SEBA Master: Web Application Engineering

8. High Performance Web Applications

Prof. Dr. Florian Matthes, SS21

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

8. High Performance Web Applications

- Database Options for Web Applications
  - NoSQL Databases
  - Multi-model and Polyglot Persistence
- Scaling Web Applications
  - Bottlenecks
  - Scalability Patterns
  - Scaling with MongoDB

NoSQL - Not Only SQL

Next generation databases mostly addressing some of the points: **being non-relational, distributed, open-source** and **horizontally scalable.**

[Stefan Edlich (2009)]

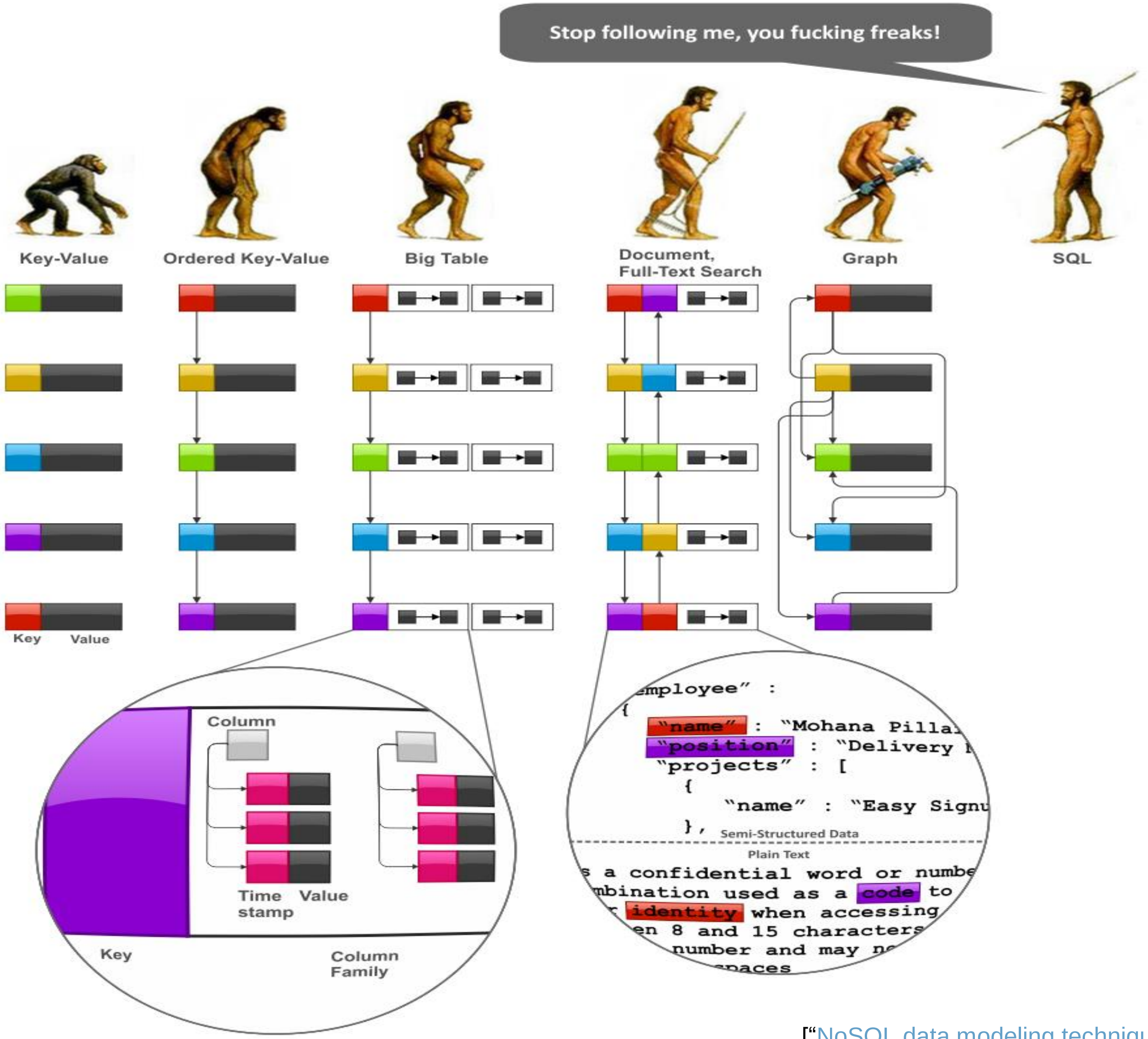
The original intention has been **modern web-scale databases**. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as:

- schema-free
- easy replication support
- simple API
- eventually consistent / BASE (not ACID)
- a huge amount of data

The misleading term "nosql" (the community now translates it mostly with "**not only sql**") should be seen as an alias to something like the definition above.

[http://nosql-databases.org/ - Stefan Edlich]

NoSQL Databases for Web Applications



["NoSQL data modeling techniques." Ilya Katsov (2012)]

Databases for Web Applications

Key-value Stores - Redis

Key-value store – data is stored in unstructured records consisting of a **key** and the **values** associated with that record.

[Aerospike]

- Example applications
  - Storing realtime stock prices
  - Cache management, session management, …
  - Circular log buffers (call center logs)
  - Realtime analytics
  - Leaderboards (sorted sets for maintaining high-score tables)
  - Realtime communication and broadcasting updates
  - Many more ..

- Disk-backed, in-memory database
- Master-slave replication, automatic failover
- Support for multiple data types
- Lua scripting capabilities
- Transaction support
- Values can be set to expire
- Pub/sub to implement messaging

- Dataset size limited to computer RAM (but can span multiple machines' RAM with clustering)
- Not suitable for complex applications with complex data models
- Not suited for interconnected (graph) data
- As the volume of data increases maintaining *unique keys* becomes more difficult and requires some complexity in generating character strings that will remain unique over a large set of keys

Databases for Web Applications

Bigtable

Bigtable is a sparse, **distributed multi-dimensional** sorted map for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.

- Example applications
  - Gmail
  - Google web indexing
  - Google earth
  - Google finance
  - Google book search
  - Blogger.com
  - Youtube
  - …..

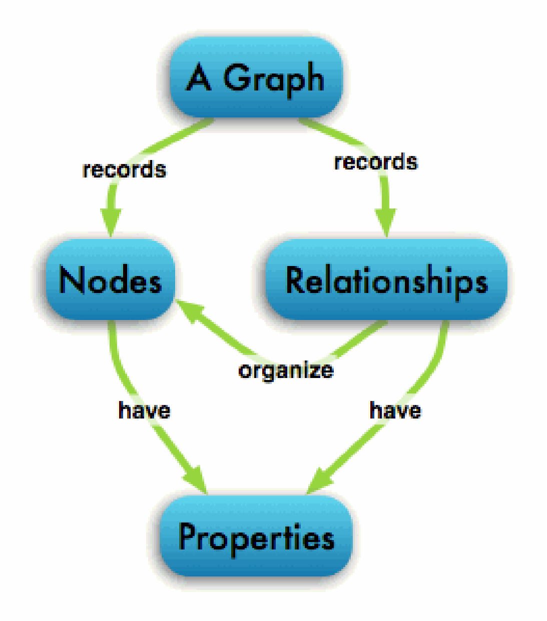
- **Performance** of queries in petabytes of data
- **Scalability** by adding horizontally systems
- **Availability** across the globe
- **High speed retrieval** independent of location
- **Data reliability** even when disks fail
- **Storage size** by harnessing the power of cheap disks
- **Versioning** of data

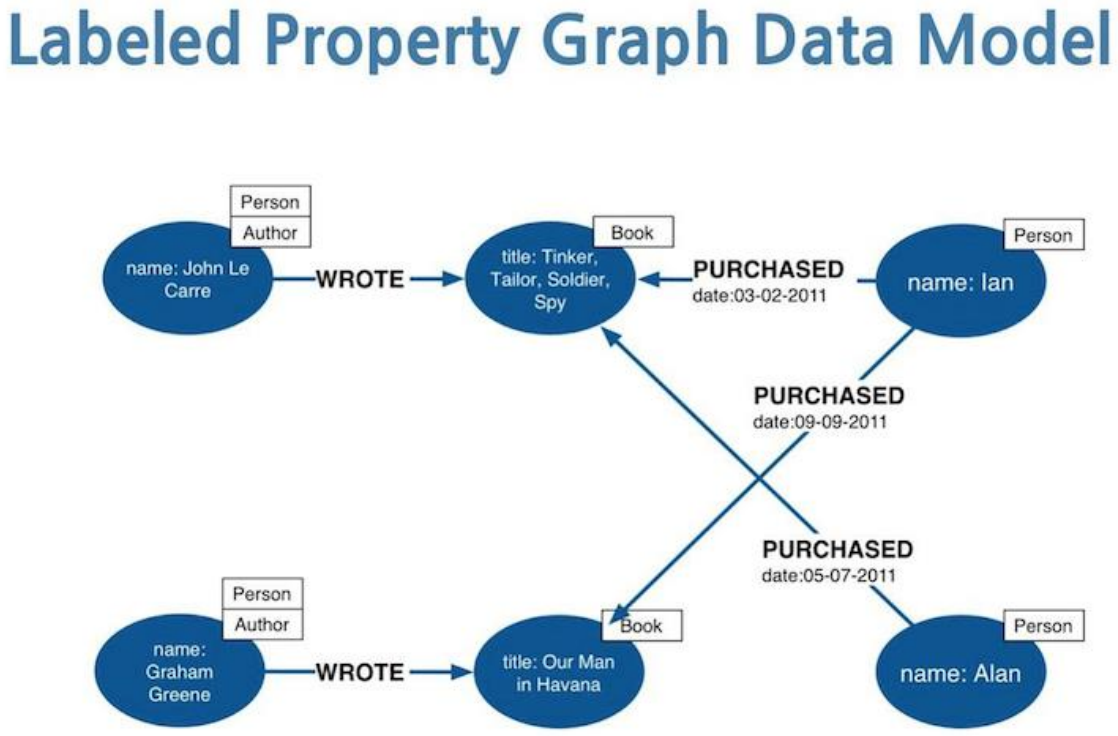
- No support for (RDBMS-style) multi-row transactions
- Offers no consistency guarantees for multi-row updates or cross-table updates (not even eventually)
- Lacks the freeform nature of JSON documents
- No join functionality

Databases for Web Applications

Graph Databases

While other databases compute relationships expensively at query time, a graph database stores **connections** as **first-class citizens**, readily available for any “join-like” navigation operation. Accessing those already persistent connections is an efficient, constant-time operation.




Labeled Property Graph Data Model

Databases for Web Applications

Graph Databases

- Example applications
  - Social networks – highly connected data
  - E-commerce – recommendations
  - Path finding – least cost path
  - Applications that suite on data-first schema (bottom-up approach)

- Powerful and simple data model
- Connected data locally indexed
  - Extremely fast for connected data
- Allows fast deep graph traversals
- Support for transactions
- Easy to query

- May not support sharding/network partitioning (high degree of interconnectedness b/w nodes)
- Requires rewiring your brain ☺
- Conceptual shift – different way of thinking
- Introducing more complexity

Databases for Web Applications

SQL-on-Hadoop Engines

With **SQL-on-Hadoop** technologies, it's possible to access big data **stored in Hadoop** by using the familiar **SQL language**. Users can plug in almost any reporting or analytical tool to analyze and study the data.

