

# Data sharding and replication

**Genoveva Vargas Solar**

French Council of scientific research, LIG-LAFMIA, France

[Genoveva.Vargas@imag.fr](mailto:Genoveva.Vargas@imag.fr)

<http://www.vargas-solar.com>

# Data all around

As of 2011 the global size  
Data in healthcare was  
estimated to be  
**150 Exabytes**  
(161 billion of  
Gigabytes)

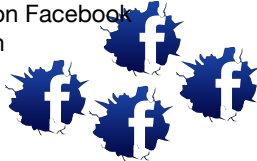


By 2014 it is  
anticipated  
there will be  
**400 million  
wearable wireless  
health monitors**



Data 2.0 is the availability of data from  
everywhere, not just from traditional  
sources. It is the ability to  
manage and analyze data from  
multiple sources, such as  
social media, sensors, and  
cloud storage, to gain  
insights and make decisions.

**30 billion  
pieces of content**  
are shared on Facebook  
every month



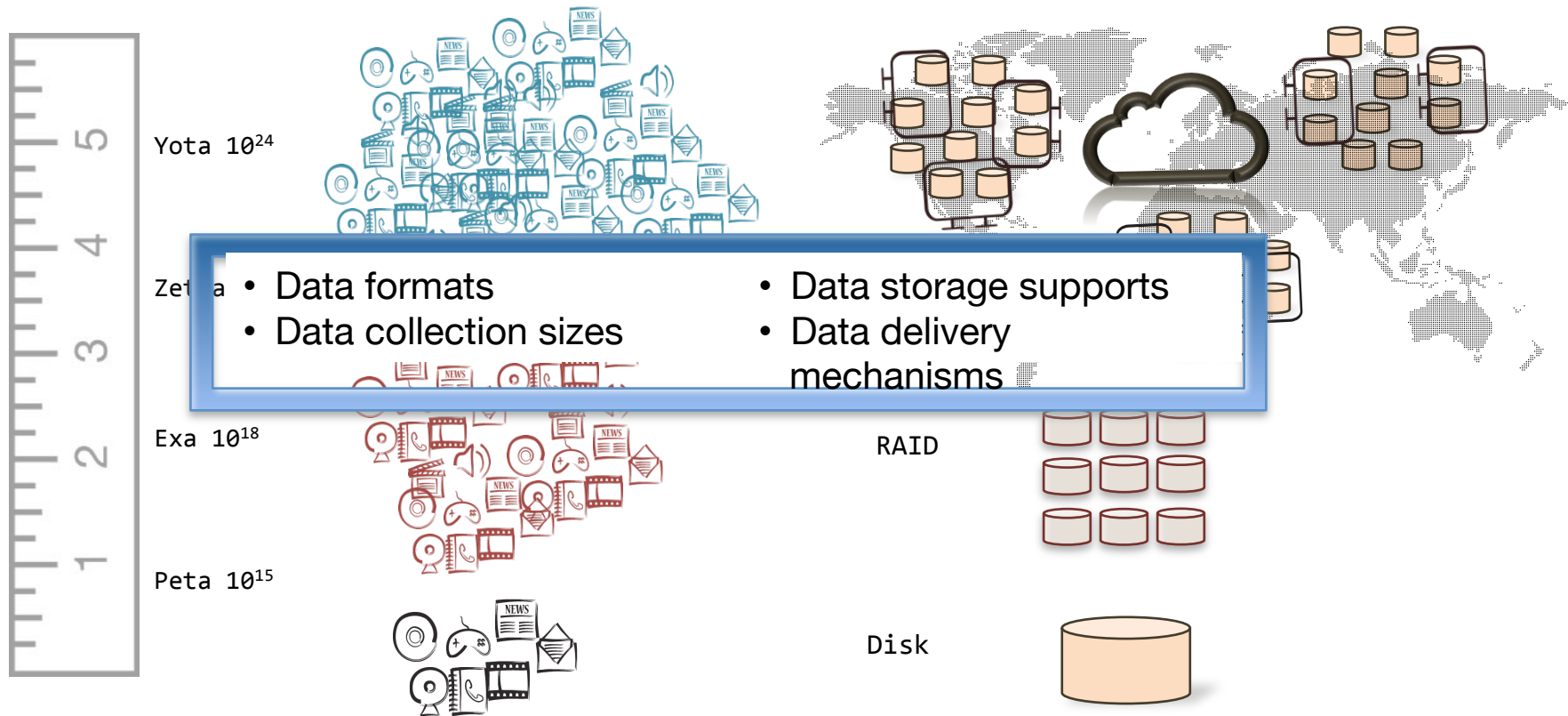
**40 million Tweets**  
are sent per day about  
200  
monthly active users

**40 billion+  
hours video**

are watched on YouTube  
each month



# Storing and accessing huge amounts of data



# This part of the course is about



Your Ultimate Guide to the  
Non-Relational Universe!

[including a **historic** [Archive](#) 2009-2011]  
**News Feed** covering some changes [here](#) !

<http://nosql-database.org>



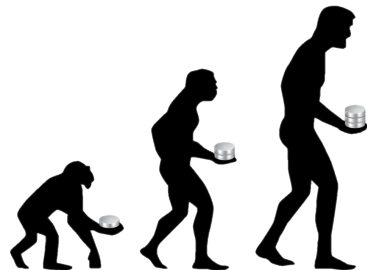
Debate on whether NoSQL stores and relational systems are better or worse ...  
***that is not the point***





# This session is absolutely about

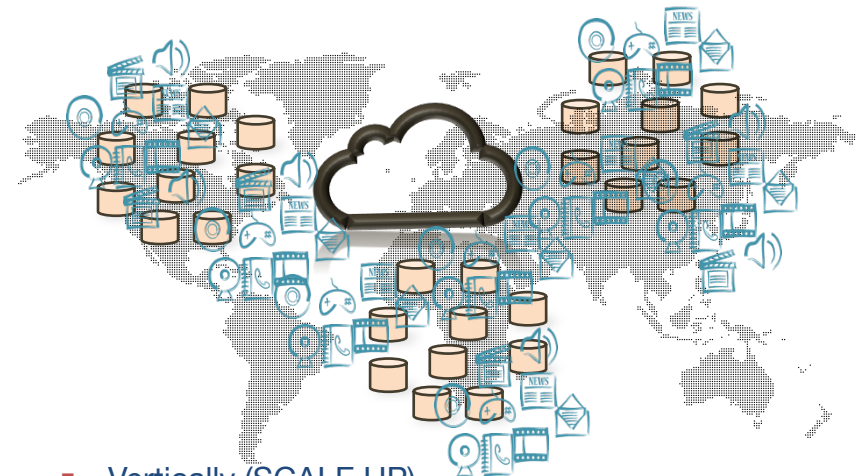
**Alternative for managing multiform and multimedia data  
collections *according to different properties and requirements***



# Scaling database systems



- A system is scalable if increasing its resources (CPU, memory, disk) results in **increased performance** proportionally to the added resources
- Improving performance means serving more units of work for handling larger units of work like when data sets grow
- Database systems have been scaled by buying bigger faster and more expensive machines



- Vertically (SCALE UP)
  - Add resources (CPU, memory) to a single node in a system
- Horizontally (SCALE OUT)
  - Add more nodes to a system

# NoSQL stores characteristics

There is no standard definition of what NoSQL means. The term began with a workshop organized in 2009, but there is much argument about what databases can truly be called NoSQL.

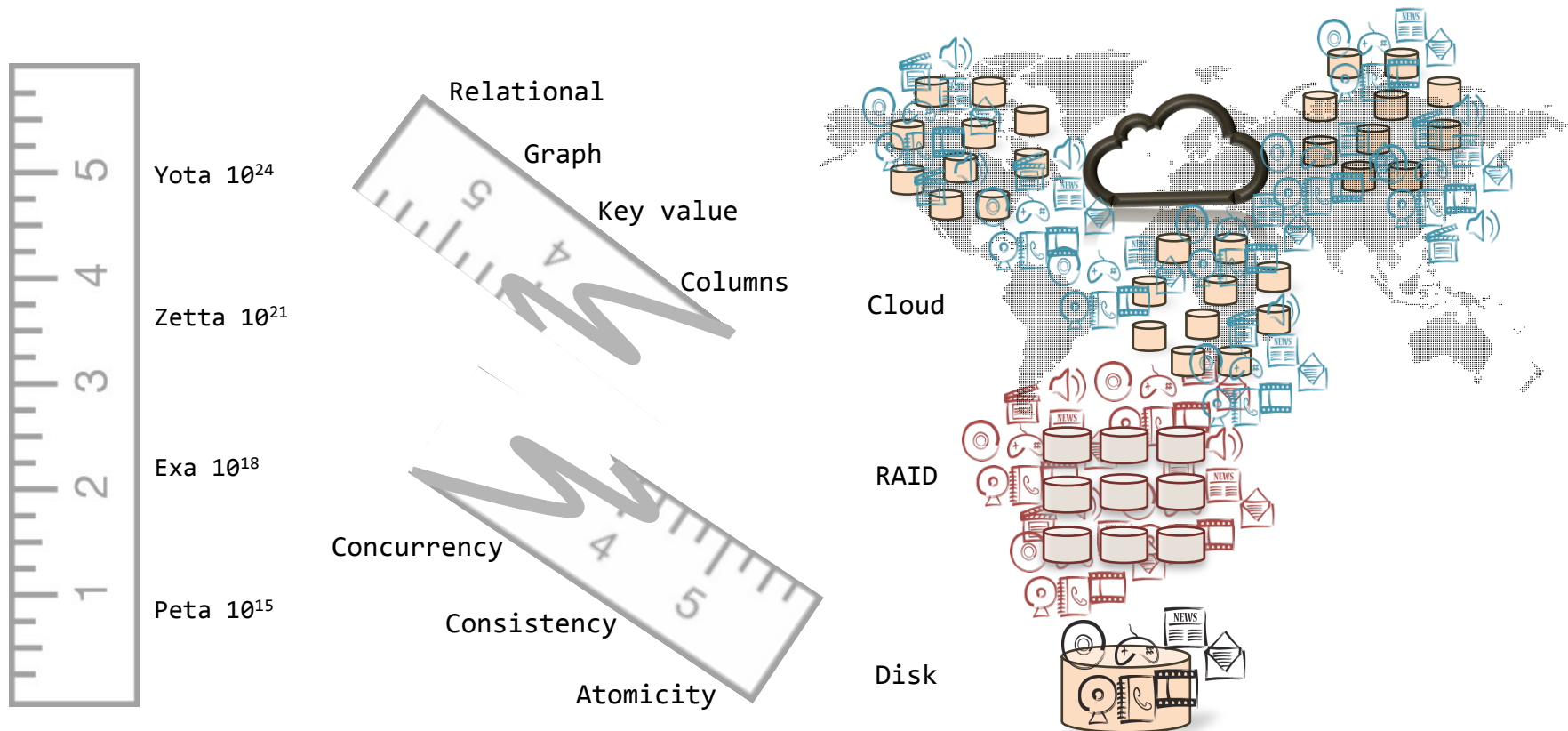
But while there is no formal definition, there are some common characteristics of NoSQL databases

- ☐ they don't use the relational data model, and thus don't use the SQL language
- ☐ they tend to be designed to run on a cluster
- ☐ they tend to be Open Source
- ☐ they don't have a fixed schema, allowing you to store any data in any record

- Simple operations
  - Key lookups reads and writes of one record or a small number of records
  - No complex queries or joins
  - Ability to dynamically add new attributes to data records
- Horizontal scalability
  - Distribute data and operations over many servers
  - Replicate and distribute data over many servers
  - No shared memory or disk
- High performance
  - Efficient use of distributed indexes and RAM for data storage
  - Weak consistency model
  - Limited transactions

