DISTRIBUTING DATA ENSURING CAP PROPERTIES

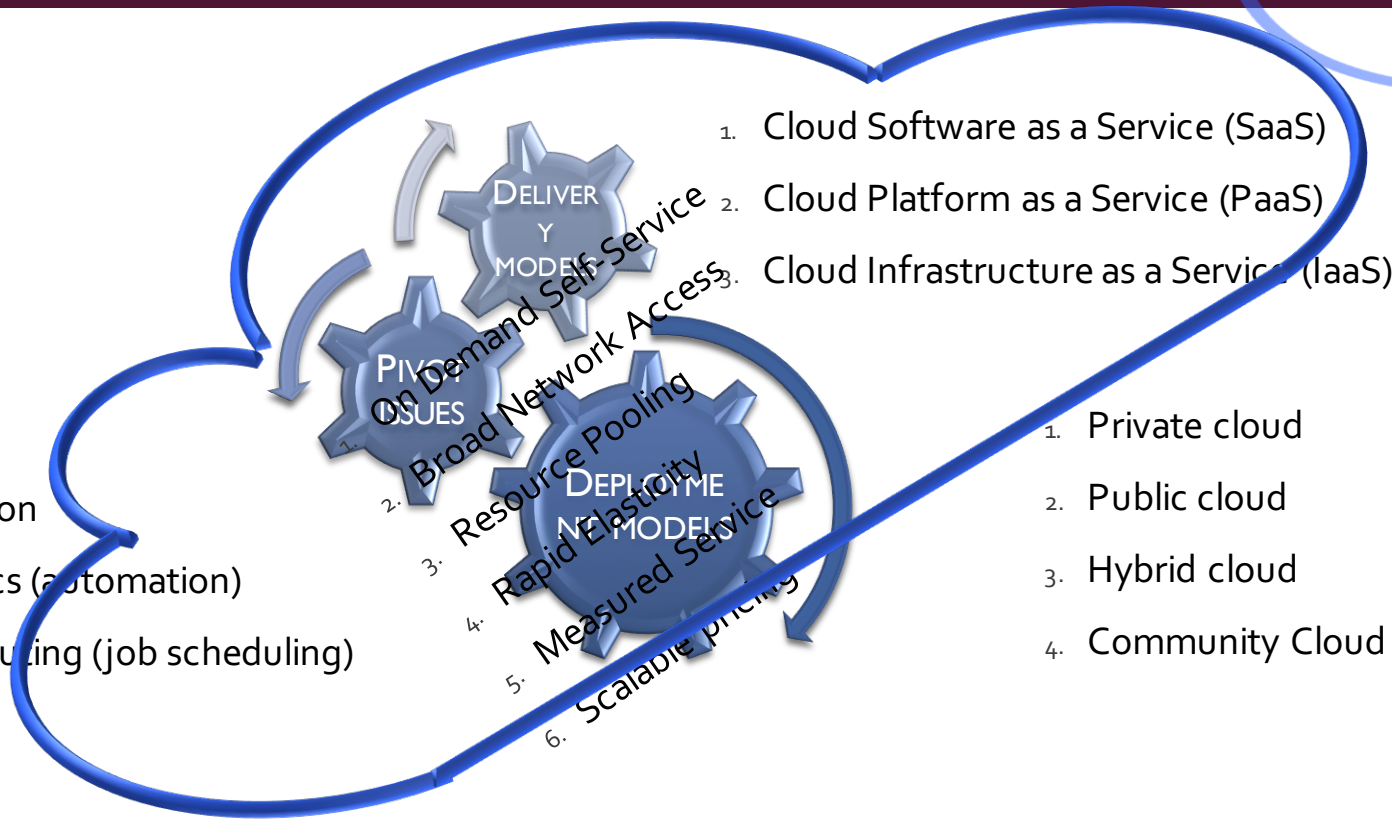
GENOVEVA VARGAS SOLAR

FRENCH COUNCIL OF SCIENTIFIC RESEARCH, LIG-LAFMIA, FRANCE

Genoveva.Vargas@imag.fr

http://www.vargas-solar.com/data-management-services-cloud

CLOUD IN A NUTSHELL



1. Virtualization
2. Autonomics (automation)
3. Grid Computing (job scheduling)

1. Cloud Software as a Service (SaaS)
2. Cloud Platform as a Service (PaaS)
3. Cloud Infrastructure as a Service (IaaS)

1. Private cloud
2. Public cloud
3. Hybrid cloud
4. Community Cloud

HOW TO MAP DATA MANAGEMENT IN THE CLOUD ?

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?