- Query data stored in HDFS or Hbase with standard SQL commands
- Query plan translates the request to a directed acyclic graph (DAG) of MapReduce tasks which can be executed by the MapReduce framework
- Impala even circumvent completely the MapReduce framework

Examples

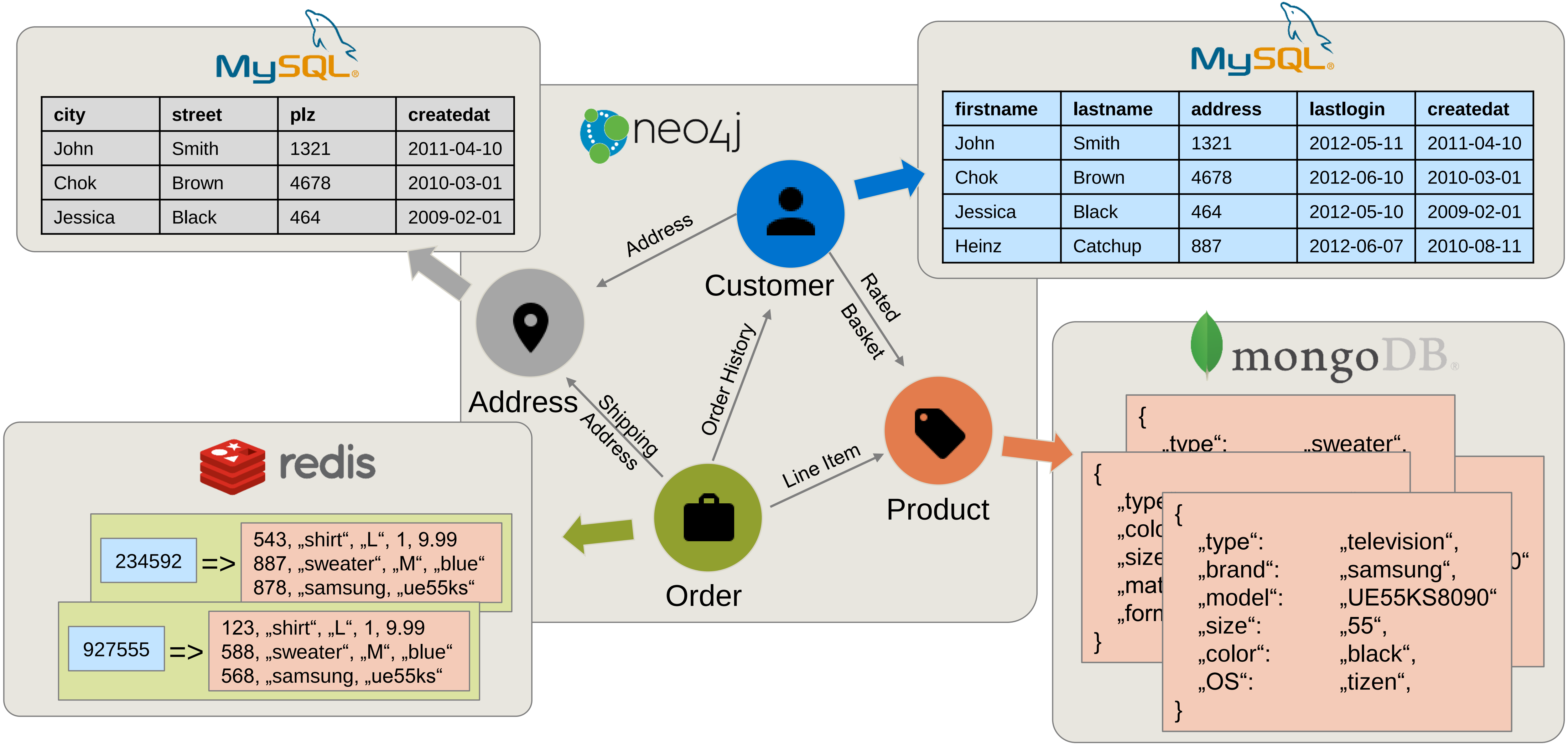
Apache Hive, CitusDB, Cloudera Impala, Concurrent Lingual, Hadapt, InfiniDB, JethroData, MammothDB, Apache Drill, MemSQL, Pivotal HawQ, Progress DataDirect, ScleraDB, Simba and Splice Machine.

- Easier to write SQL dialects than MapReduce jobs
- Low maintenance and simple to learn
- Low refactoring cost for existing applications
- New technologies with open problems

- Little control of storage
  - Most deployments must be over HDFS
  - Must support many different storage formats
- Little control of metadata and resource management
- No real-time queries and row level updates
- Not designed for OLTP

Databases for Web Applications

Polyglot Persistence – Example Scenario



city	street	plz	createdat
John	Smith	1321	2011-04-10
Chok	Brown	4678	2010-03-01
Jessica	Black	464	2009-02-01

firstname	lastname	address	lastlogin	createdat
John	Smith	1321	2012-05-11	2011-04-10
Chok	Brown	4678	2012-06-10	2010-03-01
Jessica	Black	464	2012-05-10	2009-02-01
Heinz	Catchup	887	2012-06-07	2010-08-11

```
234592 => 543, „shirt“, „L“, 1, 9.99
          887, „sweater“, „M“, „blue“
          878, „samsung, „ue55ks“

927555 => 123, „shirt“, „L“, 1, 9.99
          588, „sweater“, „M“, „blue“
          568, „samsung, „ue55ks“
```

```
{
  „type“:      „sweater“,
```

```
{
  „type
  „colo
  „size
  „mat
  „form
}
```

```
{
  „type“:      „television“,
  „brand“:     „samsung“,
  „model“:     „UE55KS8090“
  „size“:      „55“,
  „color“:     „black“,
  „OS“:        „tizen“,
}
```

Databases for Web Applications

Polyglot Persistence

If you have scenarios where you have a mixture of different data models, the idea of polyglot persistence comes up. **Polyglot persistence** is using different data storage technologies to handle varying data storage needs in order to stay performant.

If you have structured data with some differences
→ Use a document store

If you have a lot of relations between entities and want to query them efficiently
→ Use a graph database

If you manage the data structure yourself and do not need complex queries
→ Use a key-value store

If you have structured transactional data where all objects have equal attributes
→ Use a relational database

- Natural mapping of data into DB
- DB optimized for the data format
- Queries are tailored for your data format
- Focus on writing business logic
- Environment scales well

- Data has to be stored redundantly and has to be kept in sync
- Several technologies involved
- Administation effort is huge
- Know-how for several technologies required
- No cross queries capabilites

Databases for Web Applications

Multi-model Databases

A **multi-model database** is a database that can **store**, **index** and **query** data in more than one model. It supports several technologies such as: relational database, document-oriented database, graph database or triplestore and provides **one single backend layer** for accesing these data models.

Characeristics:

- „One size fits all“
- Parallel Async Execution
- Cross query capabilities
- All data is in the same datastore
- No synchronization is needed

- Handle multi-model data
- One System implements fault tolerance
- One system guarantees inter-modal data consistency
- Unified query language for multi-modal data
- Less server cost as only one database server needs to be replicated / distributed
- Less administration overhead

- A complex system with several integrated database engines
- Immature and developing
- Young technology with many challenges and open problems
- Small community compared to traditional databases

Databases for Web Applications

Multi-model Databases

Database	Key-value	SQL	Document	Graph	Object	Full-Text	Spatial	License	Transactions
OrientDB	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Apache 2 License	Full ACID, even distributed
ArangoDB	Yes	No	Yes	Yes	No	No	Yes	Apache 2 License	Full ACID, pessimistic locking, configurable durability
Couchbase	Yes	Yes	Yes	No	Yes	Yes	Yes	Apache 2 License	Single server or distributed
FoundationDB	Yes	Yes	Yes	Yes	No	No	No	Proprietary	Full ACID, multi-key, cross-node
MarkLogic	Yes	Yes	Yes	Yes	No	Yes	Yes	Proprietary	Full ACID
CrateDB	Yes	Yes	Yes	No	Yes	No	No	Apache 2 License	Eventual consistency, Optimistic concurrency control

Outline

8. High Performance Web Applications

- Database Options for Web Applications
  - NoSQL Databases
  - Multi-model and Polyglot Persistence
- Scaling Web Applications
  - Bottlenecks
  - Scalability Patterns
  - Scaling with MongoDB

Bottleneck - Definition

- A **bottleneck** has the same meaning as a bottleneck in traffic systems: a section or a route with a **carrying capacity substantially below** that of the other sections of the route.

- Tasks within your **critical path** that consume the most time.

- The critical path varies from task to task but is **common across several user paths.**

- *Example*:
  A database-backed system in which you spend 60 percent of the request-response cycle waiting for the database to respond.

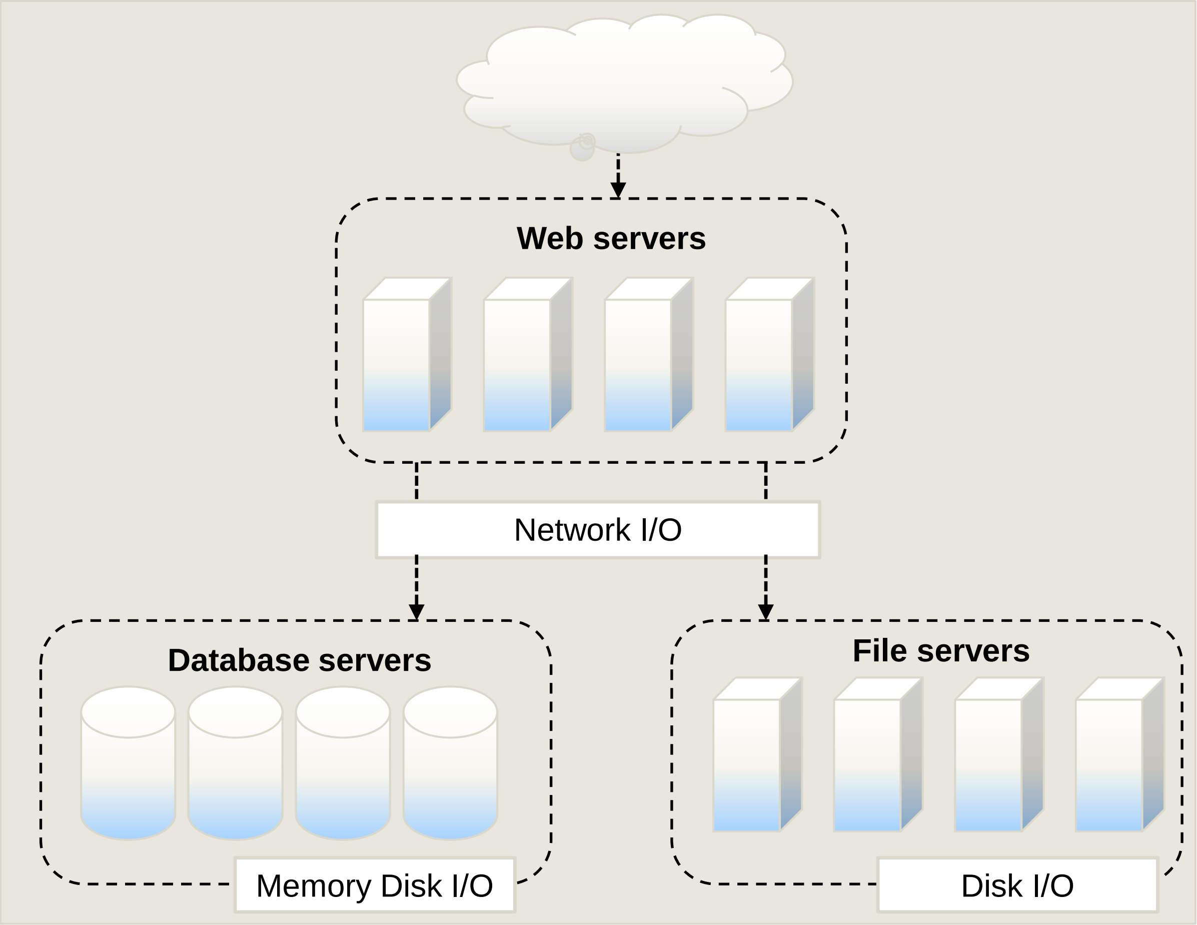
When to deal with bottlenecks?