Next generation databases mostly addressing some of the points: being **non-relational**, **distributed**, **open-source** and **horizontally scalable** [<http://nosql-database.org>]

# Dealing with huge amounts of data



## so now we have NoSQL databases

<https://www.youtube.com/watch?v=jyx8iP5tfCI>

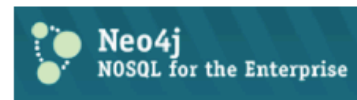
examples include

- Data model
- Consistency
- Storage
- Durability
- Availability
- Query support

Data stores designed to scale simple

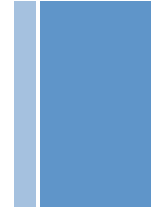
OLTP-style application loads

Read/Write operations  
by thousands/millions  
of users



We should also remember Google's **Bigtable** and Amazon's **SimpleDB**. While these are tied to their host's cloud service, they certainly fit the general operating characteristics

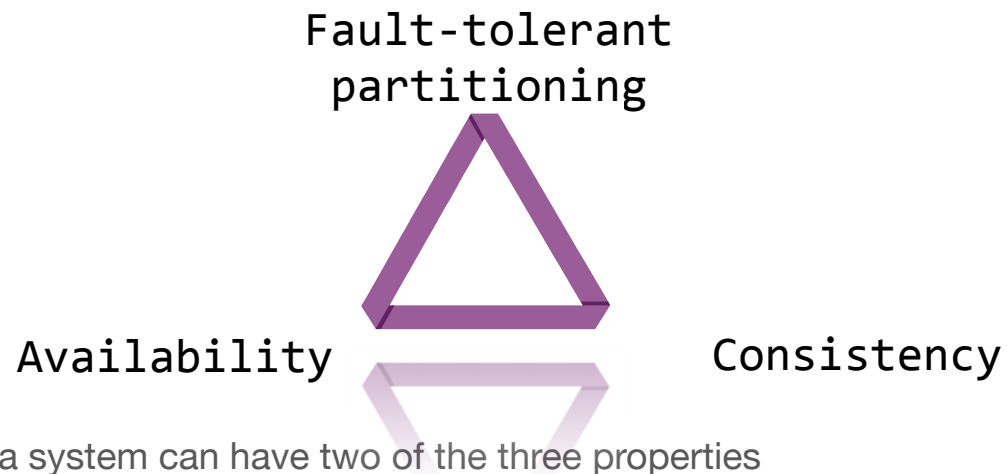
# How to map data management to Big Data requirements



*How to “map” the components of the reference architecture to (virtual) machines in the cloud.*

- How data is collected, transformed, integrated, loaded, **stored**, **modeled**?
- How to **partition** data and functions?  
(load balancing)
- How the **consistency** of the data is maintained ( vs availability)
- What **programming model**?
- Whether and how to **cache**?

# Problem statement: How much to give up?



- CAP theorem<sup>1</sup>: a system can have two of the three properties
- NoSQL systems sacrifice **consistency**

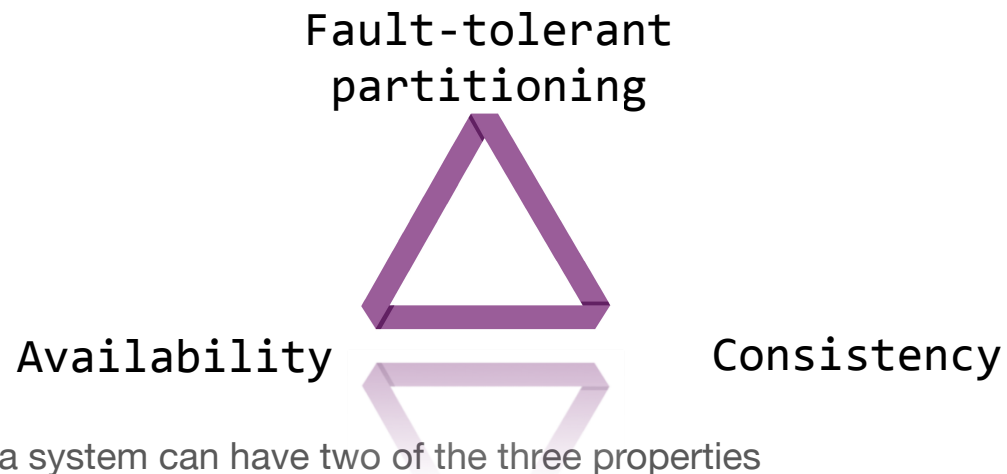
<sup>1</sup> Eric Brewer, "Towards robust distributed systems." PODC. 2000 <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>

Key-Value	SYSTEM	CONCURREN CY CONTROL	DATA STORAGE	REPLICATION	TRANSACTION
	Redis	Locks	RAM	Asynchronou s	No
	Scalaris	Locks	RAM	Synchronous	Local
	Tokyo	Locks	RAM/Disk	Asynchronou s	Local
	Voldemort	MVCC	RAM/BDB	Asynchronou s	No
	Riak	MVCC	Plug in	Asynchronou s	No
	Membrain	Locks	Flash +Disk	Synchronous	Local
	Membase	Locks	Disk	Synchronous	Local
	Dynamo	MVCC	Plug in	Asynchronou s	No
	SimpleDB	Non	S3	Asynchronou s	No
Document	MongoDB	Locks	Disk	Asynchronou s	No
	CouchDB	MVCC	Disk	Asynchronou s	No
	Terrastore	Locks	RAM+	Synchronous	L
	Hbase	Locks	HADOOP	Asynchronous	L
	HyperTable	Locks	Files	Synchronous	L
	Cassandra	MVCC	Disk	Asynchronous	L
	BigTable	Locs +stamps	GFS	Both	L
	PNuts	MVCC	Disk	Asynchronous	L
	MySQL-C	ACID	Disk	Synchronous	Y
	VoltDB	ACID/no Lock	RAM	Synchronous	Y
Relational	Clustrix	ACID/no Lock	Disk	Synchronous	Y
	ScaleDB	ACID	Disk	Synchronous	Y
	ScaleBase	ACID	Disk	Asynchronous	Y
	NimbusDB	ACID/no Lock	Disk	Synchronous	Y

Cattell, Rick. "Scalable SQL and NoSQL data stores." ACM SIGMOD Record 39.4 (2011): 12-27



# Problem statement: How much to give up?





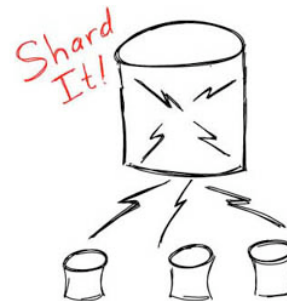
- CAP theorem<sup>1</sup>: a system can have two of the three properties
- NoSQL systems sacrifice **consistency**

<sup>1</sup> Eric Brewer, "Towards robust distributed systems." PODC. 2000 <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>

# NoSql Stores: availability and performance

14

- Replication 
  - Copy data across multiple servers (each bit of data can be found in multiple servers)
  - Increase data availability
  - Faster query evaluation
- Sharding 
  - Distribute different data across multiple servers
  - Each server acts as the single source of a data subset
- Orthogonal techniques



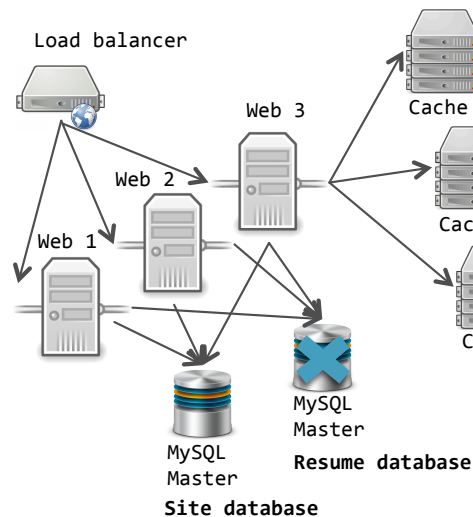
# Replication: pros & cons

- Data is more available
  - Failure of a site containing E does not result in unavailability of E if replicas exist
- Performance
  - Parallelism: queries processed in parallel on several nodes
  - Reduce data transfer for local data
- Increased updates cost
  - Synchronisation: each replica must be updated
- Increased complexity of concurrency control
  - Concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented

# Sharding: why is it useful?



- Scaling applications by reducing data sets in any single databases
- Segregating data
- Sharing application data
- Securing sensitive data by isolating it



- Improve read and write performance
  - Smaller amount of data in each user group implies faster querying
  - Isolating data into smaller shards accessed data is more likely to stay on cache
  - More write bandwidth: writing can be done in parallel
  - Smaller data sets are easier to backup, restore and manage
- Massively work done
  - Parallel work: scale out across more nodes
  - Parallel backend: handling higher user loads
  - Share nothing: very few bottlenecks
- Decrease resilience improve availability
  - If a box goes down others still operate
  - But: Part of the data missing

# Sharding and replication

- Sharding with no replication: unique copy, distributed data sets
  - (+) Better concurrency levels (shards are accessed independently)
  - (-) Cost of checking constraints, rebuilding aggregates
  - Ensure that queries and updates are distributed across shards
- Replication of shards
  - (+) Query performance (availability)
  - (-) Cost of updating, of checking constraints, complexity of concurrency control
- Partial replication (most of the times)
  - Only some shards are duplicated

# NoSQL STORES: Data management properties

18

- Indexing
  - Distributed hashing like
    - ▶ *Memcached* open source cache
    - In-memory indexes are scalable when distributing and replicating objects over multiple nodes
  - Partitioned tables
- High availability and scalability: eventual consistency
  - Data fetched are not guaranteed to be up-to-date
  - Updates are guaranteed to be propagated to all nodes eventually
- Shared nothing horizontal scaling
  - Replicating and partitioning data over many servers
  - Support large number of simple read/write operations per second (OLTP)
- No ACID guarantees
  - Updates eventually propagated but limited guarantees on reads consistency
  - BASE: basically available; soft state, eventually consistent ▶
  - Multi-version concurrency control

# Sharding on Mongo

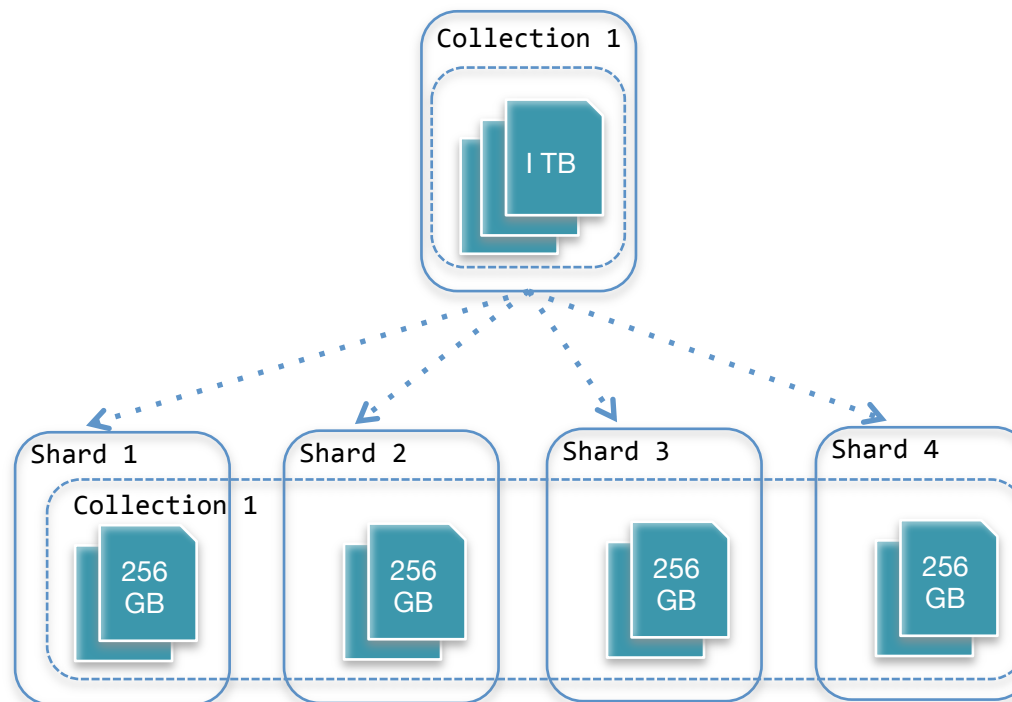
# Mongodb sharding

- Database systems with large data sets and high throughput applications can challenge the capacity of a single server.
  - High query rates can exhaust the CPU capacity of the server.
  - Larger data sets exceed the storage capacity of a single machine.
  - Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.
- To address these issues of scales, database systems have two basic approaches: vertical scaling and sharding
  - Vertical scaling adds more CPU and storage resources to increase capacity.
    - Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately more expensive than smaller systems.
    - Additionally, cloud-based providers may only allow users to provision smaller instances.
    - As a result there is a practical maximum capability for vertical scaling.
  - Sharding, or horizontal scaling, divides the data set and distributes the data over multiple servers, or shards.
    - Each shard is an independent database,
    - Collectively, the shards make up a single logical database