SHARDING

- A database shard is a ***horizontal partition*** in a database or search engine
  - Rows of a database table are held separately, rather than being split into columns (which is what normalization and vertical partitioning do, to differing extents)
  - Each individual partition is referred to as a ***shard*** or ***database shard***
  - A shard, may be located on a **separate** database server or physical location

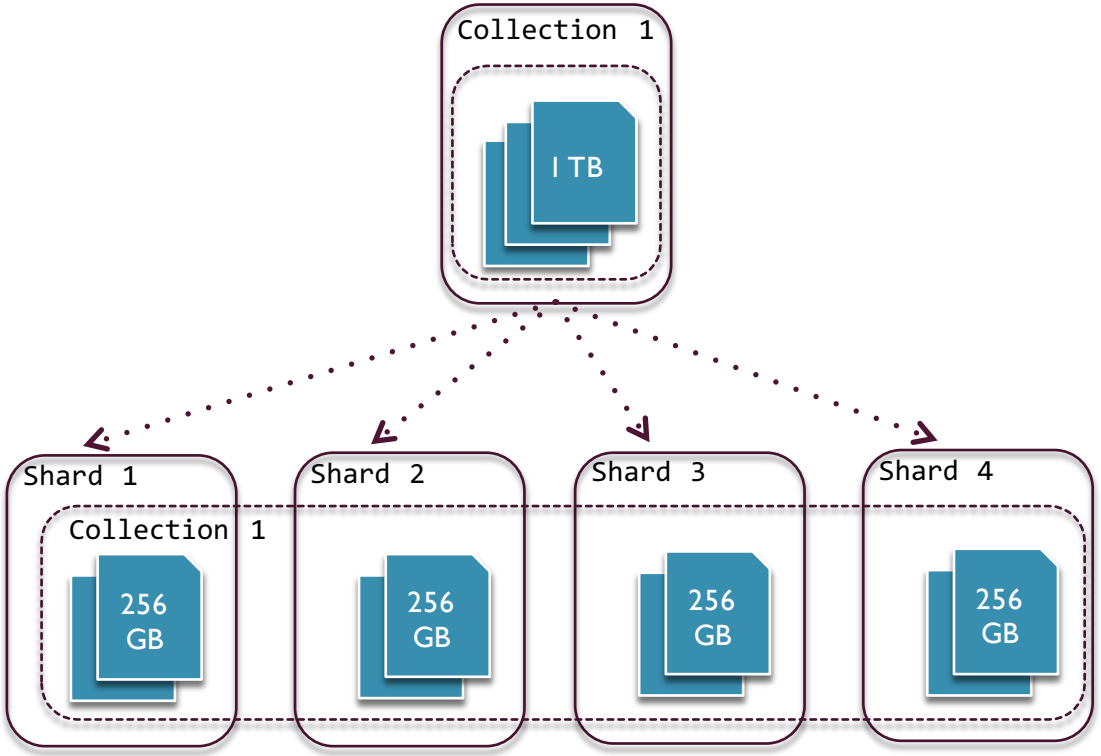
SHARDING STRATEGIES

USE CASE MONGODB

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

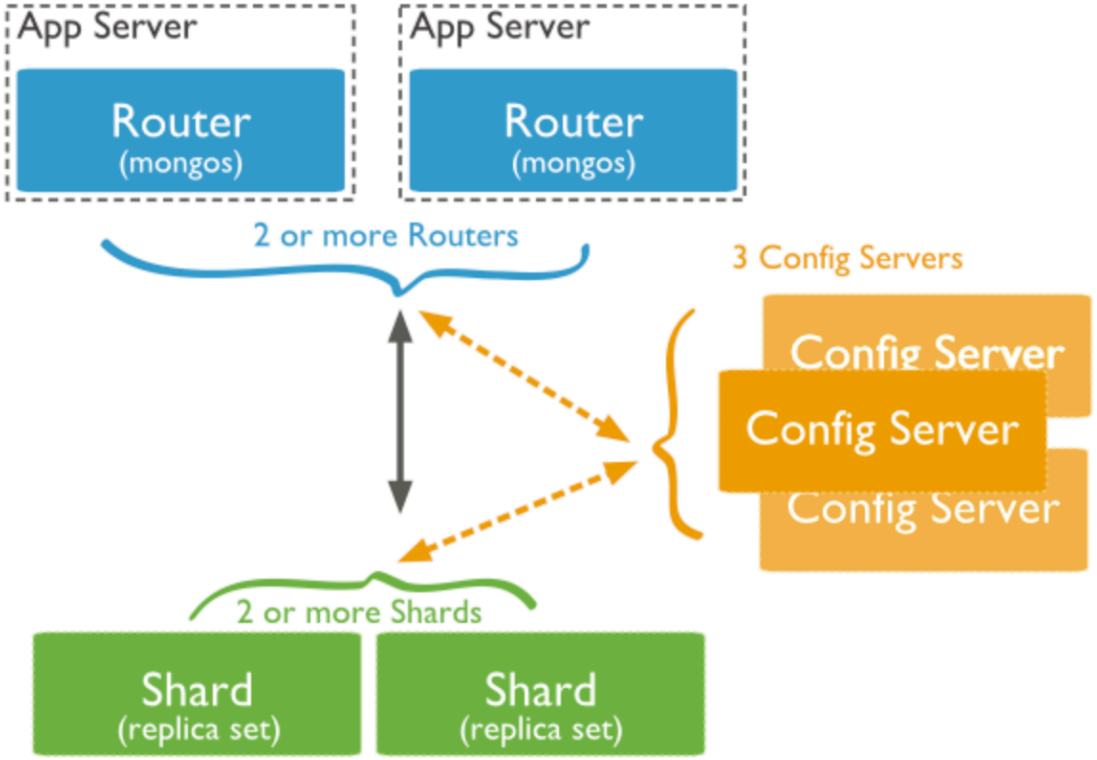
SHARDING DATA



SHARDING

- 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

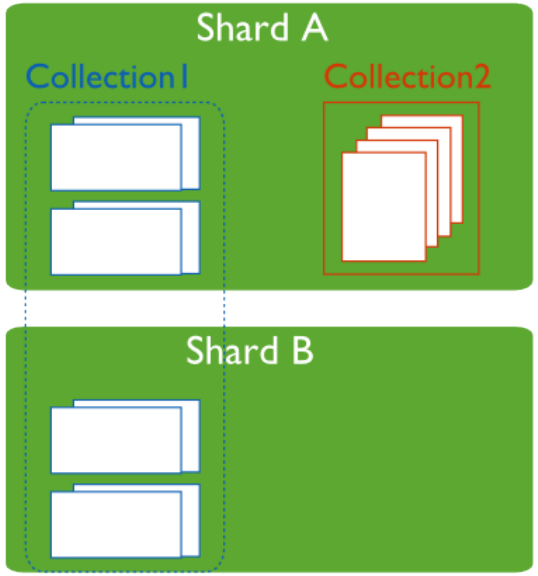


SHARDING IN MONGO

- **Shards (mongd)** 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 `Sharding Version` section displays information on the config database:

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

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 database has sharding enabled, and the primary shard for the database.

```
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 documents across chunks
  - the tag information, if any, for shard key range(s)

```
<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

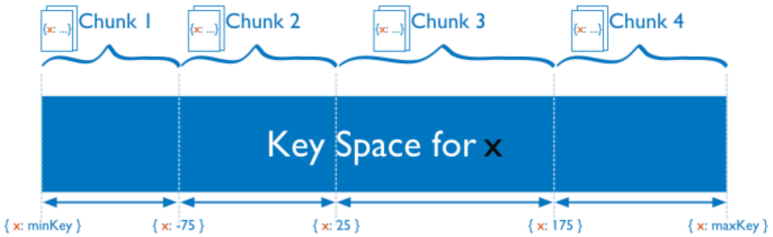
- Config servers store the metadata in the config database. The mongos instances cache this data and use it to route reads and writes to shards
- MongoDB only writes data to the config server in the following cases:
  - To create splits in existing chunks.
  - To migrate a chunk between shards.
- MongoDB 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

