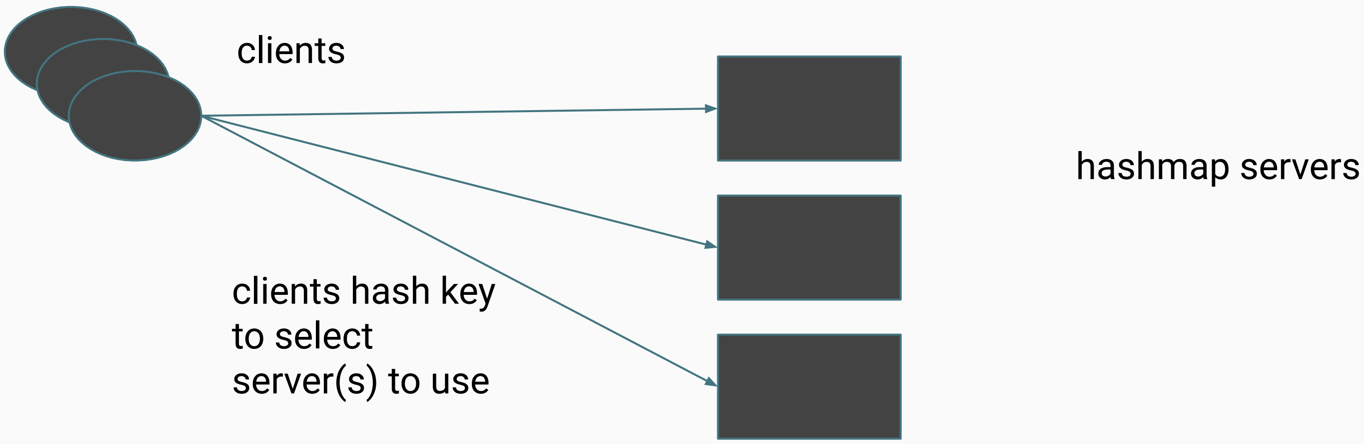


This is memcached



Does it even count as a database?

System architecture



Redis: Sort of Memcached++

- Cluster mode: Centralized partitioning and shard assignment
- Ability to snapshot the memory to disk in the background for backups
- Background server-side replication to other cache servers
- Values have types
- Complex types (such as list, map) and atomic container ops
- Pub/sub channels

```
{
    ~~"cameraId": 256213,~~
    "orgId": 1098,
    "timestamp": 1705348871,
    "maxSoundLevel": ~~"35 dB"~~ 35,
    "ambientLightLux": 77,
    "objectDetections": {
        "ship",
        "island",
        "mermaid"
    },
}
```

✅

Data size: 40TB ⚠️

Writes/sec: high

Isolation: row

Pattern 1: ID and time range

Reads/sec: low

Recency: ~immediate

Pattern 2: last value by org ID

Reads/sec: high

Recency: immediate

Pattern 3: Daily summary

Reads/sec: one / day

K/V Cache - Pros and Cons

- **Unlimited** writes/s
- **Unlimited** reads/s
- **Unlimited** total data size
- **Unchangeable** data layout
- **Inflexible** access patterns
- **Difficult** to read all the data at once
- **Consistent** performance
- **Key-level** atomicity

Cost driver is RAM == storage size

K/V Cache - Summary

Good for:

Data with hot keys

Anything where lowest latency is a must

Atomic complex type operations (redis)

Caches (can have usage or time based eviction)