- You should never spend time optimizing a component of your application we only *feel* will be a bottleneck.
- "**Premature optimization** is the root of all evil (or at least most of it) in programming." [Donald Knuth]
- Before spending time on improving the performance of a system we need to step back and decide what components would benefit most of our attention.
- *Example*: Spending a long time working on a fix that gives us a 1 percent performance boost, while we could have gotten 10 percent in half the time.

  → We need to identify our bottlenecks.

Application Areas by Software Component

- Break your application down into **logical** *components.*
- *Example*: logical components of Flickr



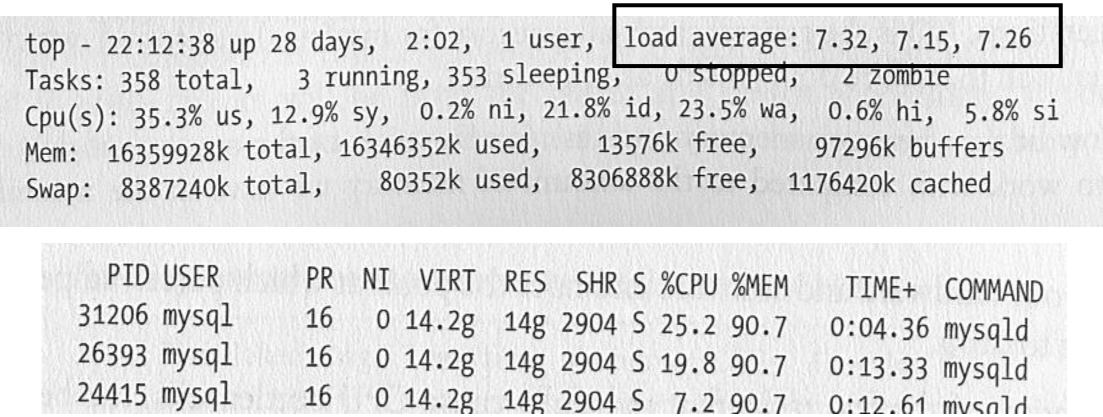
Application Areas by Hardware Component

- After identifying logical components, they can be examined in detail.
- How is the time spent within each component?
- The execution path within the component level has to be identified **down to the hardware level**.

- Areas to be considered: CPU, disk I/O, memory I/O, network I/O, memory and swap, external services and black boxes, and databases.

CPU Usage

- CPU processing speed is almost never a bottleneck in web applications.
- Exceptions: processing image, audio, or video data, cryptography.
- Simple tool to get an overview of who’s consuming CPU time: *top* under Unix:



```
top - 22:12:38 up 28 days,  2:02,  1 user,  load average: 7.32, 7.15, 7.26
Tasks: 358 total,   3 running, 353 sleeping,   0 stopped,   2 zombie
Cpu(s): 35.3% us, 12.9% sy,  0.2% ni, 21.8% id, 23.5% wa,  0.6% hi,  5.8% si
Mem:  16359928k total, 16346352k used,    13576k free,    97296k buffers
Swap:  8387240k total,    80352k used,  8306888k free,  1176420k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31206 mysql     16   0 14.2g  14g 2904 S 25.2 90.7   0:04.36 mysqld
26393 mysql     16   0 14.2g  14g 2904 S 19.8 90.7   0:13.33 mysqld
24415 mysql     16   0 14.2g  14g 2904 S  7.2 90.7   0:12.61 mysqld
```

- The *load average* statistics (average of the last 1, 5, and 15 minutes) is a good quick indicator of the general state of the machine:
  - Counts the number of threads in the running or runnable states → if the load average is above the number of processes, there are processes in the queue waiting to run.
  - For systems with multiple CPUs, the *load average* should be divided by the number of processors in order to get a comparable statistics.
  - A high load average makes a machine unresponsive or slow.

[“Building scalable web sites - building, scaling, and optimizing the next generation of web applications.” Cal Henderson (2006)]

Using Profilers to Understand CPU Consumption – Example: Flame Graphs

Flame graphs are a visualization technique used to profile an application and rapidly and more precisely spot the most frequently used functions. These graphs replace or complement the previous log text output, as they give a more pleasant and simple way of profiling.

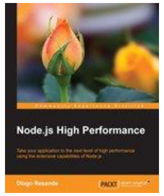
As a rule of thumb, a flame graph with a huge and wide flame means excessive CPU usage. A flame graph with high but thinner flames means low CPU usage.

Existing tools

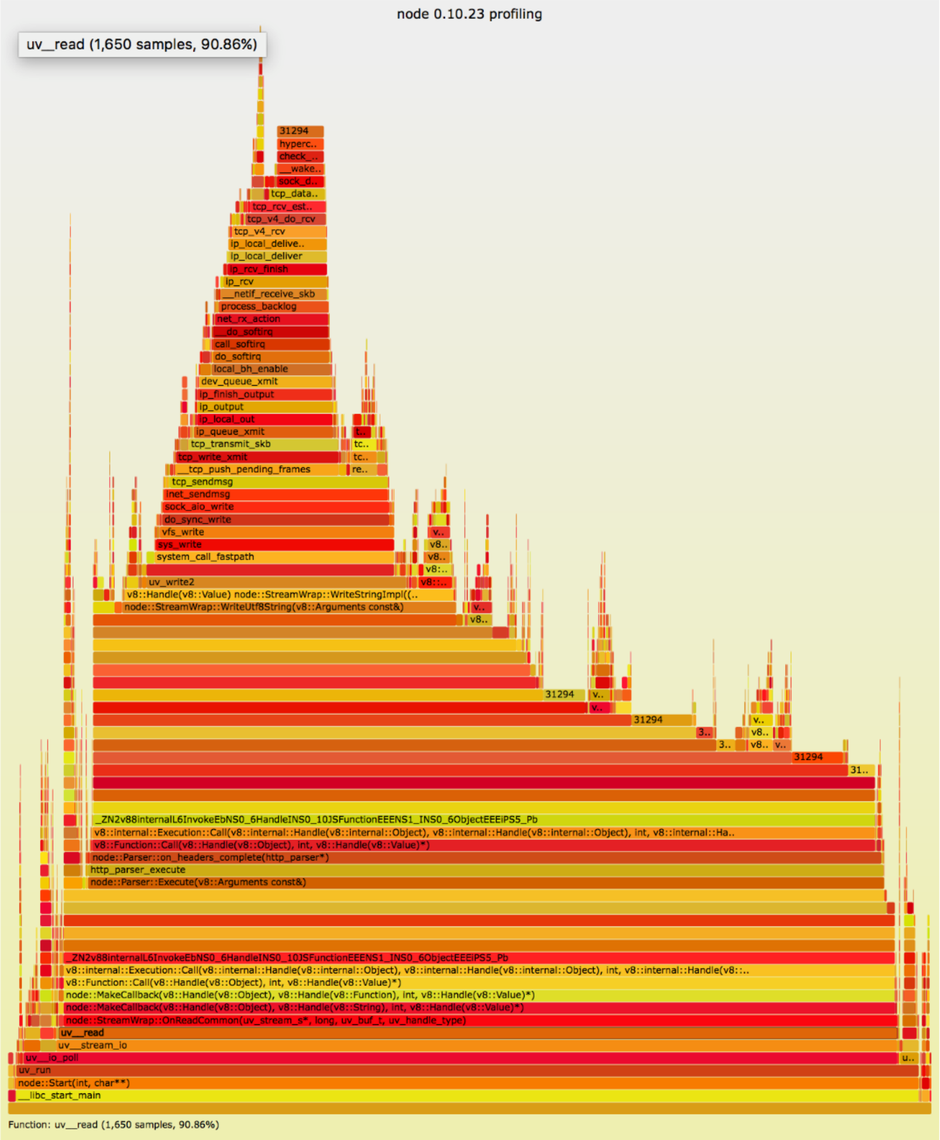
- perf
- eBPF
- SystemTap
- ktap

- DTrace
- Instruments

- Xperf.exe



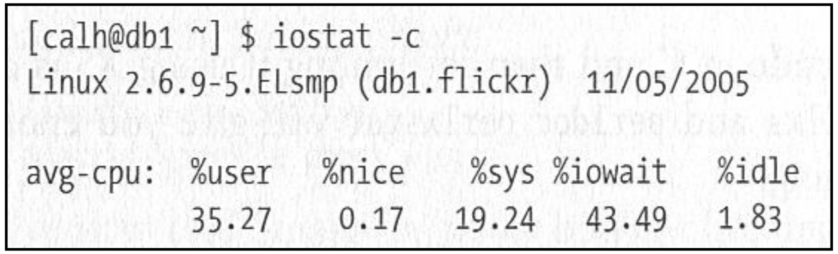
Node.js High Performance
By: Diogo Resende
Publisher: Packt Publishing
Pub. Date: August 19, 2015
Print ISBN-13: 978-1-78528-614-8
Web ISBN-13: 978-1-78528-062-7



Source: http://www.brendangregg.com/blog/2014-09-17/node-flame-graphs-on-linux.html

Disk I/O

- Covers the time spent reading and writing data from disk.
- Assuming an uncached disk, a read request has to seek to find the data, read the data from the disk platter, and move the data from disk into memory.
- Under Unix environments, disk I/O can be examined using the *iostat* utility:



```
[calh@db1 ~] $ iostat -c
Linux 2.6.9-5.ELsmp (db1.flickr)   11/05/2005

avg-cpu:  %user   %nice    %sys %iowait   %idle
          35.27    0.17   19.24   43.49    1.83
```

- *iowait* tells us how much time the processor has spent being idle while waiting for an I/O operation to complete.
- The main limiting factor for disk I/O is the speed by which the disk actually spins.
- *Example*: it takes 5,5 ms on a 5400 rpm disk, but only 2 ms on a 15000 rpm disk to seek the opposite side of the disk
- Upgrading the speed of the disk tends to be a lot cheaper than buying more disks or more machines

[“Building scalable web sites - building, scaling, and optimizing the next generation of web applications.” Cal Henderson (2006)]

Network I/O

- Network I/O refers to the rate of data that can be pushed between two hosts.