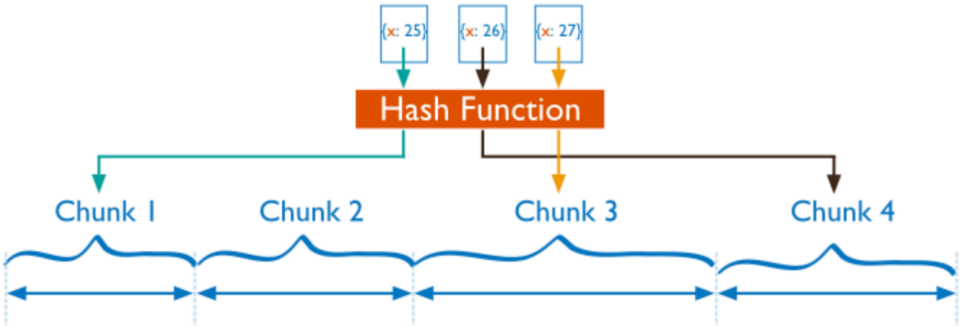
RANGE BASED SHARDING

- 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

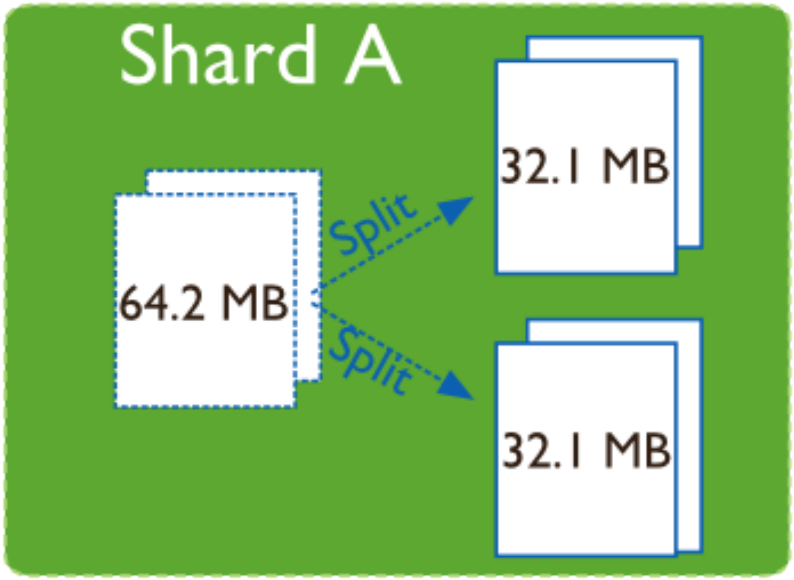
MAINTAINING A BALANCED DATA DISTRIBUTION

- 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

- 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



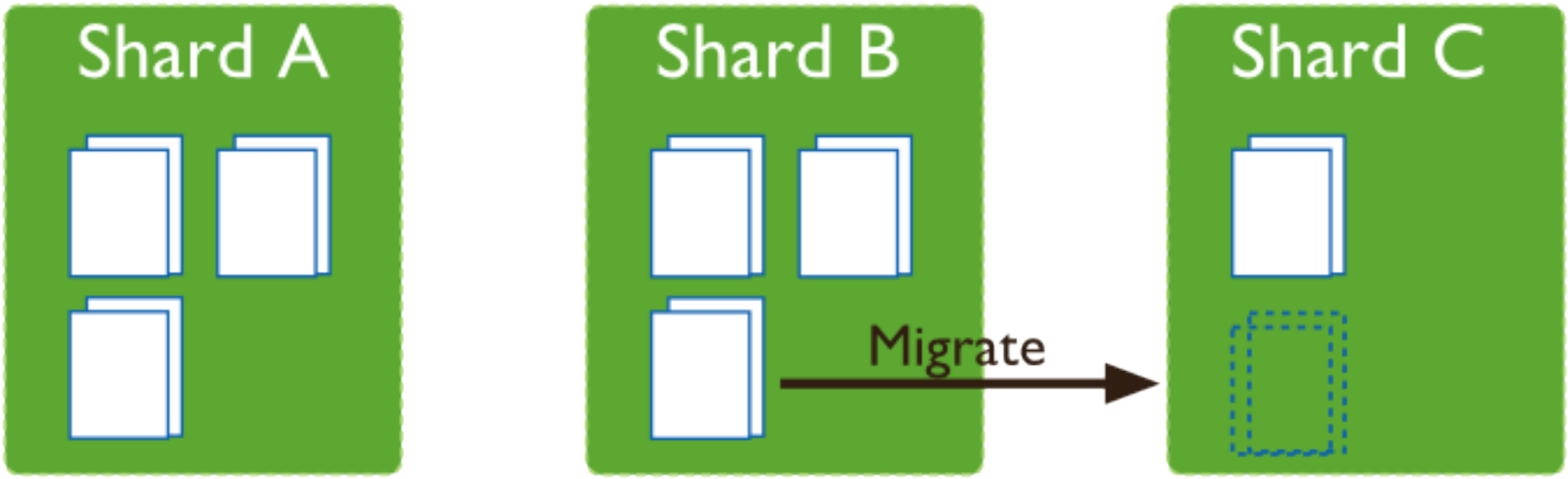
BALANCING (1)

- 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)

- 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)



ADDING AND REMOVING SHARDS FORM A CLUSTER

- 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

- *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 operations
- *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

1 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

- 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

- 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

- 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?*

hands on

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)

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

- Not linearizable
  - 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