# Sharding data

21



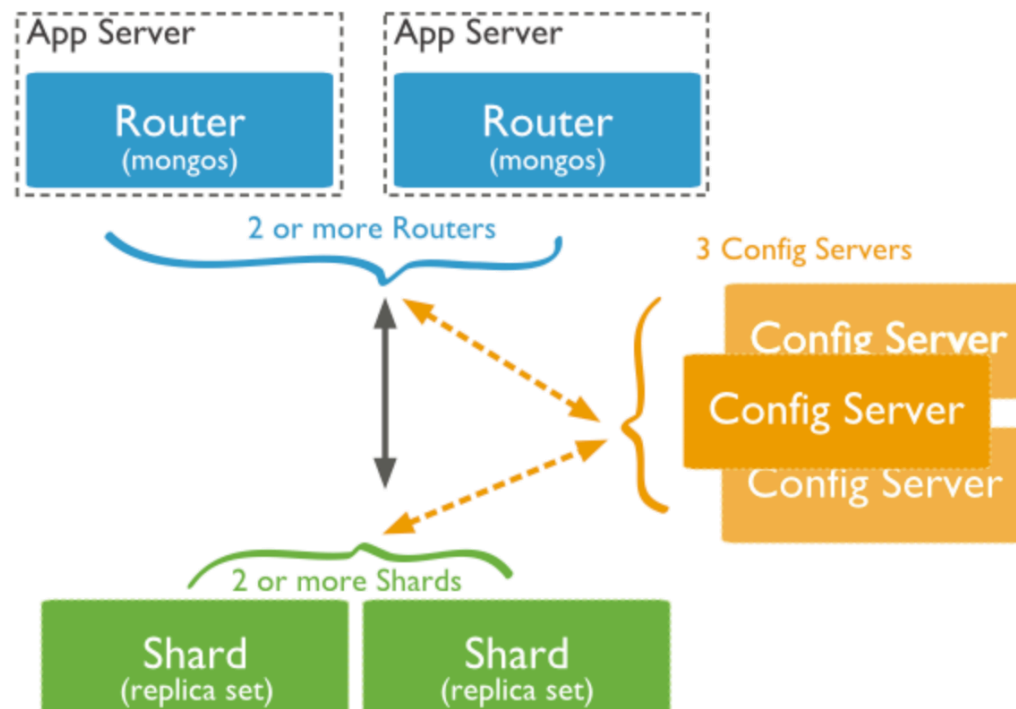
# sharding

22

- Addresses the challenge of scaling to support high throughput and large data sets:
- Reduces the number of operations each shard handles.
  - Each shard processes fewer operations as the cluster grows.
  - As a result, a cluster can increase capacity and throughput horizontally.
  - For example, to insert data, the application only needs to access the shard responsible for that record.
- Reduces the amount of data that each server needs to store.
  - Each shard stores less data as the cluster grows.
  - For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

# Sharding in mongo

23

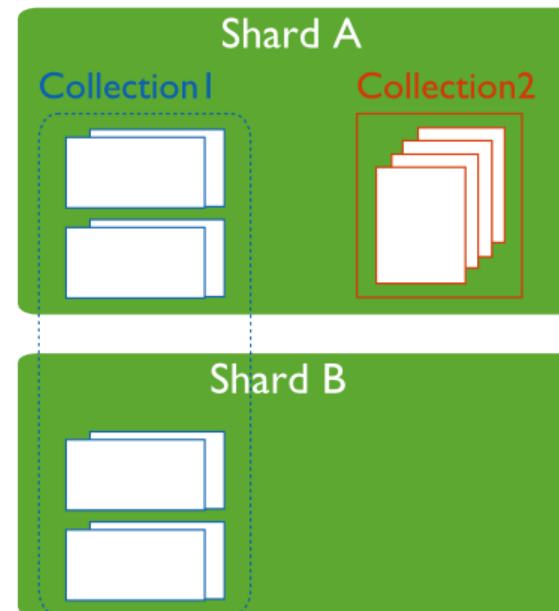


# Sharding in mongo

- **Shards (mongod)** store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set.
- **Query Routers (mongos instances)**, interface with client applications and direct operations to the appropriate shard or shards.
  - The query router processes and targets operations to shards and then returns results to the clients.
  - A sharded cluster can contain more than one query router to divide the client request load.
  - A client sends requests to one query router. Most sharded clusters have many query routers.
- **Config servers** store the cluster's metadata.
  - This data contains a mapping of the cluster's data set to the shards.
  - The query router uses this metadata to target operations to specific shards.
  - Production sharded clusters have exactly 3 config servers

# Primary shard

- Every database has a “primary” shard that holds all the un-sharded collections in that database
- To change the primary shard for a database, use the `movePrimary` command.
  - The process of migrating the primary shard may take significant time to complete, and you should not access the collections until it completes.
- When a new sharded cluster is deployed with shards that were previously used as replica sets, all existing databases continue to reside on their original shard
- Databases created subsequently may reside on any shard in the cluster



# Shard status (1)

- Use the `sh.status()` method in the mongo shell to see an overview of the cluster.
- This reports includes which shard is primary for the database and the chunk distribution across the shards

■ The  
dat

```
--- Sharding Status ---  
sharding version: {  
  "_id" : <num>,  
  "version" : <num>,  
  "minCompatibleVersion" : <num>,  
  "currentVersion" : <num>,  
  "clusterId" : <ObjectId>  
}
```

onfig

## Shard status (2)

- The Shards section lists information on the shard(s). For each shard, the section displays the name, host, and the associated tags, if any

```
shards:
{  "_id" : <shard name1>,
   "host" : <string>,
   "tags" : [ <string> ... ]
}
{  "_id" : <shard name2>,
   "host" : <string>,
   "tags" : [ <string> ... ]
}
...
```

## Shard status (3)

- The Databases section lists information on the database(s). For each database, the section displays the name, whether the data is partitioned, and the primary shard.

```
databases:
{ "_id" : <dbname1>,
  "partitioned" : <boolean>,
  "primary" : <string>
}
{ "_id" : <dbname2>,
  "partitioned" : <boolean>,
  "primary" : <string>
}
...
```



# Shard status (4)

- The Sharded Collection section provides information on the sharding details for sharded collection(s). For each sharded collection,
  - the section displays the shard key,
  - the number of chunks per shard(s),
  - the distribution of the data across the shards,
  - the tags for the shards.

```
<dbname>.<collection>
shard key: { <shard key> : <1 or hashed> }
chunks:
  <shard name1> <number of chunks>
  <shard name2> <number of chunks>
  ...
{ <shard key>: <min range1> } --> { <shard key> : <max range1> } on : <shard name>
{ <shard key>: <min range2> } --> { <shard key> : <max range2> } on : <shard name>
...
tag: <tag1> { <shard key> : <min range1> } --> { <shard key> : <max range1> }
```

# Config servers

- Special mongod instances that store the metadata for a sharded cluster.
- Use a two-phase commit to ensure immediate consistency and reliability.
- Do not run as replica sets.
- All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata.
- A production sharded cluster has exactly three config servers
  - For testing purposes you may deploy a cluster with a single config server
  - But to ensure redundancy and safety in production, you should always use three.

# Config database

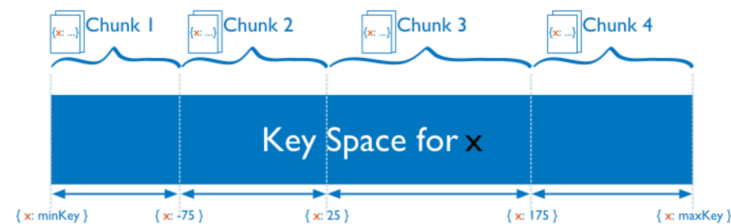
- Config servers store the metadata in the config database. The mongos instances cache this data and use it to route reads and writes to shards
- MongoDB only writes data to the config server in the following cases:
  - To create splits in existing chunks.
  - To migrate a chunk between shards.
- MongoDB reads data from the config server data in the following cases:
  - A new mongos starts for the first time, or an existing mongos restarts.
  - After a chunk migration, the mongos instances update themselves with the new cluster metadata.
  - MongoDB also uses the config server to manage distributed locks.

# Data partitioning

- MongoDB distributes data, or shards, at the collection level.
- Sharding partitions a collection's data by the shard key.
- A shard key is either an indexed field or an indexed compound field that exists in every document in the collection.
- MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards.
  - range based partitioning or hash based partitioning

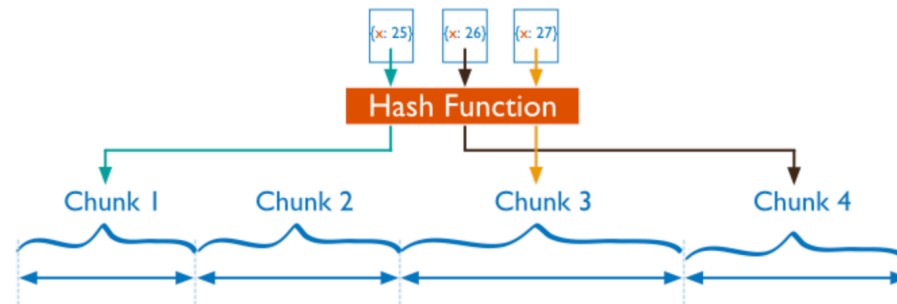
# Range based sharding

- MongoDB divides the data set into ranges determined by the shard key values
  - Consider a numeric shard key: If you visualize a number line that goes from negative infinity to positive infinity, each value of the shard key falls at some point on that line.
  - MongoDB partitions this line into smaller, non-overlapping ranges called chunks
  - chunk is range of values from some minimum value to some maximum value.
- Given a range based partitioning system, documents with “close” shard key values are likely to be in the same chunk, and therefore on the same shard



# Hash based sharding

- For hash based partitioning, MongoDB computes a hash of a field's value, and then uses these hashes to create chunks
- With hash based partitioning, two documents with “close” shard key values are unlikely to be part of the same chunk.
- This ensures a more random distribution of a collection in the cluster



