

MongoDB

A decorative horizontal bar composed of various colored segments (black, blue, yellow, cyan, light blue, dark blue, grey) arranged in a slightly wavy pattern.

Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.
www.fandsindia.com
fands@vsnl.com

Ground Rules

- ❑ **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- ❑ **If you have questions or issues, please let me know immediately.**
- ❑ **Let us be punctual.**

A decorative vertical bar on the left side of the page, composed of numerous horizontal segments in various shades of blue, black, and yellow, creating a striped effect.

Contents

What is MongoDB?

- The current SQL/NoSQL landscape
- Document-oriented vs. other types of storage
- Mongo's feature set
- Common use-cases
- Introduction to JSON

SQL has ruled for two decades

☐ Store persistent data

Storing large amounts of data on disk, while allowing applications to grab the bits they need through queries

☐ Application Integration

Many applications in an enterprise need to share information. By getting all applications to use the database, we ensure all these applications have consistent, up-to-date data

☐ Mostly Standard

The relational model is widely used and understood. Interaction with the database is done with SQL, which is a (mostly) standard language. This degree of standardization is enough to keep things familiar so people don't need to learn new things

☐ Concurrency Control

Many users access the same information at the same time. Handling this concurrency is difficult to program, so databases provide *transactions* to help ensure consistent interaction.

☐ Reporting

SQL's simple data model and standardization has made it a foundation for many reporting tools

SQL's dominance is cracking

Relational databases are designed to run on a single machine, so to scale, you need buy a bigger machine



But it's cheaper and more effective to *scale horizontally* by buying lots of machines.



Google



Bigtable

Amazon



Dynamo

www.fandsindia.com

NoSQL Databases

- 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.

NoSQL Databases

- While there is no formal definition, there are some common characteristics
 - 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