- Tool to examine network I/O: *netstat*

- Network I/O should rarely become an issue in production systems unless:
  - something has been misconfigured
  - you’re serving a huge amount of traffic
  - you have very special application requirements.

Outline

8. High Performance Web Applications

- Database Options for Web Applications
  - NoSQL Databases
  - Multi-model and Polyglot Persistence
- Scaling Web Applications
  - Bottlenecks
  - Scalability Patterns
  - Scaling with MongoDB

Example Scenario

- 10 million pages a day plus another few million Ajax interactions.

→ 116 requests per second in average (may reach double or triple of that depending on the traffic profile)

- 10 data queries per page on average

→ 1000 queries per second (QPS), or 3000 QPS at peak.

How to design a system to reach that rate, scale past it and does it all in a reliable and redundant way?

What is Scalability?

- A scalable system has three simple characteristics:
  1. The system can accommodate increased usage.
  2. The system can accommodate an increased dataset.
  3. The system is maintainable.

- Scalability is not raw speed; you can have a fast system that doesn’t scale.

- Scalability is not about using any specific technology (e.g., Java, XML), but rather based around a few core principles.

Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

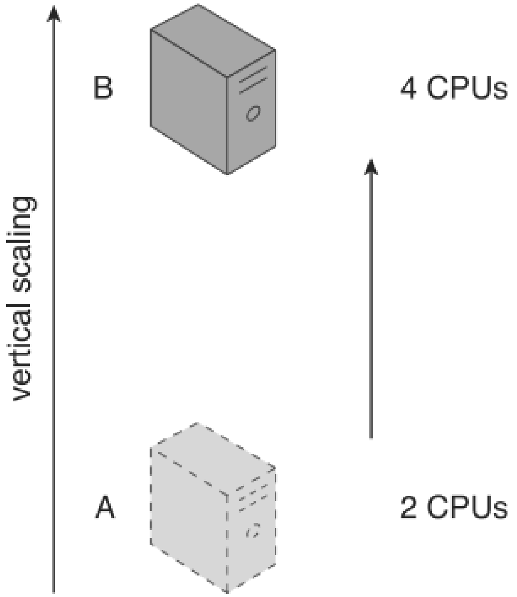
["Characteristics of scalability and their impact on performance." Bondi, André B. (2000)]

Vertical Scaling

When an existing IT resource is replaced by another with higher or lower capacity, vertical scaling is considered to have occurred

- Specifically, the replacing of an IT resource with another that has a higher capacity is referred to as **scaling up**
- Replacing an IT resource with another that has a lower capacity is considered **scaling down**

- **Pro**: it is very easy to design for vertical scaling
  → if you're certain of the ceiling for your application's usage, then vertical scaling can be a fast alternative to building truly scalable systems.

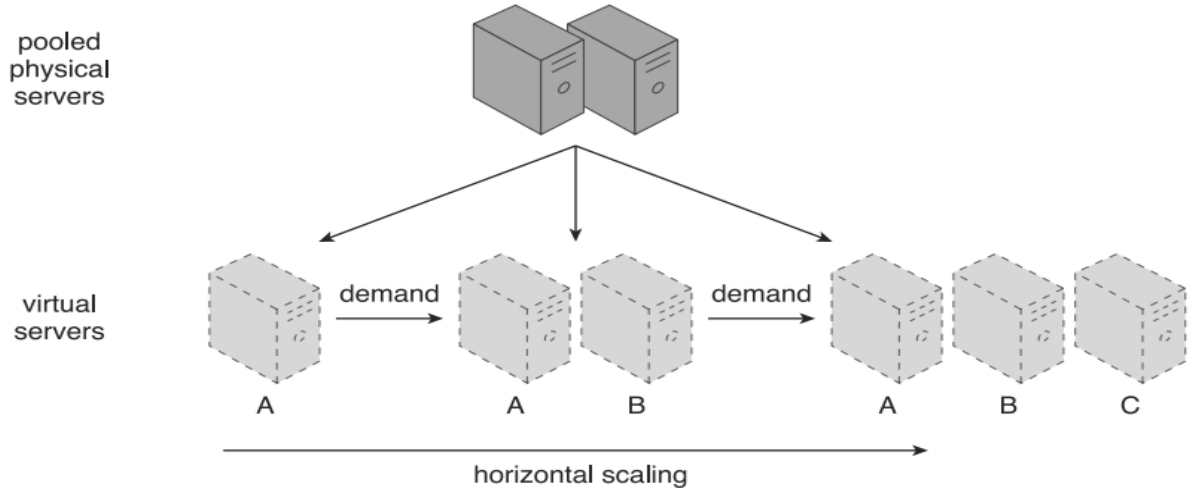


Horizontal Scaling

The allocating or releasing of IT resources that are of the same type is referred to as horizontal scaling.

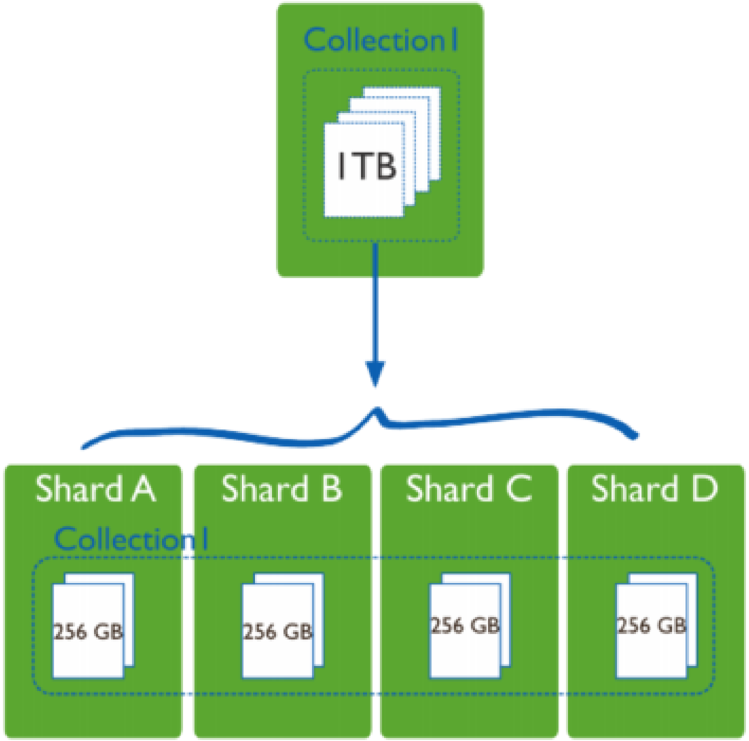
- The horizontal **allocation of resources** is referred to as **scaling out**
- Horizontal **releasing of resources** is referred to as **scaling in**
- Horizontal scaling is a common form of scaling within cloud environments

- Issue with horizontal scaling: **administration cost**
- Assumption: software configuration of all machines is exactly the same!
  → administration doesn’t scale linearly

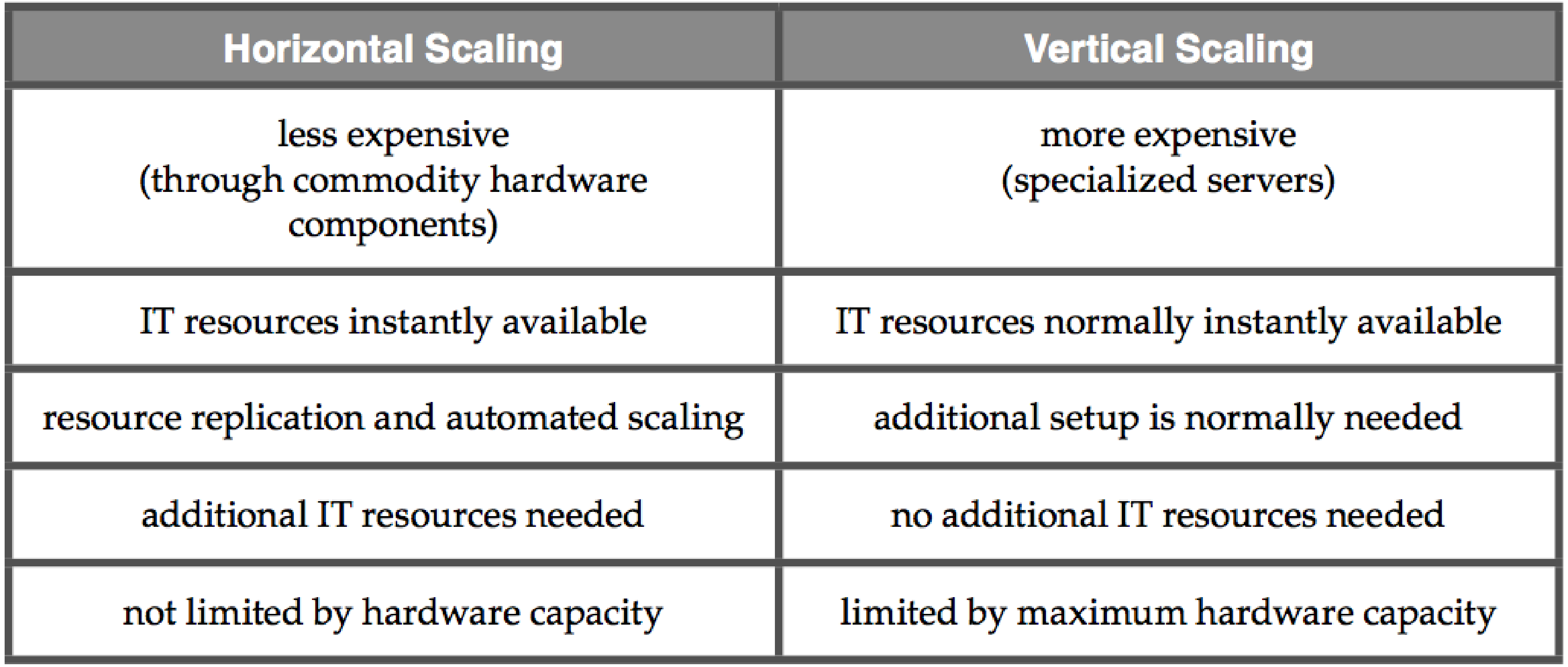


Horizontal Scaling in Databases

- Sharding (horizontal scaling) divides the **data set** and distributes the data over multiple servers, or shards
- Each shard is an independent database, and collectively, the shards make up a single logical database
- Sharding addresses the challenge of scaling to support **high throughput** and **large data sets**

