

MongoDB by 10gen (now MongoDB Inc.)

**Dr. Chetan Jaiswal,
Department of Computer Science,
Truman State University
cjaiswal@truman.edu**

Overview

In one day:
24 million transactions processed by Walmart
4000 TB (4 PB) of data uploaded to Facebook
656 million tweets on Twitter

.....



How to **store**, **query** and **process** these data efficiently?

Overview

- The problems with Relational Database:
 - Overhead for complex select, update, delete operations
 - Select: Joining too many tables to create a huge size table.
 - Update: Each update affects many other tables.
 - Delete: Must guarantee the consistency of data.
 - Not well-supported the mix of unstructured data.
 - Not well-scaling with very large size of data.

NoSQL is a good solution to deal with these problems.

Overview















- What is NoSQL:
 - NoSQL = **Non SQL** or **Not only SQL**
 - Wikipedia's definition:

A **NoSQL** database provides a mechanism for storage and retrieval of data that is modeled in **means other than** the **tabular relations** used in relational databases.

Overview – NoSQL Family

Data stored in 4 types:

- Document
- Graph
- Key-value
- Wide-column

Document Database	Graph Databases
  	 
Wide Column Stores	Key-Value Databases
   	    

What is it?

- A document-oriented database – documents encapsulate and encode data (or information) in some standard formats or encodings
- NoSQL database – non-adherence to the widely used relational database – highly optimized for retrieve and append operations
- uses BSON format (<http://bsonspec.org/>), binary JSON
- schema-less – No more configuring database columns with types
- No transactions
- No joins
- "humongous" – huge, monstrous (data)
- Jack of all trades
 - Gateway to NoSQL from SQL
 - Mixing RDBMS and KV features

What is it?

Why MongoDB?

Document-oriented

Documents (objects) map nicely to programming language data types

Embedded documents and arrays reduce need for joins

Dynamically-typed (schemaless) for easy schema evolution

No joins and **no multi-document transactions** for high performance and easy scalability

High performance

No joins and embedding makes reads and writes fast

Indexes including indexing of keys from embedded documents and arrays

Optional streaming writes (no acknowledgements)

High availability

Replicated servers with automatic master failover

Easy scalability

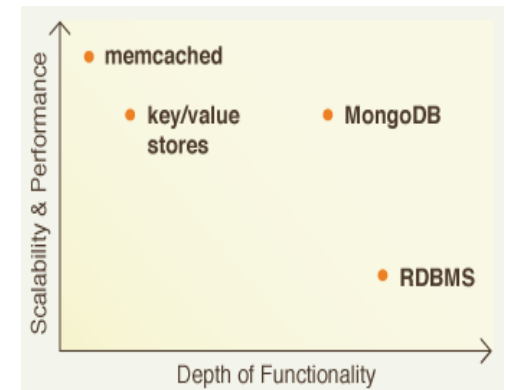
Automatic sharding (auto-partitioning of data across servers)

Reads and writes are distributed over shards

No joins or multi-document transactions make distributed queries easy and fast

Eventually-consistent reads can be distributed over replicated servers

Rich query language



Connectivity?

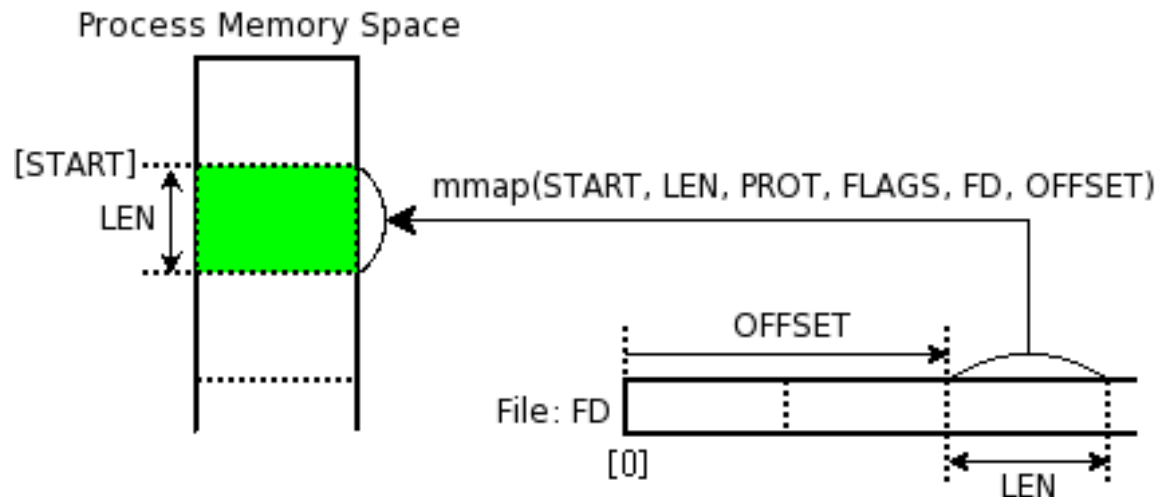
MongoDB currently has client support for the following programming languages:

mongodb.org Supported

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

The Basics

- MongoDB is a **server process** that runs on Linux, Windows and OS X.
- Clients **connect** to the MongoDB process, optionally authenticate themselves if security is turned on, and perform a sequence of actions, such as **inserts, queries and updates**.
- MongoDB stores its data in files (default location is /data/db/), and uses **memory mapped files** for data management for efficiency.
- A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource.
(mmap())



The Basics

- A MongoDB instance may have zero or more databases
- A database may have zero or more collections.
 - Can be thought of as the relation (table) in DBMS, but with many differences.
- A collection may have zero or more documents.
 - Docs in the same collection don't even need to have the same fields
 - Docs are the records in RDBMS
 - Docs can embed other documents
 - Documents are addressed in the database via a unique key
- A document may have one or more fields.
- MongoDB Indexes is much like their RDBMS counterparts.

