SE 464
Week 3

Basics of System Design: Scaling, Databases, Caches

Intro to Scaling

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2001%20-%20Types%20of%20Scaling.pdf

Traditional view of Software **Scalability**

In Data Structures & Algorithms we consider a kind of scalability:

- As **input data size** "n" gets bigger, program should run quickly.
- Complexity analysis lists program runtime as a function of input size.

For example:

- Given a list of size ***n***, mergesort takes O(n log n) time to run.
- Given a hashtable of size ***n***, finding a value takes O(1) time.

- This assumes one problem to solve, one computer, and all operations having the same cost.

Services vs Programs

- A **service** is different than a simple program because it listens for requests from clients/users, and may handle multiple requests concurrently.
- External user provides an input (request) and service outputs a response.
- Requests are usually delivered as messages that arrive over a network.
- The service runs constantly, waiting for requests that it should process.
  - Thus, you can't just run the code on your laptop. You need a machine that is always powered-on (probably located in a data center or server room).

For example:

- a website, like: https://www.ebay.com/sch/i.html?_nkw=guitar

Defining *Service Scalability*

- Roughly speaking, a service is scalable if it can easily handle growth in the number of concurrent users/requests.

Scalability metrics are measures of **work throughput**:

- Requests/queries per second
- Concurrent users
- Monthly-active users

- So far, we don't care about the **costs** to achieve this scale (time per request or number of machines required), just the scale achieved.

Scaling Challenges

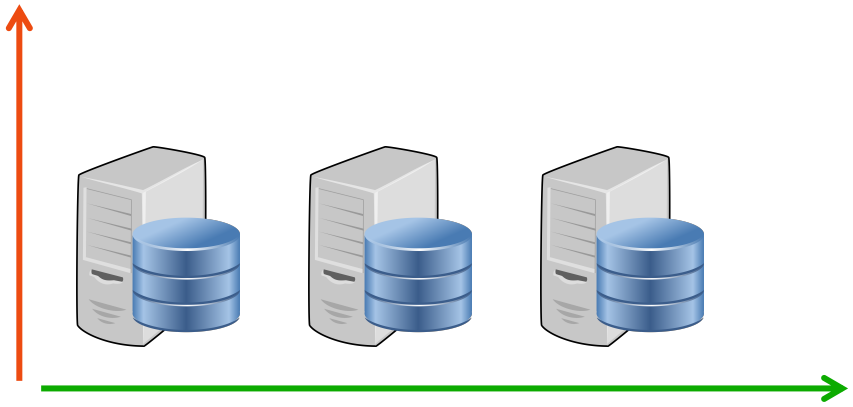
- Why is it difficult to make services big, even if *money ain't a thang?*

- Programs run on one machine, which has limited speed.
- **Coordinating** multiple machine can be difficult (who does what?)
- **Sharing data** among multiple machines is difficult (where is the data? how do we manage competing requests to change the same data?).
- More machines means there is high probability one will **fail** (die).
- Users can be distributed worldwide (communication **latency** is high).
- Service components must trust each other but ignore interference from attackers (**authentication**).
- Software **updates** must be deployed without downtime.

Vertical Scaling

- Let's assume that you're starting with a very light load and you can handle all the requests on one machine. Suddenly demand increases!
- The easiest approach to scaling is to just buy a faster machine to run your service.

Vertical scaling makes your machine(s) bigger and stronger.
Think "taller."



Horizontal scaling adds more machines.
Think of them standing side-by-side.

COMP_ENG 101: What affects computer performance?

Primarily:

- Number of CPU cores.
- Speed of each CPU core.
- (Lack of) competing processes running on the same machine.

Perhaps also:

- Amount of memory (RAM).
- Type of disk (SSD vs magnetic).
- Number of disks (parallel access).
- Type of network connectivity.
- Presence of GPUs, TPUs, and other special-purpose accelerators.

https://aws.amazon.com/ec2/instance-types/

Parallelism within a machine

- At any given time, you probably have about 100 processes (programs) "running" on your laptop (which probably has about 4 CPU cores).
- The OS kernel schedules processes, so they take turns using CPU.
- Often, processes **block** (wait) while doing input/output (IO).
  - For example, reading a file from disk, or waiting for a message to arrive from the network.
- While a process is blocked, another process can take over the CPU.
- A single process can have multiple **threads** which execute concurrently while sharing the same memory.
  - This is called Shared Memory Parallelism.

Apache web server example

```
top - 20:20:10 up 697 days, 17:04,  1 user,  load average: 0.00, 0.01, 0.00
Tasks:  91 total,   1 running,  90 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.3%us,  0.3%sy,  0.0%ni, 96.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.3%st
Mem:   1017368k total,   942832k used,    74536k free,     4988k buffers
Swap:  1048572k total,   308580k used,   739992k free,    40260k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7750 root      20   0  117m 6656 5556 S  0.0  0.7   0:00.00 sshd
 7727 root      20   0 79984 5800 4964 S  0.0  0.6   0:00.00 sshd
13729 tomcat    20   0 2666m 677m 4516 S  1.7 68.2 825:59.07 java
 7753 ec2-user  20   0  112m 3412 3008 S  0.0  0.3   0:00.00 bash
 7752 ec2-user  20   0  117m 3872 2772 S  0.0  0.4   0:00.00 sshd
 3879 root      20   0  526m  14m 2724 S  0.0  1.4   2383:27 cfn-hup
 7774 ec2-user  20   0 15308 2188 1904 R  0.0  0.2   0:00.18 top
17933 apache    20   0  823m 7888 1304 S  0.0  0.8  40:55.55 httpd.worker
17936 apache    20   0  823m 7496 1248 S  0.0  0.7  40:42.84 httpd.worker
17938 apache    20   0  823m 7712 1212 S  0.7  0.8  41:02.22 httpd.worker
 2546 root      20   0 89028 1224 1048 S  0.0  0.1  15:50.17 sendmail
17937 apache    20   0  823m 7616 1044 S  0.0  0.7  40:35.37 httpd.worker
17940 apache    20   0  823m 7168 1024 S  0.3  0.7  40:46.54 httpd.worker
17932 apache    20   0  823m 7764 1020 S  0.0  0.8  40:41.50 httpd.worker
17934 apache    20   0  823m 7444  896 S  0.3  0.7  41:05.20 httpd.worker
17939 apache    20   0  823m 7100  884 S  0.0  0.7  40:31.48 httpd.worker
17941 apache    20   0  823m 7308  836 S  0.0  0.7  40:32.70 httpd.worker
 3287 ntp       20   0 29288  944  792 S  0.0  0.1   0:56.44 ntpd
 2116 root      20   0  9356  872  728 S  0.0  0.1   0:16.25 dhclient
 2866 root      20   0  118m  824  728 S  0.0  0.1   4:59.63 crond
 2516 root      20   0 79984  700  592 S  0.0  0.1   3:43.41 sshd
 2555 smmsp     20   0 80492  640  500 S  0.0  0.1   0:05.54 sendmail
17935 apache    20   0  823m 6324  408 S  0.0  0.6  40:43.62 httpd.worker
17929 root      20   0 94836  588  192 S  0.0  0.1   1:24.51 httpd.worker
    1 root      20   0 19616  336  156 S  0.0  0.0   0:37.12 init
```

- At left is the output from the "top" command, showing process status on Linux.
- This is a t2.micro virtual machine with only one CPU core.
- It's running a webserver with at least 11 separate processes (httpd.worker).
- While one process is blocked (meaning *busy*, eg., waiting to read data from an HTML file) another process can handle a different user's request.

Cloud Computing makes scaling easier

- **Vertical scaling:** change the **instance type** of a virtual machine. Eg., upgrade from:
  - **t3.nano** (<2 cores, 0.5GB RAM, remote SSD disk) $.0052/hour …to…
  - **m5d.24xlarge** (96 cores, 386GB RAM, local NVMe SSD disk) $5.424/hour
- Vertical scaling (up or down) just requires a reboot of the VM.

- **Horizontal scaling:** purchase more VM instances.
  - The new instance will be available to use in just a few minutes.

- We call cloud computing resources "**elastic**" because you can quickly change the size and quantity of the computing resources you are using.

Vertical Scaling pros and cons

- ✓ Easy to write your programs.
- ✓ Most languages have support for multithreading.
- ✓ Most "off the shelf" software (commercial or open source) is written to run on one machine.
  Eg.: MySQL, Oracle DB, Apache, Nginx, Node.js, etc
- ✓ Modern servers can do a lot of work in parallel with ~96 cores.
- ✓ Can connects hundreds of disks to a machine before overwhelming I/O bandwidth.
- ✓ Avoids slow communication with outside machines.

- ✗ Cannot handle really huge loads.
- ✗ Cannot be scaled quickly in a fine-grained manner.
  Ie., must replace entire machine instead of just adding one more node.
- ✗ Single point of failure.
- ✗ Price/performance ratio is poor for top-of-the-line machines.
  - ✗ Motherboards with many sockets are expensive.
  - ✗ Fastest CPUs are expensive.

- **Vertical scaling is not scalable!**

Horizontal Scaling is needed for global apps

- Public Cloud Computing providers can give you lots of machines, but making good use of them is very difficult.
- Most of this class will address the coordination of execution and data in horizontally-scaled systems.

Recap

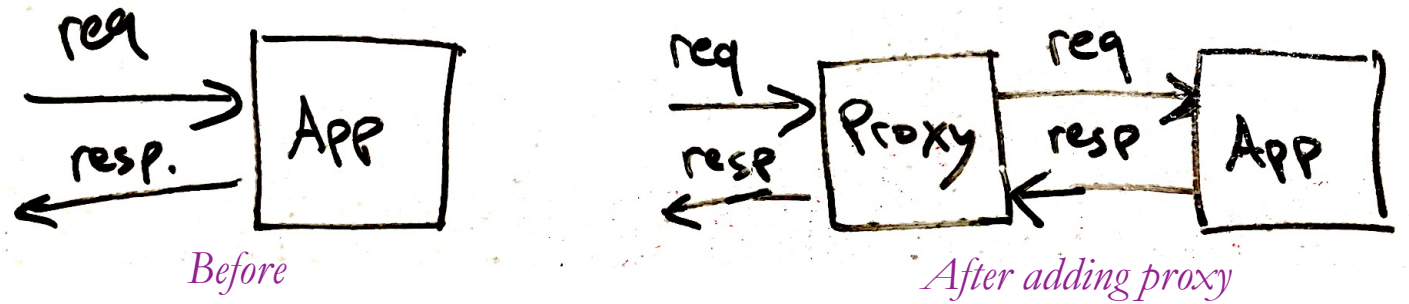
- A software **service** is a program that runs continuously, giving responses to requests.
- **Scalability** is the ability of a service to grow to handle many concurrent users (ideally an arbitrarily large number).
- Two approaches to scaling that are useful in different scenarios:
- **Vertical scaling** is upgrading your machine(s).
  - The simplest and most efficient way of scaling… but there is a ceiling.
- **Horizontal scaling** is adding more machines.
  - Coordinating a cluster of machines is complicated, but it's necessary for global scale and massive throughput.

Caching

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2004%20-%20Proxies%20and%20Caches.pdf

Proxies

- A proxy server is an **intermediary router** for requests.
- The proxy does not know how to answer requests, but it knows who to ask.
- The request is relayed to another server and the response relayed back.
- Proxies can be transparently added to **any stateless service**, like HTTP:



Before *After adding proxy*

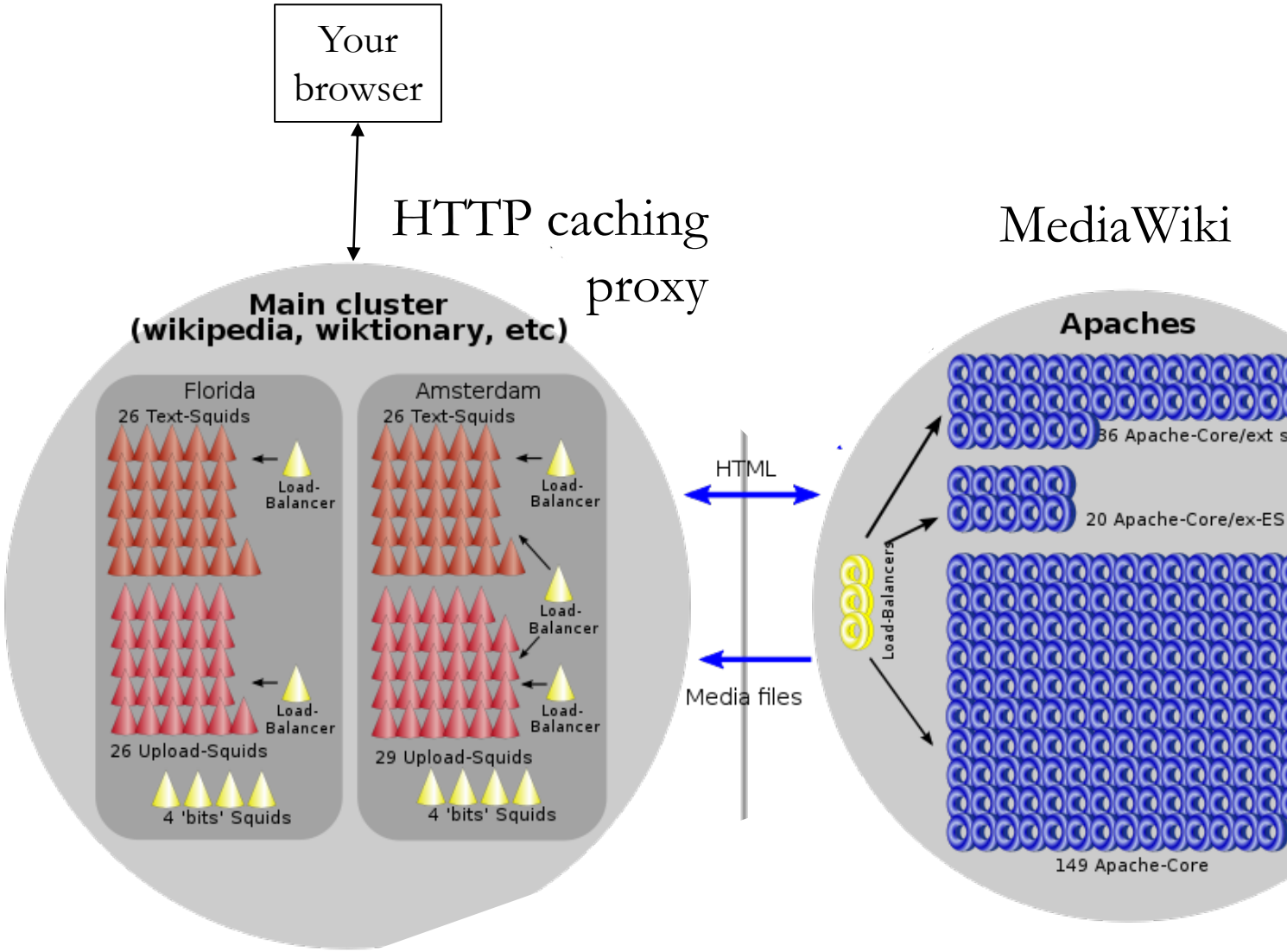
- A **load balancer** is a type of proxy that connects to many app servers.
  - The work done by the load balancer is very simple, so it can handle much more load than an application server.
  - Creates a single point of contact for a large cluster of app servers.

Front-end Cache

- Squid is a **caching** proxy.

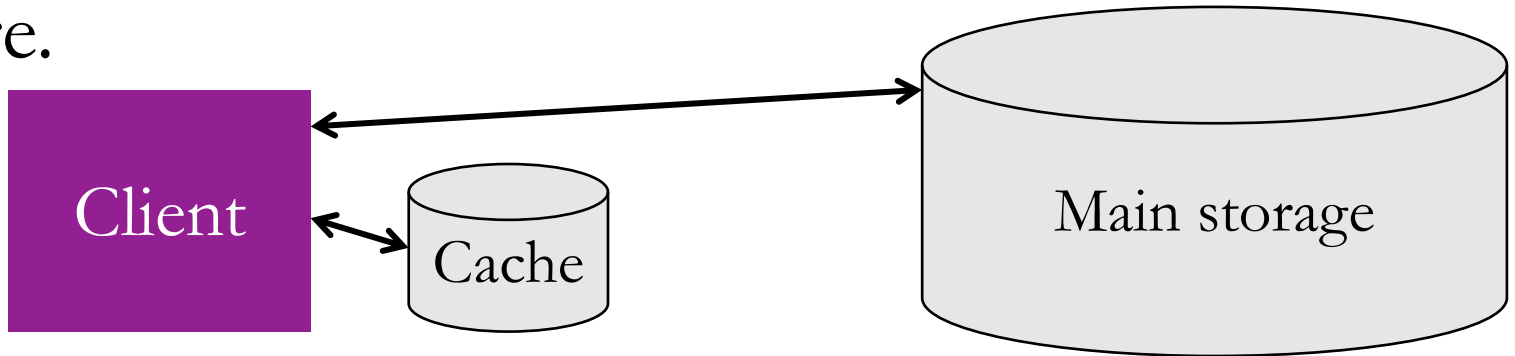
(A cache stores recently retrieved items for reuse)

- Frequent requests are found in (*hit*) the cache, without re-asking MediaWiki and accessing the shared database.
- Unusual requests are not in (*miss*) the cache, and are relayed to MediaWiki.