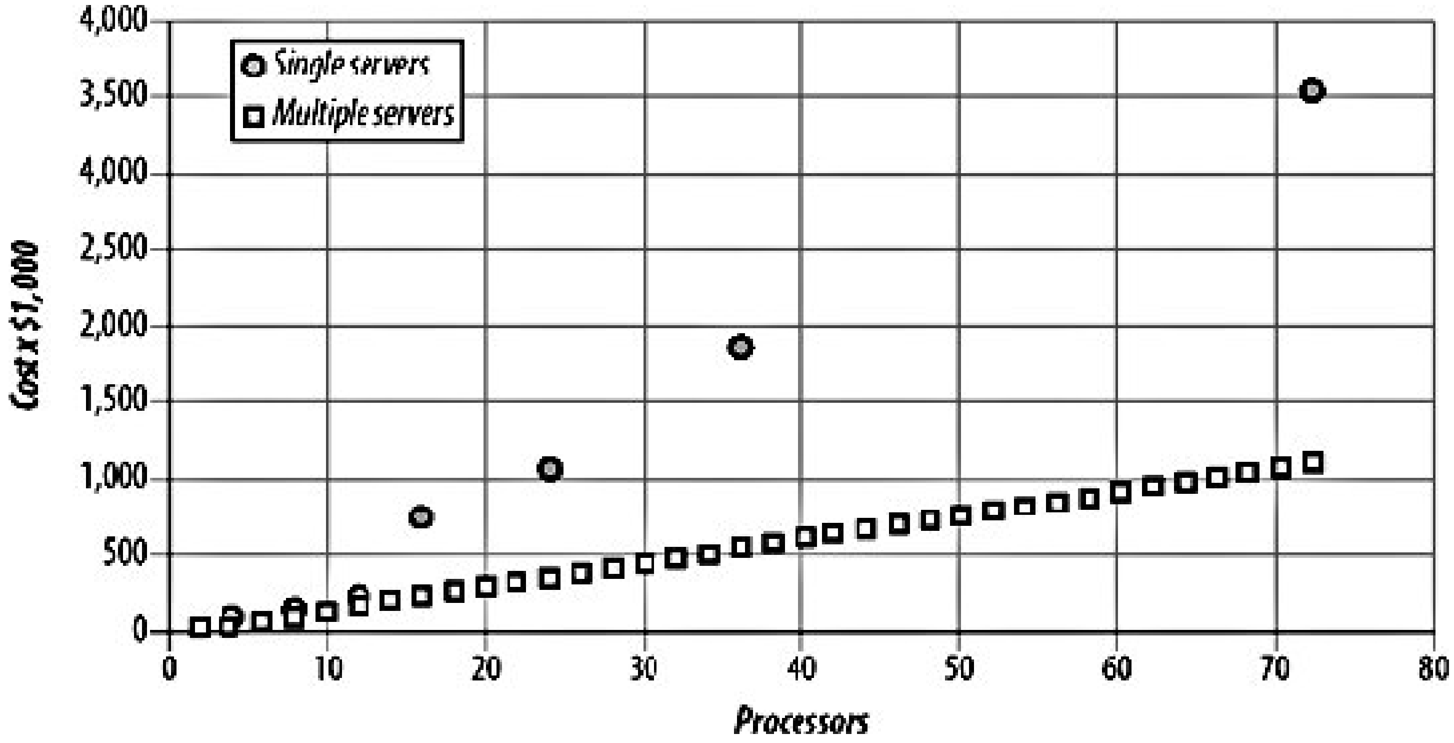


Horizontal vs Vertical Scaling



Horizontal Scaling	Vertical Scaling
less expensive (through commodity hardware components)	more expensive (specialized servers)
IT resources instantly available	IT resources normally instantly available
resource replication and automated scaling	additional setup is normally needed
additional IT resources needed	no additional IT resources needed
not limited by hardware capacity	limited by maximum hardware capacity

Horizontal and Vertical Scaling



Vertical scaling problem: the cost doesn’t scale linearly but exponential and there is a limit of commercially available servers

[“Building scalable web sites - building, scaling, and optimizing the next generation of web applications.” Cal Henderson (2006)]

Passive Redundancy

- Machines can fail: Google expects one of 10000 machines to fail completely each day.

- Your design needs to be able to cope with every component failing!
  → redundant hardware is needed; the spare standby pieces can be cold, warm, or hot.

- **Cold** means, that it requires setup and configuration (either physical or in software) before it can take over for the failed component.

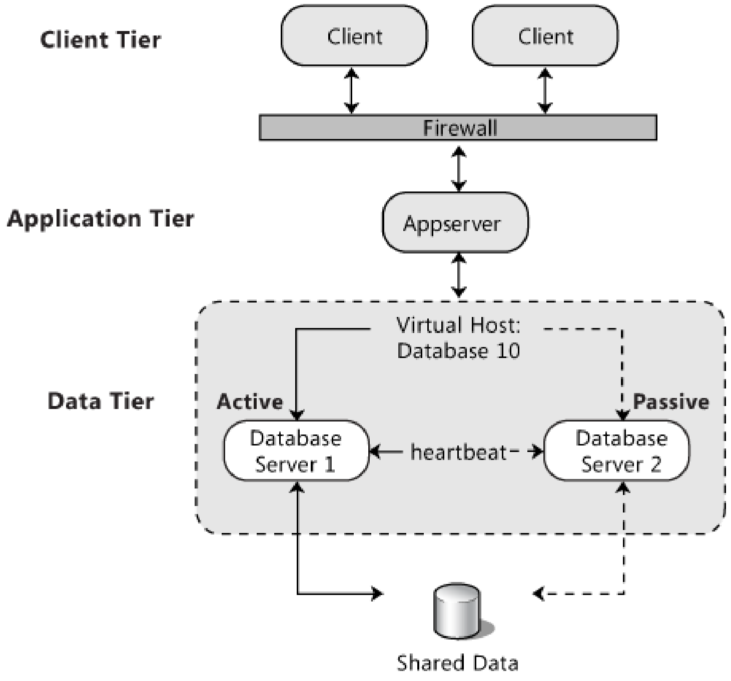
- A **warm** spare is a piece of hardware that is all configured for use and just needs to be flipped on to start it (manually).
  - **Example**: a MySQL system with a production master and a backup slave; the slave is not used in production but is ready to go immediately

- A **hot** spare component automatically takes over: the dead component is detected, and the transition happens without any user intervention.
  - **Example**: an active balancer is taking all traffic, talking to the backup balancer via a monitoring protocol; if the active one fails and the passive balancer stops getting a heartbeat it immediately knows how to take over and starts receiving and processing traffic.

Active Redundancy

- In an active/passive redundant pair, we have **one online** production device and one hot backup not being used.

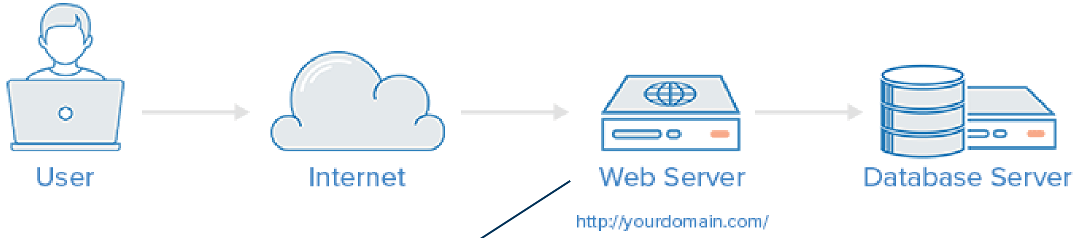
- In an active/active pair, we use both devices **simultaneously**, moving all traffic to a single device when the other fails.



Load Balancing (1)

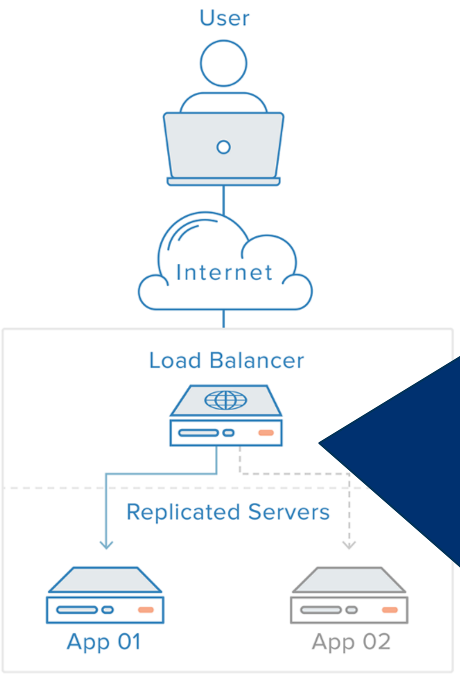
Load balancing is a key component of highly-available infrastructures commonly used **to improve the performance and reliability** of web sites, applications, databases and other services **by distributing the workload across multiple servers**.

No load balancing



Single point of failure

With load balancer

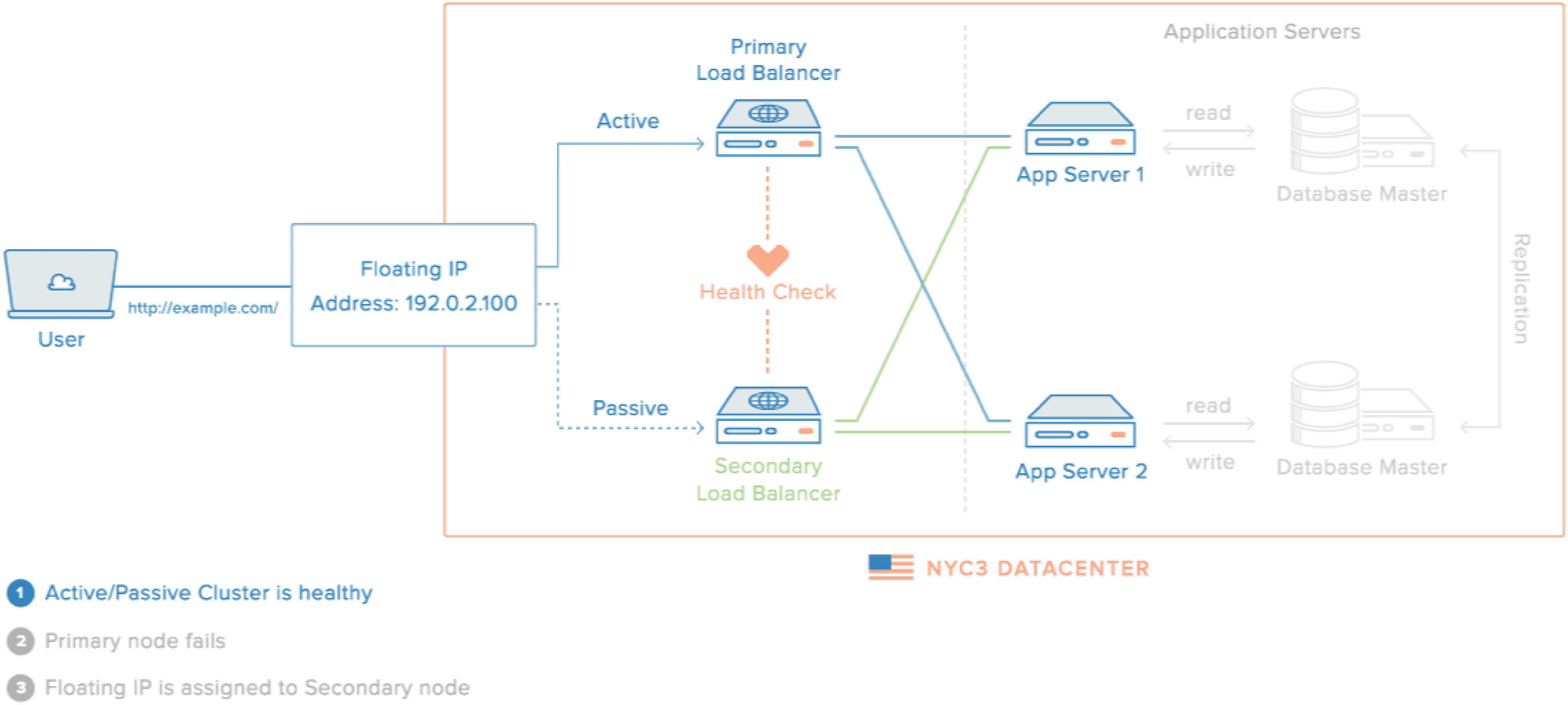


- Performs a server health check
- Chooses a server based on load balancing algorithm:
  - Round robin
  - Least connections
  - Source
- Depending on the architecture constraints state handling can be required:
  - Sticky sessions (load balancer enriches requests with required headers)