NoSQL Databases



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

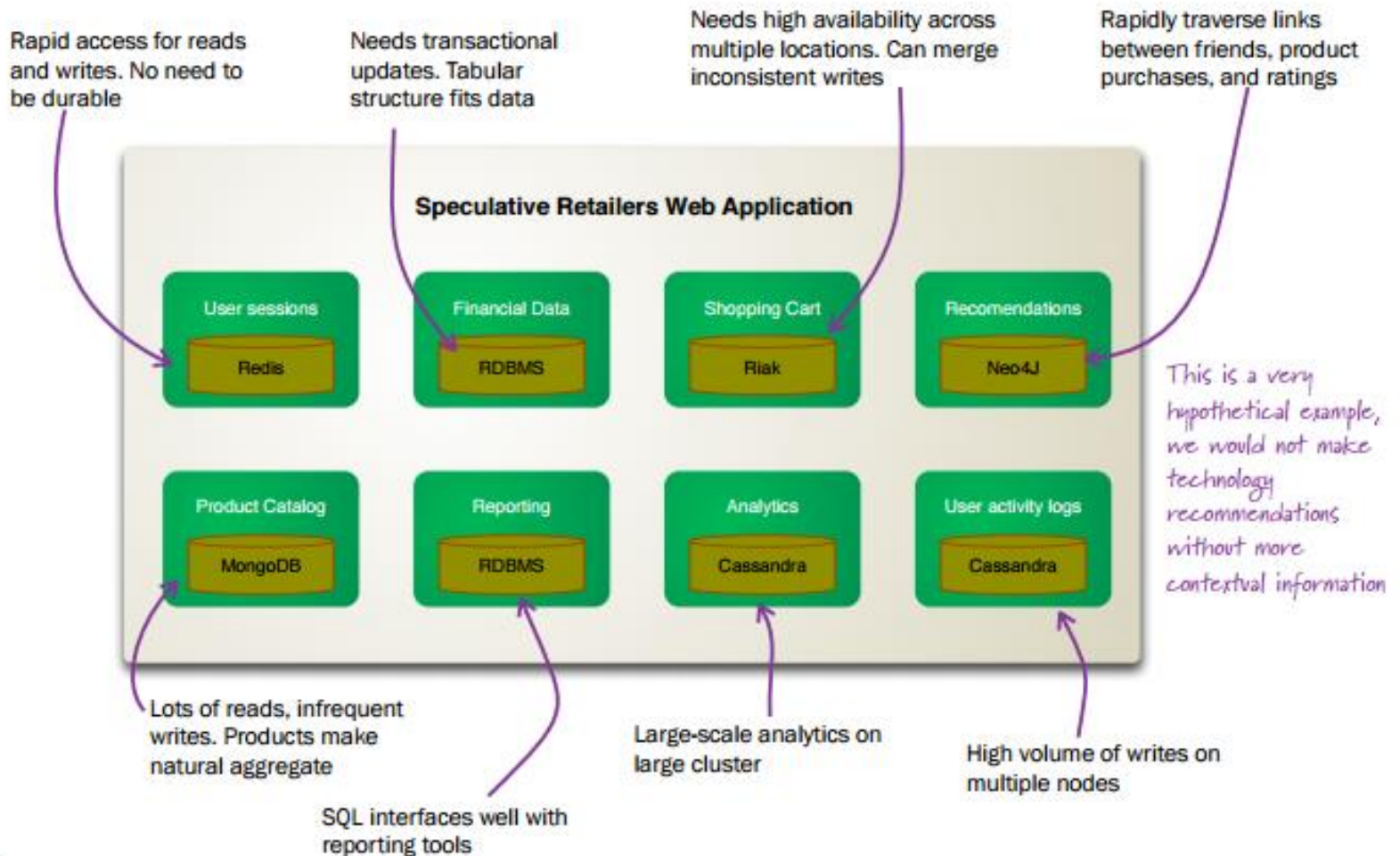
NoSQL Databases

- Main Advantages
 - Reduce Development Drag
 - Embrace Large Scale
- This does not mean Relational is dead
 - the relational model is still relevant
 - ACID transactions
 - Tools
 - Familiarity