The Basics

A Document:

```
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [
    {
      street: "123 Fake Street",
      city: "Faketon",
      state: "MA",
      zip: "12345"
    },
    {
      street: "1 Some Other Street",
      city: "Boston",
      state: "MA",
      zip: "12345"
    }
  ]
}
```

The Basics

- MongoDB is a **collection-oriented, schema-free** document database.
- By *collection-oriented*, we mean that data is grouped into sets that are called 'collections'. Each collection has a unique name in the database, and can contain an unlimited number of documents. Collections are analogous to tables in a RDBMS, except that they don't have any defined schema.
- By *schema-free*, we mean that the database doesn't need to know anything about the structure of the documents that you store in a collection. In fact, you can store documents with different structure in the same collection if you so choose.
- By *document*, we mean that we store data that is a structured collection of key-value pairs, where **keys are strings**, and **values are** any of a rich set of data types, including arrays and **documents**. We call this data format "BSON" for "Binary Serialized dOcument Notation.,,
- A **database** holds a set of collections
- A **collection** holds a set of documents

<https://docs.mongodb.com/manual/reference/bson-types/>

A BSON Document

- { author: 'joe',
- created : new Date(),
- title : 'Yet another blog post',
- text : 'Here is the text...',
- tags : ['example', 'joe'],
- comments : [- { author: 'jim',
- comment: 'I disagree'
- },
- { author: 'nancy',
- comment: 'Good post'
- }
-]
- }
- This document is a blog post, so we can store in a "posts" collection using the shell:
- > doc = { author : 'joe', created : new Date('03/28/2009'), ... }
- > db.posts.insert(doc);

BSON Query

- **Queries** are expressed as BSON documents which indicate a query pattern.

```
db.users.find({'last_name': 'Smith'})
```

```
db.users.find( { x : 3, y : "abc" } ).sort({x:1});
```

```
// select * from users where x=3 and y='abc' order by x asc;
```

```
SELECT * FROM users WHERE age>33
```

```
db.users.find({age:{$gt:33}})
```

```
SELECT * FROM users WHERE age!=33
```

```
db.users.find({age:{$ne:33}})
```

```
SELECT * FROM users WHERE name LIKE "%Joe%"
```

```
db.users.find({name:/Joe/})
```

```
SELECT * FROM users WHERE a=1 and b='q'
```

```
db.users.find({a:1,b:'q'})
```

```
SELECT * FROM users WHERE a=1 or b=2
```

```
db.users.find( { $or : [ { a : 1 }, { b : 2 } ] } )
```

```
SELECT * FROM foo WHERE name='bob' and (a=1 or b=2 )
```

```
db.foo.find( { name : "bob" , $or : [ { a : 1 }, { b : 2 } ] } )
```

```
SELECT * FROM users WHERE age>33 AND age<=40
```

```
db.users.find({'age':{$gt:33,$lte:40}})
```

```
SELECT * from users where x=3 and y='abc' order by x asc;
```

BSON Query

- The simplest type of update fully replaces a matching document with a new one. This can be useful to do a dramatic schema migration. For example, suppose we are making major changes to a user document, which looks like the following:

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "name" : "joe",
  "friends" : 32,
  "enemies" : 2
}
```

We want to move the "friends" and "enemies" fields to a "relationships" subdocument. We can change the structure of the document in the shell and then replace the database's version with an update:

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
  "friends" : 32,
  "enemies" : 2
}
> joe.username = joe.name;
"joe"
> delete joe.friends;
true
> delete joe.enemies;
true
> delete joe.name;
true
> db.users.update({"name" : "joe"}, joe);
Now, doing a findOne shows that the structure of the document has been updated:
```