Cache basics

- Caching is a general concept that applies to web browsers, computer memory, filesystems, databases, etc.
  - any time you wish to improve performance of data access.
- A **cache** is a small data storage structure designed to improve performance when accessing a large data store.
  - For now, think of our data set as a dictionary or map (storing key-value pairs).
- The cache stores the **most recently** or **most frequently** accessed data.
- Because it's small, the cache can be accessed more quickly than the main data store.



Cache **hits** and **misses**

- The cache is small, so it cannot contain every item in the data set!

When reading data:

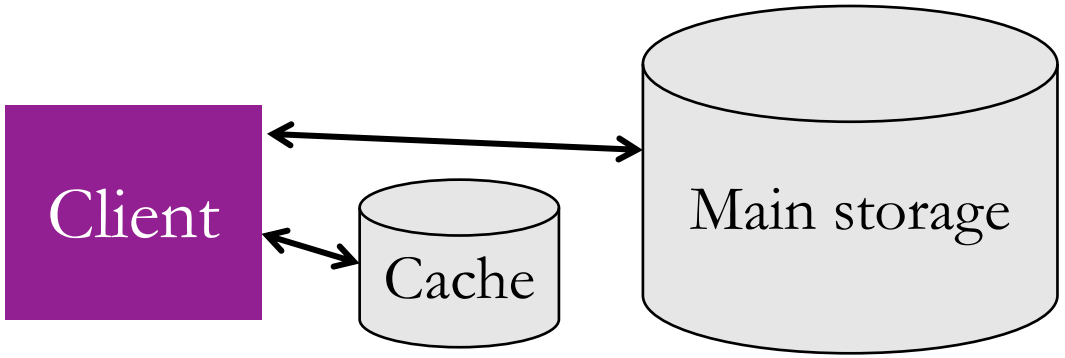
1. Check cache first, hopefully the data will be there (called a cache **hit**).
   - Record in the cache that this entry was accessed at this time.
2. If the data was not in the cache, it's a cache **miss.**
   - Get the data from the main data store.
   - Make room in the cache by **evicting** another data element.
   - Store the data in the cache and repeat Step 1.

- The most common eviction policy is LRU: least recently used

Types of Caches

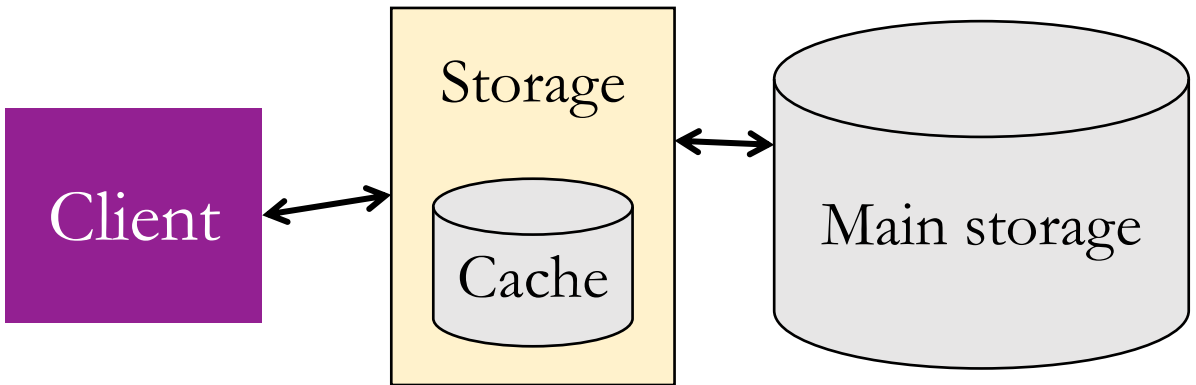
Managed Cache

- Client has direct access to both the small and large data store.
  - Client is responsible for implementing the caching logic.
- Eg.: Redis, Memcached



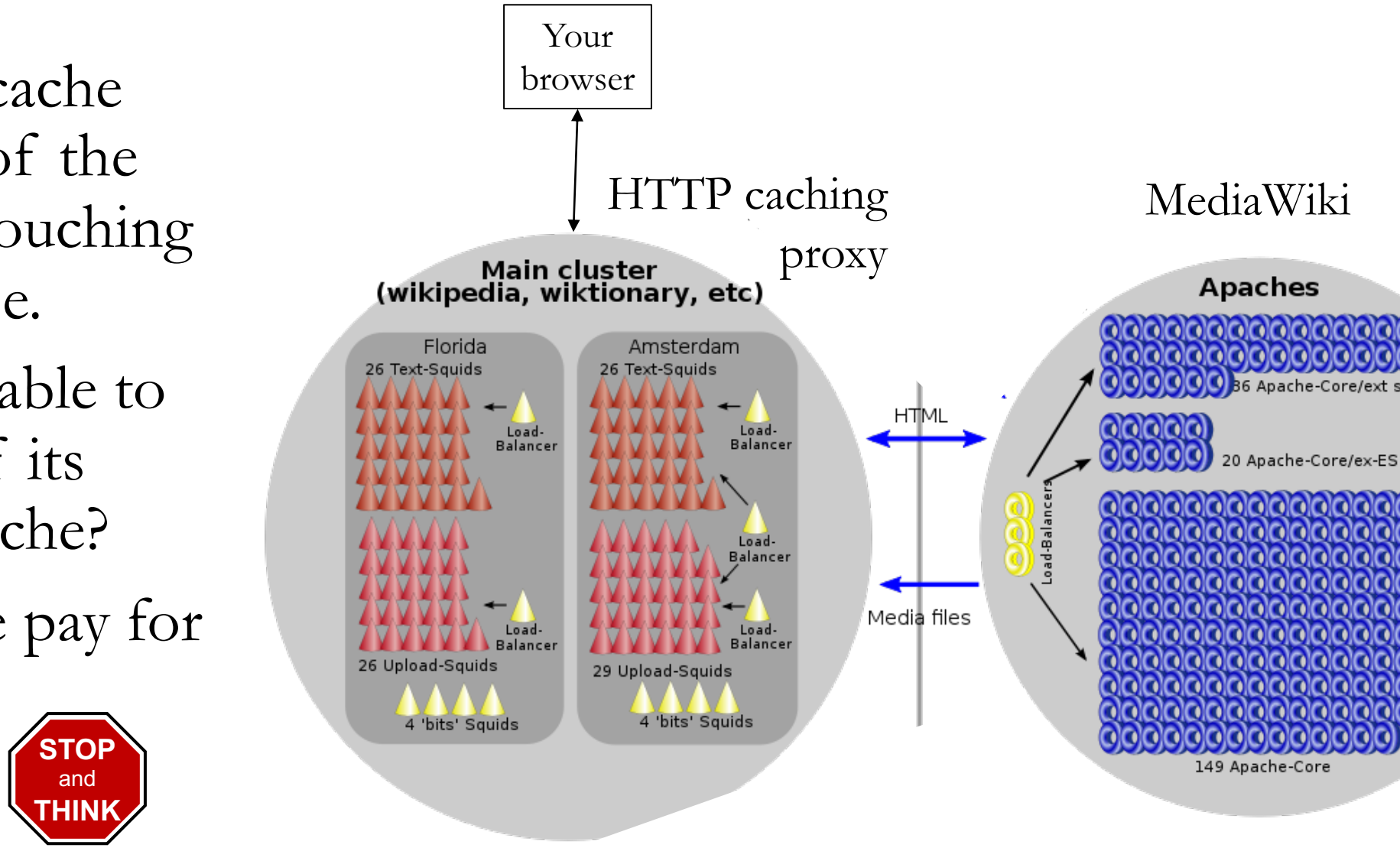
Transparent Cache

- Client connects to one data store.
- Caching is implemented inside storage "black box."
- Eg.:
  - Squid caching proxy, CDNs
  - Database server.

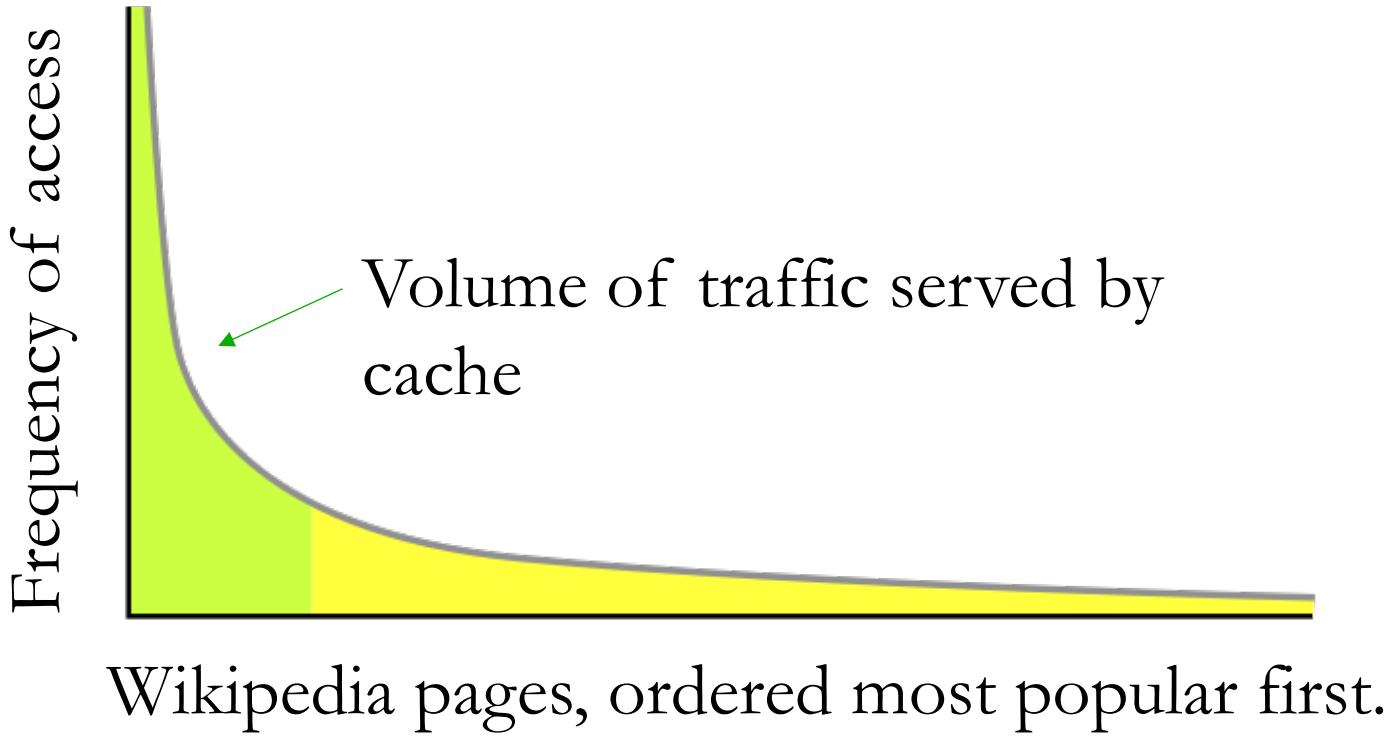


Stop and think

- A small frontend cache might serve 90% of the requests without touching the shared database.
- Why is Wikipedia able to handle so many of its requests from a cache?
- What prices do we pay for this efficiency?



"Long tail" of Wikipedia



- A small fraction of Wikipedia pages account for a large fraction of traffic.
  - Optimize performance for these pages.
  - These will naturally be stored in the frontend cache.
- The "long tail" is the large number of rarely-accessed pages.
  - Most accesses to these rare pages involve a database access

Data **writes** cause cache to be out of date!

- Remember that we can have many clients, each with its own cache.
- When data changes, out-of-date copies of data may be cached and returned to clients. Eg., a Wiki article is edited. What to do?

Three basic solutions:

- **Expire** cache entries after a certain TTL (time to live)
- After writes, send new data or an invalidation message to all caches. This creates a **coherent cache**. But it adds performance overhead.
- Don't every change your data! For example, create a new filename every time you add new data. This is called **versioned data**.

HTTP support caching well

- HTTP is **stateless**, so the same response can be saved and reused for repeats of the same request.
- HTTP has different **methods** GET/PUT/POST/DELETE.
  - GET requests can be cached, others may not because they modify data.
- HTTP has **Cache-Control headers** for both client and server to enable/disable caching and control expiration time.

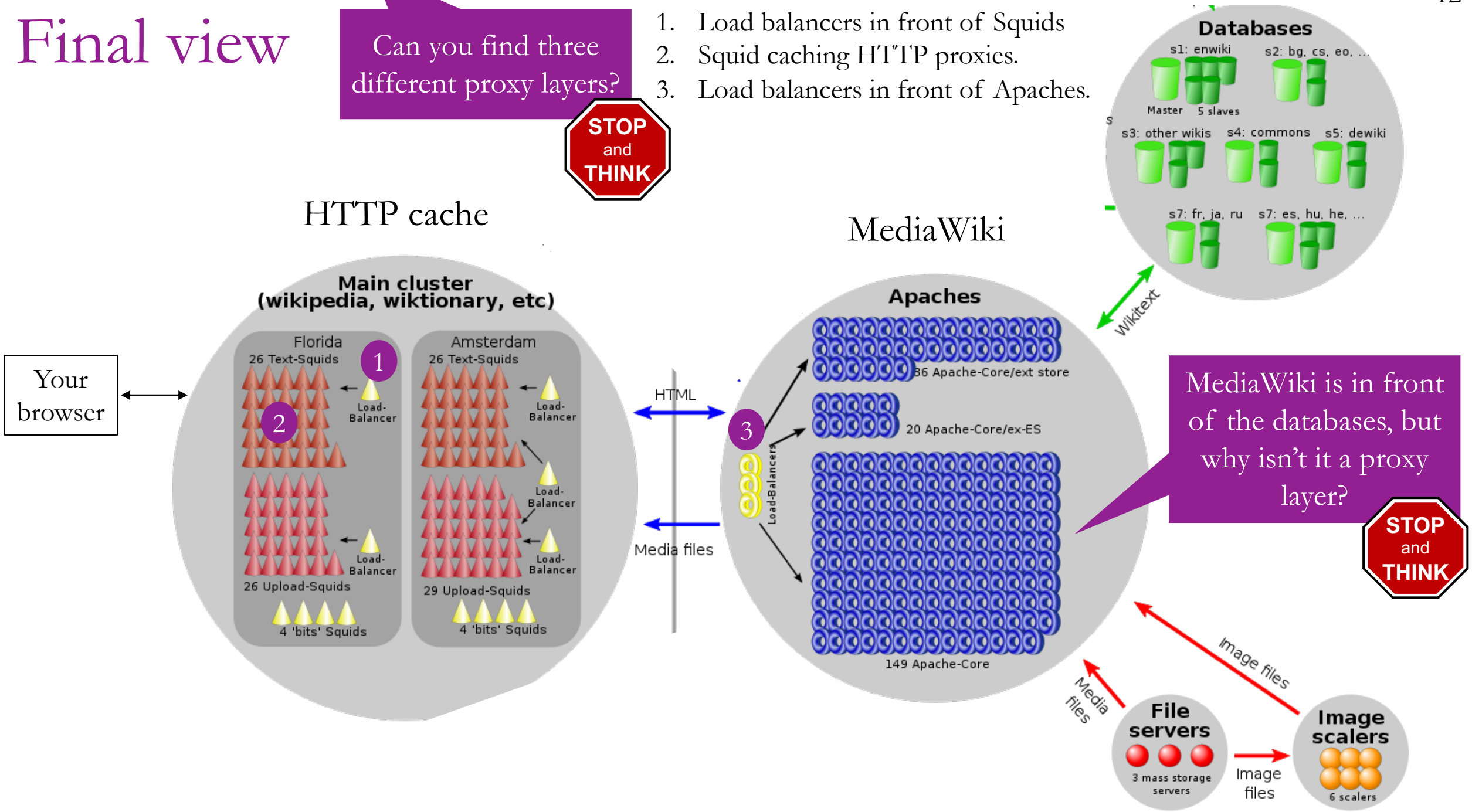
- These features allow a web browser to skip repeated requests.
- Also, an HTTP caching proxy, like Squid, is compatible with any web server and can be *transparently* added.

Final view

Can you find three different proxy layers?

STOP and THINK

1. Load balancers in front of Squids
2. Squid caching HTTP proxies.
3. Load balancers in front of Apaches.



Review

- Introduced **proxies** and **caching**.
- A proxy is an intermediary for handling requests.
  - Useful both for **caching** and **load balancing** (discussed later).
- Often, many of a service's requests are for a few popular documents.
  - Caching allows responses to be saved and repeated for duplicate requests.
- HTTP has built-in support for caching.