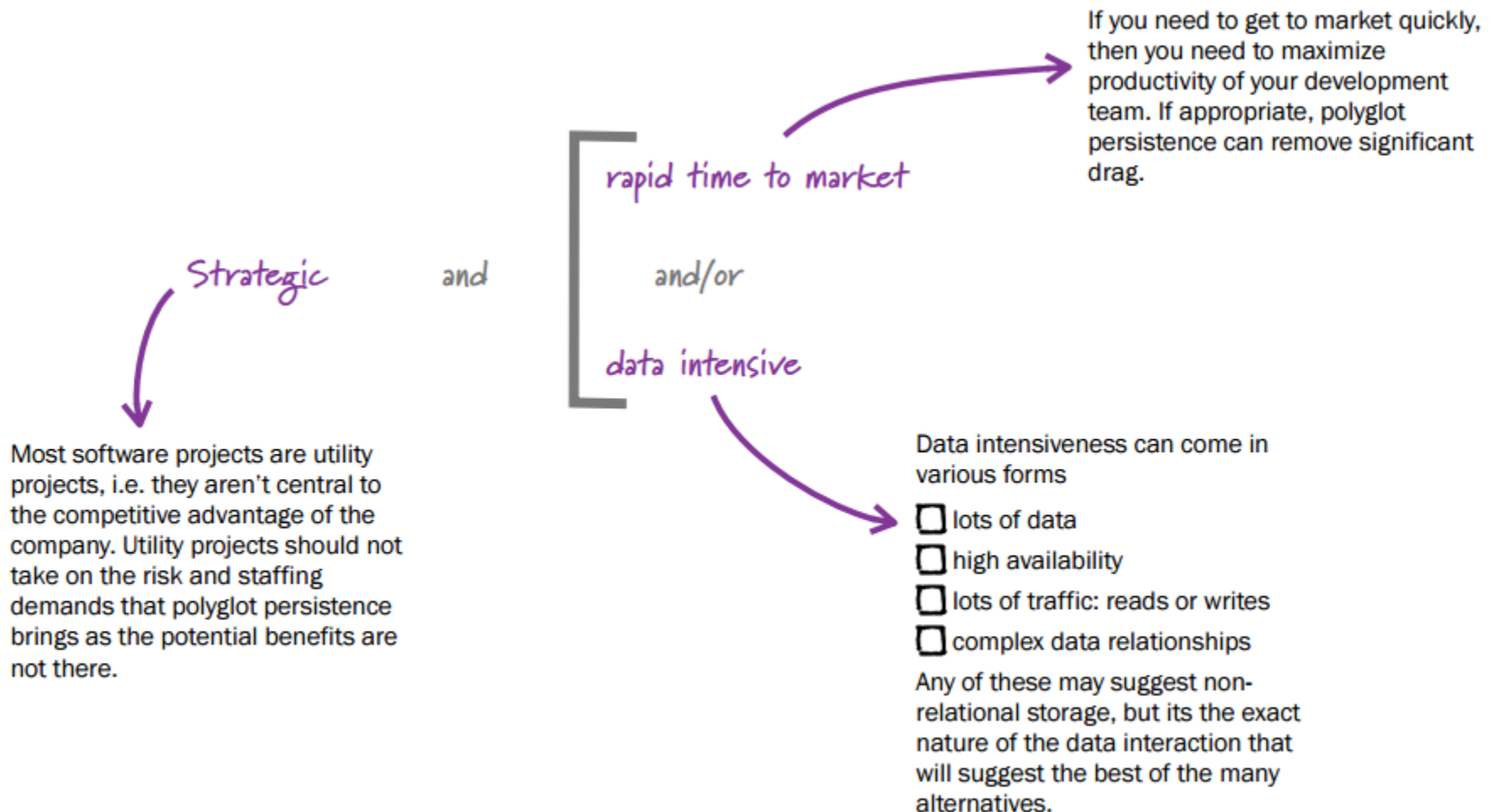
Polyglot Persistence

- Using multiple data storage technologies, chosen based upon the way data is being used by individual applications. Why store binary images in relational database, when there are better storage systems?

What might Polyglot Persistence look like?



Candidates for polyglot persistence?

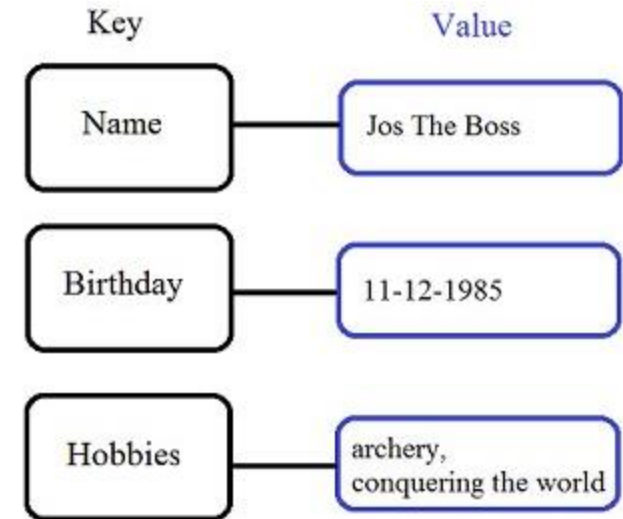


Four NoSQL db Types/Models

- Key-Value store
- Document Data Model
- Column-Oriented database
- Graph Database
- (Key-Value + Document Data Model) = Aggregate Oriented

Key-Value Stores

- Key-value stores are the least complex of the NoSQL databases.
- Simplicity makes it the most scalable of the NoSQL database types, capable of storing huge amounts of data.