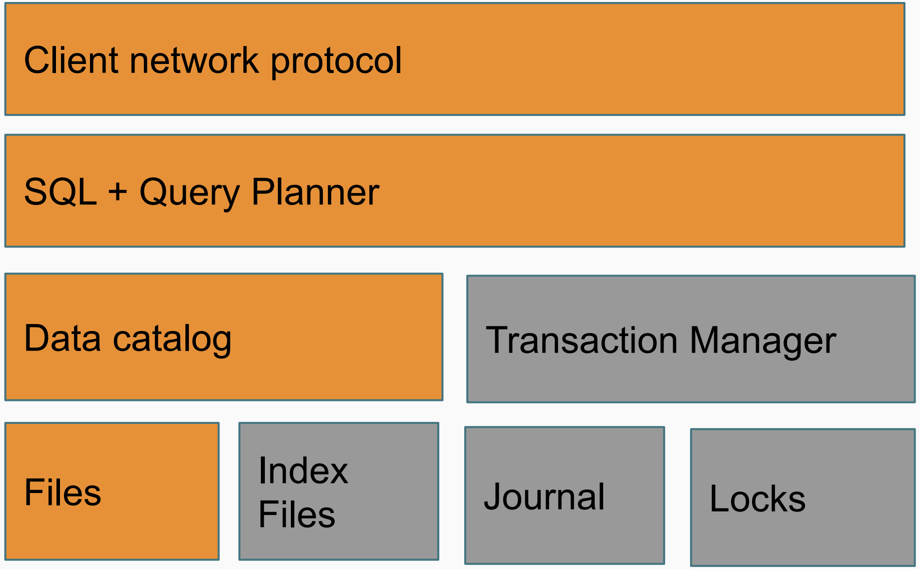
Bad for:

Large datasets ($10/GB /mo!)

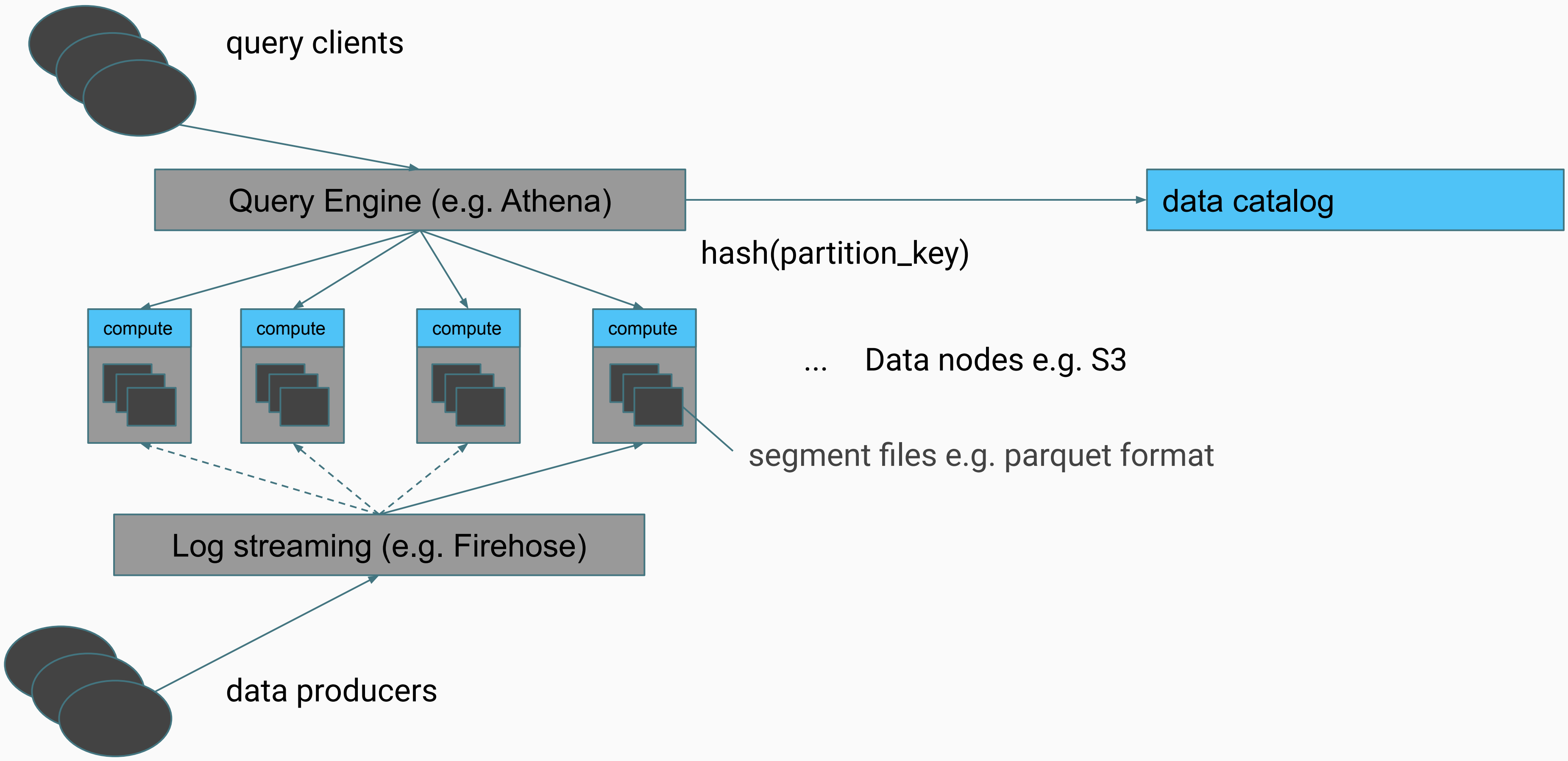
Data that cannot be regenerated (memcached)