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "username" : "joe",
  "relationships" : {
    "friends" : 32,
    "enemies" : 2
  }
}
```

Example - Blog

- Example: A Blog
- A blog post has an author, some text, and many comments
- The comments are unique per post, but one author has many posts •
- How would you design this in SQL?
- Collections for posts, authors, and comments
- References by manually created ID
- ```
post = {
 id: 150,
 author: 100,
 text: 'This is a pretty awesome post.',
 comments: [100, 105, 112]
}
author = {
 id: 100,
 name: 'Michael Arrington',
 posts: [150]
}
comment = {
 id: 105,
 text: 'Whatever this sux.'
}
```



# Example – Blog – Another way

- Collection for posts
- Embed comments, author name
- ```
post = {  
    author: 'Michael Arrington',  
    text: 'This is a pretty awesome post.',  
    comments: [  
        'Whatever this post sux.',  
        'I agree, lame!'  
    ]  
}
```

Why is this one better?

- Embedded objects brought back in the same query as parent object
- Only 1 trip to the DB server required
- Objects in the same collection are generally stored contiguously on disk
- Spatial locality = faster
- If the document model matches your domain well, it can be much easier to comprehend than nasty joins

Example – Blog – Another way

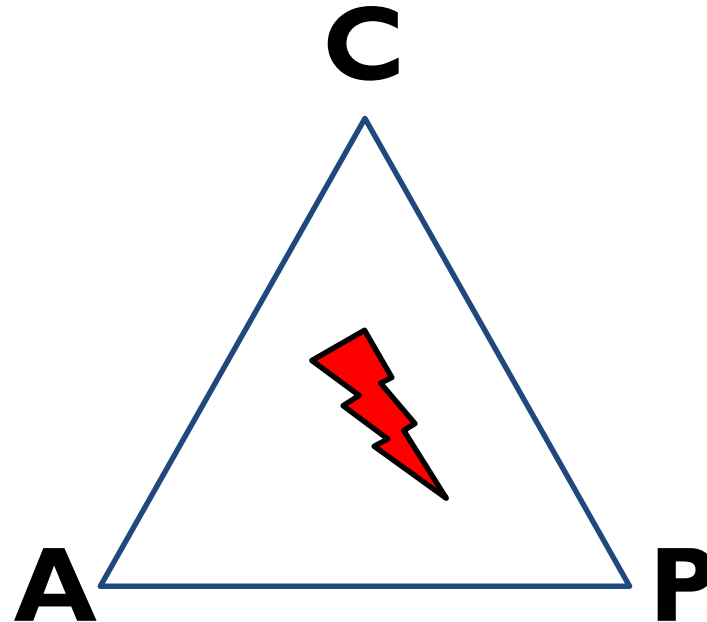
- Schema design in MongoDB is very different from schema design in a relational DBMS.
- However it is still very important and the first step towards building an application.
- In relational data models, conceptually there is a "correct" design for a given entity relationship model independent of the use case.
- This is typically a third normal form normalization. One typically only diverges from this for performance reasons.
- In MongoDB, the schema design is not only a function of the data to be modeled but also of the use case.
- The schema design is optimized for our most common use case.
- This has pros and cons – that use case is then typically highly performant; however there is a bias in the schema which may make certain ad hoc queries a little less elegant than in the relational schema.
- As we design the schema, the questions we must answer are:
 1. When do we embed data versus linking (see below)? Our decisions here imply the answer to question #2:
 2. How many collections do we have, and what are they?
 3. When do we need atomic operations? These operations can be performed within the scope of a BSON document, but not across documents.