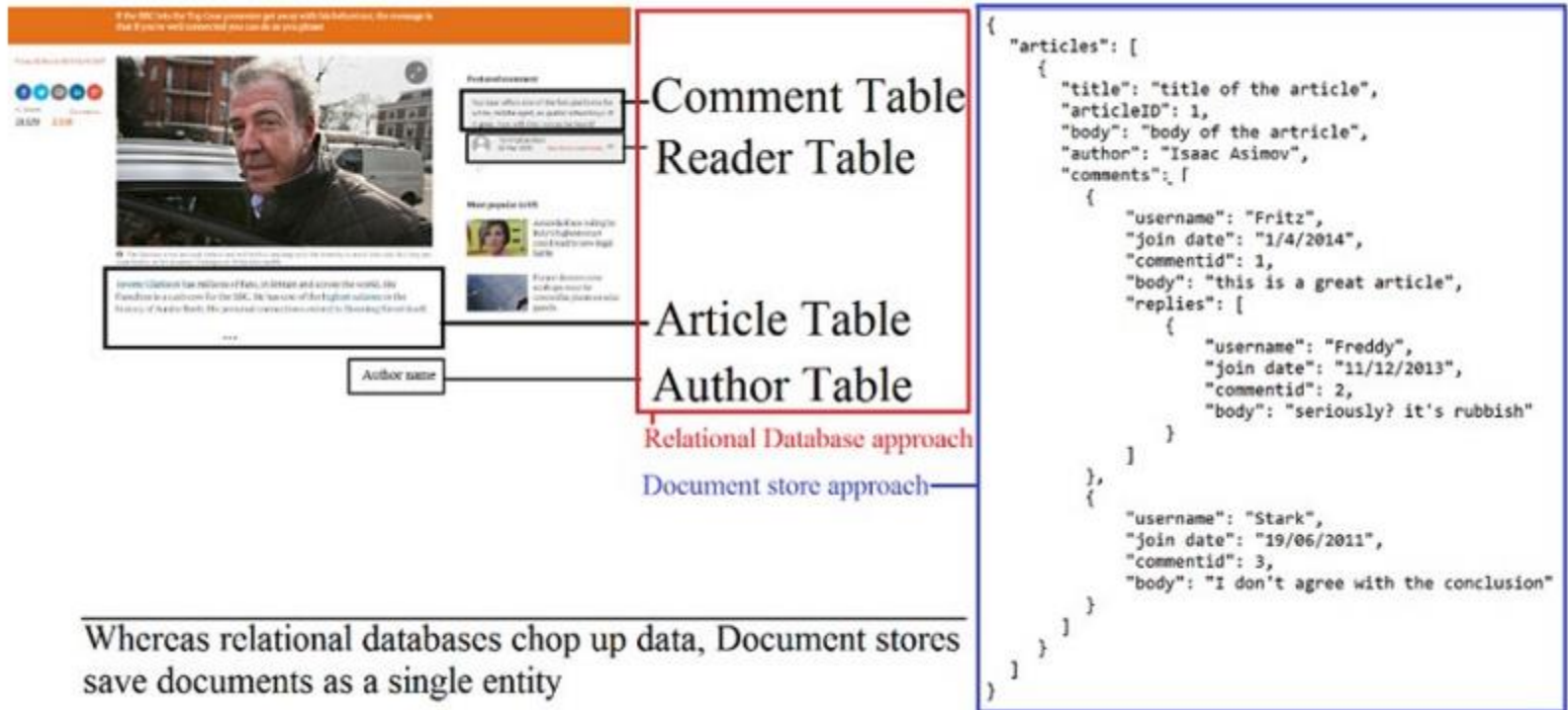
Redis, Voldemort,
Riak, and Amazon's
Dynamo.

Document Stores

- ❑ Document stores are one step up in complexity from key-value stores
- ❑ Assume a certain document structure that can be specified with a schema.
- ❑ Document stores appear the most natural among the NoSQL database types because they're designed to store everyday documents as is, and they allow for complex querying and calculations on this often already aggregated form of data.