Analytics Database

Analytics database



Approximate System Architecture



Typical analytics database Properties

- Heavy focus on efficient storage query processing - pushing compute down to data instead of pulling data to compute
- Data is append-only, added via a log stream, removed in big swaths
- Provides high level SQL interface
- Queries typically take 10 sec - 5 min
- Nearly unlimited size, a single query can scan terabytes in seconds

Columnar data format (Parquet)

{"red":1, "blue": "Hi", "roses": True, "violets": "yippee"}
{"red":3, "blue": "Hi again", "roses": True, "violets": "garrrbage"}
{"red":2, "blue": "Hi", "roses": False, "violets": "some text"}

red: int (delta coding)

1
+2
-1

blue: string (references)

"Hi"
"Hi again"
@0

roses: boolean (bitmask)

0x6

Very high compression at the field level!
Often 80-90%+

Columnar data format (Parquet)

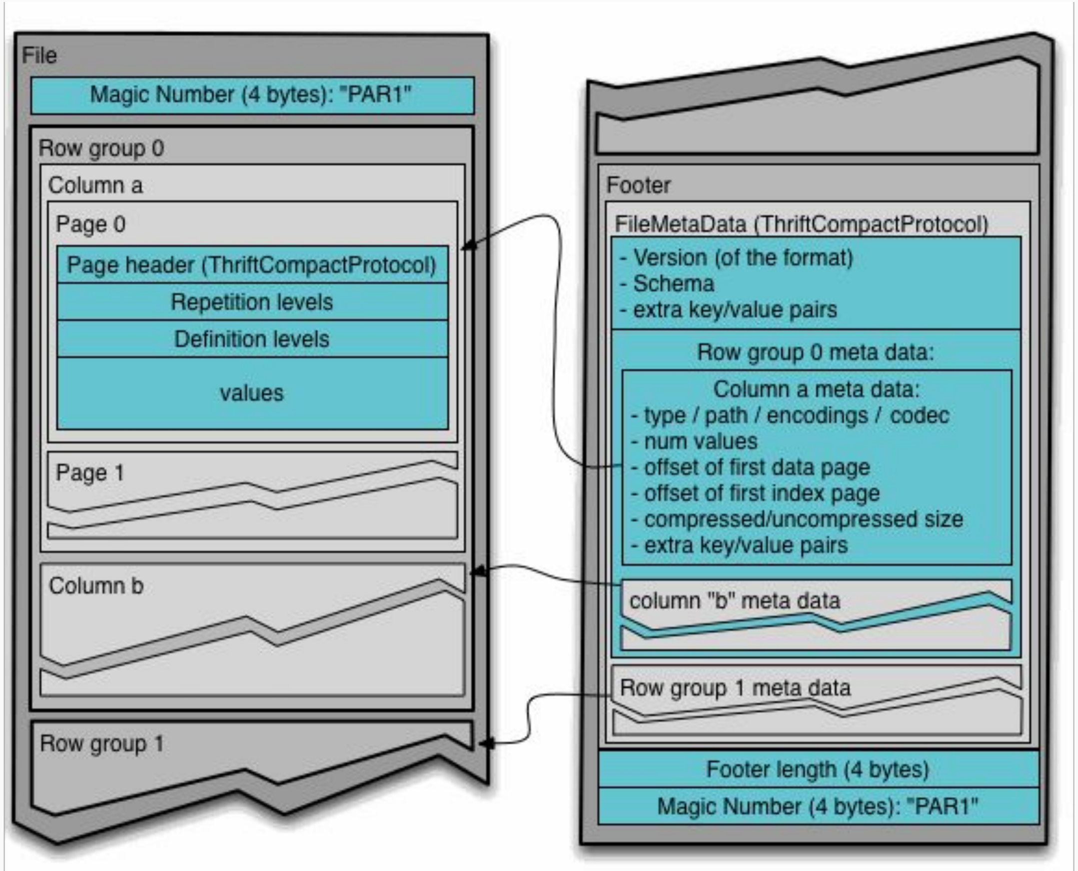


image: https://github.com/apache/parquet-format

Only need to read the columns you care about for a query

Less data stored, even less data read!