# Tag aware sharding

- MongoDB allows administrators to direct the balancing policy using tag aware sharding.
- Administrators create and associate tags with ranges of the shard key
- Assign those tags to the shards.
- The balancer migrates tagged data to the appropriate shards and ensures that the cluster always enforces the distribution of data that the tags describe
  - Tags are the primary mechanism to control the behavior of the balancer and the distribution of chunks in a cluster.
  - Most commonly, tag aware sharding serves to improve the locality of data for sharded clusters that span multiple data centers

# Range and hash based sharding

- **Range based partitioning** supports more efficient range queries
  - Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks
  - It can result in an uneven distribution of data, which may negate some of the benefits of sharding
    - If the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard.
    - In this situation, a small set of shards may receive the majority of requests and the system would not scale very well
- **Hash based partitioning**, ensures an even distribution of data
  - Hashed key values results 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



# Maintaining a balanced data distribution

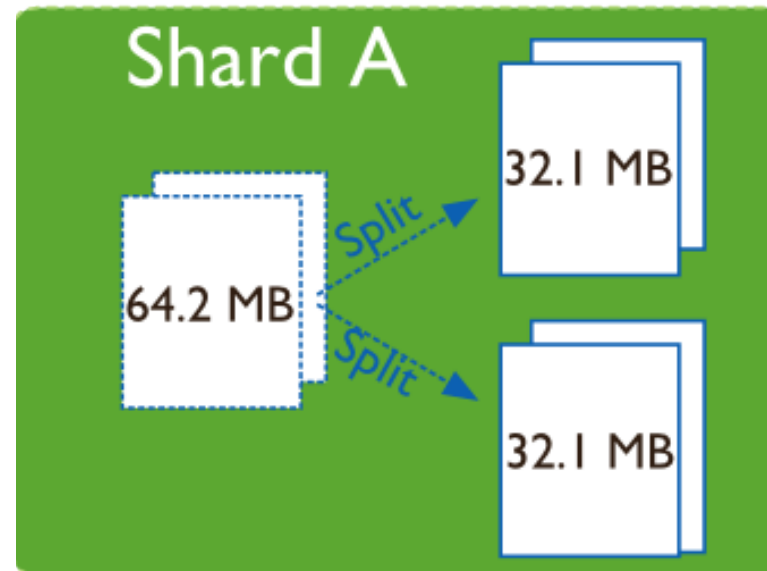
37

- The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster
  - particular shard contains significantly more chunks than another shard
  - a size of a chunk is significantly greater than other chunk sizes
- MongoDB ensures a balanced cluster using two background process: **splitting** and the **balancer**

# splitting

38

- Background process that keeps chunks from growing too large.
- When a chunk grows beyond a specified chunk size,
  - MongoDB splits the chunk in half
  - Inserts and updates triggers splits.
    - Splits are an efficient meta-data change.
    - To create splits, MongoDB does not migrate any data or affect the shards



# Balancing (1)

- The balancer is a background process that manages chunk migrations.
- Runs from any of the query routers in a cluster
- When the distribution of a sharded collection in a cluster is uneven
  - The balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances
  - For example: if collection users has 100 chunks on shard 1 and 50 chunks on shard 2, the balancer will migrate chunks from shard 1 to shard 2 until the collection achieves balance

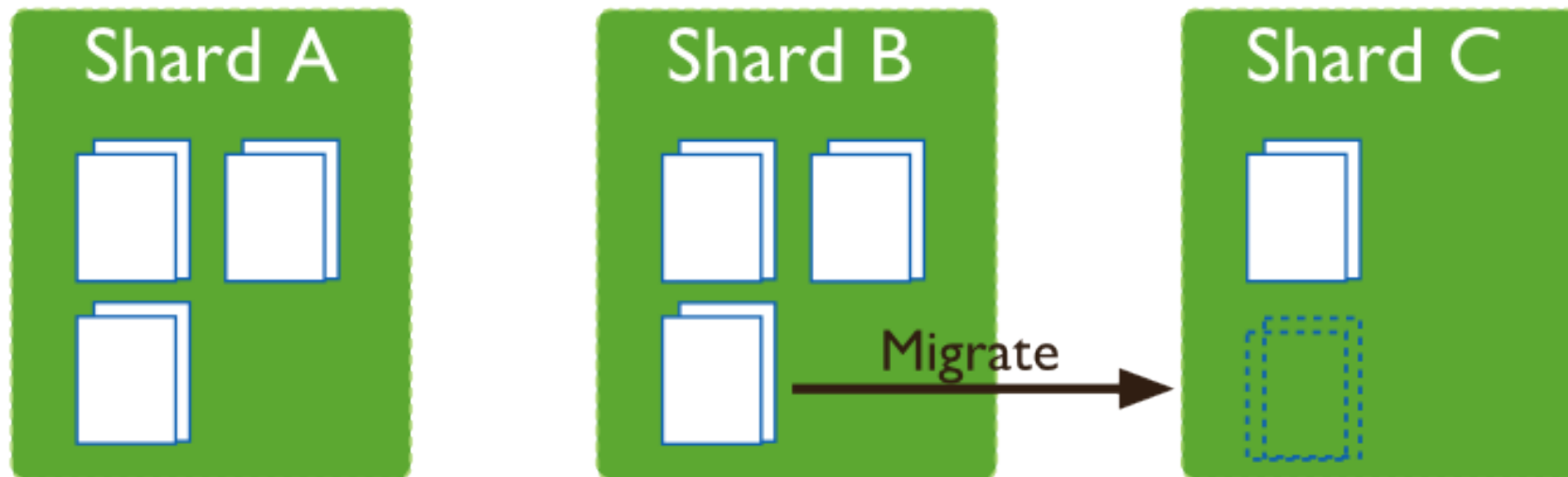
## Balancing (2)

40

- The shards manage chunk migrations as a background operation between an origin shard and a destination shard.
  - During a chunk migration, the destination shard is sent all the current documents in the chunk from the origin shard.
  - Next, the destination shard captures and applies all changes made to the data during the migration process.
  - Finally, the metadata regarding the location of the chunk on config server is updated
  - If there's an error during the migration, the balancer aborts the process leaving the chunk unchanged on the origin shard.
  - MongoDB removes the chunk's data from the origin shard after the migration completes successfully

## Balancing (3)

41



# Adding and removing shards from a cluster

42

- Adding a shard to a cluster creates an imbalance since the new shard has no chunks.
  - MongoDB begins migrating data to the new shard immediately
  - It can take some time before the cluster balances.
- When removing a shard, the balancer migrates all chunks from a shard to other shards.
- After migrating all data and updating the meta data, you can safely remove the shard

# Sharding vs. horizontal partitioning

43

- *Horizontal partitioning* splits one or more tables by row, usually within a single instance of a schema and a database server.
  - Reduce index size (and thus search effort) provided that there is some obvious, robust, implicit way to identify in which table a particular row will be found, without first needing to search the index,
  - e.g., the classic example of the 'CustomersEast' and 'CustomersWest' tables, where their zip code already indicates where they will be found
- Sharding goes beyond this: it partitions the problematic table(s) in the same way, but across potentially multiple instances of the schema
  - Search load for the large partitioned table can now be split across multiple servers (logical or physical), not just multiple indexes on the same logical server
  - Splitting shards across multiple isolated instances requires more than simple horizontal partitioning
  - Sharding splits large partitionable tables across the servers, while smaller tables **are replicated as complete units**





# Consistency issues



# Consistency

- Consistency is an important area of study in distributed systems
- Data consistency summarizes the validity, accuracy, usability and integrity of related data between applications and across an (IT) enterprise
  - ensures that each user observes a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users or processes
- Data Consistency problems may arise at any time but are frequently introduced during or following recovery situations when backup copies of the data are used in place of the original data


# Consistency models

- Consistency model is a guarantee about the relation between and update to an object and the access to an updated object
  - used in distributed systems like distributed shared memory systems or distributed data stores (such as a file systems, databases, optimistic replication systems or Web caching).
  - The system supports a given model if operations on memory follow specific rules.
  - The data consistency model specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable
- *Linearizability* (strict or atomic consistency)
- *Serializability*: ensures a global ordering of transactions 
- *Sequential consistency*: ensures a global ordering of operation 
- *Casual consistency*: ensures partial orderings between dependent operations  
- Eventually consistent transactions: ensure different order of updates in all copies eventually converge same value

# Eventual consistency

- Specific form of **weak consistency** used in many large distributed databases
- Requires that **all changes** to a replicated piece of data **eventually reach all affected replicas**
  - Storage system guarantees that if no updates are made to the object, eventually all accesses will return the last updated value
  - If no failures occur, the maximum size of inconsistency window can be determined based on factors like: communication delays, the load of the system, the number of replicas involved in the replication scheme
- **Conflict resolution** is not handled & responsibility is pushed up to the programmer in the event of conflicting updates
- *Domain Name System*
  - Updates to a name are distributed according to a configured pattern in combination to time-controlled caches
  - Eventually all clients will see the update
  - Given enough time over which no changes are performed, all updates will propagate through the system and all replicas will be synchronized

# Properties

- CAP theorem by Brewer: consistency, availability and partition tolerance are three desired properties of any shared-data system
  - Conjuncture: *Maximum two of them can be guaranteed at a time (!)* 
  - Ideally we expect a service to be available during the whole period time of network connection: *if the network is available the service should be available too*
- To achieve good performance parameters, requests need to be processed by a distributed system:
  - Increasing the number of servers, the probability of any of them or network communication failing is also invreaed
  - A system must be designed in such a way that this failure be transparent (or minimize the impact) to the client

<sup>1</sup> Eric Brewer, "Towards robust distributed systems." PODC. 2000 <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>

# Playing with cap

- Availability and partition: achieve as low latency as possible combined with high performance as possible (e.g., Cassandra)
- Consistency and partition tolerance: mirroring database clusters between different data centres to achieve quicker response by splitting workload into different sub tasks and then execute them simultaneously across all available nodes/servers
- Stock market prices and number of stock available have to be up to date (consistency level)
- E-commerce site: not good for a business if a customer finds out the product is out of stock after she submitted the payment (consistency level)

# Eventual consistency

- Writes to one replica will eventually appear at other replicas
- If all replicas have received the same set of write they will have the same values for all data
- Weak form of consistency that does not restrict the ordering of operations on different keys → programmers should reason about all possible orderings and exposing many inconsistencies to users
- Examples:
  - After Alice updates her profile, she might not see that update after a refresh
  - Or if Alice and Bob are commenting back and forth on a blog post, Carol might see a random non-contiguous subset of that conversation

# Consistency vs. High availability

51

- The application designer must know how the database consistency is obtained and the costs of inconsistency or anomalies to decide whether to implement eventual consistency with high availability
- Dealing with consistency abnormalities is intuitive and difficult: it depends on thinking the correct sequence of operations and therefore more difficult than high consistency
  - Atomic commitment protocols taking care of resources blocking
  - Time stamps and versions associated to a store system for identifying newest versions
  - E.g. PNUTS (Yahoo!)

# Eventual consistency: Pros and cons

52

- Easy to achieve
- Database servers separated from the larger database cluster by a network partition can still accept (NoSQL systems)
- Often strongly consistent (cf. Amazon SimpleDB inconsistency window < 500 ms. ; Amazon S3 < 12 seconds; Cassandra < 200 ms)
- Not precise definition: not clear what is *eventually consistent state*?
- A DB always returning the value 42 is eventually consistent even if 42 was never written?
- Eventually all accesses return the last updated value thus the DB cannot converge to an arbitrary value
- What values can be returned before the eventual state of the DB is reached? If replicas have not yet converged, what guarantees can be made on the data returned?
- The last updated value? How to know what version of data item was converged to same state in all replicas?