Scaling Databases - SQL vs No SQL

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.

https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2013%20-%20Distributed%20NoSQL%20Databases.pdf

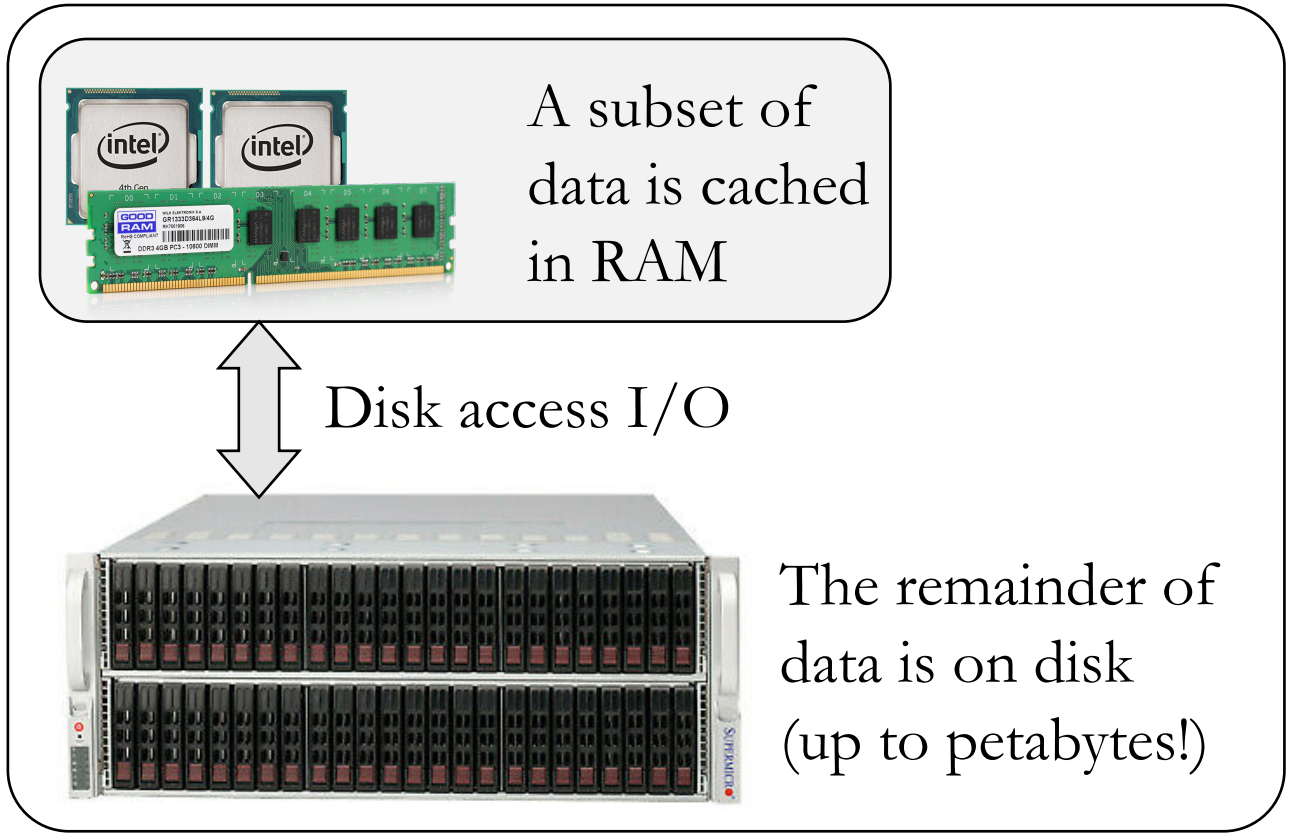
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2009%20-%20SQL%20Database%20Scaling.pdf

Recap: Storage and Relational Databases

- **Persistent** storage requires special consideration due to slow performance and lack of language-level support.
  - **RAID** combines multiple disks for better capacity, storage, and fault tolerance.
- Databases solve lots of problems:
  - **scalability, persistence, indexing, concurrency,** etc.
  - Filesystems can solve some, but not all, of these problems.
- **Relational (SQL) databases** store data in tables.
- Developer defines the DB **schema** first (tables, columns, keys).
  - Rows are added during DB operation, and they must fit the schema.
- **Indexes** let us find rows quickly with value of one or more column.
- SQL query language lets us run analysis code "close to" data storage (filtering, aggregation – sum, count, min, max, avg, etc.).

Memory vs disk access in databases

- Remember that computers have a hierarchy of storage.
- RAM is 100,000x faster but ~100x smaller than disk.
- Database servers operate much faster when accessing data that is cached in RAM (memory).
  - RAM can be up to ~1TB.



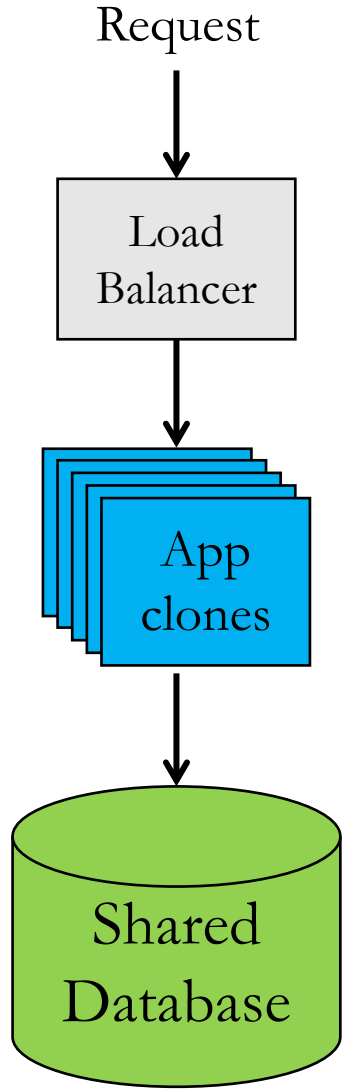
- **Goal**: fit entire **active data set** in RAM.
- Database/OS automatically cache most frequent data in RAM.

Databases are performance bottlenecks

- Why is load balancer not the bottleneck in this design?

  - Load balancer does much less work per request than the database.
- Why not create clones of the database?

  - Traditional scalable service design relies on a single shared database for **coordination**. App clones share state through the database.
  - However, we'll learn some tricks in this lecture.



Relational Database performance optimizations

- **Query planners** optimize order of table access and use of indexes:
  - SELECT * FROM user NATURAL JOIN post
    WHERE post.date > "2010-01-01" AND user.birth_year < 1920;

- RAM is used to store the most important data and indexes.
- Responses can be cached and replayed if data has not changed.

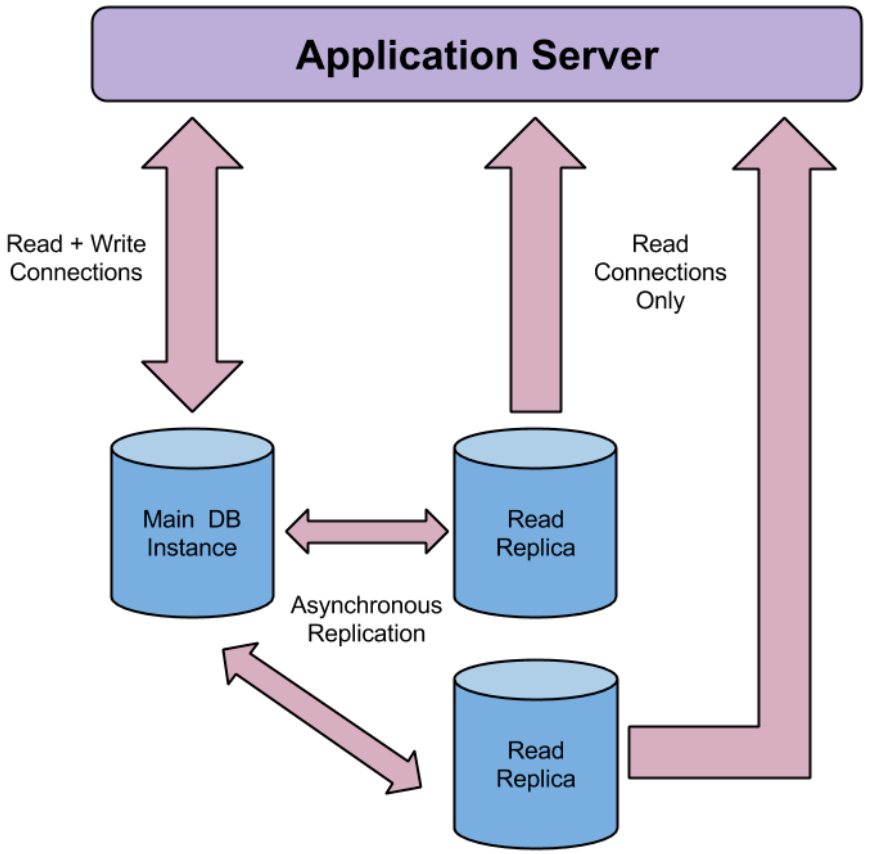
To avoid a database bottleneck:

- Avoid unnecessary queries (cache data in the frontend).
- Buy a really fast machine, with plenty of RAM for caching.
- Use the fastest possible disks (SSDs, RAID).

} *Vertical Scaling*

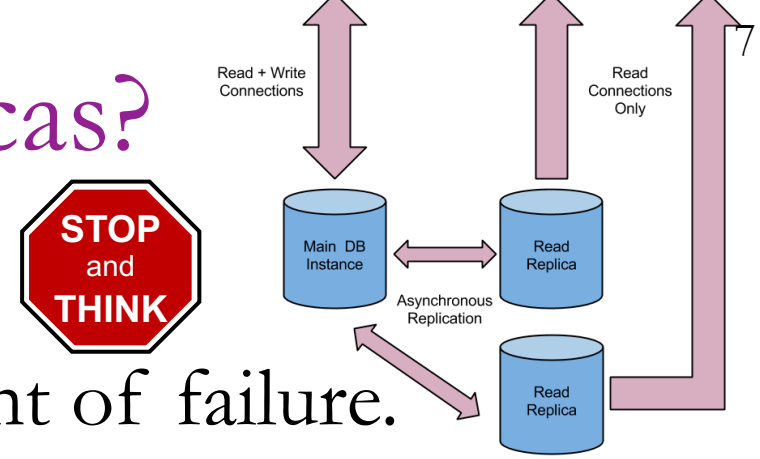
- Use **read replicas** or **sharding** – *Horizontal Scaling*

Read replicas



- Often, > 95% of DB traffic is **reads**.
- **Replica** servers each have a **full copy** of all the data, and they can handle read requests (SELECT).
- All writes (UPDATE, DELETE) must go to the **Primary** server (a.k.a. Main, Master)
- Data changes are pushed to read replicas.
- However, replicas may be slightly behind the primary, so read requests that are sensitive to consistency should use the primary.
- Too many replicas would make the data push process a bottleneck in the primary.

What limits the number of read replicas?



- This design is not infinitely scalable.
- The Primary is a central bottleneck and single point of failure.
- If there are N replicas, Primary must send N copies of each write.
- If there are R times as many reads as writes, and we want to equalize load on Primary and Replicas (to the max machine capacity), we get:

$$\text{primary\_load} = \text{repl\_load}$$

$$\text{primary\_reads} + \text{primary\_writes} + \text{data\_xfer} = \text{repl\_reads} + \text{repl\_writes} + \text{data\_xfer}$$

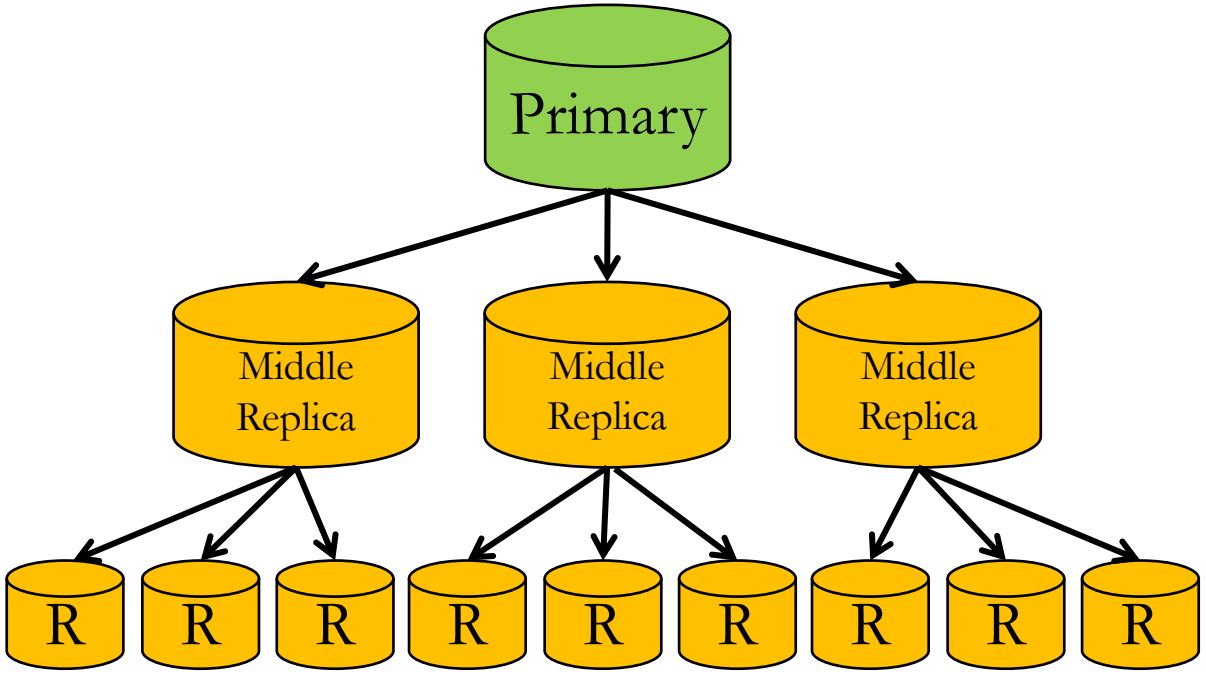
$$0 + 1 + N = R + 0 + 1$$

$$N = R$$

- Here, the optimal number of replicas is directly proportional to the ratio of reads to writes, perhaps about ten in a typical application.

Ideas for greater scaling of reads?

Multi-level replication can extend read-scalability



This is a kind of **horizontal scaling** for database reads.

Where do read requests go?

- To the bottom level replicas. (nine are shown in this diagram)

Why not read from middle replicas?

- Like the primary, they are busy pushing writes to their many children.

Where do write requests go?

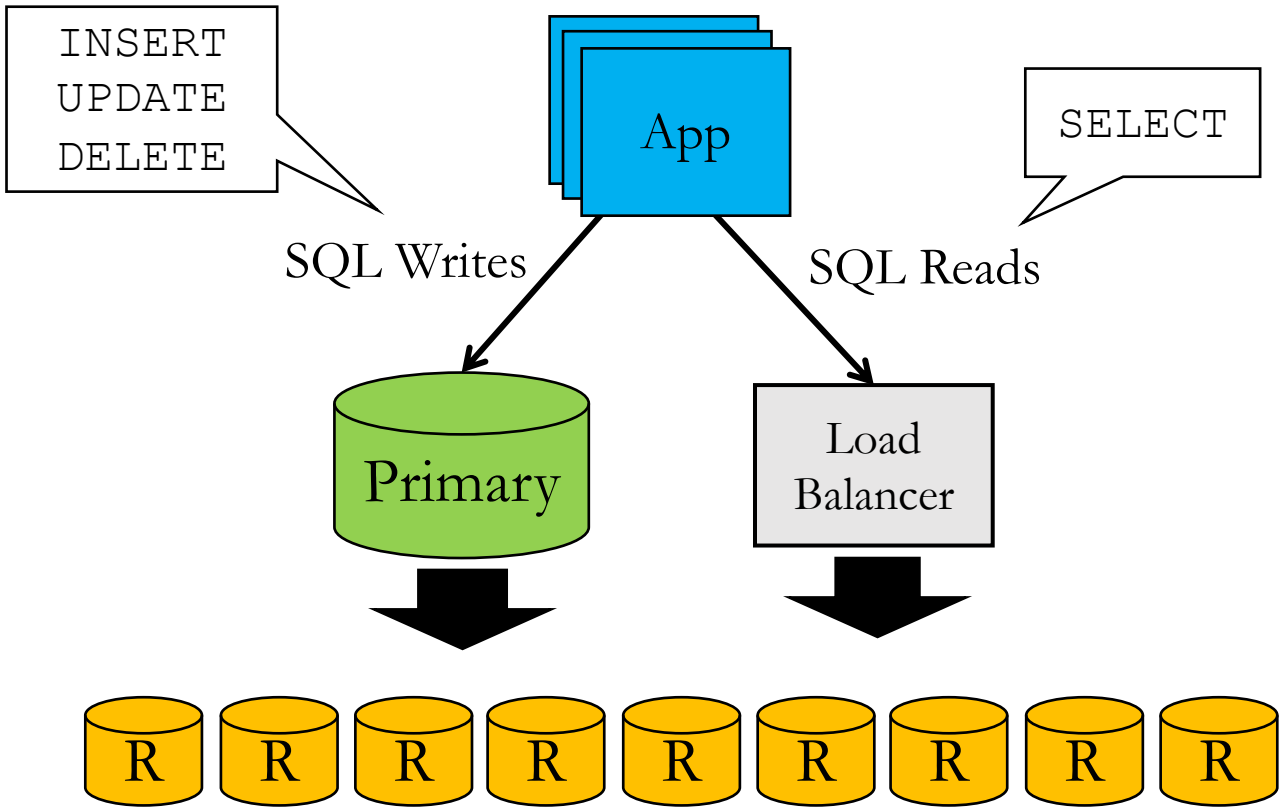
- To the one primary.