Much faster than general purpose compression such as gzip, lzma

Why is the metadata in the footer?

```
{
    "cameraId": ~~256213~~ {uuid},
    "orgId": 1098,
    "timestamp": 1705348871,
    "maxSoundLevel": ~~"35 dB"~~ 35,
    "ambientLightLux": 77,
    "objectDetections": {
        "ship",
        "island",
        "mermaid"
    },
}
```


Data size:	40TB
Writes/sec:	high
Isolation:	row

❌ **Pattern 1: ID and time range**

Reads/sec:	low
Recency:	~immediate

❌ **Pattern 2: last value by org ID**

Reads/sec:	high
Recency:	immediate

✅ **Pattern 3: Daily summary**

Reads/sec:	one / day

Many other flavors and options

- "Data Warehouse"
- Lots of cloud based options
- Spanner (Google)
- high-end commercial DB vendors e.g. MS SQL Server

Analytics DB - Pros and Cons

- **Unlimited** writes/s
- **Unlimited** reads/s
- **Unlimited** total data size
- **Migratable** data layout
- **Flexible** access patterns
- **Easy** to read all the data at once (very efficient with specific time range, column selection)
- **Very Slow** interactive performance; delayed data
- **Append only**

Cost driver is usually quantity of data ingested / scanned

Analytics DB - Summary

Good for:

Data that is used as a source for reports, analytics/graphs, cron jobs and ad-hoc debugging

Data that doesn't need to be queryable by a device or user

High volume, high throughput processing

Logs

Bad for:

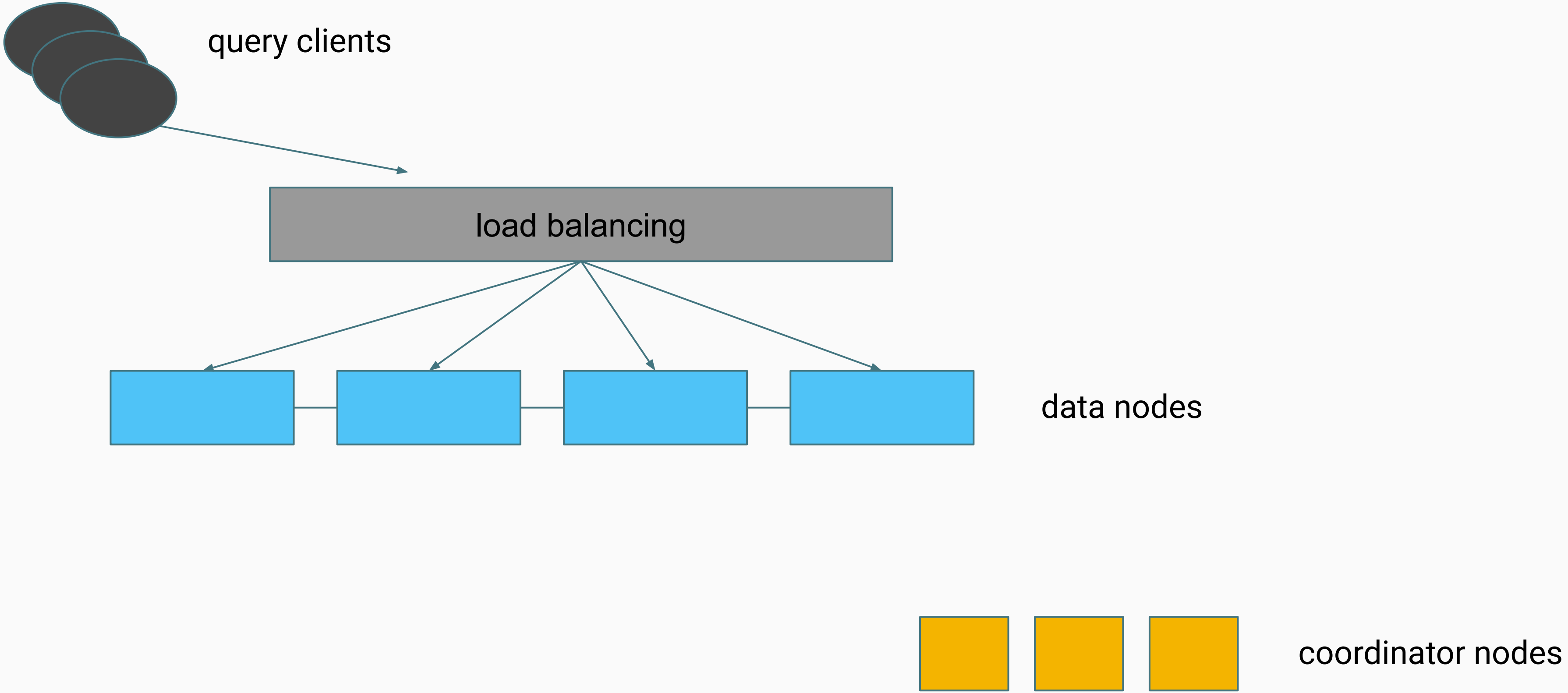
Anything interactive

Repeated queries: scanning is extremely fast but relatively expensive due to sheer size

Data that gets modified / replaced

Text Search Database

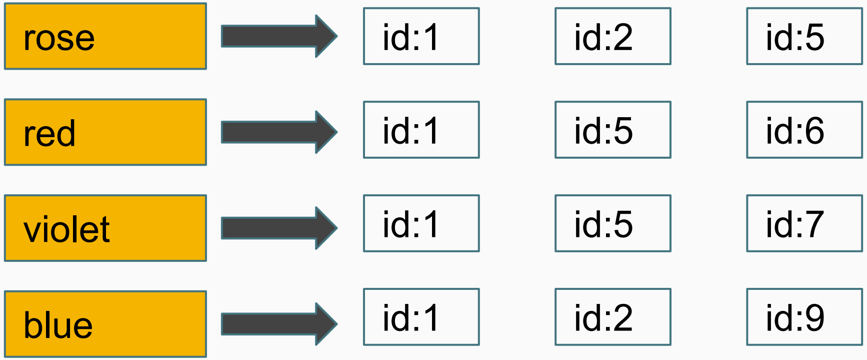
Elasticsearch System Architecture



Typical search database Properties

- Each record is treated as a "document" which is indexed by "terms"
- Text indexes are inverted. The rows are the terms and the columns are the document IDs
- Extremely efficient for boolean AND queries even for relatively non-selective predicates
- High indexing throughput

Inverted Index



"documents" are stored separately

"terms" are extracted during indexing

document id is appended to each term's "posting list"

variable length integer, delta-encoded, sorted document id list
(document ids that contain the term)

Results are obtained by doing a linear n-way merge of the lists in memory at high speed

"rose" and "red" -> [1, 5]
"rose" and "red" and "violet" and "blue" -> [1]

```
{
    "cameraId": "ab815fb6",
    "ts": 1705351366,
    "eyePatch": false,
    "parrot": false,
    "hat": "tri-corner",
    "vest": "six button",
    "beard": "two braids",
    "boots": "knee-high",
    "belt buckle": "enormous",
    "swagger": "comical",
    .......(+ maybe 100s more)
}
```

Data size:	10TB
Writes/sec:	high
Isolation:	row
Patterns: lookup by AND query	
Reads/sec:	high
Recency:	recent

✅

🔎 `swagger:comical vest:"six button" -parrot`

Text search DB - Pros and Cons

- **Champion** at boolean queries that combine low-selectivity clauses
- **Very high** writes/s (many millions easy)
- **Very high** reads/s (many millions easy)
- **Very large** total data size (high compression ratio - 100TB easy)
- **Changeable** index layout
- **Difficult** to read all the data at once
- **Delayed** data visibility
- **Updates** inefficient

Cost driver is usually storage size

Text search DB - Summary

Good for:

Text or metadata where a typical query combines multiple filters (or "keywords") in an AND configuration

Logs that need interactive querying!