# Eventual consistency: more issues

53

- Requires that writes to one replica will eventually appear at other replicas
- If all replicas have received the same set of writes, they will have the same values for all data
- Problem: it does not restrict the ordering of operations on different keys → programmer should reason about all possible orderings
  - *What is the effect on the application if a database read returns an arbitrarily old value?*
  - *What is the effect on the application if the database sees a modification happen in the wrong order?*
  - *What is the effect on the application of another client modifying the database as I try to read it?*
  - *What is the effect that my database updates have on other clients trying to read the data?*



<http://vargas-solar/data-management-services-cloud/>

# Linearizability (1)

- First introduced as a consistency model by Herlihy and Wing in 1987
- **History:** sequence of invocations and responses made of an object by a set of threads.
  - Each invocation of a function will have a subsequent response.
  - This can be used to model any use of an object.
- Given two threads, A and B, both attempt to grab a lock, backing off if it's already taken
- This would be modelled as both threads invoking the lock operation, then both threads receiving a response, one successful, one not

A invokes lock	B invokes lock	A gets 'failed' response	B gets 'successful' response
----------------	----------------	--------------------------	------------------------------

- A **sequential history** is one in which all invocations have immediate responses. A sequential history should be trivial to reason about, as it has no real concurrency
- Is this example sequential?

# Linearizability (2)

- A **history is linearizable** if:
  - its invocations and responses can be reordered to yield a sequential history
  - that sequential history is correct according to the sequential definition of the object
  - **if a response preceded an invocation in the original history, it must still precede it in the sequential reordering**
- It is the last point which is unique to *linearizability*, and is thus the major contribution of Herlihy and Wing
- Reordering B's invocation below A's response yields a sequential history. This is easy to reason about, as all operations now happen in an obvious order

---

A invokes lock	A gets 'failed' response	B invokes lock	B gets 'successful' response
----------------	--------------------------	----------------	------------------------------

---

- Unfortunately, it doesn't match the sequential definition of the object (it doesn't match the semantics of the program)

# Linearizability (3)

- A should have successfully obtained the lock, and B should have subsequently aborted

---

B invokes lock	B gets 'successful' response	A invokes lock	A gets 'failed' response
----------------	------------------------------	----------------	--------------------------

---

- This is another correct sequential history. It is also a linearization!
  - Note that the definition of linearizability only precludes responses that precede invocations from being reordered
  - Since the original history had no responses before invocations, we can reorder it as we wish. Hence the original history is indeed linearizable.
- An object (as opposed to a history) is linearizable if all valid histories of its use can be linearized
  - Note that this is a much harder assertion to prove



# Serializability (1)

- In concurrency control of databases, transaction processing, and transactional applications (e.g., transactional memory and software transactional memory), both centralized and distributed,
  - A transaction schedule (*history*) is *serializable* if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e., sequentially without overlapping in time
  - Transactions are normally executed concurrently (they overlap), since this is the most efficient way.
  - Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control

# Serializability (2)

## ■ Not linearizable

A invokes lock    A gets 'successful' response    B invokes unlock    B gets 'successful' response    A invokes unlock    B gets 'successful' response

- This history is not valid because there is a point at which both A and B hold the lock;

- It cannot be reordered to a valid sequential history without violating the ordering rule

- However, under *serializability*, B's unlock operation may be moved to before A's original lock, which is a valid history (assuming the object begins the history in a locked state)

B invokes unlock    B gets 'successful' response    A invokes lock    A gets 'successful' response    A invokes unlock    A gets 'successful' response



# Sequential consistency

- Sequential consistency is one of the consistency models used in the domain of concurrent programming (e.g. in distributed shared memory, distributed transactions, etc.).
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*
- A system provides sequential consistency if
  - every node of the system sees the (write) operations on the same memory part (page, virtual object, cell, etc.) in the same order,
  - although the order may be different from the order as defined by real time (as observed by a hypothetical external observer or global clock) of issuing the operations
- The sequential consistency is weaker than strict consistency
  - which would demand that operations are seen in order in which they were actually issued,
  - which is essentially impossible to secure in distributed system as deciding global time is impossible and





# Casual consistency

- A system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order
  - Condition-writes that are potentially causally related must be seen by all processes in the same order
  - Concurrent writes may be seen in a different order on different machines
- When a node performs a read followed later by a write, even on a different variable
  - the first operation is said to be causally ordered before the second
  - because the value stored by the write may have been dependent upon the result of the read
- A read operation is causally ordered after the earlier write on the same variable that stored the data retrieved by the read
- Two write operations performed by the same node are defined to be causally ordered, in the order they were performed
  - After writing value  $v$  into variable  $x$ , a node knows that a read of  $x$  would give  $v$ , so a later write could be said to be (potentially) causally related to the earlier one
- Finally, e this causal order can be forced to be transitive: that is, we say that if operation  $A$  is (causally) ordered before  $B$ , and  $B$  is ordered before  $C$ ,  $A$  is ordered before  $C$
- This is weaker than sequential consistency, which requires that all nodes see all writes in the same order



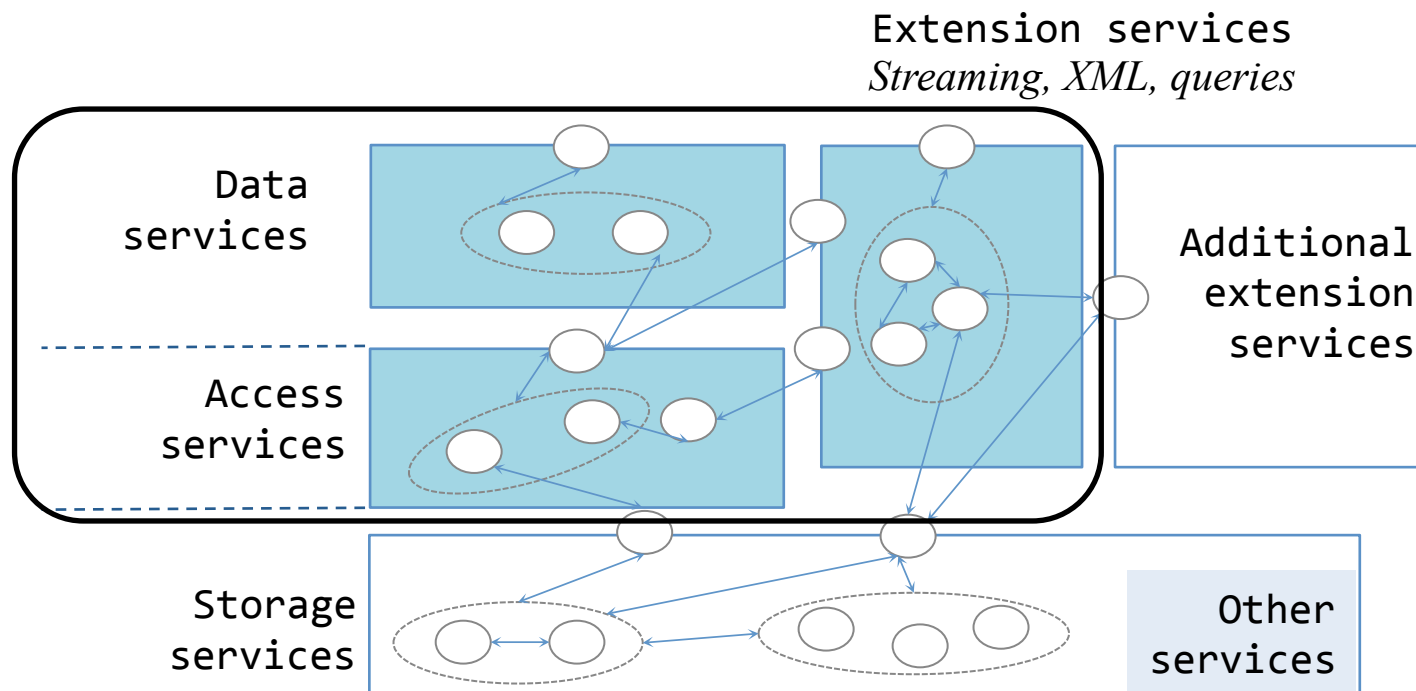
# CAP THEOREM

- **Consistency:** requires that each operation executed within a distributed system where data is spread among more servers ended with the same result as if executed on one server with all data
- **Availability:** requires that sending a request to any functional node should be enough for a requester to get a response (a system is therefore tolerant to failure of other nodes caused for example by network throughput problems)
- **Partition tolerance:** a distributed system consists of servers interconnected by a network. During network communication failures are frequent. Temporary communication interruption among a server must not cause the whole system to respond incorrectly

## **Tailoring data storage services**

# Service based HQ evaluation

64



# Service based HQ evaluation

65

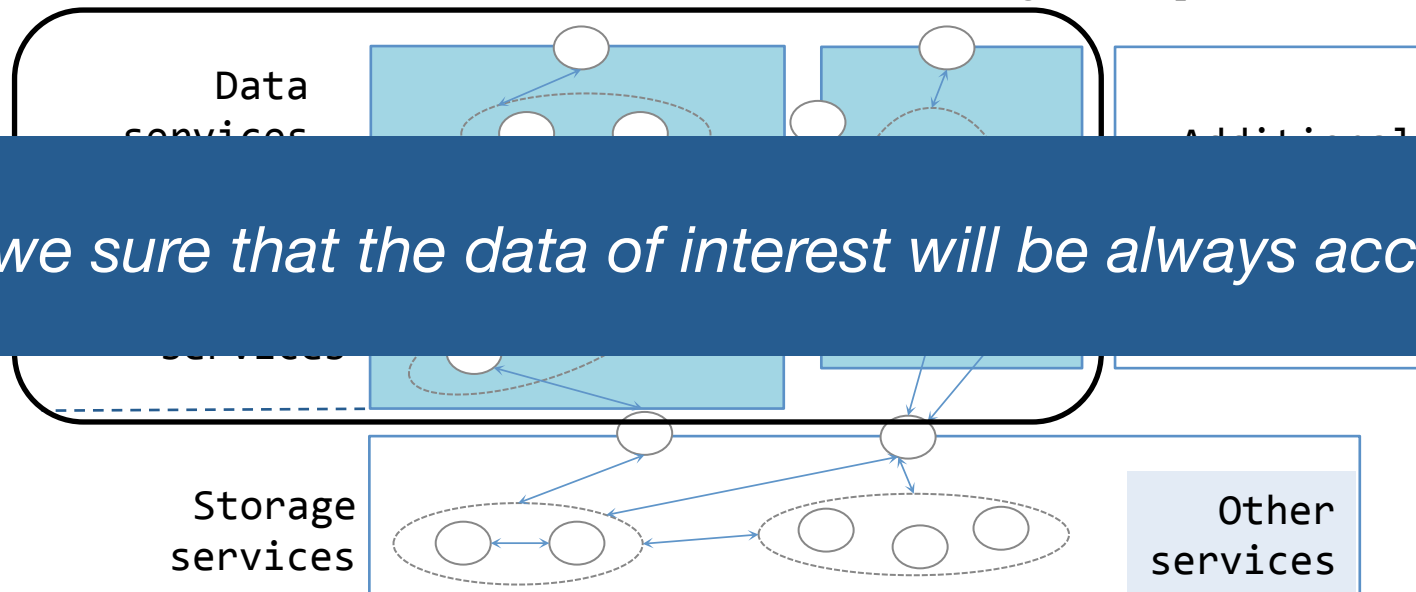
Extension services  
*Streaming, XML, queries*