 4. What indexes will we create to make query and updates fast?
 5. How will we shard? What is the shard key?

Index

- Indexes in MongoDB are similar to indexes in RDBMS.
- MongoDB supports indexes on any field or sub-field contained in documents
- MongoDB defines indexes on a percollection level.
- All MongoDB indexes use a B-tree data structure.
- **Choosing Indexes**
- A second aspect of schema design is index selection. As a general rule, where you want an index in a relational database, you want an index in Mongo.
- The `_id` field is automatically indexed.
- Fields upon which keys are looked up should be indexed.
- Sort fields generally should be indexed.
- The MongoDB profiling facility provides useful information for where an index should be added that is missing.
- Note that adding an index slows writes to a collection, but not reads. Use lots of indexes for collections with a high read : write ratio (assuming one does not mind the storage overage). For collections with more writes than reads, indexes are expensive as keys must be added to each index for each insert.
- Geospatial indexes... GIS

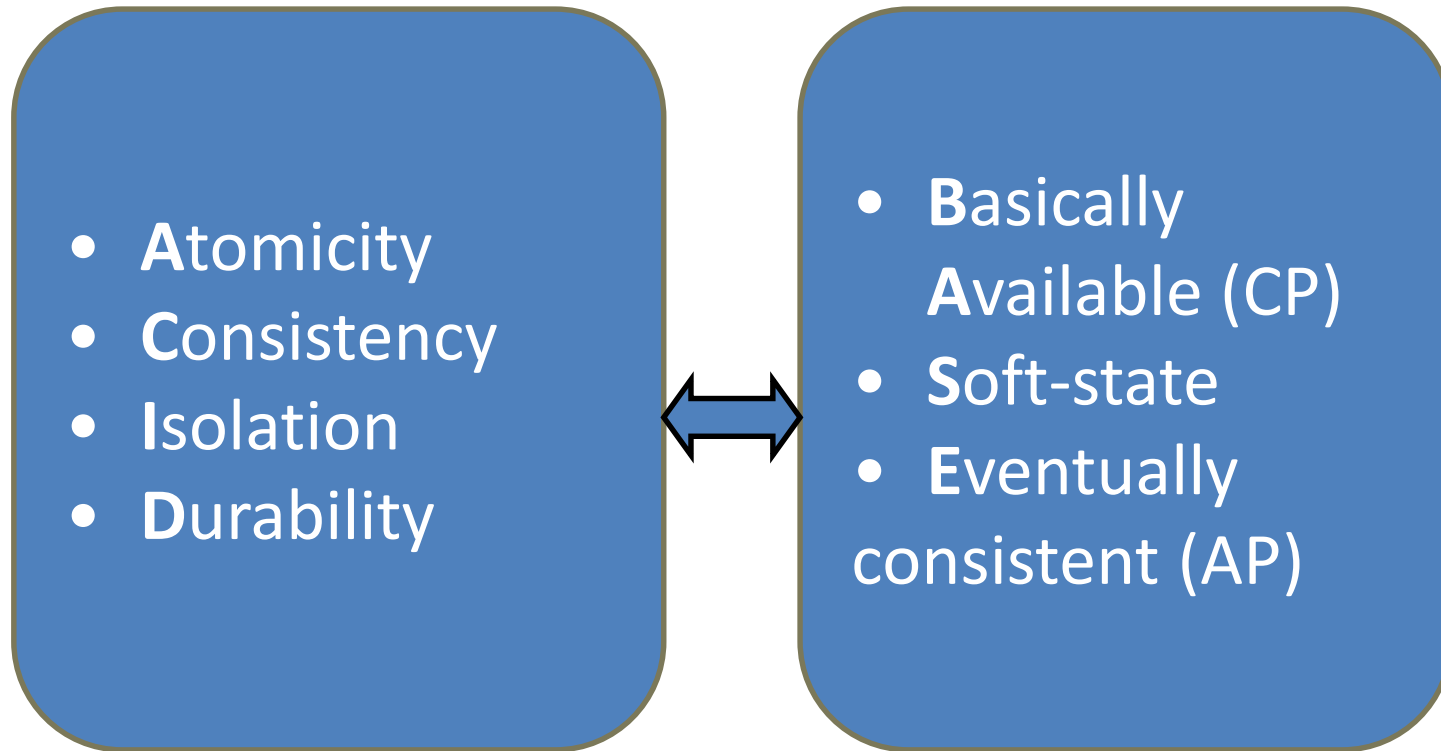
Theory of noSQL: CAP

- Many nodes
- Nodes contain *replicas of partitions* of data
- **Consistency**
 - all replicas contain the same version of data
- **Availability**
 - system remains operational on failing nodes
- **Partition tolerance**
 - multiple entry points
 - system remains operational on system split

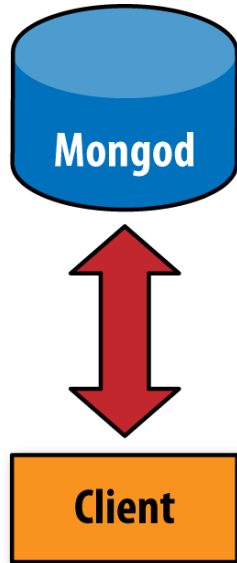


CAP Theorem:
satisfying all three at the
same time is impossible

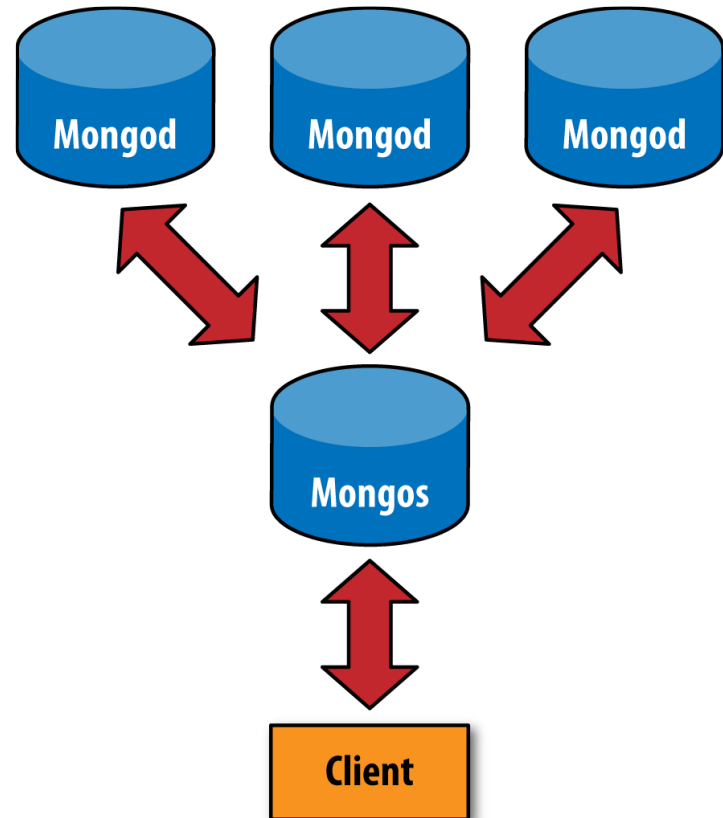
ACID - BASE



Sharding



Nonsharded client connection



Sharded client connection

Sharding

➤ `cluster = new ShardingTest({"shards" : 3, "chunksize" : 1})`

Running this command creates a new cluster with three shards (*mongod* processes) running on ports 30000, 30001, and 30002. By default, *ShardingTest* starts a *mongos* on port 30999. A routing process called *mongos* is started in front of the shards. This router keeps a “table of contents” that tells it which shard contains which data. Applications can connect to this router and issue requests normally. The router, knowing what data is on which shard, is able to forward the requests to the appropriate shard(s). If there are responses to the request, the router collects them, merges them, and sends them back to the application.

➤ `db = (new Mongo("localhost:30999")).getDB("test")`