Document Stores

MongoDB and CouchDB



Column Oriented

- Traditional relational databases are row-oriented, with each row having a row-id and each field within the row stored together in a table.

Apache HBase, Facebook's Cassandra, Hypertable, and the grandfather of wide-column stores, Google BigTable.

Name	ROWID
Jos The Boss	1
Fritz Schneider	2
Freddy Stark	3
Delphine Thewiseone	4

Birthday	ROWID
11-12-1985	1
27-1-1978	2
16-9-1986	4

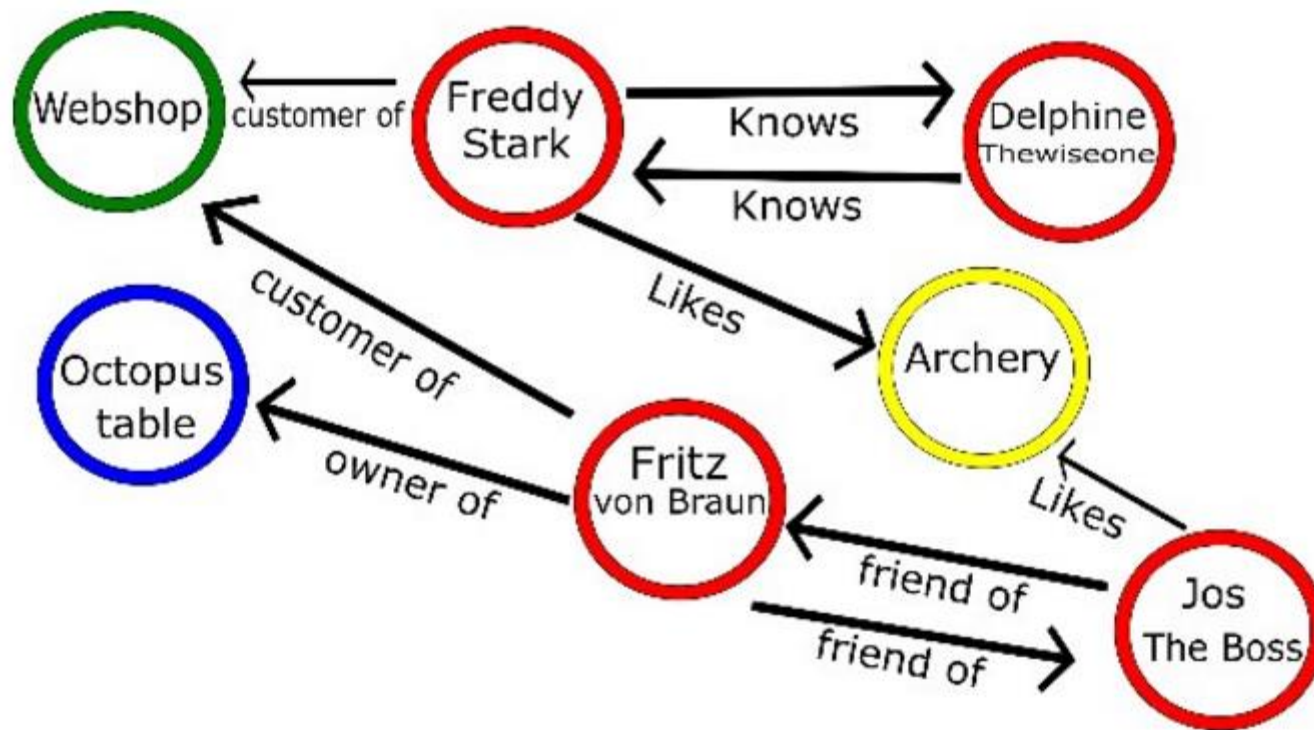
Hobbies	ROWID
archery	1, 3
conquering the world	1
building things	2
surfing	2
swordplay	3
lollygagging	3

A column-oriented database stores each column separately

Graph Databases

- The last big NoSQL database type is the most complex one, geared toward storing relations between entities in an efficient manner. When the data is highly interconnected, such as for social networks, scientific paper citations, or capital asset clusters, graph databases are the answer. Graph or network data has two main components:
 - *Node* : The entities themselves. In a social network this could be people.
 - *Edge*: The relationship between two entities. This relationship is represented by a line and has its own properties. An edge can have a direction, for example, if the arrow indicates who is whose boss.