Data  
services

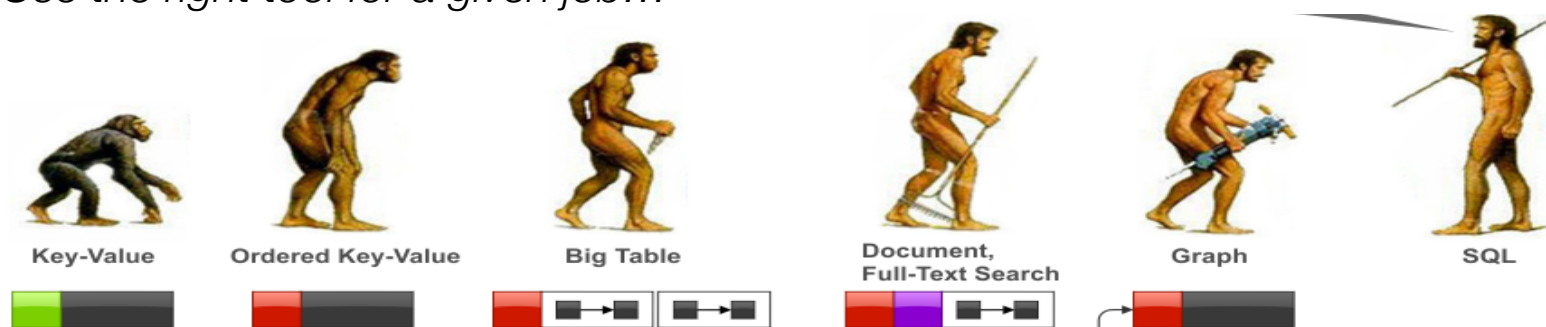
*Are we sure that the data of interest will be always accessible?*

Storage  
services

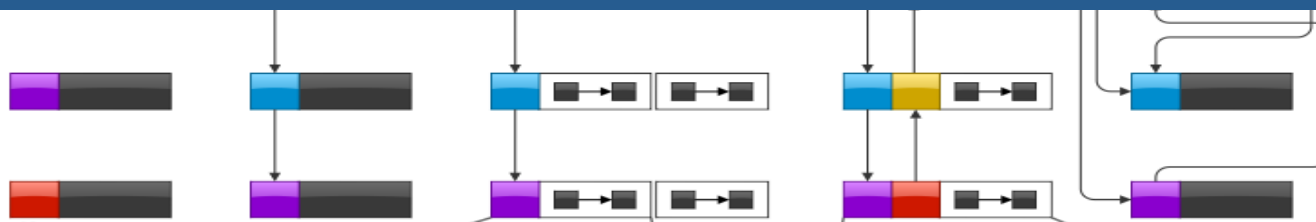
Other  
services



*Use the right tool for a given job...*



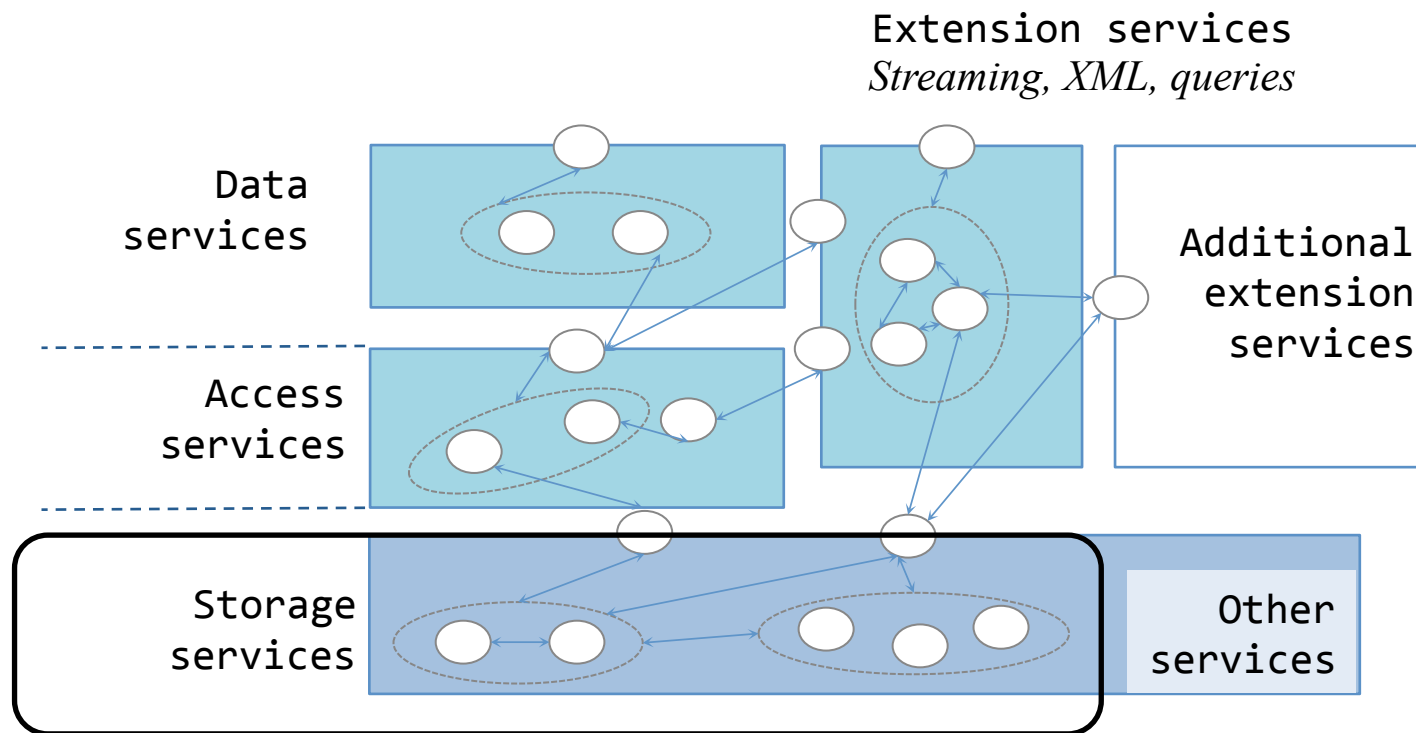
*Lack of standardization of models and data storage technologies*



**(Katsov-2012)**

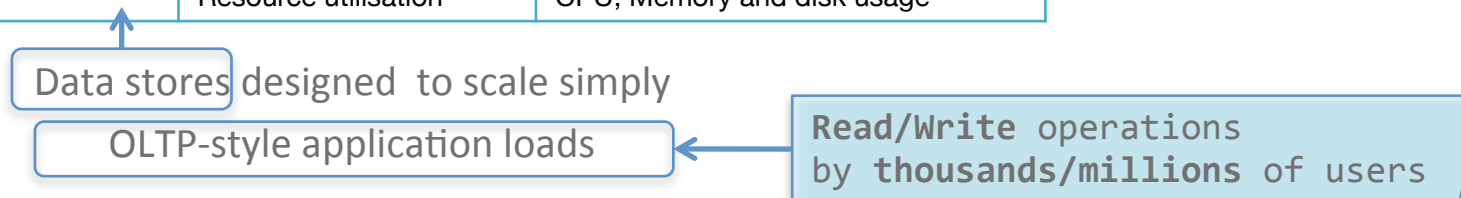
# Polyglot persistence

67



# Quality driven benchmark<sup>1</sup>

CHARACTERISTIC	SUBCHARACTERISTIC	METRIC
Reliability	Maturity	API changes
	Availability	Downtime <sup>3</sup>
	Fault tolerance	Node down throughput <sup>3</sup>
	Recoverability	Time to stabilize on node up <sup>3</sup>
Performance and efficiency	Time behaviour	Throughput, latency <sup>2</sup>
	Resource utilisation	CPU, Memory and disk usage <sup>4</sup>



<sup>1</sup>Yahoo Cloud Serving Benchmark, <https://github.com/brianfrankcooper/YCSB/wiki>

<sup>2</sup> Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. SoCC '10, ACM, New York, NY, USA (2010)

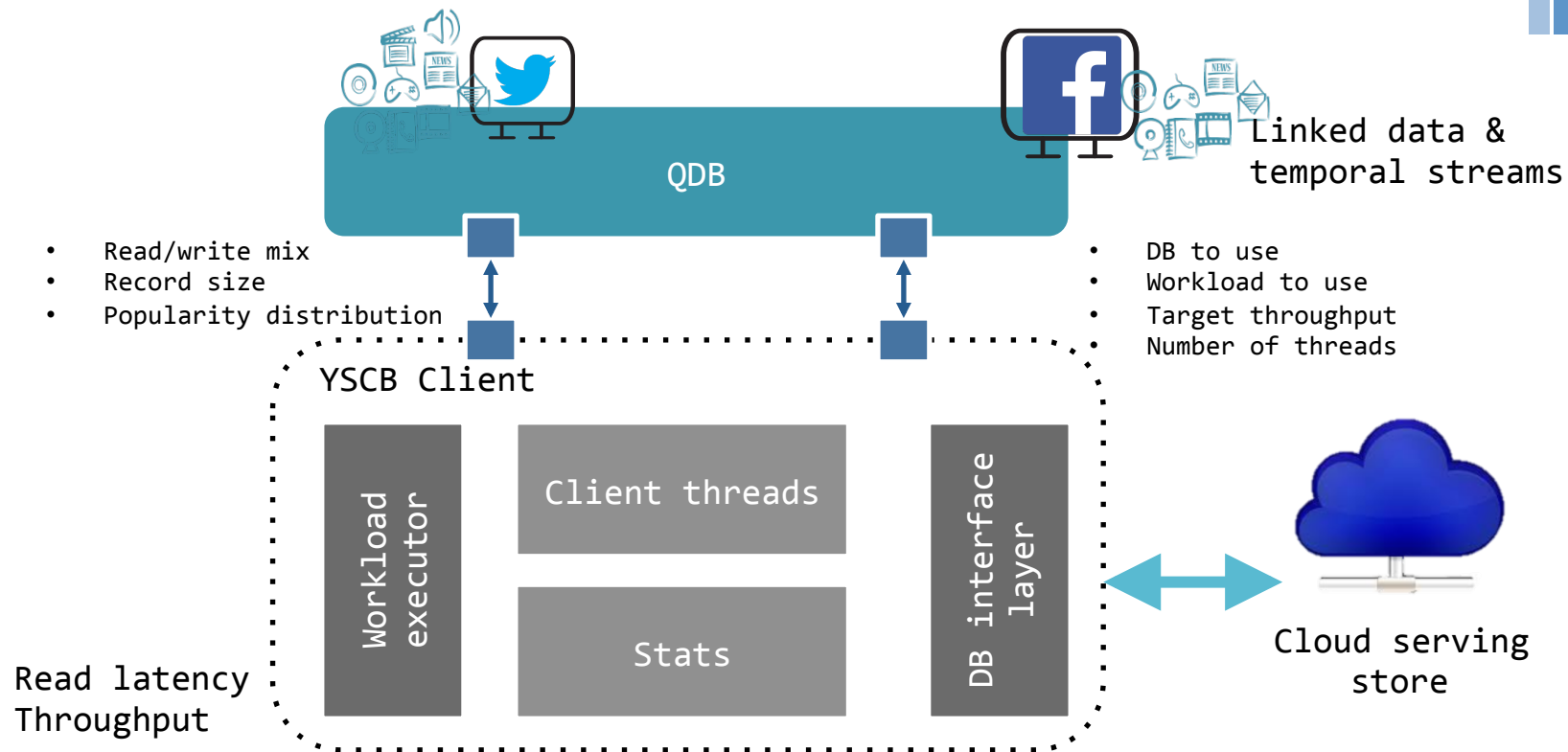
<sup>3</sup> Nelubin, D., Engber, B.: Failover Characteristics of leading NoSQL databases. Tech. rep., Thumbtack Technology (2013)