```
> for (var i=0; i<100000; i++) {  
... db.users.insert({"username" : "user"+i, "created_at" : new Date()});  
... }  
> db.users.count()  
100000
```

Sharding

```
> sh.status()
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "localhost:30000" }
{ "_id" : "shard0001", "host" : "localhost:30001" }
{ "_id" : "shard0002", "host" : "localhost:30002" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : false, "primary" : "shard0001" }
```


Sharding

- `sh.enableSharding("test")`
- `db.users.ensureIndex({"username" : 1})`
- `sh.shardCollection("test.users", {"username" : 1})` //array cannot be the shard key, but there can be compound shard key i.e. with multiple fields provided an index is defined on both the fields
- `sh.status()`

```
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "localhost:30000" }
{ "_id" : "shard0001", "host" : "localhost:30001" }
{ "_id" : "shard0002", "host" : "localhost:30002" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
test.users chunks:
shard0001 4
shard0002 4
shard0000 5
{ "username" : { $minKey : 1 } }--> { "username" : "user1704" }
on : shard0001
{ "username" : "user1704" }--> { "username" : "user24083" }
on : shard0002
{ "username" : "user24083" }--> { "username" : "user31126" }
on : shard0001
{ "username" : "user31126" }--> { "username" : "user38170" }
on : shard0002
{ "username" : "user38170" }--> { "username" : "user45213" }
on : shard0001
{ "username" : "user45213" }--> { "username" : "user52257" }
on : shard0002
{ "username" : "user52257" }--> { "username" : "user59300" }
```

Sharding

➤ Choosing Shard Key??

➤ See: <https://docs.mongodb.com/manual/core/sharding-shard-key/>

➤ Use case:

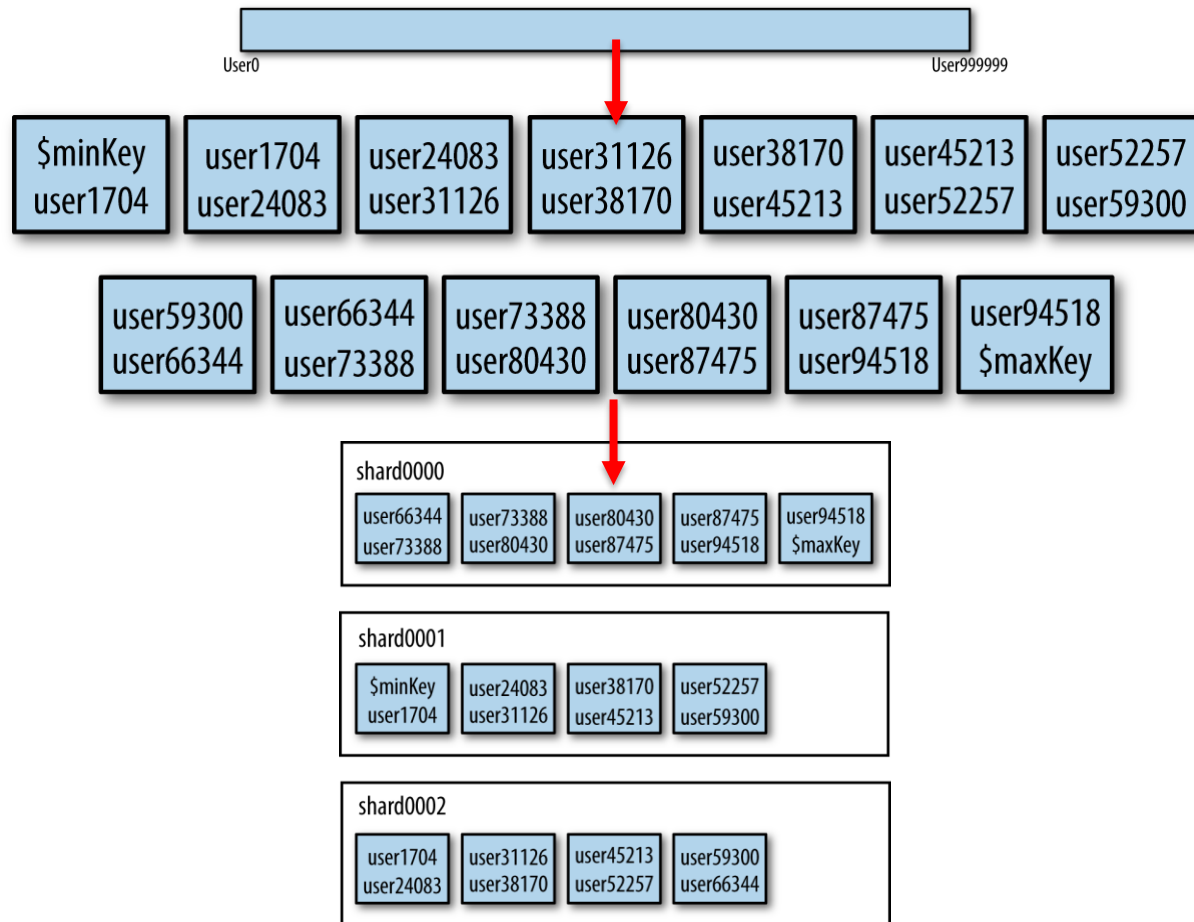
Online Shopping

Blogs

Hotel/Flight Reservations

Population Data

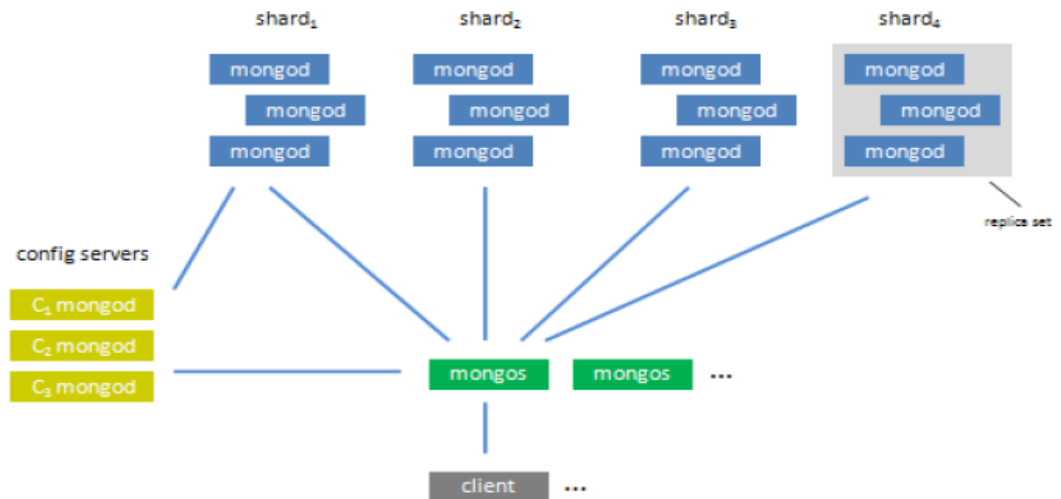
Sharding



Sharding

Sharding

- Automatic balancing for changes in load and data distribution
- Easy addition of new machines without down time
- Scaling to one thousand nodes
- No single points of failure
- Automatic failover



- One or more shards, **each shard holds a portion of the total data** (managed automatically). Reads and writes are automatically routed to the appropriate shard(s).
- **Each shard is backed by a replica set** – which just holds the data for that shard.
 - A replica set is one or more servers, each holding copies of the same data. At any given time one is primary and the rest are secondaries.
 - If the primary goes down one of the secondaries takes over automatically as primary.
 - All writes and consistent reads go to the primary, and all eventually consistent reads are distributed amongst all the secondaries. (slaveOk option)
 - A client can block until a write operation has been replicated. (getLastError command, to N servers or majority, timeout)
- Multiple **config servers**, each one holds a copy of the meta data indicating **which data lives on which shard**.