Source: https://www.digitalocean.com/community/tutorials/what-is-load-balancing

Load Balancing (2)

In action



1. Active/Passive Cluster is healthy
2. Primary node fails
3. Floating IP is assigned to Secondary node

Source: https://www.digitalocean.com/community/tutorials/what-is-load-balancing

Outline

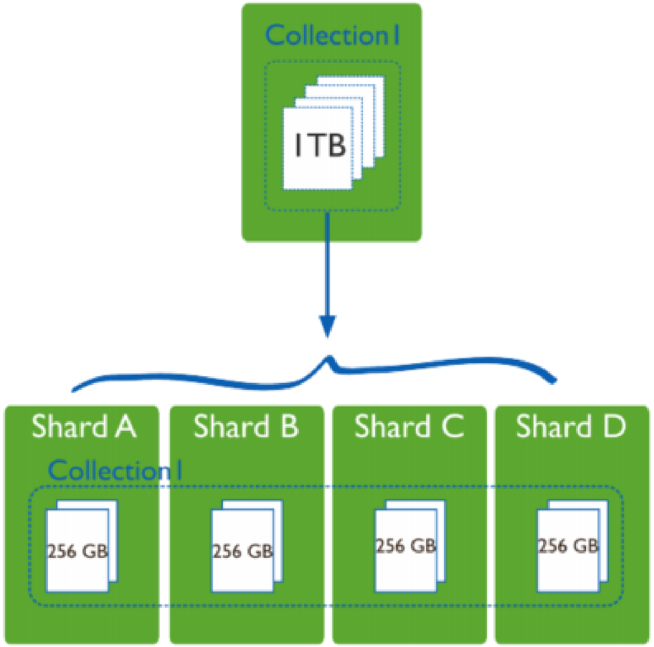
8. High Performance Web Applications

- Database Options for Web Applications
  - NoSQL Databases
  - Multi-model and Polyglot Persistence
- Scaling Web Applications
  - Bottlenecks
  - Scalability Patterns
  - Scaling with MongoDB

Sharding (1)

- Sharding divides a **data set** and distributes the data over multiple servers, or shards
- Each shard is an *independent database*, and collectively, the shards make up a single logical database
- Sharding addresses the challenge of scaling to support **high throughput** and **large data sets**

- **Available implementations:**
  - MySQL Fabric
  - Postgres - pg_shard
  - Oracle’s RAC (Real Application Cluster) – quite expensive
  - MongoDB sharded clusters
  - Graph database - optimally partitioning large graphs across a set of servers is near-impossible (NP complete) problem!

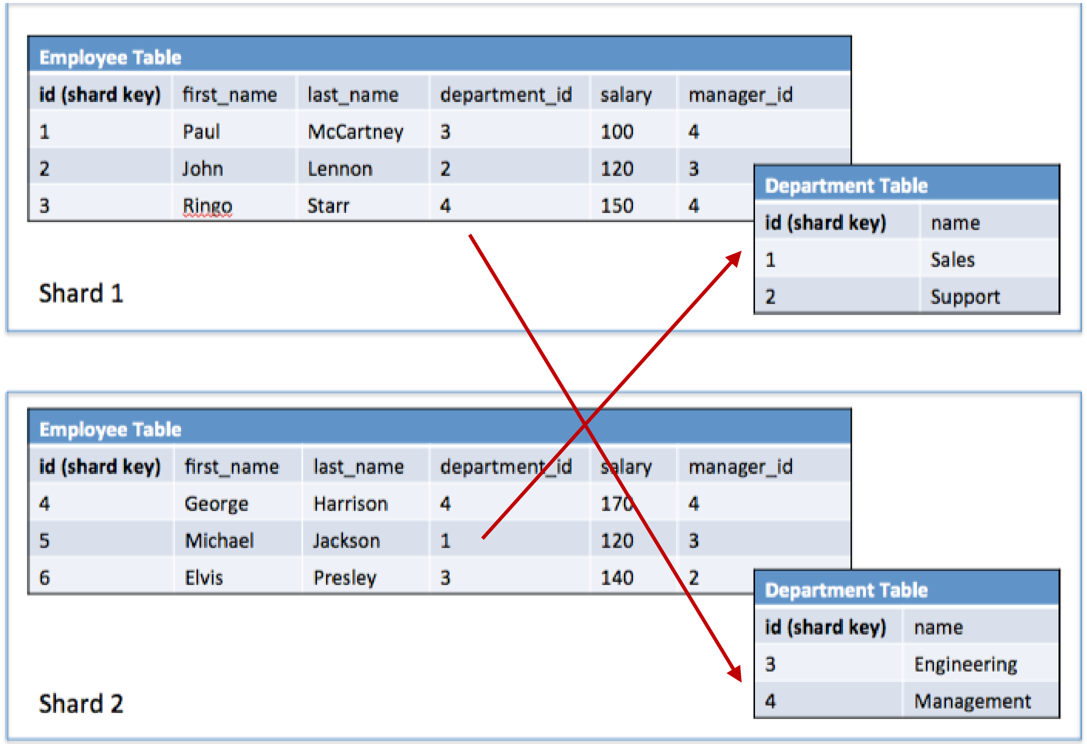


Sharding (2)

Challenges

- Distributed queries
- Avoidance of cross-shard joins
- Handling sub-queries
- Auto-increment primary key management
  - *Keys in MongoDB (12-byte): 4-byte timestamp, 3-byte machine id, 2-byte process id, and 3-byte counter*
- Determine the optimum method for sharding the data

Example #1
Choosing 'id' as the shard key presents a shard conflict, because *there is no guarantee that all employees are in the same shard as their corresponding departments*.



Shard 1

Employee Table

id (shard key)	first_name	last_name	department_id	salary	manager_id
1	Paul	McCartney	3	100	4
2	John	Lennon	2	120	3
3	Ringo	Starr	4	150	4

Department Table

id (shard key)	name
1	Sales
2	Support

Shard 2

Employee Table

id (shard key)	first_name	last_name	department_id	salary	manager_id
4	George	Harrison	4	170	4
5	Michael	Jackson	1	120	3
6	Elvis	Presley	3	140	2

Department Table

id (shard key)	name
3	Engineering
4	Management

[Example from ScaleBase]

Sharding (3)

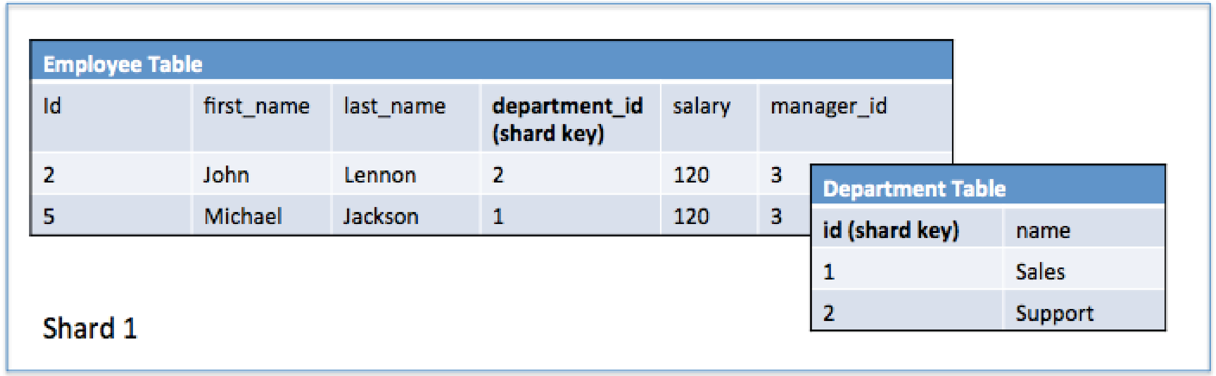
Challenges

Example #2

Choose ‘department_id’ as the ‘Employee Table’ shard key

The outcome

- The join query was optimized as a result of all department-related data being stored in the same partition
- No cross-joins exist between partitions
- Statements can now safely be executed on all partitions



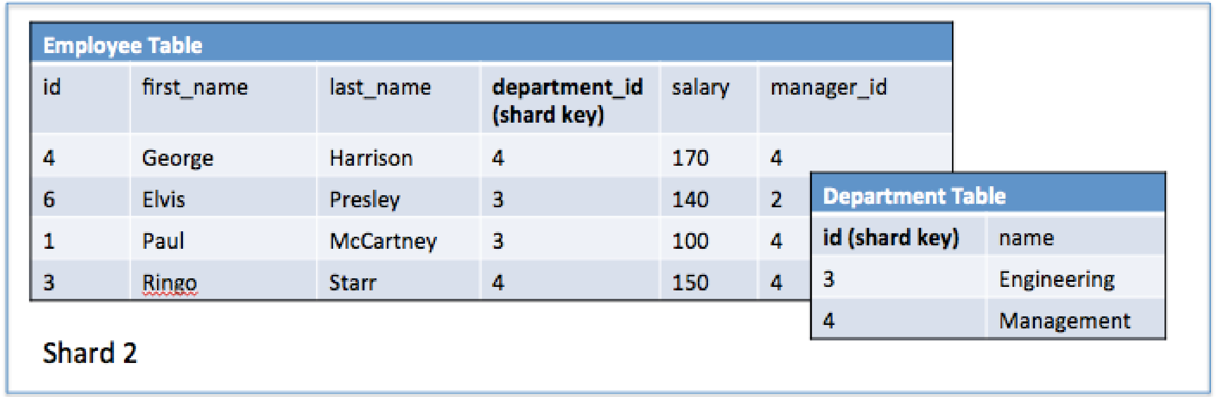
Employee Table

Id	first_name	last_name	**department_id (shard key)**	salary	manager_id
2	John	Lennon	2	120	3
5	Michael	Jackson	1	120	3

Department Table

id (shard key)	name
1	Sales
2	Support

Shard 1



Employee Table

id	first_name	last_name	**department_id (shard key)**	salary	manager_id
4	George	Harrison	4	170	4
6	Elvis	Presley	3	140	2
1	Paul	McCartney	3	100	4
3	Ringo	Starr	4	150	4

Department Table

id (shard key)	name
3	Engineering
4	Management

Shard 2

[Example from ScaleBase]