Can we add more replication levels (to achieve arbitrary *width*)?

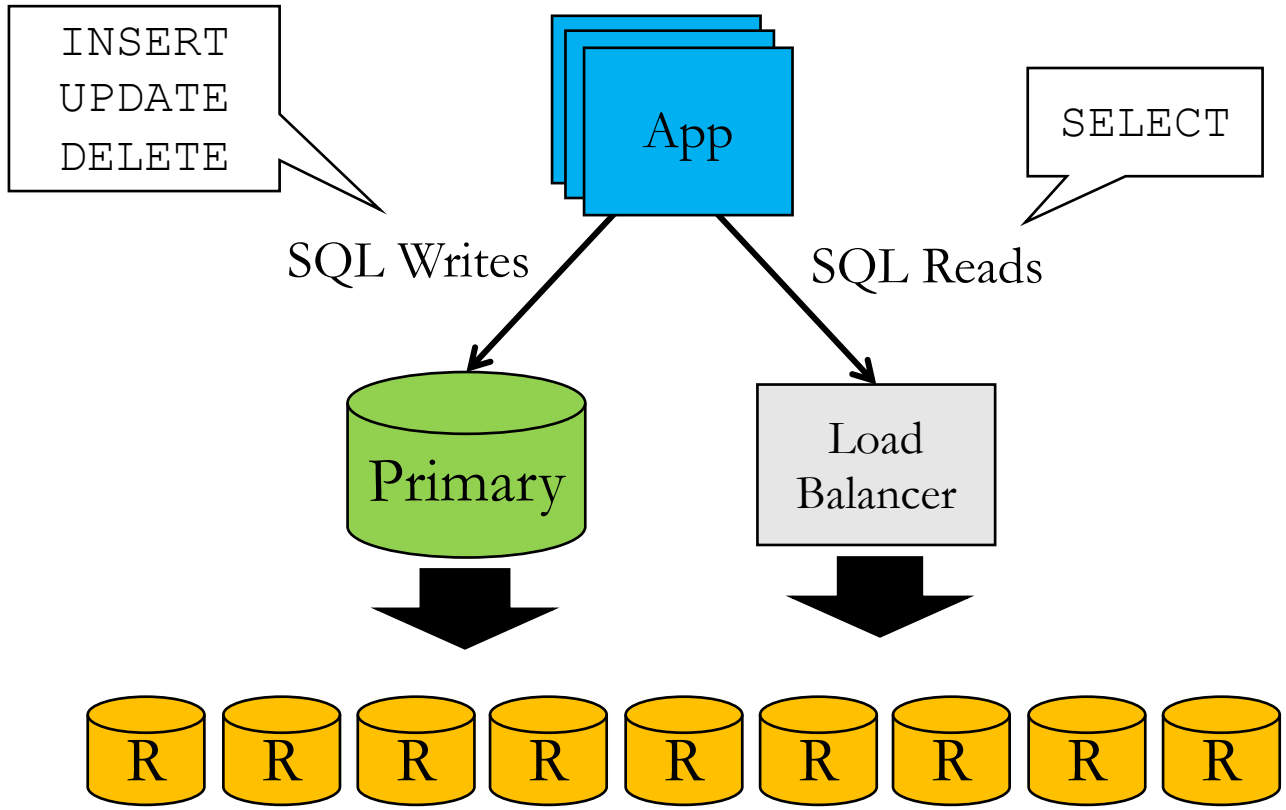
- Yes, but each level adds more **delay** between write at primary and data availability at read replicas.

How to use read-replicas?

- Put a load balancer in front of all the read replicas.
- This can be a NAT-type local LB or a simple software library. (eg.)



Replication shortcomings?

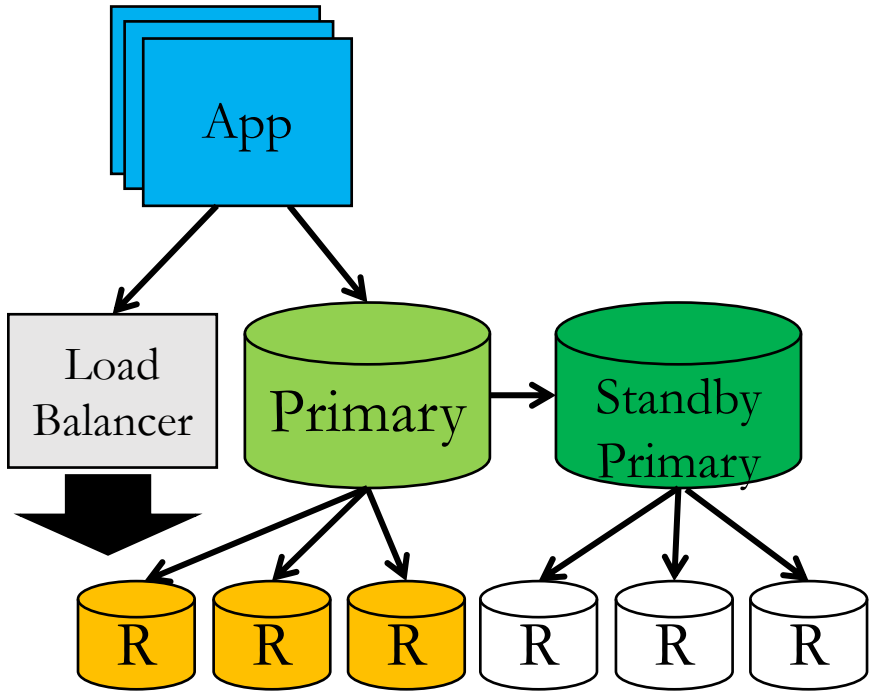


- **Writes** are not scalable. They are all handled by one DB machine.
- **Capacity** is not scalable. All the data must fit on each DB machine.
- Primary is a **single point of failure**.

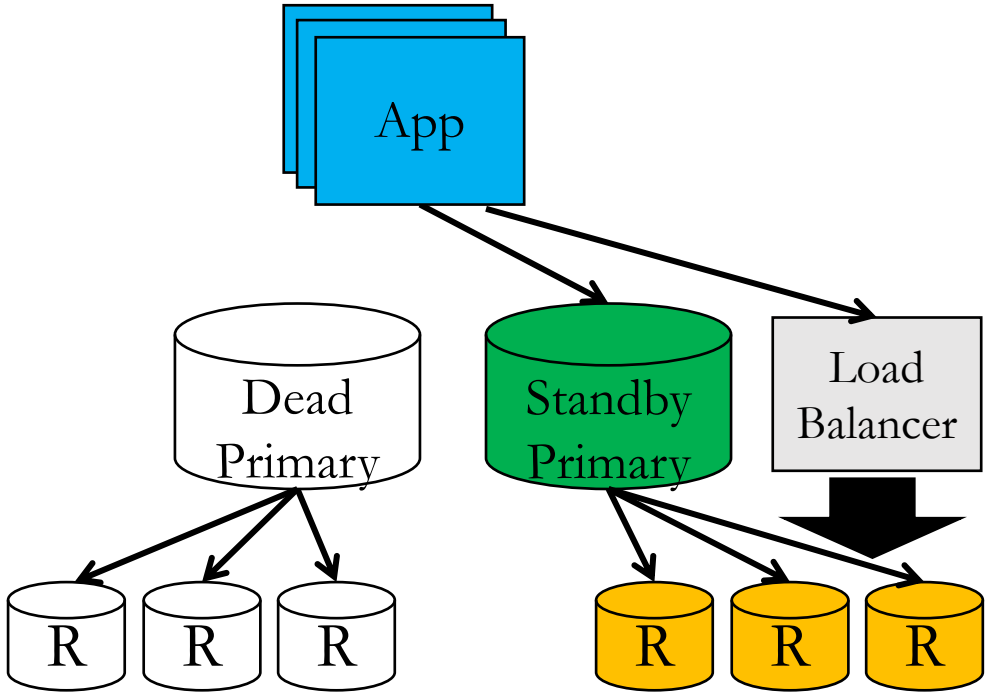
Primary-primary failover for robustness

- Keep a "standby" primary ready to take over if the main primary fails.
- App will switch over to Standby if the main primary stops responding.

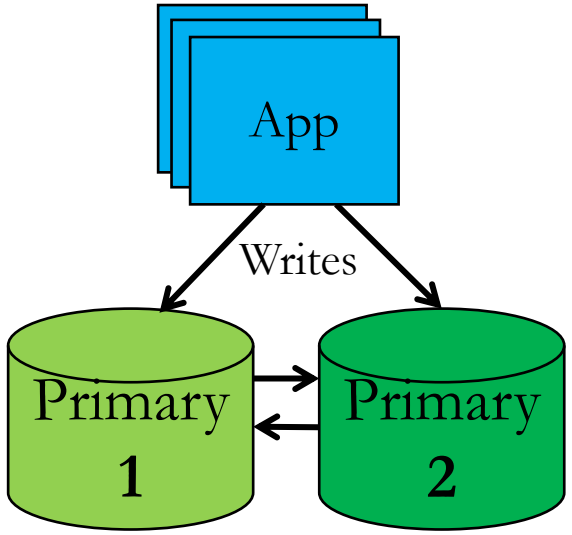
Normal



After Failure



Why not allow writes to multiple primarys?



- Each Primary still must handle all the writes, though indirectly.
- Thus, the same performance bottleneck remains.

- Also, data can become **inconsistent** if operations happen concurrently.

How to scale **writes** and storage **capacity**?

- We already tried vertical scaling.
- How to implement **horizontal** scaling of a writes and capacity?

Some kind of **partitioning** is needed:

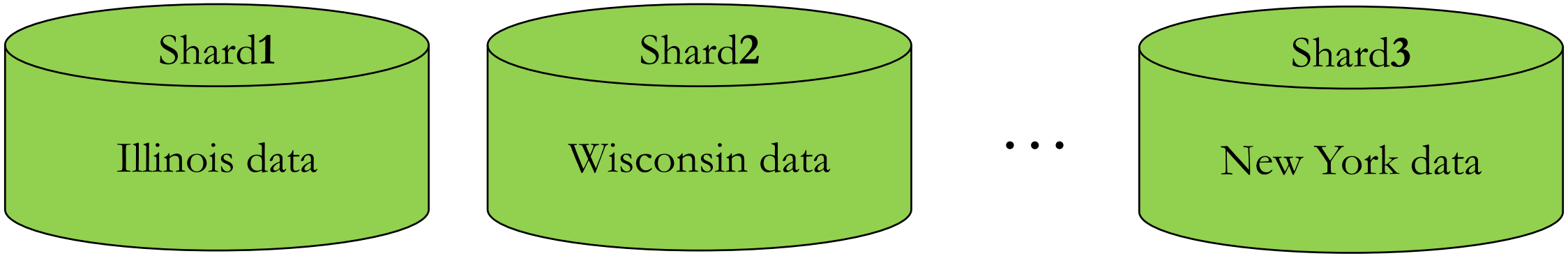
- **Functional partitioning**:
  - Create multiple databases storing different categories/types of data.
  - Eg.: three separate databases for: accounts, orders, and customers.
  - Cons:
    - Limits queries joining rows in tables in different DBs
    - Only a few functional partitions are possible. It's not highly scalable.
- **Data partitioning** is a more general approach…

Functional partitioning divides by **tables**

Data partitioning divides by **rows**

Sharding *(data partitioning)* relational databases

- Divide your data universe into disjoint subsets is called **shards**.
- For example: Consider parallelizing Facebook's database…
  - Maybe put Illinois users in one machine, Wisconsin in another, etc.
  - Each node stores rows for all tables, but only a subset of rows.



- **Sharding key** determines assignment of rows to shards.
- Relational databases usually don't support sharding natively, it must be somehow hacked at the application level.

Sharding example

Shard0

User

user	name
0	Steve
2	Yingyi
4	Alex

Post

user	date	text
0	04-25	Hi there…
0	04-27	Still tea…
2	03-12	Web scal…
2	04-25	Tips and…

Shard1

User

user	name
1	Guannan
3	Clarissa

Post

user	date	text
3	04-05	Box pl…
1	04-27	Sound…
1	03-12	Random…
3	04-27	Northw…

- In this example, `shard_id = user % 2`
- How to implement query for all posts by Steve?

All the data we need must be on Shard 0.

```
SELECT * FROM Post NATURAL JOIN User WHERE user=0?
```

Sharding example 2

Shard0

User

user	name
0	Steve
2	Yingyi
4	Alex

Post

user	date	text
0	04-25	Hi there…
0	04-27	Still tea…
2	03-12	Web scal…
2	04-25	Tips and…

Friend

user	friend
0	1
0	2
0	3
0	4
6	0

- How to implement query for latest 10 posts from Steve's friends?

```
SELECT * FROM User
 NATURAL JOIN Friend
 JOIN Post ON
    Post.user=
    Friend.friend
WHERE
User.name="Steve"
ORDER BY date DESC
LIMIT 10;
```

- Steve may be friends with users in all the shards; all shards must be queried.
- Query above will not work verbatim: user=0 row only exists in Shard0.
- Each shard can supply ten latest posts, app must manually merge them and choose the latest ten.

Sharding conclusions

Pros

Because each row is stored once:

✓**Capacity** scales.

✓Data is **consistent.**

If sharding key is chosen carefully:

✓Data will be **balanced**.

✓Many queries will involve only one or a few shards. There is no central bottleneck for these.

Cons

✗Cannot use plain SQL.

✗Queries must be manually adapted to match sharding.

✗If sharding key is chosen poorly, shard load will be imbalanced, either by capacity or traffic.

✗Some queries will involve all the shards. The capacity for handling such queries is limited by each single machine's speed.

Some Simple Scaling math

- **N** nodes
- **R** total request rate *(requests per second or another time frame)*
- Each node has the capacity to handle a maximum rate of requests **C**.

- If each request is sent to one node:
  - $R_{max} = NC$
- If each request is sent to a constant **k** number of nodes:
  - $R_{max} = NC/k = \mathcal{O}(NC)$

} *Scalable* (increases with N)

- If each request is sent to all nodes:
  - $R_{max} = C$ ← *Not Scalable*

Summary

- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind primary, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **data partitioning**.
  - Each data row/element is assigned a single "home"
  - If not, consistency is very tricky (write race conditions for transactions).
- **Sharding** is data partitioning for SQL/relational DBs.
  - Works well for queries that can be handled within a single shard.
  - Sharding divides data along just one dimension, so inevitably some queries will involve all the nodes, and thus will not be scalable.
- Next time… NoSQL databases for more horizontal scaling!