- One or more **routers**, each one acts as a server for one or more clients. Clients issue queries/updates to a router and the router routes them to the appropriate shard while consulting the config servers.
- One or more clients, each one is (part of) the user's application and issues commands to a router via the mongo client library (driver) for its language.

mongod is the server program (data or config). **mongos** is the router program.

Sharding

- Sharding is the partitioning of data among multiple machines in an order-preserving manner.
- Sharding is performed on a per-collection basis.
- Each shard stores multiple "chunks" of data, based on the shard key. MongoDB distributes these chunks evenly.
- In MongoDB, *sharding is the tool for scaling a system, and replication is the tool for data safety, high availability, and disaster recovery.*
 - The two work in tandem yet are orthogonal concepts in the design.
- MongoDB's auto-sharding scaling model shares many similarities with Yahoo's PNUTS and Google's BigTable.

Chunks on shard key **location**:

Machine 1	Machine 2	Machine 3
Alabama → Arizona	Colorado → Florida	Arkansas → California
Indiana → Kansas	Idaho → Illinois	Georgia → Hawaii
Maryland → Michigan	Kentucky → Maine	Minnesota → Missouri
Montana → Montana	Nebraska → New Jersey	Ohio → Pennsylvania
New Mexico → North Dakota	Rhode Island → South Dakota	Tennessee → Utah
	Vermont → West Virginia	Wisconsin → Wyoming

Sharding

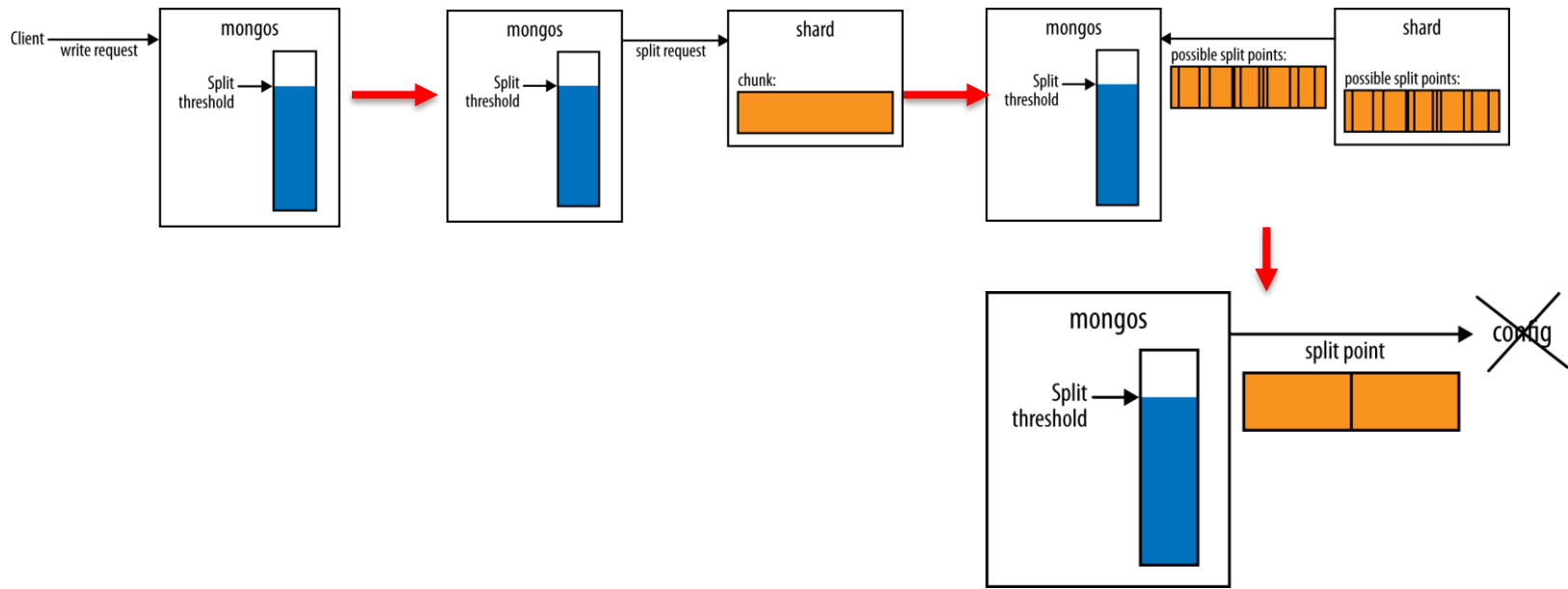
- A **chunk is a contiguous range of data** from a particular collection.
 - Chunks are described as a triple of collection, minKey, and maxKey.
 - Thus, the shard key K of a given document assigns that document to the chunk where $\text{minKey} \leq K < \text{maxKey}$.
- Chunks grow to a maximum size, usually 64MB.
 - Once a chunk has reached that approximate size, the chunk *splits* into two new chunks.
 - When a particular shard has excess data, chunks will then *migrate* to other shards in the system. The addition of a new shard will also influence the migration of chunks.
 - Balancing is necessary when the load on any one shard node grows out of proportion with the remaining nodes.
 - In this situation, the **data must be redistributed to equalize load** across shards.
- If it is possible that a single value within the shard key range might grow exceptionally large, it is best to **use a compound shard key** instead so that further discrimination of the values will be possible.

Config DB Processes

- Each config server has a complete copy of all chunk information.
- A two-phase commit is used to ensure the consistency of the configuration data among the config servers.
- Note that config server use their own replication model; they are not run in as a replica set.
- If any of the config servers is down, the cluster's meta-data goes read only.
 - However, even in such a failure state, the MongoDB cluster can still be read from and written to.

Sharding

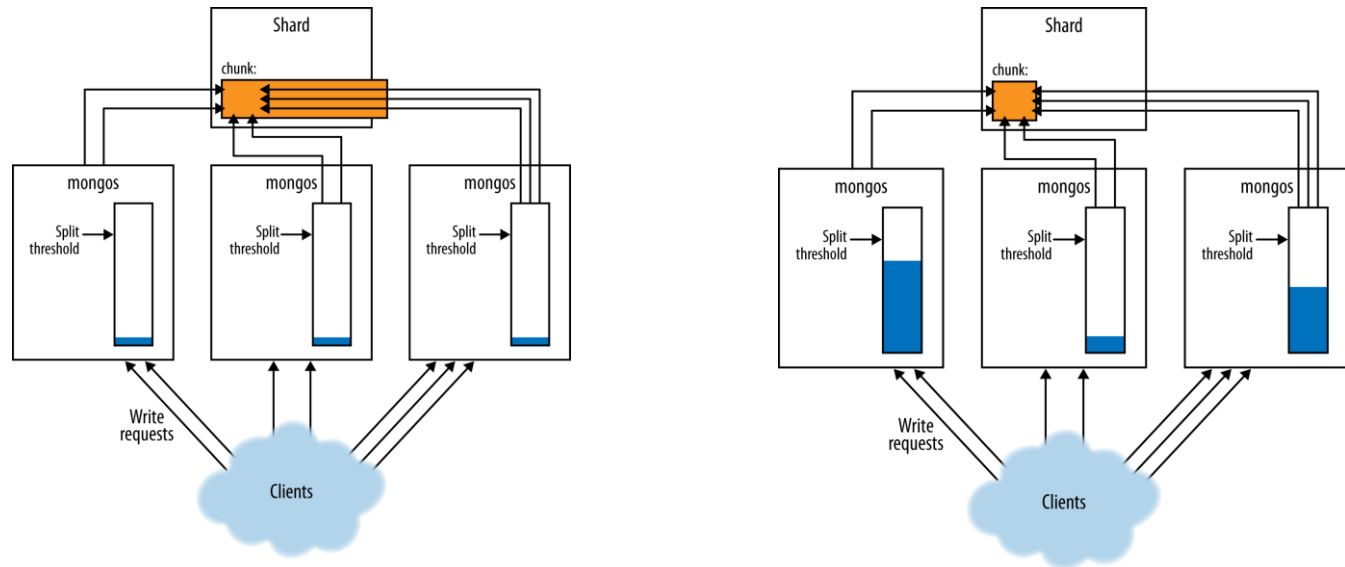