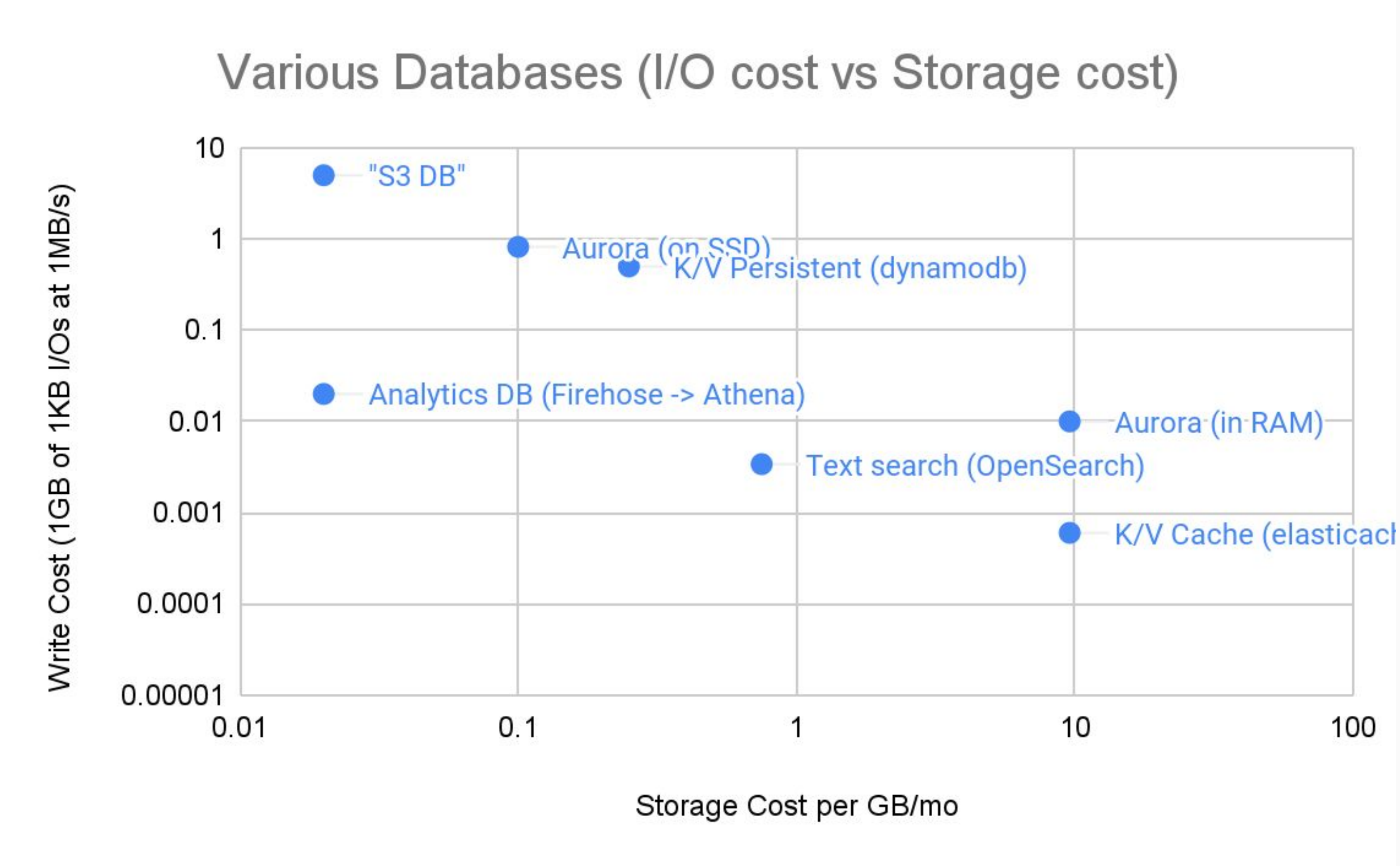
Bad for:

Transactional applications (where writes must be effective immediately)

Large records (expensive storage)

Bulk queries

Interesting trend



Plugging Related Courses

CS 145: Data Management and Data Systems (Aut)

CS 245: Principles of Data-Intensive Systems (Win)

CS 244B: Distributed Systems (Spr)

CS 246: Mining Massive Data Sets (Win)

CS 349D: Cloud Computing Technology (Spr) (Grad students)