Recall from Lecture 9: Scaling SQL Databases

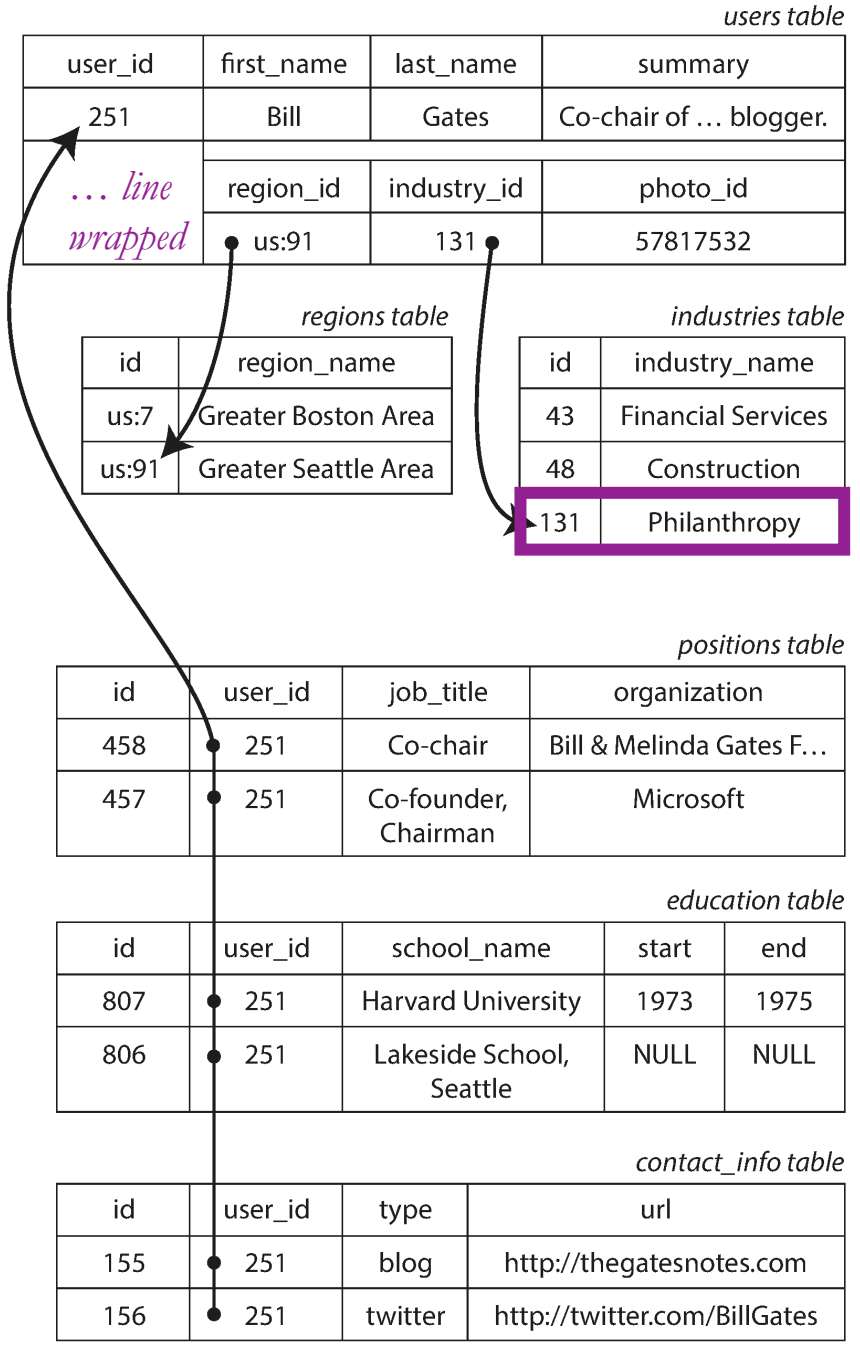
- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind master, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **data partitioning**.
  - Each data row/element is assigned a single "home" (or a constant number).
  - If not, each node must accept all writes, which is not scalable.
- **Sharding** is data partitioning for SQL/relational DBs.
  - Works well for queries that can be handled within a single shard.
  - Sharding divides data along just one dimension, so inevitably some queries will involve all the nodes, and thus will not be scalable.

Review of **Sharding**:

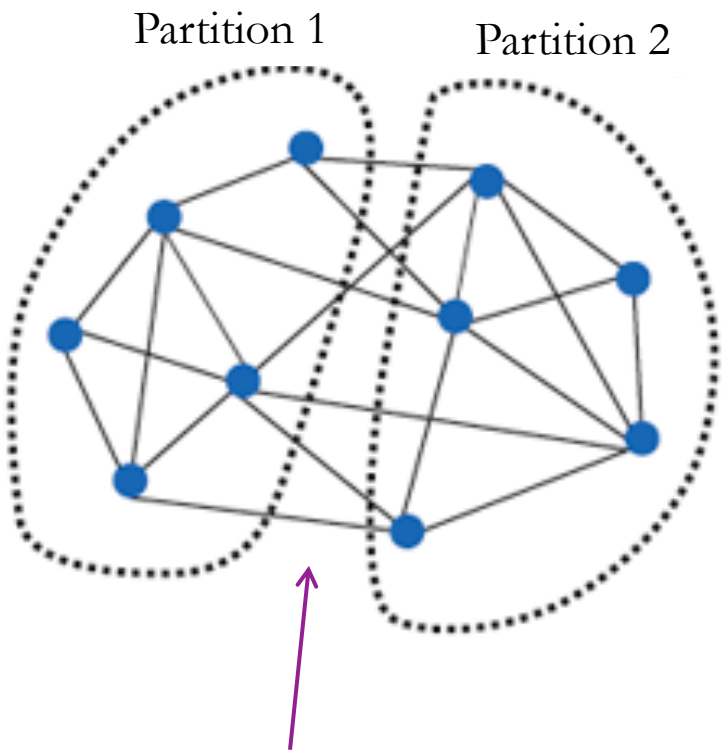
- **Splits data** among many machines.
- Accept **writes** on all machines.
- But the data partitioning is done manually.
  - Programmer chooses a sharding key or rule, and to write code that joins results from the different shards.
- Works well for queries that can be handled within a single shard.
- If we keep the relational model, with **normalized** data, many queries will involve all the nodes, so scaling is limited.
- NoSQL databases all solve this problem by **denormalizing** data, meaning that data is duplicated to isolate queries to one node.

Normalized data

- A **normalized** relational database has no duplication of data.
- References (foreign keys) point to shared data.
- Eg., at right, the Philanthropy industry is shared by many LinkedIn users.
  - In effect, many users are related to each other by all being linked to that industry
- To optimally partition the rows into shards, we could solve a balanced graph partitioning problem.



Graph partitioning model for DB sharding



Edges between partitions imply data transfers between nodes.

- Nodes represent database rows.
- Edges represent references (foreign keys)

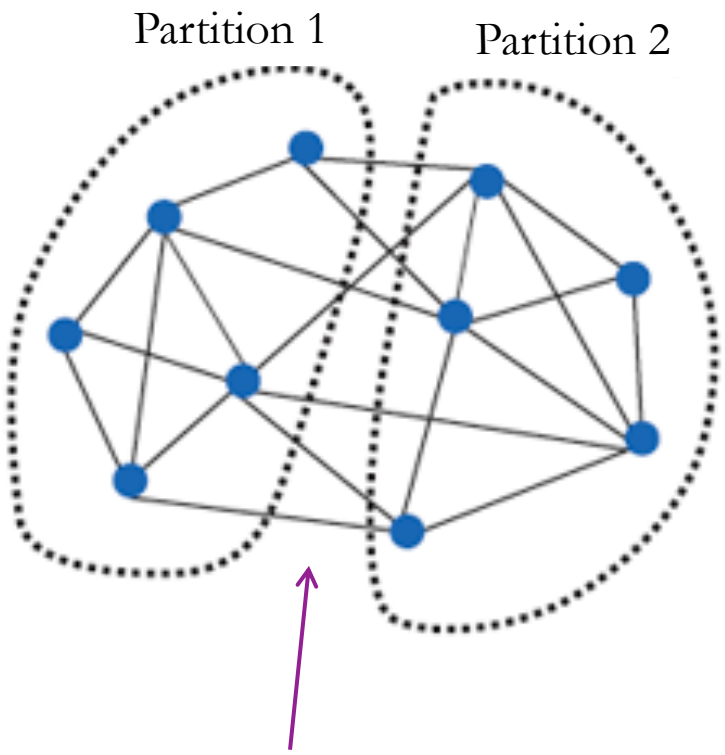
Task: assign the rows to shards (partition the nodes),

Such that:

- Total edges between partitions is minimized.
  (Need to fetch data from another shard for a JOIN is minimized.)
- Nodes per partition is roughly balanced.
  (Data stored on each shard is balanced.)

Reference: Facebook's social graph sharding: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

Partitioning challenges

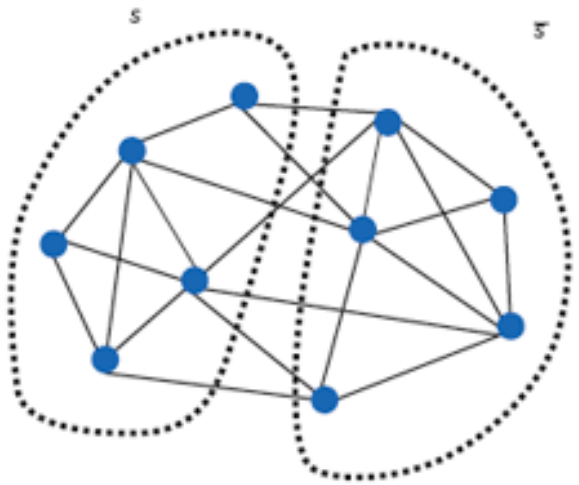


Edges between partitions imply data transfers between nodes.

- Solving this problem is NP-complete. (But we can approximately solve it fairly well.)
- This model's cost function is too simplistic:
  - Some rows are fetched more often.
  - An edge can pull data transitively from lots of nodes. Ie., the cost of a reference can vary dramatically.
- Even with an optimal partitioning, we still have data references between partitions.
- How does the data (graph structure) affect the solution quality?
  - Random interconnections hurt.
  - Nodes with high degree (many edges) hurt.
  - Structured, independent relationships are easy.
  - Nodes with one edge (spurs) are easy.

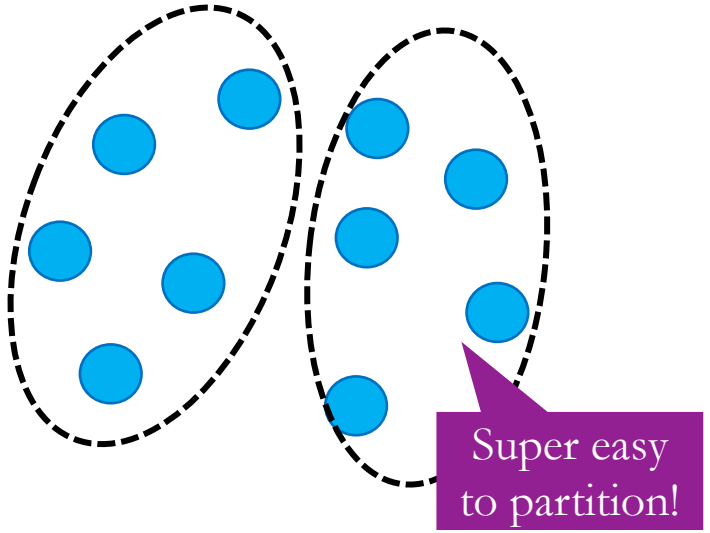
The jump from SQL to NoSQL

SQL sharding



- Eliminating the edges (references) would make the data partitioning problem trivial!
- Foreign keys (references) and JOINs (dereferences) are fundamental to SQL and relational databases.

NoSQL partitioning



- Removing the ability to create references gives us a **NoSQL** database.
- Instead of following references with JOINs, we store **denormalized** data, with copies of referenced data.

From Normalized … to Denormalized Data

SQL

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

… line wrapped

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

NoSQL

user_id	value
251	{ "first_name": "Bill", "last_name:" "Gates", "summary": "Co-chair of … blogger.", "region": "Greater Seattle Area", "industry": "Philanthropy", "photo_id": 57817532, "positions": [{ "job_title": "Co-chair", "organization": "Bill & Melinda …" }, { "job_title": "Co-founder, Chairman", "organization": "Microsoft" }], "education": [{ "school_name": "Harvard University", "start": 1973, "end": 1975 } { "school_name": "Lakeside School, …"}] }

NoSQL rationale

key	value
user251	{ "first_name": "Bill", "last_name": "Gates", "summary": "Co-chair of … blogger.", "region": "Greater Seattle Area", "industry": "Philanthropy", "photo_id": 57817532, "positions": [{ "job_title": "Co-chair", "organization": "Bill & Melinda …" }, { "job_title": "Co-founder, Chairman", "organization": "Microsoft" }], "education": [{ "school_name": "Harvard University", "start": 1973, "end": 1975 } { "school_name": "Lakeside School, …"}] }
user444	{ first_name: "Steve",

One **key** is indexed.

Precise format of **value** varies by NoSQL DB type.

Why just one column?

- Without references, it's impossible to define finite/fixed columns (a schema).
- Consider "positions": we don't know how many position columns to add.
- Some NoSQL DBs allow multiple columns, but each row can have different columns ("wide columns")

Why just one table?

- Some NoSQL DBs allow multiple tables, but since rows can have any format, it's kind of meaningless.

NoSQL DBs are **key-value stores**.

Hashing is the basis of distributed NoSQL DBs

- A ***hash*** is an algorithm that takes a value and returns a pseudo-random value derived from it.
- It's a *constant* but *unpredictable* mapping
  - A long sequence of arithmetic operations
- MD5 is a standard hash function:
  - `"Steve"` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `"steve"` → `2666b87c682f5072f62bab0955d485ce`
  - `"Janice"` → `3837607db4754c036425cb1b2a7c8766`
  - `"1"` → `b026324c6904b2a9cb4b88d6d61c81d1`
  - `"Steve"` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `tale_of_two_cities.txt` *(806,878 characters)* → `3ab56b74562a714a5638f94446581977`

This hash is case sensitive

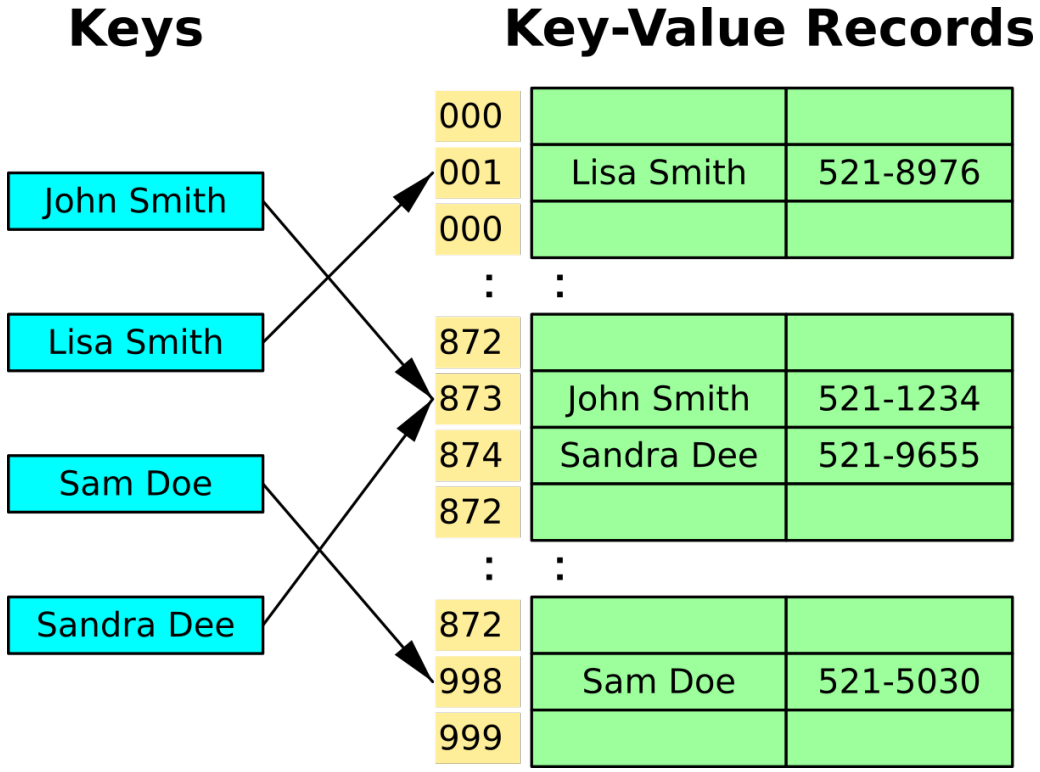
- The same input always gives the same output
- Length of the input can vary, but output has fixed length

Hash Table

- Stores (***key***, ***value***) pairs
  - This abstract data type is called a ***dictionary***, or ***map***.
- For example:
  - A word and its definition.
    - "word" → "a single distinct meaningful element of speech or writing, …"
    - "hash" → "a dish of cooked meat cut into small pieces and cooked again, …"
  - A database table's primary key and the rest of the columns in the row:
    - StaffID → [StfFirstName, StfLastName, StfStreetAddress, StfCity, StfState, ...]
    - 98005 → ["Suzanne", "Viescas", "15127 NE 24th, #383", …]
    - 98007 → ["Gary", "Hallmark", "Route 2, Box 203B", …]

Hash Table mechanics

- ***Hash the key*** to determine the address where the value is stored

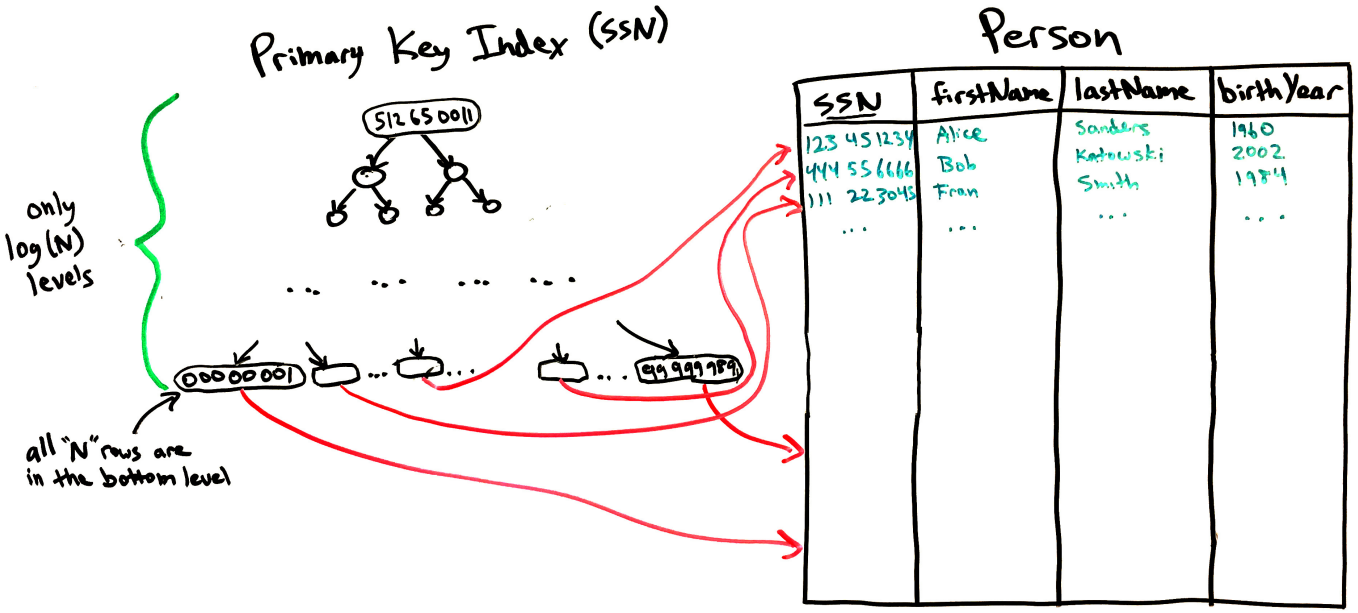


- If the address is already filled, then use the next open slot
  - This is called a *collision* and there are other strategies besides "linear probing"