If one of the config servers is down when a *mongos* tries to do a split, the *mongos* won't be able to update the metadata. All config servers must be up and reachable for splits to happen. If the *mongos* continues to receive write requests for the chunk, it will keep trying to split the chunk and fail. As long as the config servers are not healthy, splits will continue not to work and all the split attempts can slow down the *mongos* and shard involved. This process of *mongos* repeatedly attempting to split a chunk and being unable to is called a *split storm*. The only way to prevent split storms is to ensure that your config servers are up and healthy as much of the time as possible. You can also restart a *mongos* to reset its write counter (so that it is no longer at the split threshold).



Sharding

Another issue is that *mongos* might never realize that it needs to split a large chunk. There is no global counter of how big each chunk is. Each *mongos* simply calculates whether the writes it has received have reached a certain threshold. This means that if your *mongos* processes go up and down frequently a *mongos* might never receive enough writes to hit the split threshold before it is shut down again and your chunks will get larger and larger.

- make the chunk size smaller than you actually want it to be. This will prompt splits to happen at a lower threshold.
- leave *mongos* processes up, when possible, instead of spinning them up when they are needed and then turning them off when they are not. However, some deployments may find it too expensive to run *mongos* processes that aren't being used.



Migration

The balancer is responsible for migrating data. It regularly checks for imbalances between shards and, if it finds an imbalance, will begin migrating chunks. Although the balancer is often referred to as a single entity, each mongos plays the part of “the balancer” occasionally.

Once a mongos has become the balancer, it checks its table of chunks for each collection to see if any shards have hit the balancing threshold. This is when one shard has significantly more chunks than the other shards (the exact threshold varies: larger collections tolerate larger imbalances than smaller ones).

If an imbalance is detected, the balancer will redistribute chunks until all shards are within one chunk of one another. If no collections have hit the balancing threshold. The mongos stops being the balancer.

Assuming that some collections have hit the threshold, the balancer will begin migrating chunks. It chooses a chunk from the overloaded shard and asks the shard if it should split the chunk before migrating. Once it does any necessary splits, it migrates the chunk to a machine with fewer chunks.

An application using the cluster does not need be aware that the data is moving: all reads and writes are routed to the old chunk until the move is complete. Once the metadata is updated, all mongos processes attempting to access the data in the old location will get an error. These errors should not be visible to the client: the mongos will silently handle the error and retry the operation on the new shard.