<sup>4</sup> Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing 30(7) (Jul 2004)



# Quality driven benchmark

69



# Ongoing work

- **QDB benchmark** extends YCSB: *FaultTolerance, Recoverability and TimeBehaviour*
  - Pivot data model for representing NoSQL stores data models
  - Sample application: Shopping system<sup>1</sup> (ProductInfo)
  - Document data stores: MongoDB, Couchbase, VoltDB, Redis, Neo4J
    - Cluster of four Ubuntu 12.04 servers deployed with extra large VM instances (8 virtual cores and 14 GB of RAM) in Windows Azure<sup>2</sup>
- Distributed **polyglot (big) database** engineering
  - Model2Roo: engineering data storage solutions for given data collections
  - ExSchema for supporting the maintenance of a polyglot storage solution

<sup>1</sup> McMurtry, D., Oakley, A., Sharp, J., Subramanian, M., Zhang, H.: Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence Microsoft patterns & practices, Microsoft (2013)

<sup>2</sup> <http://www.windowsazure.com/>

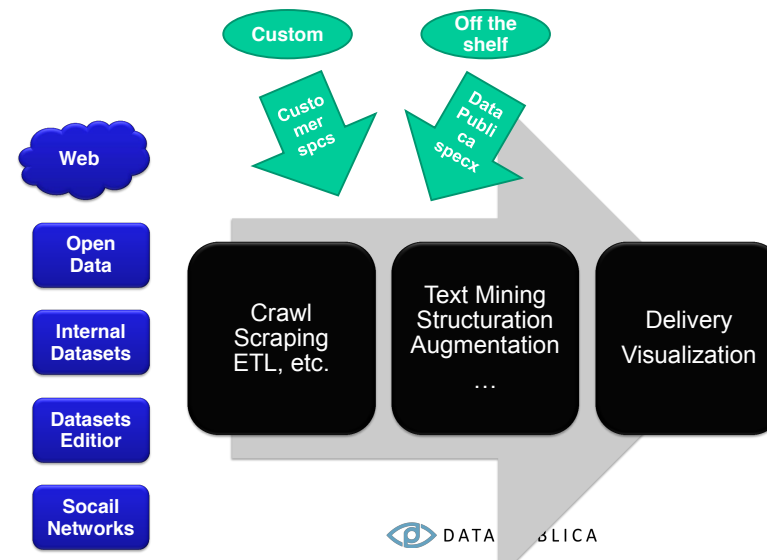
<sup>3</sup> <http://forge.puppetlabs.com/puppetlabs/>

<sup>4</sup> Yahoo Cloud Serving Benchmark, <https://github.com/brianfrankcooper/YCSB/wiki>

# Challenge: open data

71

- Data journalism
  - <http://datauy.org/que-es-una-hackaton/>
- Open data services:
  - <http://www.data-publica.com>
- Open data repositories
  - <http://www.infotecarios.com/hackathon-repositorios-de-datos-abiertos-open-data-segunda-parte/>
- Open knowledge
  - <https://okfn.org>
- Datos en Uruguay
  - <http://datauy.org/asi-fue-el-dia-mundial-de-los-datos-abiertos-2013/>



# Challenge: polyglot meets XPeranto

Given a data collection coming from different social networks stored on NoSQL systems (Neo4J and Mongo) *[possibly according to a strategy combining sharding and replication techniques]*, extend the UnQL pivot query language considering

- Data processing operators adapted to query different data models (graph, document). Example query Neo + Mongo and what about Join, Union ...
- Assuming concurrent CRUD operations to the stores can you expect query results to be consistent ? How can you tag your results or implement a sharding strategy in order to determine whether results are consistent?
- Querying data represented on different models: How can you exploit the structure of the different stores for expressing queries ? Provide adapted operators? Give generic operators and then rewrite queries?
- Normally, Polyglot solutions tend to solve some data processing issues in the application code. This can be penalizing. Discuss the challenges to address for ensuring that your queries will be able to scale as the collection grows.

# Challenge: expected results

- Give the principle of your proposal through a **partial programming solution**, of the operators of your UnQL extension, detail the query evaluation process if U want your solution to scale
  - We ask U to sketch the solution on the polyglot database that we provide consisting of mongo, Neo4J stores
  - <https://github.com/jccastrejon/edbt-unql>
  - Technical requirements: VMware player 5

# When is polyglot persistence pertinent?

- Application essentially composing and serving web pages
  - They only looked up page elements by ID, they had **different needs or availability, concurrency** and **no need to share** all their data
  - A problem like this is much better suited to a NoSQL store than the corporate relational DBMS
- **Scaling** to lots of traffic gets harder and harder to do with **vertical scaling**
  - Many NoSQL databases are designed to operate over clusters
  - They can **tackle larger volumes of traffic and data** than is realistic with a single server

# Conclusions

- Data are growing big and more heterogeneous and they need new adapted ways to be managed thus the NoSQL movement is gaining momentum
- Data heterogeneity implies different management requirements this is where polyglot persistence comes up
  - Consistency – Availability – Fault tolerance theorem: find the balance !
  - Which data store according to its data model?
  - A lot of programming implied ...

**Open opportunities** if you're interested in this topic!





**Merci**

**Thanks**

**Gracias**

*Contact:* Genoveva Vargas-Solar, CNRS, LIG-LAFMIA  
[Genoveva.Vargas@imag.fr](mailto:Genoveva.Vargas@imag.fr)  
<http://www.vargas-solar.com/teaching/>

**Open source polyglot persistence tools**

<http://code.google.com/p/exschema/>

<http://code.google.com/p/model2roo/>



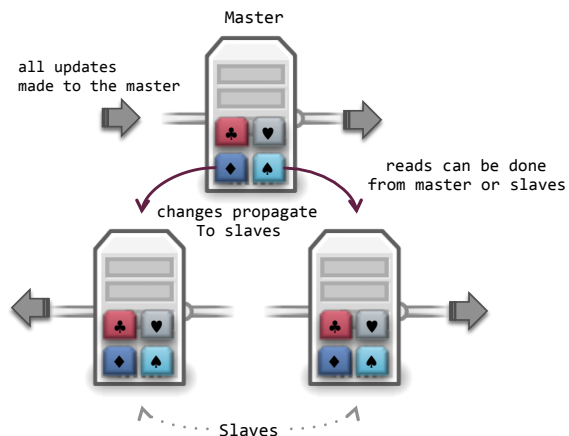
# References

- Eric A., Brewer "Towards robust distributed systems." PODC. 2000
- Rick, Cattell "Scalable SQL and NoSQL data stores." ACM SIGMOD Record 39.4 (2011): 12-27
- Juan Castrejon, Genoveva Vargas-Solar, Christine Collet, and Rafael Lozano, ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications, In Proceedings of the International Conference on Software Maintenance, Demo Paper, IEEE, 2013
- M. Fowler and P. Sadalage. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Pearson Education, Limited, 2012
- C. Richardson, Developing polyglot persistence applications, <http://fr.slideshare.net/chris.e.richardson/developing-polyglotpersistenceapplications-gluecon2013>

## NOSQL STORES: AVAILABILITY AND PERFORMANCE



# Replication master - slave

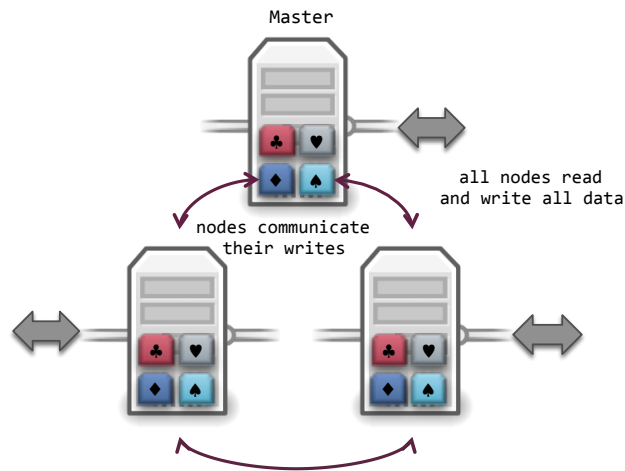


- Makes one node the authoritative copy/replica that handles writes while replica synchronize with the master and may handle reads
- All replicas have the same weight
  - Replicas can all accept writes
  - The lose of one of them does not prevent access to the data store
- Helps with read scalability but does not help with write scalability
- Read resilience: should the master fail, slaves can still handle read requests
- Master failure eliminates the ability to handle writes until either the master is restored or a new master is appointed
- Biggest complication is consistency
  - Possible write – write conflict
  - Attempt to update the same record at the same time from to different places
- Master is a bottle-neck and a point of failure

# Master-slave replication management

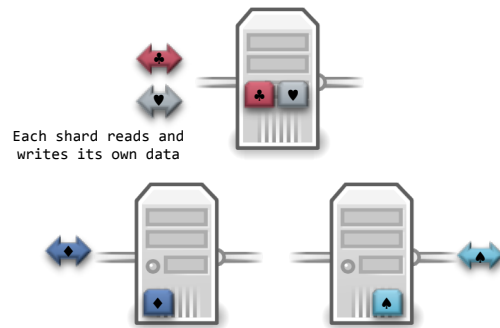
- Masters can be appointed
  - Manually when configuring the nodes cluster
  - Automatically: when configuring a nodes cluster one of them elected as master. The master can appoint a new master when the master fails reducing downtime
- Read resilience
  - Read and write paths have to be managed separately to handle failure in the write path and still reads can occur
  - Reads and writes are put in different database connections if the database library accepts it
- Replication comes inevitably with a dark side: **inconsistency**
  - Different clients reading different slaves will see different values if changes have not been propagated to all slaves
  - In the worst case a client cannot read a write it just made
  - Even if master-slave is used for hot backups, if the master fails any updates on to the backup are lost

# Replication: peer-To-Peer



- Allows writes to any node; the nodes coordinate to synchronize their copies
- The replicas have equal weight
- Deals with inconsistencies
  - Replicas coordinate to avoid conflict
  - Network traffic cost for coordinating writes
  - Unnecessary to make all replicas agree to write, only the majority
  - Survival to the loss of the minority of replicas nodes
  - Policy to merge inconsistent writes
  - Full performance on writing to any replica

# Sharding



- Ability to distribute both data and load of simple operations over many servers, with no RAM or disk shared among servers
  - A way to horizontally scale **writes**
  - Improve **read** performance
  - Application/data store support
- Puts different data on separate nodes
    - Each user only talks to one server so she gets rapid responses
    - The load should be balanced out nicely between servers
  - Ensure that
    - data that is accessed together is clumped together on the same node
    - that clumps are arranged on the nodes to provide best data access

# Sharding

## Database laws

- Small databases are fast
- Big databases are slow
- Keep databases small



## Principle

- Start with a big monolithic database
  - Break into smaller databases
  - Across many clusters
  - Using a key value

*Instead of having one million customers information on a single big machine ....*

*100 000 customers on **smaller** and **different** machines*

# Sharding criteria

- Partitioning
  - Relational: handled by the DBMS (homogeneous DBMS)
  - NoSQL: based on ranging of the k-value
- Federation
  - Relational
    - Combine tables stored in different physical databases
    - Easier with denormalized data
  - NoSQL:
    - Store together data that are accessed together
    - Aggregates unit of distribution

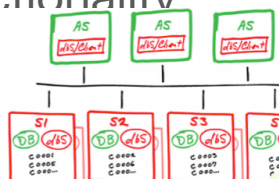




# Sharding

## Architecture

- Each application server (AS) is running DBS/client
- Each shard server is running
  - a database server
  - replication agents and query agents for supporting parallel query functionality