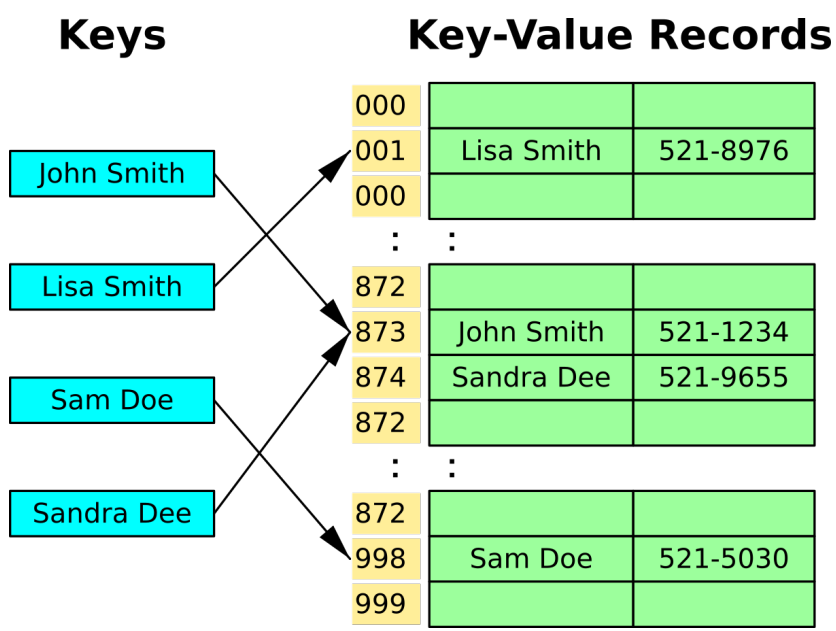
Hash Indexes in SQL databases

- A hash table is an alternative to a search tree
  - It lets you find the data in one step!
  - However, it does not support efficient range queries.
    - A hash table scatters data randomly, so walking though a range is difficult.

Tree-based table index

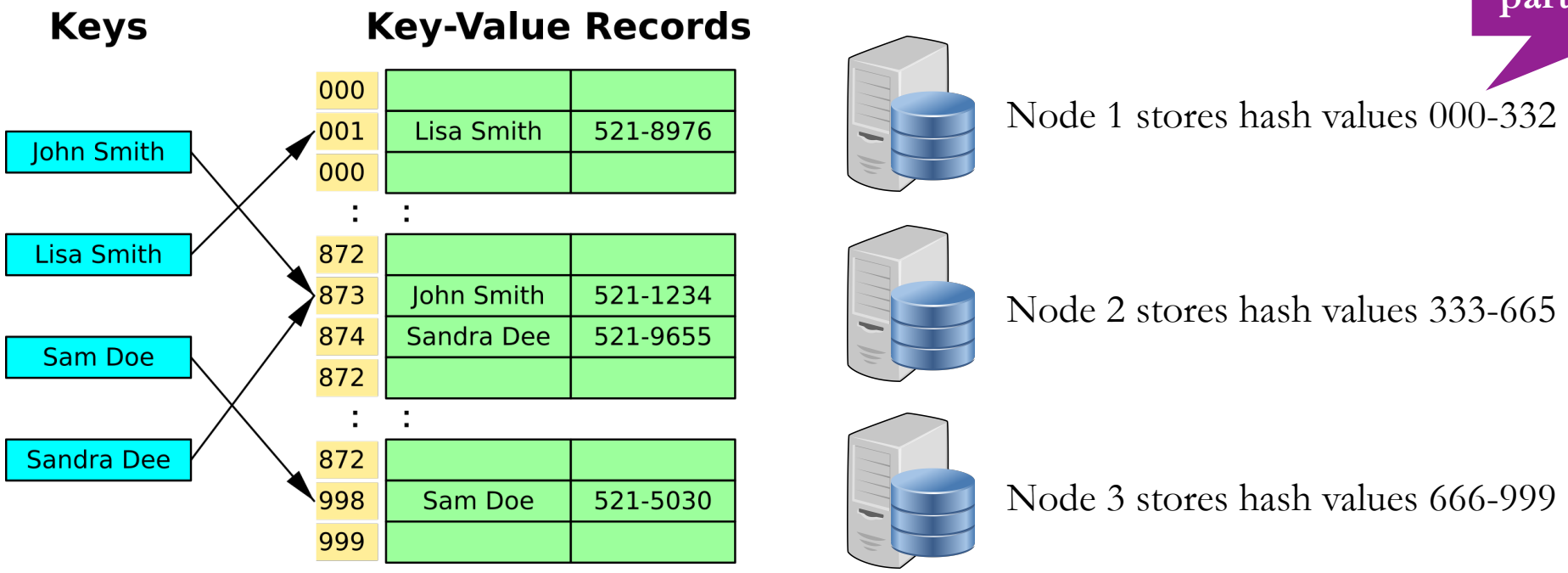


Hash-based table index



Distributed Hash Table

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



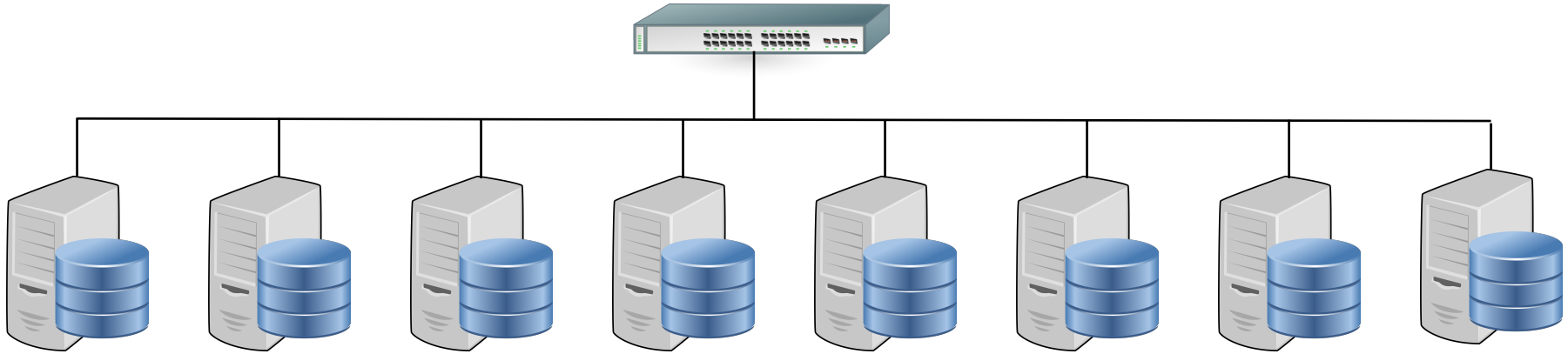
DHT is a NoSQL database

- NoSQL databases are distributed key-value stores
- Like one big table with just a primary key
- They have a map/dictionary interface, do not support SQL queries.
  - You can only:
    - ***get*** a value for a key.
    - ***put*** a value for a key.
  - Each operation only affects the node(s) storing that key
    - Very **scalable**! (can grow large without slowing down)
- If we wanted to support full SQL, JOINs would have to pull data from many nodes in the cluster and performance would be slow.

What limits the scalability?

Distributed, **shared-nothing** architecture

- Create a **cluster** of computers connected to each other.
- Each **node** in the cluster stores a fraction of the data set.



- Distributed database examples:
  - MongoDB, Cassandra, Amazon DynamoDB, …
- Distributed filesystems also use the same basic idea:
  - Hadoop HDFS, Google File System (Colossus, BigTable), Amazon S3, …

NoSQL downsides :(

key	value
user251	{ "first_name": "Bill", "last_name:" "Gates", "summary": "Co-chair of … blogger.", "region": "Greater Seattle Area", "industry": "Philanthropy", "photo_id": 57817532, "positions": [{ "job_title": "Co-chair", "organization": "Bill & Melinda …" }, { "job_title": "Co-founder, Chairman", "organization": "Microsoft" }], "education": [{ "school_name": "Harvard University", "start": 1973, "end": 1975 } { "school_name": "Lakeside School, …"}] }
user444	{ "first_name": "Steve",

- Just one indexed column (the key).
  - Because index is built with hash-based partitioning.
- Denormalized data is duplicated.
  - Wastes space.
  - Cannot be edited in one place.
  - Eg., "Greater Seattle Area" is repeated in many user profiles instead of "region:91"
- References are possible, but:
  - Following the reference requires another query, probably to another node.
  - There is no constraint checking (refs can become invalid after delete).

Normalization thought experiment

key	value
user251	{ "first_name": "Bill", "last_name": "Gates", "summary": "Co-chair of … blogger.", "region": "us:91" "industry": 131 "photo_id": 57817532, "positions": [458, 457], "education": [807, 806] }
reg:us:91	{ "region_name": "Greater Seattle Area" }
ind:131	{ "industry_name": "Pilanthropy"
pos:458	{ "user_id": 251, "job_title": "Co-chair", "organization": "Bill and Melinda Gates …" }
pos:457	{ "user_id": 251, "job_title": "Co-founder, Chairman", "organization": "Microsoft"
edu:807	{ "user_id": 251, "school_name": "Harvard University", "start": 1973, "end": 1975 }

- What happens if we try to store normalized data, like this, in a NoSQL database?
- Is it possible?
- Why isn't it done?

- It's possible, but you would need many serial queries to many different DB nodes to fetch the user's profile.
- References are not enforced by the schema, so they can become broken.

Summary

- **Data partitioning** is necessary to divide write load among nodes.
  - Should minimize references between partitions.
  - Can be treated as a graph partitioning problem.
  - SQL sharding was a special case of data partitioning, done in app code.
- **NoSQL** databases make partitioning easy by eliminating references.
- Without references, data becomes **denormalized**.
  - Duplicated data consumes more space, can become inconsistent.
- **NoSQL databases** are very scalable, but they provide only a very simple **key-value** abstraction. One key is indexed.
- **Distributed Hash Table** can implement a NoSQL database.
  - The hash space is divided evenly between storage nodes.
  - Client computes hash of key to determine which node should store data.

Choosing a Database

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2015%20-%20Choosing%20a%20Database.pdf

Recall the goals of a Database:

- **Scalability** – work with data larger than computer's RAM.
- **Persistence** – keep data around after your program finishes.
- **Indexing** – efficiently sort & search along various dimensions.
- **Concurrency** – multiple users or applications can read/write.
- **Analysis** – SQL query language is concise yet powerful. Avoid transferring lots of data. Run analysis on the storage machines.
- **Separation of concerns** – decouples app code from storage engine. Also, allows apps to be stateless, allowing parallelism.

Less importantly:

- **Integrity** – restrict data type, disallow duplicate entries.
- **Deduplication** – save space, keep common data consistent.
- **Security** – different users can have access to specific data.

Data storage options

	Examples	Use cases
SQL Relational DB	MySQL, Oracle	Structured data. Transactional data.
Column-oriented DB	Snowflake, BigQuery	SQL queries for analytics on huge datasets (OLAP).
Search engine	Elastic search	Searchable text documents.
Document store	MongoDB	Semi-structured data (JSON docs).
Distributed cache	Redis	In-memory cache with expiration. Very fast.
NoSQL DB	Cassandra, Dynamo	Huge data to be accessed in parallel.
Cloud object store	S3, Azure Blobs	Images, videos, & other static content.
Cluster filesystem	Hadoop dist. fs.	Files to be processed in huge parallel computation.
Networked filesystem (NAS)	NFS, EFS, EBS	App is designed to write to local file system, but we want that storage to be scalable and shared.

With so many options, choosing the "right" storage option is difficult!
Dozens of other DBs exist, but these examples are popular and representative examples.

SQL Relational Databases

- The most common, traditional solution.
- Data is organized into **tables**, with **foreign keys** to cross reference.
  - The format of the data (schema) is predefined. Consistent, not flexible.
- SQL language run common data analyses *inside* the database:

```
SELECT category, avg(price) FROM products GROUP BY category;
```

  - Running calculations on the storage machine helps performance.
    Data transfer (I/O) is a bottleneck in most systems.
  - Reduces data transfer between app and data store.
    Above query just returns a short answer over the network.
- Supports **transactions** (sequence of ops to be committed *all or none*).
- Works very well up to a certain size.
  - Writes must happen on **one "master" machine**.
  - **Read-replicas** give read scaling (w/delay). **Sharding** can help write scaling.

Database transaction example: *spending gift card balance*

```
-- 1. start a new transaction
START TRANSACTION;

-- 2. get the gift card balance
SELECT @cardBalance:=balance FROM giftCards
  WHERE cardId=23902;

-- 3. insert a new order for customer 145
INSERT INTO orders(orderDate, status,
                   customerNumber)
  VALUES('2021-02-22', 'In Process', 145);

-- 4. get the newly-created order id
SELECT @orderId:=LAST_INSERT_ID();
```

```
-- 5. Insert order line items
INSERT INTO orderdetails(orderNumber, product,
                         quantity, priceEach)
  VALUES(@orderId,'S18_1749', 3, '136'),
        (@orderId,'S18_2248', 5, '55');

-- 6. deduct from balance
UPDATE giftCards SET balance=(@cardBalance-683)
  WHERE cardId=23902;

-- 7. end the transaction (commit changes)
COMMIT;
```

Why must these steps be completed *atomically* (together)?

- Prevent card balance from being spent twice.
- Prevent clients from seeing the order without line items.

The first is a race condition, the second is an inconsistency.

Transactions on a distributed (NoSQL) DB?

- Transactions less common on NoSQL DBs because they are slow.
- Often, transactions are not necessary because 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 the keys involved (the lock prevents reads/writes).
     - All replicas must agree to the lock.
     - Multiple competing lock requests may occur in parallel, but one must be chosen, so multiple rounds of communication may be needed to agree.
  2. Execute the transaction on all replicas. Wait for all to confirm.
  3. Unlock the keys involved (let reads/writes proceed).
- Reference: Google's Spanner OSDI 2012 paper

How to implement a distributed lock?

- A lock requires an atomic conditional write operation, like:

```
PUT("key", "new_val") IF
GET("key") == "old_val";
```

- Many NoSQL databases support something like this (Cassandra, Mongo).

- Or if you don't care too much about scalability:
  - Store your transactional data in a SQL Database.
  - Or use a SQL Database to implement a lock used to control access in NoSQL.

NoSQL transaction to deduct $1 from an account:

```
id=37; // my unique client id
while(GET("lock") != id) {
  // get the lock if possible
  PUT("lock", id)
  IF GET("lock") == 0;
}
// do my 2-step transaction:
x = GET("balance");
PUT("balance", x - 1);

// release the lock
PUT("lock", 0);
```

Throughput/scaling limitations

Data store	Examples	Throughput limitations
SQL Relational DB	MySQL, Oracle	All writes to primary. Read-replication adds delay.
Column-oriented DB	Snowflake, BigQuery	All use a scalable data partitioning method, such as hashing.
Search engine	Elastic search	
Document store	MongoDB	
Distributed cache	Redis	
NoSQL DB	Cassandra, Dynamo	
Cloud object store	S3, Azure Blobs	
Cluster filesystem	Hadoop dist. fs.	
Networked filesystem	NFS, EFS, EBS	One machine, many disks (RAID).

Data abstractions

Data store	Examples	Data abstraction
SQL Relational DB	MySQL, Oracle	Tables, rows, columns
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns
Search engine	Elastic search	JSON, text
Document store	MongoDB	Key → JSON
Distributed cache	Redis	Key → value (lists, sets, etc.)
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)
Cloud object store	S3, Azure Blobs	K-V / Filename-contents
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents
Networked filesystem	NFS, EFS, EBS	Filename-contents

Highly structured

Semi-structured

Files with data "blobs"

Files may have some internal structure, but the storage API is not aware of it and makes no use of it.