If the mongos is unable to retrieve the new chunk location because the config servers are unavailable, it will return an error to the client. This is another reason why it is important to always have config servers up and healthy.

Limitations

- No referential integrity
- High degree of denormalization means updating something in many places instead of one
- Lack of predefined schema is a double-edged sword
- You must have a model in your app
- Objects within a collection can be completely inconsistent in their fields
- Systems with a heavy emphasis on complex transactions such as banking systems and accounting. These systems typically require multi-object transactions, which MongoDB doesn't support. It's worth noting that, unlike many "NoSQL" solutions, MongoDB does support atomic operations on single documents. As documents can be rich entities; for many use cases, this is sufficient.
- Traditional Non-Realtime Data Warehousing. Traditional relational data warehouses and variants (columnar relational) are well suited for certain business intelligence problems – especially if you need SQL to use client tools (e.g. MicroStrategy) with the database. For cases where the analytics are realtime, the data very complicated to model in relational, or where the data volume is huge, MongoDB may be a fit.
- Problems requiring SQL.

What it is Good for?

- Archiving and event logging
- Document and Content Management Systems - as a document-oriented (JSON) database, MongoDB's flexible schemas are a good fit for this.
- ECommerce. Several sites are using MongoDB as the core of their ecommerce infrastructure (often in combination with an RDBMS for the final order processing and accounting).
- Gaming. High performance small read/writes are a good fit for MongoDB; also for certain games geospatial indexes can be helpful.
- High volume problems. Problems where a traditional DBMS might be too expensive for the data in question. In many cases developers would traditionally write custom code to a filesystem instead using flat files or other methodologies.
- Mobile. Specifically, the server-side infrastructure of mobile systems. Geospatial key here.
- Operational data store of a web site MongoDB is very good at real-time inserts, updates, and queries. Scalability and replication are provided which are necessary functions for large web sites' real-time data stores. Specific web use case examples:
 - content management
 - comment storage, management, voting
 - user registration, profile, session data
- Projects using iterative/agile development methodologies. Mongo's BSON data format makes it very easy to store and retrieve data in a documentstyle / "schemaless" format. Addition of new properties to existing objects is easy and does not generally require blocking "ALTER TABLE" style operations.
- Real-time stats/analytics

References

- MongoDB: The Definitive Guide, Second Edition, by Kristina Chodorow
- <https://dms.sztaki.hu/>
- <http://www.inf.elte.hu/>
- <http://docs.mongodb.org/manual/>