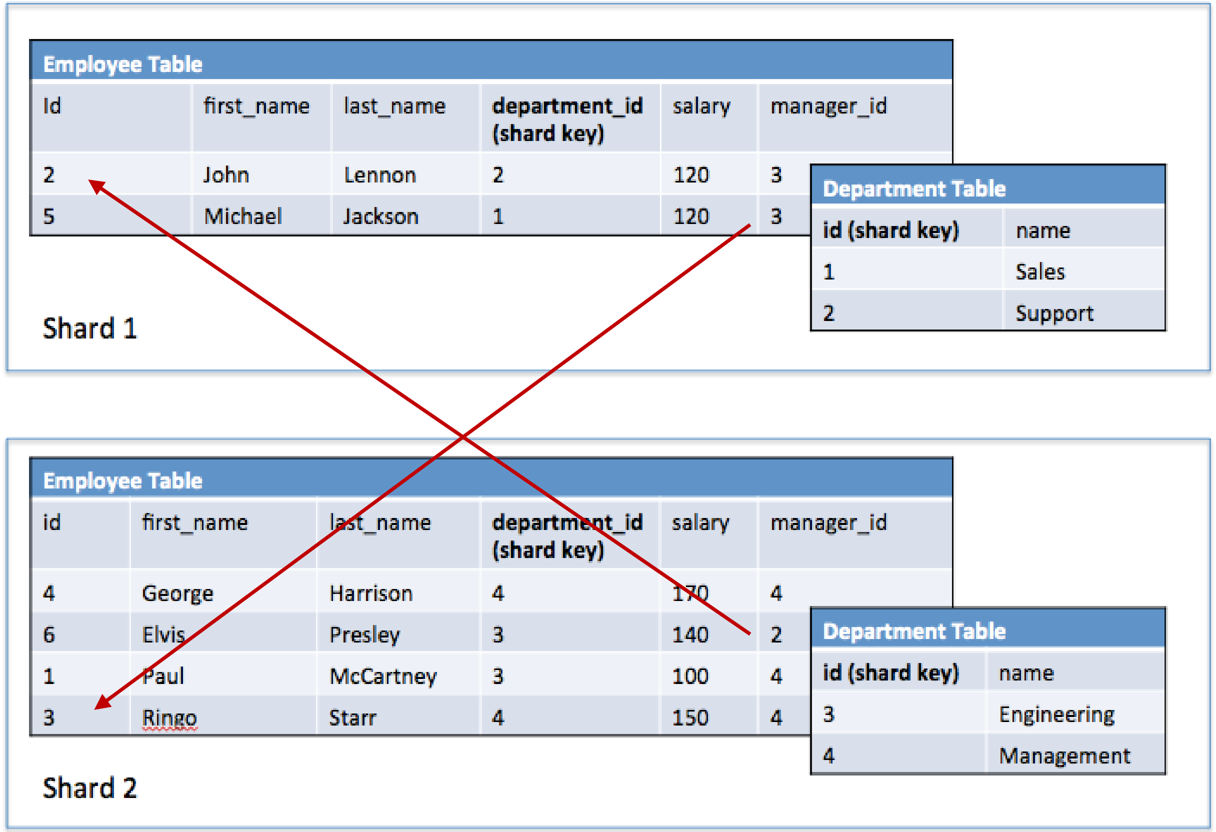
Sharding (4)

Challenges

‘Select e.first_name, e.last_name, m.first_name, m.last_name from employee e join employee m on e.manager_id=m.id’

Example #3
Join the ‘Employee Table’ together with itself to find a manager ➔ there is no guarantee they are in the same shard.

The employee tables are not capable of being sharded by both ‘id’ and ‘manager_id’ at the same time.



Employee Table

Id	first_name	last_name	**department_id (shard key)**	salary	manager_id
2	John	Lennon	2	120	3
5	Michael	Jackson	1	120	3

Department Table

id (shard key)	name
1	Sales
2	Support

Shard 1

Employee Table

id	first_name	last_name	**department_id (shard key)**	salary	manager_id
4	George	Harrison	4	170	4
6	Elvis	Presley	3	140	2
1	Paul	McCartney	3	100	4
3	Ringo	Starr	4	150	4

Department Table

id (shard key)	name
3	Engineering
4	Management

Shard 2

Distributed data can become quite complex if not handled correctly!

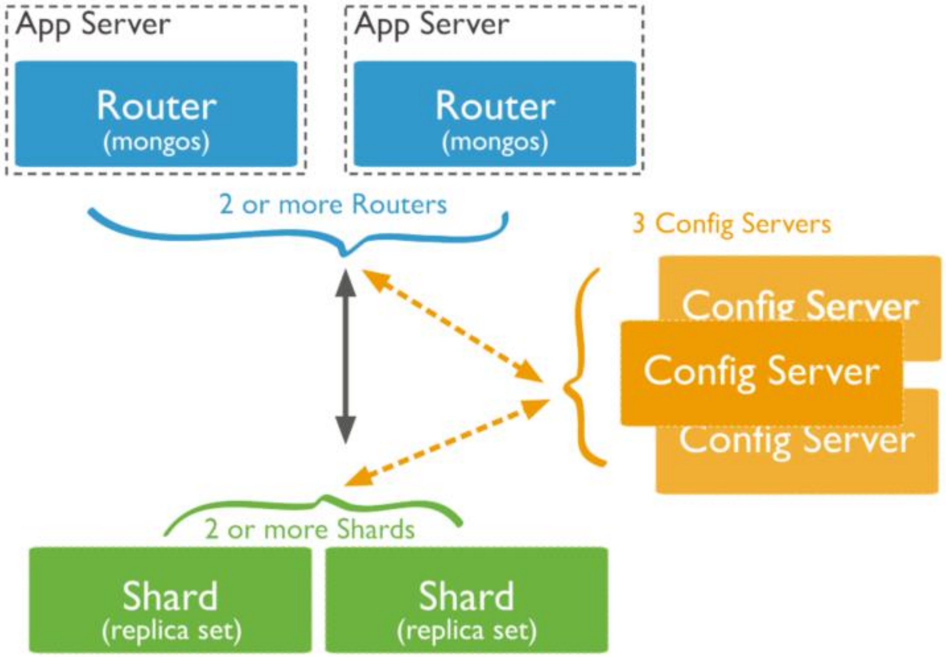
[Example from ScaleBase]

Sharding in MongoDB

MongoDB supports sharding through the configuration of a **sharded clusters**.

Sharded cluster has the following components:

- **Shards** store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set (group of mongodb instances that maintain the same data set; *covered in next section*).
- **Query routers or mongos** is a router that determines where reads and writes go.
- **Config servers** store the cluster's metadata. It persists the location of particular shard key ranges.



Sharded Cluster

Query routers

- Mongos has *no* persistent state and knows only what data is in which shard by tracking the metadata information stored in the configuration servers.
- Mongos performs operations in sharded data by *broadcasting* to all the shards that holds the documents of a collection or target only shard based on the shard key. Usually, multi-update and remove operations are a broadcast operations.

Config server

- If config server goes down ➔ the entire shard cluster goes down
  - Create replica sets of config server [upto 50 servers to ensure availability]
- It updates the metadata information whenever there is a chunk split or chunk migration.

Steps to Enable Sharding

- MongoDB distributes data at the *collection level*. Sharding partitions a collection’s data by the **shard key**.

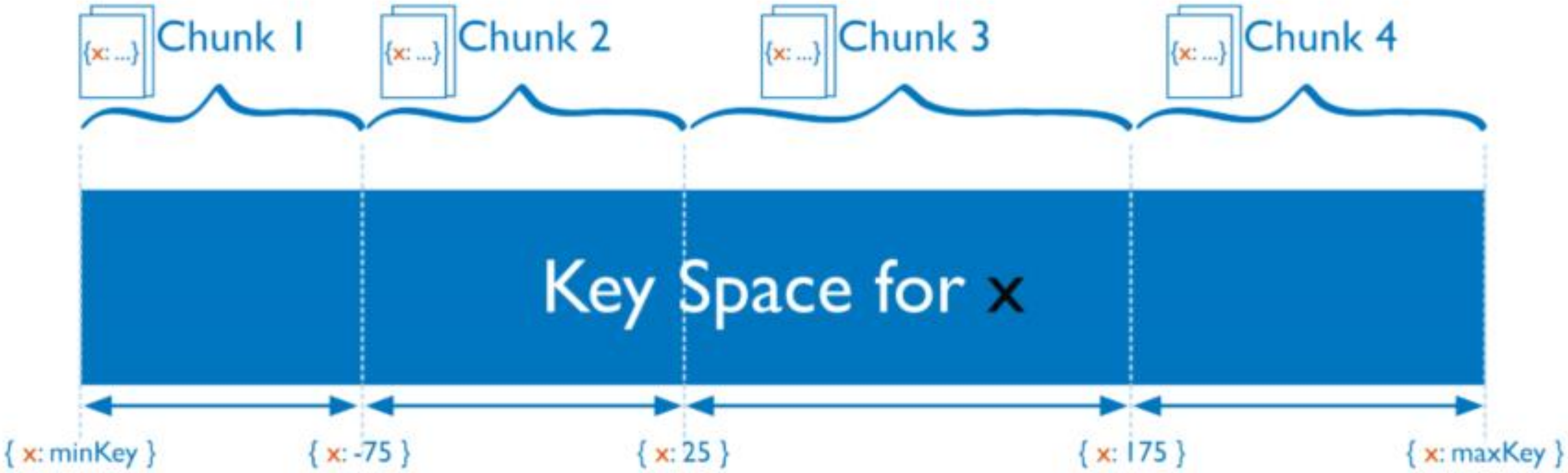
1. Determine which field(s) will be used for the shard key
2. Create a unique index on the key field(s)
   - db.databaseName.collectionName.ensureIndex({ _id: “hashed” })
3. Enable sharding on the collection
   - sh.shardCollection(“databaseName.collectionName”, { _id: “hashed” })

- Shard key cannot exceed *512 bytes*.
- Shard key index cannot be a *text* index or a *geospatial* index.
- Shard key is immutable!