Column-oriented Relational Databases

Previously, we saw that:

- Read-Replication and Sharding allow lots of **parallel** reads and writes.
- This is useful for **OLTP** applications (Online Transaction Processing).

OLAP (Online Analytics Processing) involves just a few huge queries

- Eg., Over the past three years, in which locations have customers been most responsive to our mailed-to-home coupons?
- Analytics queries involve **scanning** tables, not using indexes.
- Must be parallelized over many nodes.
- The workload is mostly **reads**, with occasional importing of new data.

Column-oriented DBs are optimized for SQL analytics workloads.

Many choices for semi-structured, scalable stores!

Data store	Examples	Data abstraction
SQL Relational DB	MySQL, Oracle	Tables, rows, columns
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns
Search engine	Elastic search	JSON, text
Document store	MongoDB	Key → JSON
Distributed cache	Redis	Key → value (lists, sets, etc.)
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)
Cloud object store	S3, Azure Blobs	K-V / Filename-contents
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents
Networked filesystem	NFS, EFS, EBS	Filename-contents

Semi-structured

- Best choice depends on the structure of data being stored.

Distributed data store comparison

Copied from: https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

MongoDB stores JSON objects

- Below, "_id" is the sharding key used for data partitioning:

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date('Jun 23, 1912'),
    death: new Date('Jun 07, 1954'),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],
    views : NumberLong(1250000)
}
```

MongoDB (3.2) – *a "document store"*

- **Main point:** JSON document store.
- **Best used:** If you like JSON. If documents change frequently, and you want to keep a history of changes.
- **For example:** For most things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.

MongoDB (3.2) – *a "document store"*

- Written in: C++
- **Main point: JSON document store**
- License: AGPL (Drivers: Apache)
- Protocol: Custom, binary (BSON)
- Master/slave replication (auto failover with replica sets)
- Sharding built-in
- Queries are javascript expressions
- Run arbitrary javascript functions server-side
- Geospatial queries
- Multiple storage engines with different performance characteristics
- Performance over features
- Document validation
- Journaling (efficiently keeping previous versions of documents)
- Powerful aggregation framework
- Text search integrated
- GridFS to store big data + metadata (not actually a FS)
- Has geospatial indexing
- Data center aware
- **Best used:** If you need dynamic queries. If you prefer to define indexes, not map/reduce functions. If you need good performance on a big DB. If documents change frequently, and you want to keep a history of changes.
- **For example:** For most things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.

ElasticSearch also stores JSON documents

- But it's designed to **index every word** in the document and to handle advanced queries:


```
{
    "date": "2020-04-15",
    "txt": "$1,000 in donations buy lunch for ambulance, victim assistance workers ... WATERTOWN, N.Y. (WWNY) - Two anonymous donations at a downtown Watertown restaurant are buying first responders lunch. Vito's Gourmet received the donations totaling $1,000 from people who wanted to give back to the community. On Wednesday, owner Todd Tarzia delivered gift certificates to Guilfoyle Ambulance and the Victims Assistance Center for each of its employees. \u201cOne of the donors in particular specifically designated first responders as who they wanted the lunch to go to and we thought, well, geez, first responders don\u2019t all sit around a lunch table, especially in the world with the virus right now so what can we do to get lunch to them on their schedule. So we decided to make up gift certificates for amounts that\u2019s enough for everyone to have lunch,\u201d said Tarzia. \"To get a gift like this is just so thoughtful and our people are going to be very thankful for this,\" said Bruce Wright, CEO, Guilfoyle Ambulance. …",
    "title": "$1,000 in donations buy lunch for ambulance, victim assistance workers",
    "lang": "en",
    "url": "https://www.wwnytv.com/2020/04/15/donations-buy-lunch-ambulance-victim-assistance-workers/"
}
```


- How could you use a simple Distributed Hash Table (eg., Redis or Cassandra) to implement an index of all the words in this document? User wants to search for "Watertown AND gift".
- An inverted index. For each word in document, store this document id under the word key.

ElasticSearch – *a "search engine"*

- **Main point:** Advanced Search

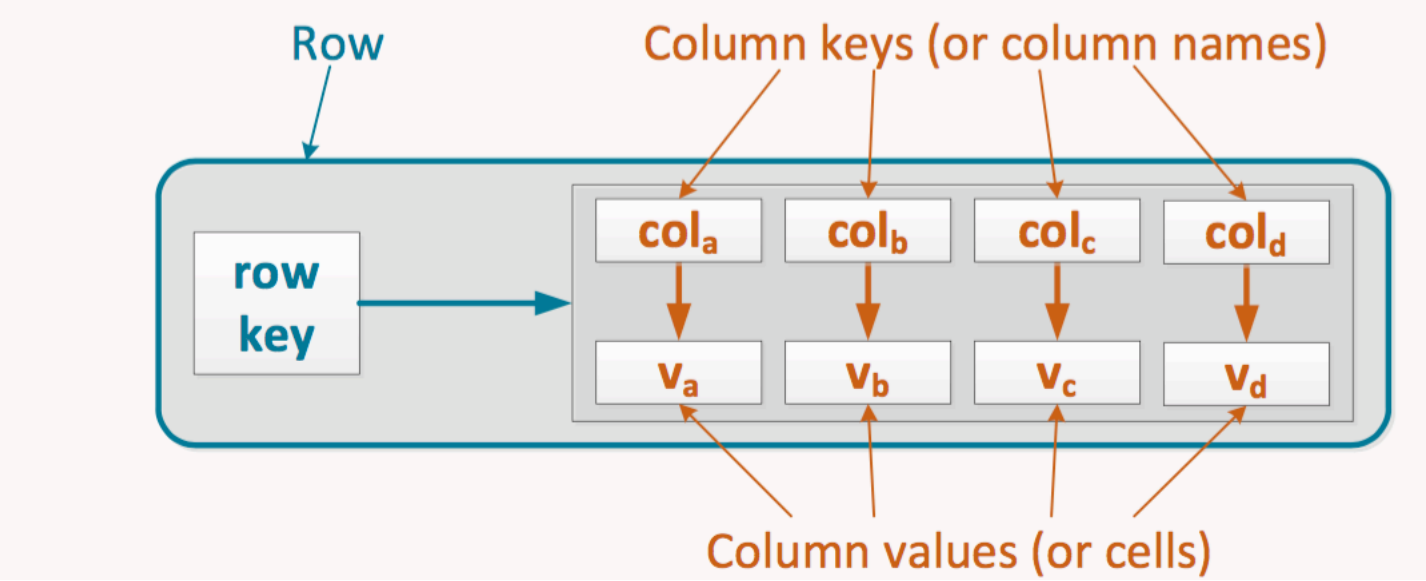
- **Best used:** When you have objects with (flexible) fields (or plain text), and you need search by all words in the document, or you need to construct complex search queries (AND, OR, NOT, …)

- **For example:** Full-text document search, a leaderboard system that depends on many variables.

ElasticSearch (0.20.1) – *a "search engine"*

- Written in: Java
- **Main point: Advanced Search**
- License: Apache
- Protocol: JSON over HTTP (Plugins: Thrift, memcached)
- Stores JSON documents
- Has versioning
- Parent and children documents
- Documents can time out
- Very versatile and sophisticated querying, scriptable
- Write consistency: one, quorum or all
- Sorting by score (!)
- Geo distance sorting
- Fuzzy searches (approximate date, etc) (!)
- Asynchronous replication
- Atomic, scripted updates (good for counters, etc)
- Can maintain automatic "stats groups" (good for debugging)
- **Best used:** When you have objects with (flexible) fields, and you need "advanced search" functionality.
- **For example:** A dating service that handles age difference, geographic location, tastes and dislikes, etc. Or a leaderboard system that depends on many variables.

Cassandra rows (NoSQL, 2d key-value store)



Each row's value is a map of "**columns**" to value. Column names are indexed within the row.

Row key is the hashing key that determines on which nodes the row is stored.

7b976c48...	name: Bill Watterson	state: DC	birth_date: 1953
7c8f33e2...	name: Howard Tayler	state: UT	birth_date: 1968
7d2a3630...	name: Randall Monroe	state: PA	
7da30d76...	name: Dave Kellett	state: CA	

Columns are defined separately for each row!

Details: https://tech.ebayinc.com/engineering/cassandra-data-modeling-best-practices-part-1/

Cassandra bridge information example

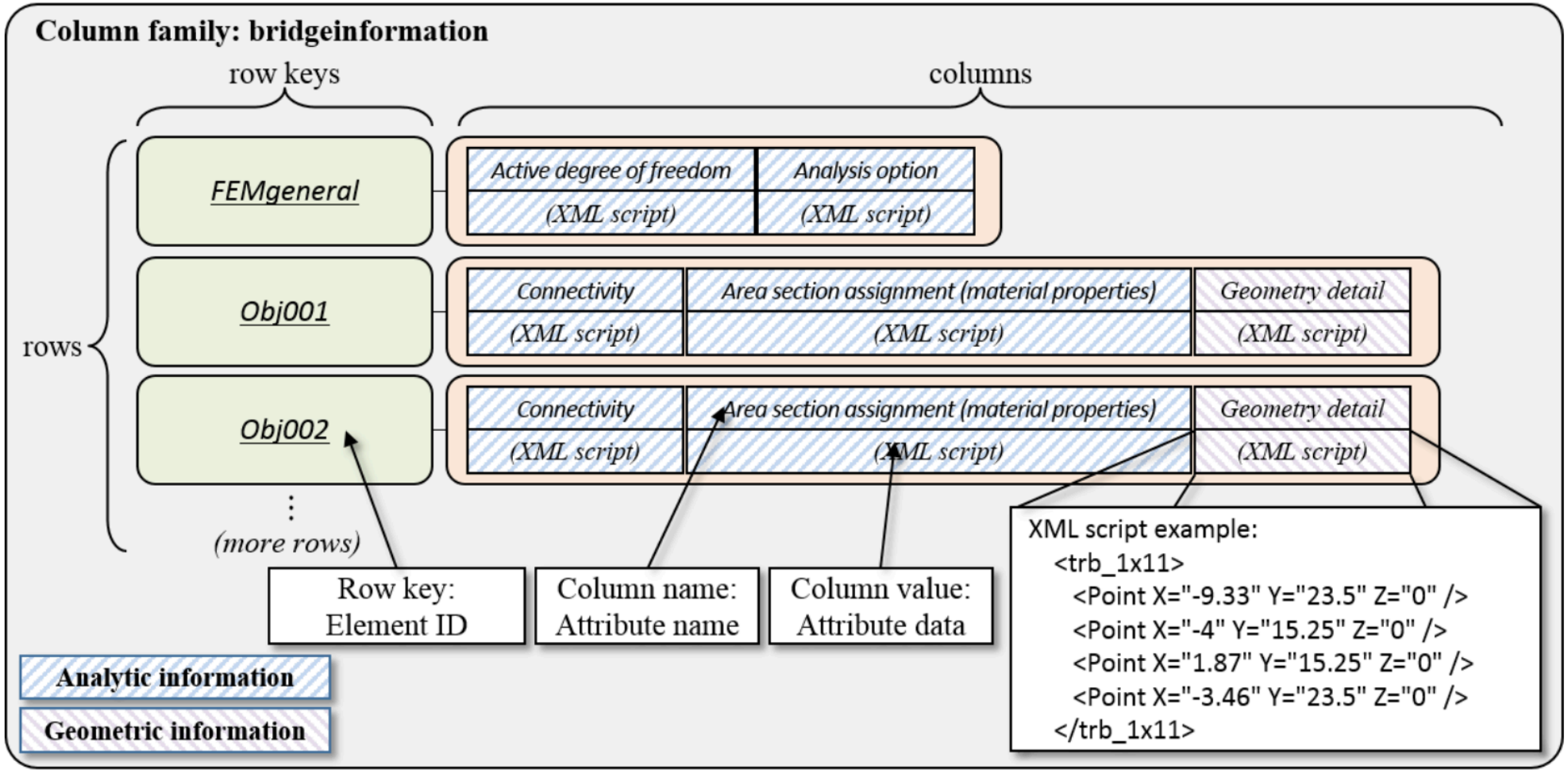


Fig. 6 Data schema of bridge information model on Apache Cassandra

https://www.researchgate.net/publication/301630614_A_NoSQL_data_management_infrastructure_for_bridge_monitoring

Cassandra – *a "NoSQL database"*

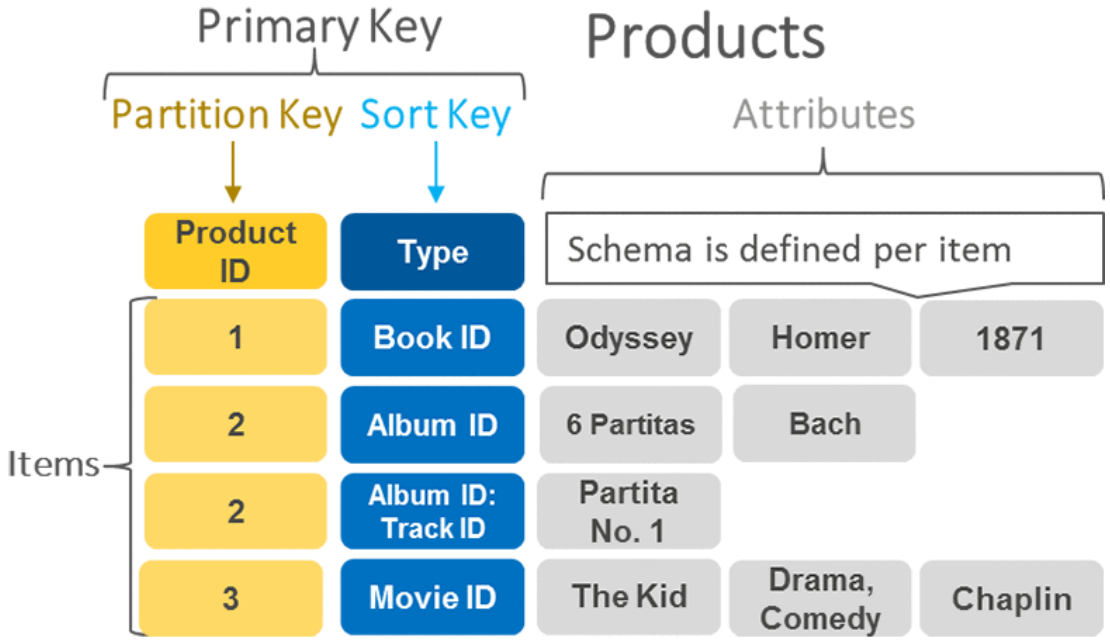
- **Main point:** Store huge datasets.
- **Best used:** When you need to store data so huge that it doesn't fit on server, but still want a friendly familiar interface to it.
- **For example:** Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.

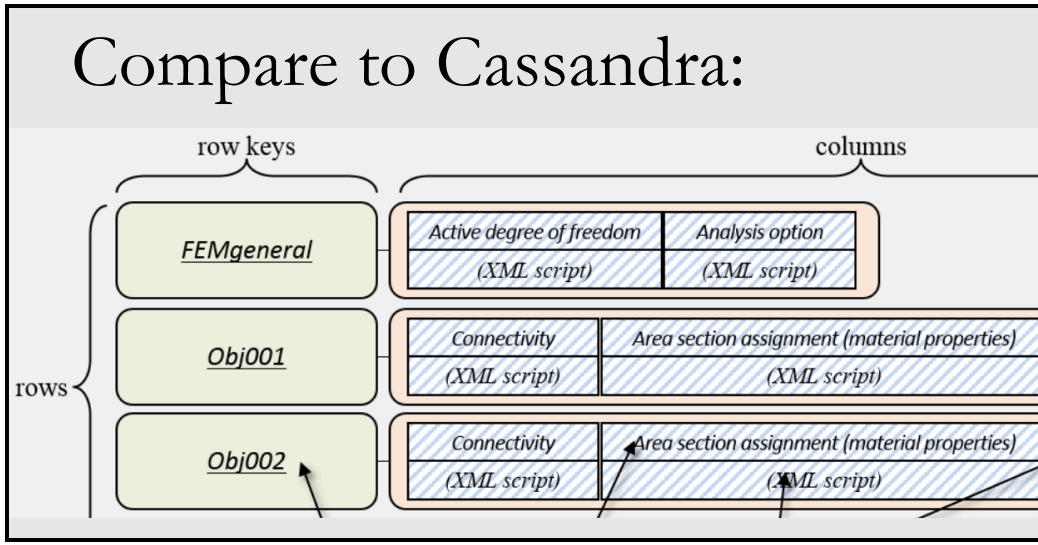
Cassandra (2.0) – *a "NoSQL database"*

- Written in: Java
- **Main point: Store huge datasets in "almost" SQL**
- License: Apache
- Protocol: CQL3 & Thrift
- CQL3 is very similar to SQL, but with some limitations that come from the scalability (most notably: no JOINs, no aggregate functions.)
- Querying by key, or key range (secondary indices are also available)
- Tunable trade-offs for distribution and replication (N, R, W)
- Data can have expiration (set on INSERT).
- Writes can be much faster than reads (when reads are disk-bound)
- Map/reduce possible with Apache Hadoop
- All nodes are similar, as opposed to Hadoop/HBase
- Very good and reliable cross-datacenter replication
- Distributed counter datatype.
- You can write triggers in Java.
- **Best used:** When you need to store data so huge that it doesn't fit on one server, but still want a friendly familiar interface to it.
- **For example:** Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.

DynamoDB is a 2D key-value store, like Cassandra

- **Partition Key** (like Cassandra's row key) is hashed to find partition.
- **Sort Key** (optional) allows efficient range queries within the partition key.
  - Together, the Partition and Sort keys form the **Primary Key**.
- **Attributes** are key-value pairs stored under the Primary Key.

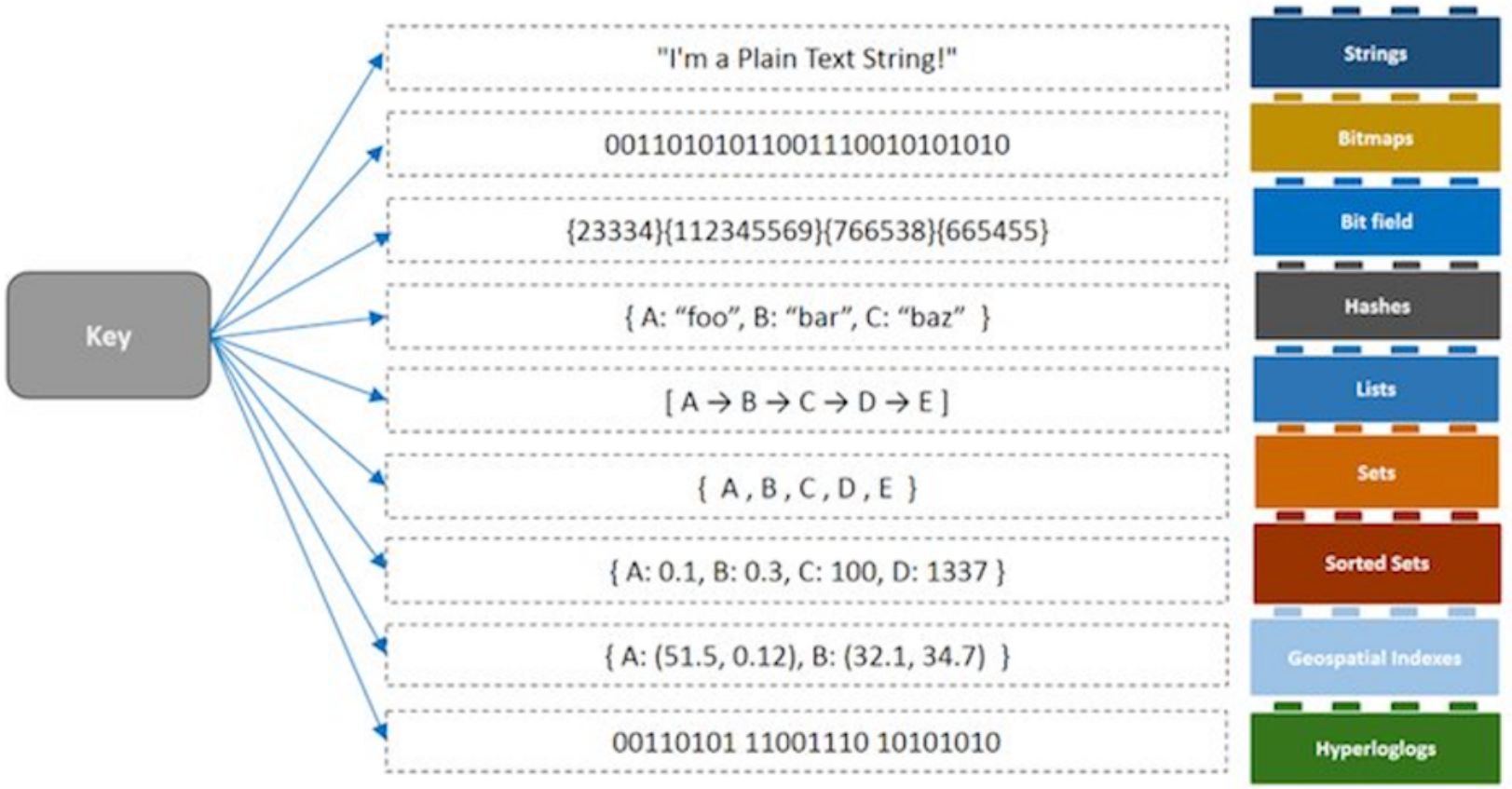




Reference: https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/

Redis DB/cache values

- All data is stored in RAM (not just disk), for high performance.
- Redis understands many types of data values:
  - allows operations like "add to a set" that modify (or get) part of a value.



Distributed Caches

- For example: Redis, Memcached, ElastiCache, Riak

- Originally developed in order to reduce load on relational databases.
  - Cache responses to frequent DB requests or other materialized application data.
  - Always support timed **expiration** of data.
- Use the same basic key-value abstraction as NoSQL distributed DBs.
- Store data across many nodes.
- Have the same data consistency issues as NoSQL databases.
- Often optimized to do everything in-memory,
  - but most also store data persistently to disk.

So… distributed caches and NoSQL databases are very similar.

Comparison

NoSQL Database

- Items are **permanent**/persistent.
- All items are stored on **disk** (some are cached in RAM).
- **Scale** is the primary goal.

Distributed Cache

- Items **expire**.
- Items are stored in RAM (though maybe persisted to disk).
- **Speed** is the primary goal.
- RAM capacity is limited.
  - Once capacity is reached, start evicting oldest/least-used items.

Comparison

CDN / Reverse Proxy Cache

- Cache common HTTP responses.
- Transparent to the application.
- Just configure the cache's origin

Distributed Cache

- Cache common's used data that contributes to responses.
- For example:
  - A leaderboard in header of ever HTML page.
  - Session information for the user.

Redis (V3.2) – *a "cache"*

- **Main point:** Blazing fast storage.

- **Best used:** For rapidly changing data with a foreseeable database size (should fit mostly in memory). Also, for caching data than can be rebuilt from another data store.

- **For example:** To store real-time stock prices. Real-time analytics. Leaderboards. Real-time communication. And wherever you used memcached before.

Redis (V3.2) – *a "cache"*

- Written in: C
- **Main point: Blazing fast**
- License: BSD
- Protocol: Telnet-like, binary safe
- Disk-backed in-memory database,
- Master-slave replication, automatic failover
- Simple values or data structures by keys
- but complex operations like ZREVRANGEBYSCORE.
- INCR & co (good for rate limiting or statistics)
- Bit and bitfield operations (eg. to implement bloom filters)
- Has sets (also union/diff/inter)
- Has lists (also a queue; blocking pop)
- Has hashes (objects of multiple fields)
- Sorted sets (high score table, good for range queries)
- Lua scripting capabilities
- Has transactions
- Values can be set to expire (as in a cache)
- Pub/Sub lets you implement messaging
- GEO API to query by radius (!)

- **Best used:** For rapidly changing data with a foreseeable database size (should fit mostly in memory).

- **For example:** To store real-time stock prices. Real-time analytics. Leaderboards. Real-time communication. And wherever you used memcached before.

Filesystem choices

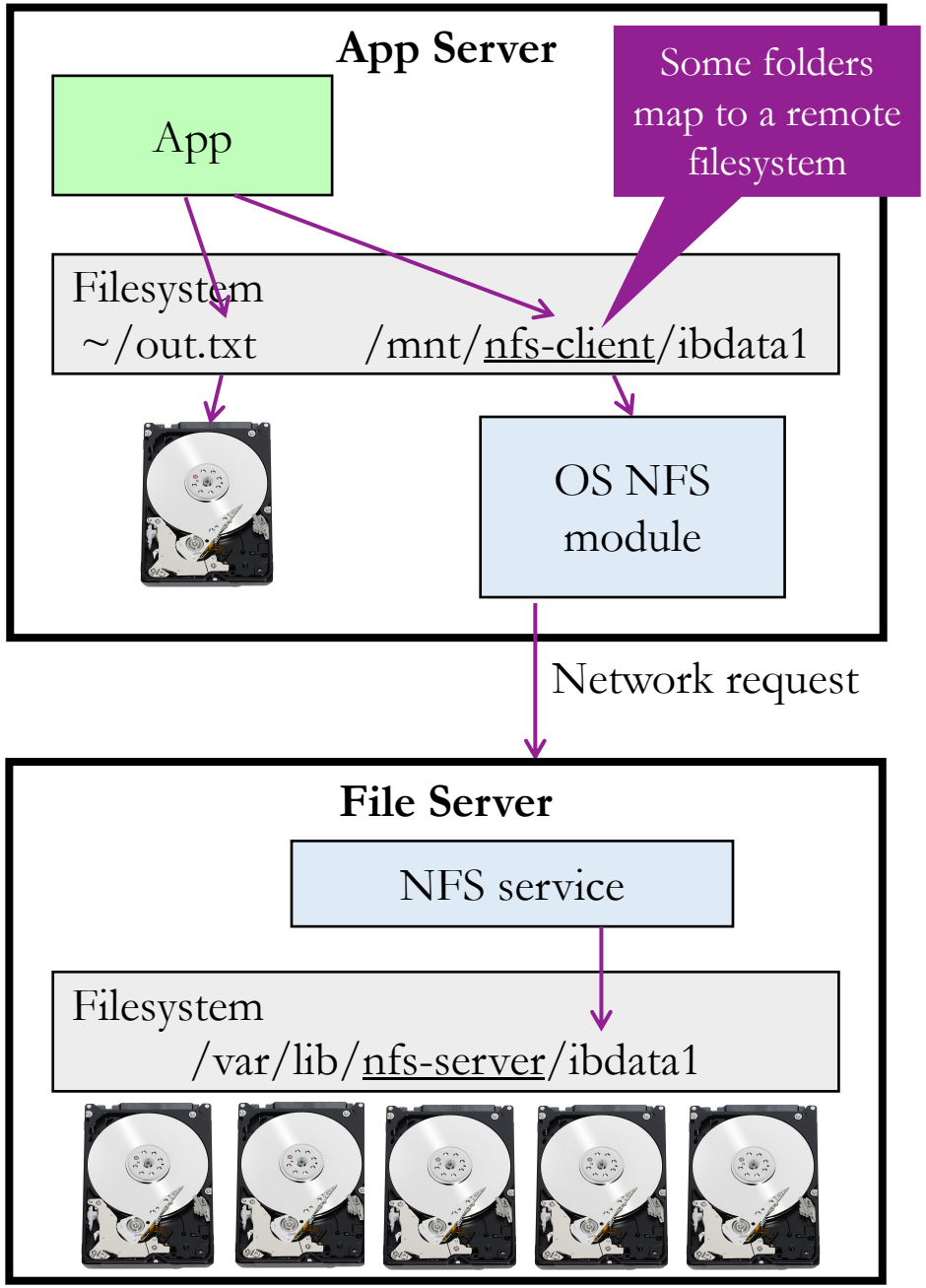
Data store	Examples	Data abstraction
SQL Relational DB	MySQL, Oracle	Tables, rows, columns
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns
Search engine	Elastic search	JSON, text
Document store	MongoDB	Key → JSON
Distributed cache	Redis	Key → value (lists, sets, etc.)
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)
Cloud object store	S3, Azure Blobs	K-V / Filename-contents
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents
Networked filesystem	NFS, EFS, EBS	Filename-contents

Files with arbitrary data

Networked file system

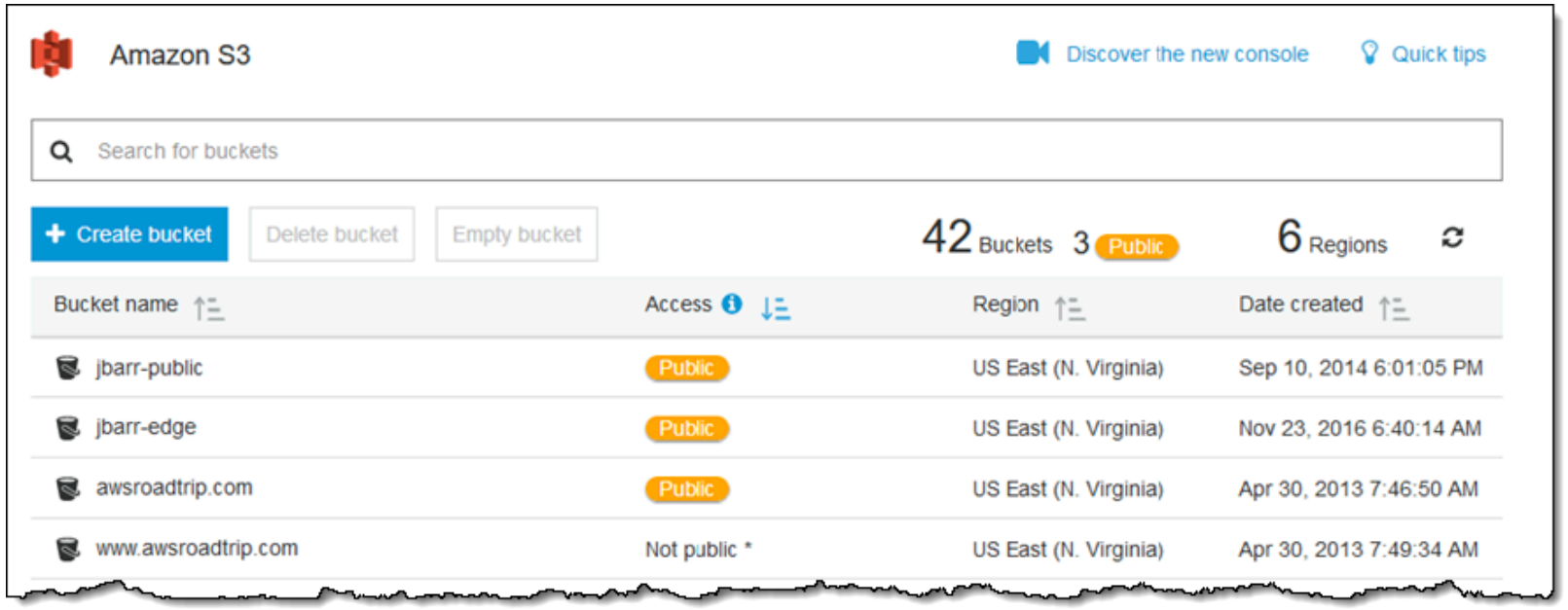
- Eg., NFS (unix), SMB (Windows).
- Managed by the OS.
- Provides a regular filesystem interface to applications by *mounting* the remote drive.

- Not too useful in modern applications, but may be necessary if your app is built to work directly with a local file system.
- Modern apps should instead interact with cloud-based storage services.



Cloud object store (S3)

- A flexible general-purpose file store for cloud apps.
- Managed by cloud provider. Capacity available is "unlimited."
- Provides a network API for accessing files (maybe REST).
- In other words, app accesses files like a remote database.
- Often provides a public HTTP GET interface to access files:
  - Can be easily connected to CDN or urls used directly



S3 example for hosting media files on web

- https://stevetarzia.com/localization

Browser view:

Matlab batphone scripts and data v1.0 (1.2MB) (ma
run). Please report any bugs or problems to me.

Matlab audio scripts v1.0 (0.4MB) (unfortunately,
run). Please report any bugs or problems to me. Th
these scripts:

- basic recordings (4.0 GB)
- HVAC off recordings (1.6 GB)
- lecture noise recordings (4.4 GB)

HTML:

```
<li><p><a href="mobisys11_batphone_v1.0.tar.gz">Matlab
batphone scripts and data v1.0 (1.2MB)</a> (may require some
toolkits to run).  Please report any bugs or problems to
me.</li>

<li><p><a href="mobisys11_scripts_v1.0.tar.gz">Matlab audio
scripts v1.0 (0.4MB)</a> (unfortunately, requires several
toolkits to run).  Please report any bugs or problems to me.
The following data is needed for these scripts:

<ul><li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_passive.tar.
gz">basic recordings (4.0 GB)</a>

<li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_HVAC_off.tar
.gz">HVAC off recordings (1.6 GB)</a>

<li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_lectures.tar
.gz">lecture noise recordings (4.4 GB)</a></ul>
```

Hadoop File System (HDFS)

- When you need to use Hadoop/Spark to do distributed processing.
- Data is too big to move it for analysis.
- Allows data to reside on the same machines where computation happens, thus making processing efficient.

- Hadoop distributed filesystem and its distributed processing tools were designed to work together.

Recap – Choosing a data store

Data store	Examples	Data abstraction	
SQL Relational DB	MySQL, Oracle	Tables, rows, columns	Highly structured
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns	
Search engine	Elastic search	JSON, text	Semi-structured
Document store	MongoDB	Key → JSON	
Distributed cache	Redis	Key → value (lists, sets, etc.)	
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)	
Cloud object store	S3, Azure Blobs	K-V / Filename-contents	Files with data "blobs"
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents	
Networked filesystem	NFS, EFS, EBS	Filename-contents	

Your choice depends mainly on the **structure** of data and pattern of **access**.