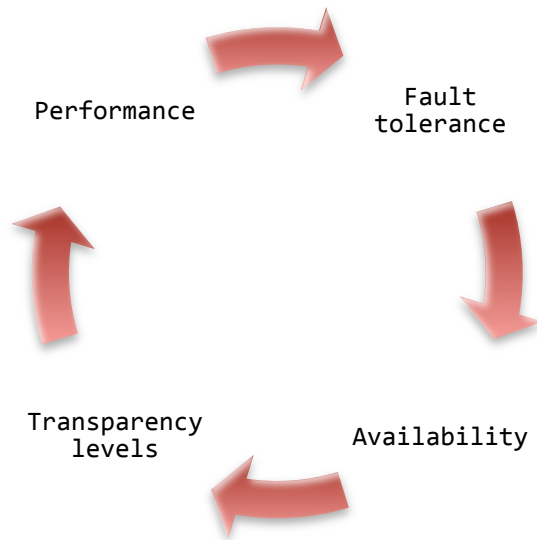
*Customers data is partitioned by ID in shards using an algorithm  $d$  to determine which shard a customer ID belongs to*

## Process

- Pick a dimension that helps sharding easily (customers, countries, addresses)
- Pick strategies that will last a long time as repartition/re-sharding of data is operationally difficult
- This is done according to two different principles
  - ▶ **Partitioning:** a partition is a structure that divides a space into two parts
  - ▶ **Federation:** a set of things that together compose a centralized unit but each individually maintains some aspect of autonomy

# Replication: aspects to consider

- Conditioning



- Important elements to consider

- Data to duplicate
- Copies location
- Duplication model (master – slave / P2P)
- Consistency model (global – copies)

→ **Find a compromise !**

# PARTITIONING

A PARTITION IS A STRUCTURE THAT DIVIDES A SPACE INTO TOW PARTS



# Background: distributed relational databases

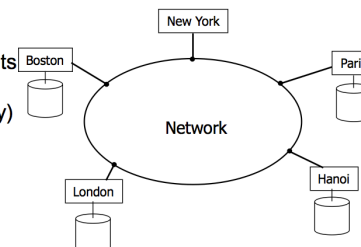
88

- External schemas (views) are often subsets of relations (contacts in Europe and America)
- Access defined on subsets of relations: 80% of the queries issued in a region have to do with contacts of that region
- Relations partition
  - Better concurrency level
  - Fragments accessed independently
- Implications
  - Check integrity constraints
  - Rebuild relations



## Background (distributed relational DBMS)

- External schemas (views) are often subsets of relations (offices of the same size in Paris, NY, Hanoi)
- Access are often defined on subsets of relations (80% of queries issued in a city affect the project of the city)
- Partition of relations
  - Better concurrency level (fragments are accessed independently)
- Price to pay
  - integrity constraints checking
  - rebuilding relations



- EMP(ENO, ENAME, TITLE)
- PROJ(PNO, PNAME, BUDGET, LOC)
- PAY(TITLE, SAL)
- ASG(ENO, PNO, DUR, RESP)

# Fragmentation

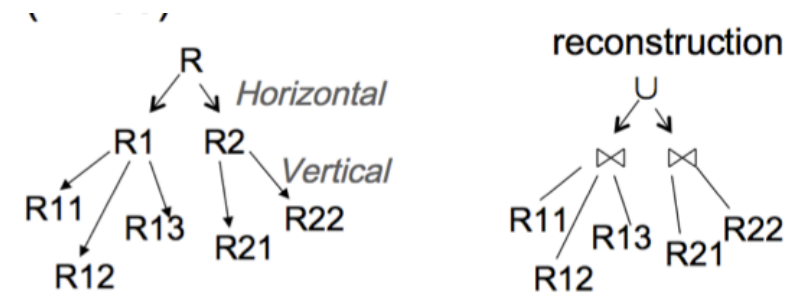
## ■ Horizontal

- Groups of tuples of the same relation
- Budget  $< 300\ 000$  or  $\geq 150\ 000$
- Not disjoint are more difficult to manage

## ■ Vertical

- Groups attributes of the same relation
- Separate budget from loc and pname of the relation project

## ■ Hybrid



# Fragmentation: rules

## Vertical

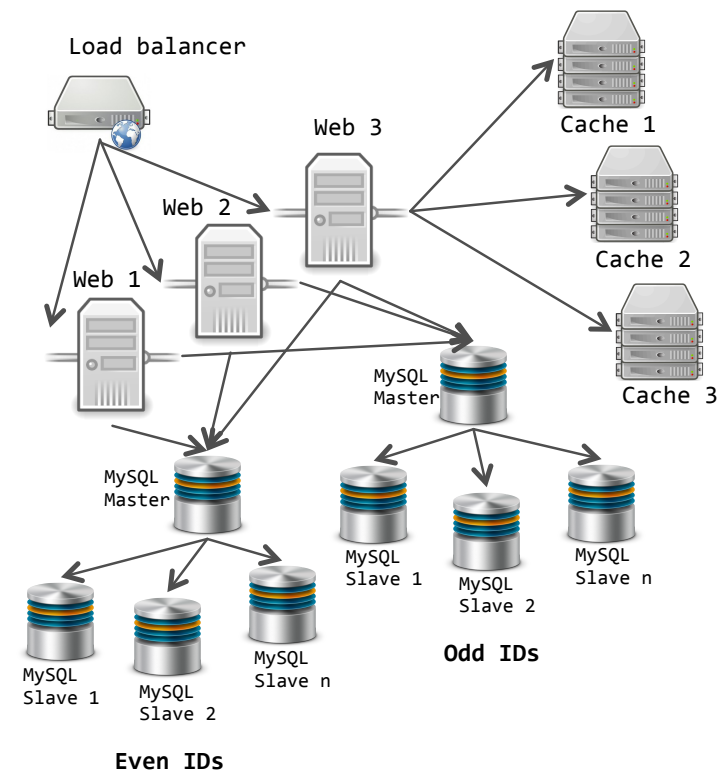
- Clustering
  - Grouping elementary fragments
  - Budget and location information in two relations
- Splitting
  - Decomposing a relation according to affinity relationships among attributes

## Horizontal

- Tuples of the same fragment must be statistically homogeneous
  - If  $t_1$  and  $t_2$  are tuples of the same fragment then  $t_1$  and  $t_2$  have the same probability of being selected by a query
- Keep important conditions
  - Complete
    - Every tuple (attribute) belongs to a fragment (without information loss)
    - If tuples where budget  $\geq 150\,000$  are more likely to be selected then it is a good candidate
  - Minimum
    - If no application distinguishes between budget  $\geq 150\,000$  and budget  $< 150\,000$  then these conditions are unnecessary

# Sharding: horizontal partitioning

- The entities of a database are split into two or more sets (by row)
- In relational: same schema several physical bases/servers
  - Partition contacts in Europe and America shards where they zip code indicates where they will be found
  - Efficient if there exists some robust and implicit way to identify in which partition to find a particular entity
- Last resort shard
  - Needs to find a sharding function: modulo, round robin, hash – partition, range - partition



# FEDERATION

A FEDERATION IS A SET OF THINGS THAT TOGETHER COMPOSE A CENTRALIZED UNIT BUT EACH INDIVIDUALLY MAINTAINS SOME ASPECT OF AUTONOMY

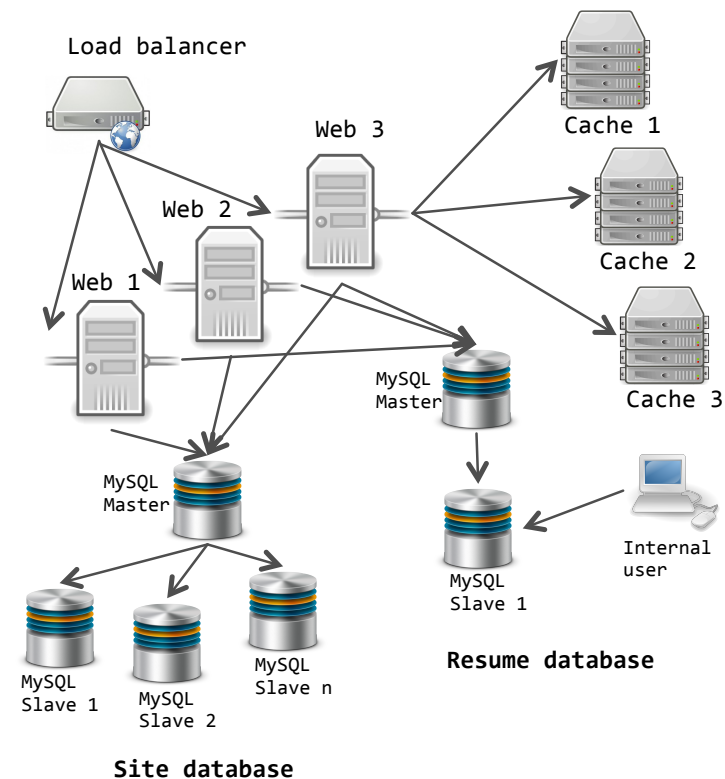




# FEDERATION: vertical SHARDING

93


- Principle
  - Partition data according to their logical affiliation
  - Put together data that are commonly accessed
- The search load for the large partitioned entity can be split across multiple servers (logical and physical) and not only according to multiple indexes in the same logical server
- Different schemas, systems, and physical bases/ servers
- Shards the components of a site and not only data



## NOSQL STORES: PERSISTENCY MANAGEMENT



# «*memcached*»

- «*memcached*» is a memory management protocol based on a cache:
  - Uses the key-value notion
  - Information is completely stored in RAM
- «*memcached*» protocol for:
  - Creating, retrieving, updating, and deleting information from the database
  - Applications with their own «*memcached*» manager (Google, Facebook, YouTube, FarmVille, Twitter, Wikipedia)

# Storage on disc (1)

- For efficiency reasons, information is stored using the RAM:
  - Work information is in RAM in order to answer to low latency requests
  - Yet, this is not always possible and desirable
- **The process of moving data from RAM to disc is called "*eviction*"; this process is configured automatically for every bucket**

## Storage on disc (2)

- NoSQL servers support the storage of key-value pairs on disc:
  - **Persistency**—can be executed by loading data, closing and reinitializing it without having to load data from another source
  - **Hot backups**—loaded data are stored on disc so that it can be reinitialized in case of failures
  - **Storage on disc**—the disc is used when the quantity of data is higher than the physical size of the RAM, frequently used information is maintained in RAM and the rest is stored on disc

# Storage on disc (3)

- Strategies for ensuring:
  - Each node maintains in RAM information on the key-value pairs it stores.  
Keys:
    - may not be found, or
    - they can be stored in memory or on disc
  - The process of moving information from RAM to disc is asynchronous:
    - The server can continue processing new requests
    - A queue manages requests to disc
- **In periods with a lot of writing requests, clients can be notified that the server is temporarily out of memory until information is evicted**



## NOSQL STORES: CONCURRENCY CONTROL



# Multi version concurrency control (MVCC)



- **Objective:** Provide concurrent access to the database and in programming languages to implement transactional memory
- **Problem:** If someone is reading from a database at the same time as someone else is writing to it, the reader could see a half-written or inconsistent piece of data.
- Lock: readers wait until the writer is done
- MVCC:
  - Each user connected to the database sees a snapshot of the database at a particular instant in time
  - Any changes made by a writer will not be seen by other users until the changes have been completed (until the transaction has been committed)
  - When an MVCC database needs to update an item of data it marks the old data as obsolete and adds the newer version elsewhere → multiple versions stored, but only one is the latest
  - Writes can be isolated by virtue of the old versions being maintained
  - Requires (generally) the system to periodically sweep through and delete the old, obsolete data objects