Data Partitioning

Range-based Sharding

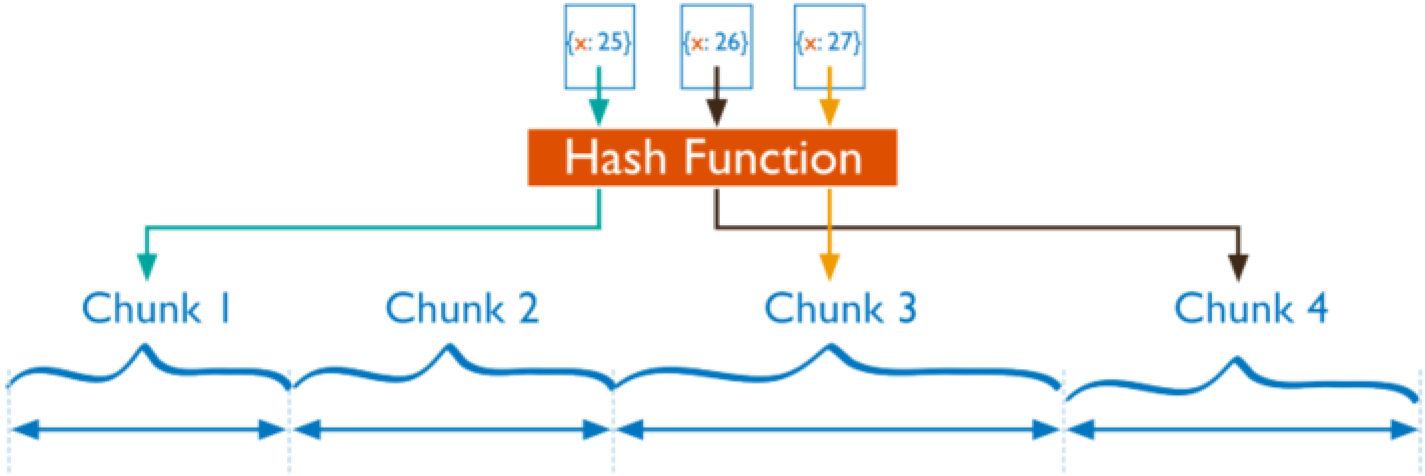
- **Numeric shard key**



Data Partitioning

Hash-based Sharding

- Range-based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding
- Hash-based partitioning ensures an even distribution of data at the expense of efficient range queries
  - Hashed key values result in random distribution of data across chunks and therefore shards
- Random distribution makes it more likely that a range query on the shard key will not be able to target a few shards but would more likely query every shard in order to return a result



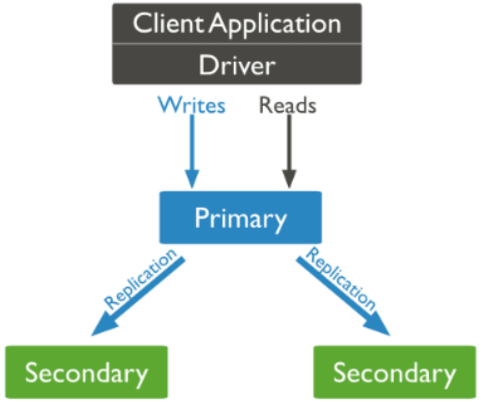
Replication in MongoDB

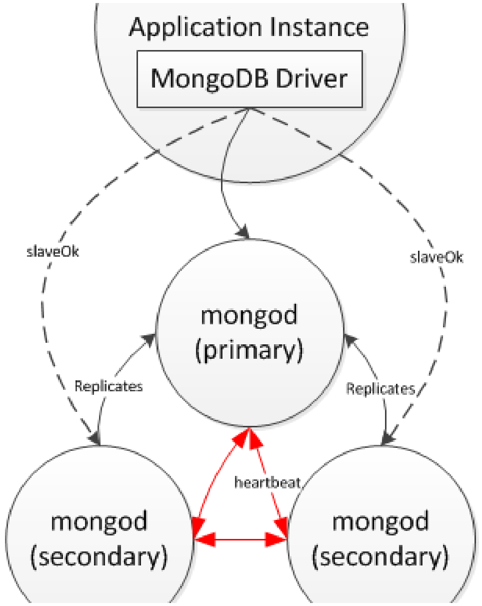
- A **replica set** is a group of mongod instances that maintain the same data set.

- A replica set contains several data-bearing nodes
  - One and only **primary** node
  - **Secondary** nodes
  - Optionally one arbiter node
    - do not maintain a data set
    - plays a role in the election of a primary during election process

- Replica sets have automatic failover
  - Heartbeat manages automatic failover

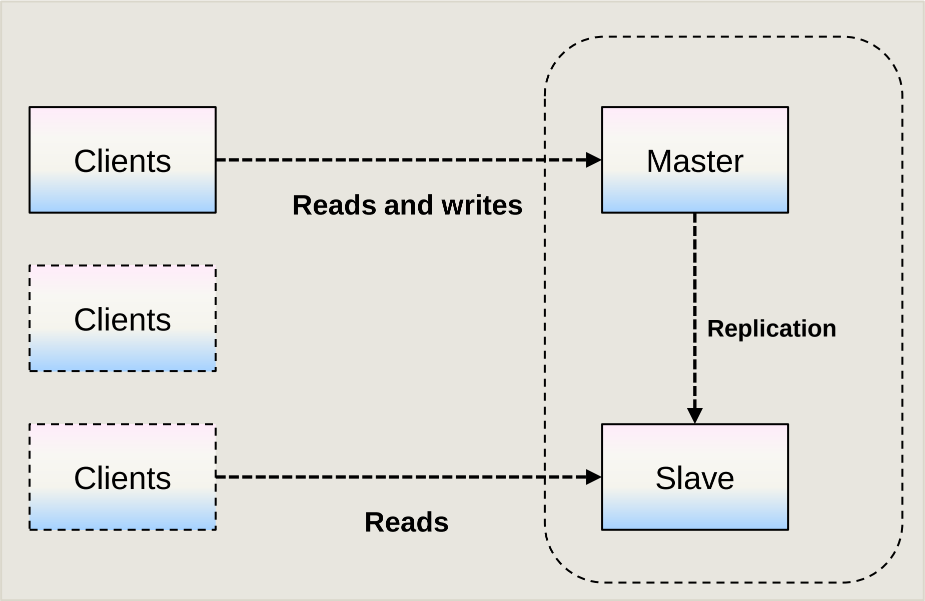
A replica set can have up to 50 members





Primary-secondary Replication

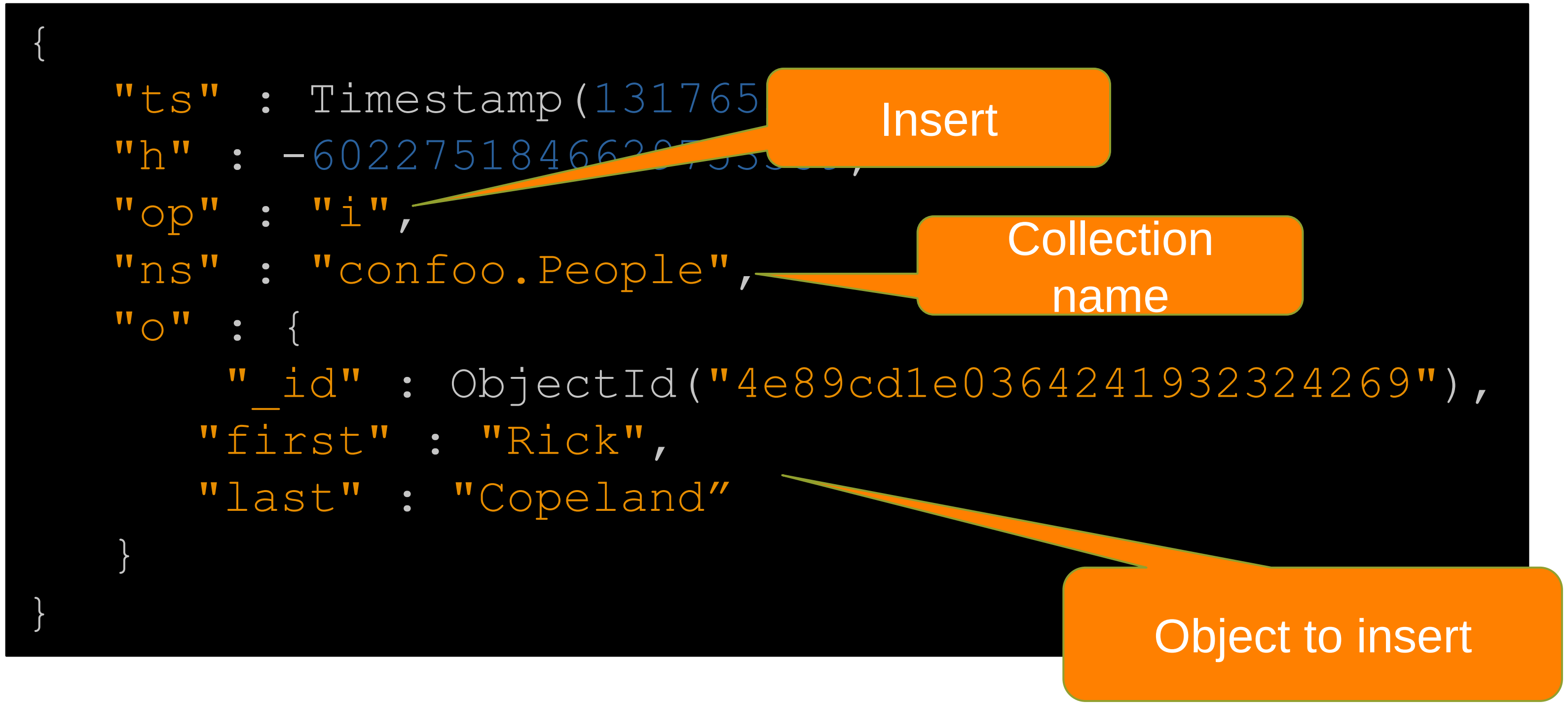
- Primary handles all writes
  - Driver applies write on the primary
  - Records the operations on the primary’s **oplog** (operation log)
  - Secondary nodes replicate this log and apply the operations to their data sets
    - This is an asynchronous operation



Replication

The operation log (oplog)

- Special collection (oplog) records operations idempotently
- Secondaries read from primary oplog and replay operations locally
- Space is pre-allocated and fixed for the oplog



```
{
    "ts" : Timestamp(131765
    "h" : -602275184662[illegible],
    "op" : "i",
    "ns" : "confoo.People",
    "o" : {
        "_id" : ObjectId("4e89cd1e0364241932324269"),
      "first" : "Rick",
      "last" : "Copeland”
    }
}
```

Secondary Nodes

- All members of the replica set can accept read operations.
  - By default primary node handles the read operation
  - *Read preferences* can be changed

- Read preference mode
  - primary - default mode
  - primaryPreferred – if primary is not available read from secondary
  - seconday – read from secondary
  - secondaryPreferred – if secondary is not available read from primary
  - nearest – respective of member's type, read from least network latency node\

- If more read capacity is needed → add more slaves.
  - Some large web applications have hundreds of slaves per master

Exercise care when specifying read preferences: Modes other than primary may return **stale data** because with *asynchronous replication*, data in the secondary may not reflect the most recent write operations (follows eventual consistency).

Replication

Failover

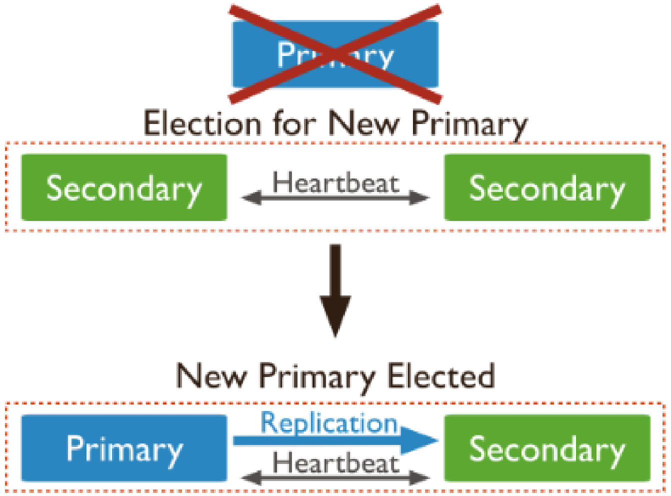
Use heartbeat signal to detect failure

When primary can't be reached (default 10 seconds), elect a new one

The election algorithm will make a "best-effort" attempt to have the secondary with the highest priority (number between 0 to 100) available call an election, and is also more likely to win .

You can also manually assign priority:

- Slower nodes with lower priority
- Backup or read-only nodes to never be primary



While an election is in process, the replica set has no primary and cannot accept writes and all remaining members become read-only!