Graph Databases



Graph databases like Neo4j also claim to uphold ACID, whereas document stores and key-value stores adhere to BASE.

<http://db-engines.com/en/ranking>

Rank			DBMS	Database Model	Score		
Jul 2019	Jun 2019	Jul 2018			Jul 2019	Jun 2019	Jul 2018
1.	1.	1.	Oracle +	Relational, Multi-model i	1321.26	+22.04	+43.47
2.	2.	2.	MySQL +	Relational, Multi-model i	1229.52	+5.89	+33.45
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model i	1090.83	+3.07	+37.42
4.	4.	4.	PostgreSQL +	Relational, Multi-model i	483.28	+6.65	+77.47
5.	5.	5.	MongoDB +	Document	409.93	+6.03	+59.60
6.	6.	6.	IBM Db2 +	Relational, Multi-model i	174.14	+1.94	-12.06
7.	7.	↑ 8.	Elasticsearch +	Search engine, Multi-model i	148.81	-0.01	+12.59
8.	8.	↓ 7.	Redis +	Key-value, Multi-model i	144.26	-1.86	+4.35
9.	9.	9.	Microsoft Access	Relational	137.31	-3.70	+4.73
10.	10.	10.	Cassandra +	Wide column	127.00	+1.82	+5.95

Recap



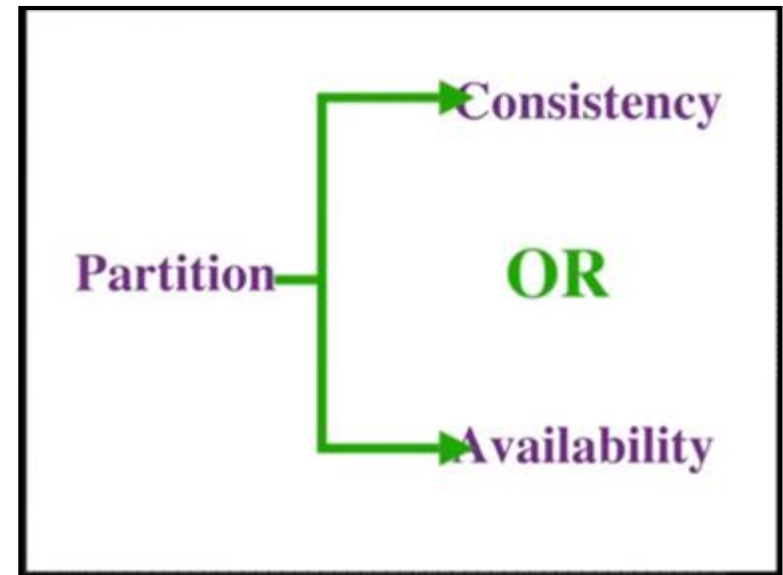
DB Name	Super Type	Subtype
mongoDB	Aggregate	Document
RavenDB	Aggregate	Document
CouchDB	Aggregate	Document
Cassandra	Aggregate	Column Family
Apache HBASE	Aggregate	Column Family
Project Voldermort	Aggregate	Key Value
Riak	Aggregate	Key Value
Redis	Aggregate	Key Value
Neo4i	Graph	

CAP theorem(wikipedia.org)

- In theoretical computer science, the CAP theorem, also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
 - Consistency (all nodes see the same data at the same time)
 - Availability (a guarantee that every request receives a response about whether it succeeded or failed)
 - Partition tolerance (the system continues to operate despite arbitrary partitioning due to network failures)

CAP theorem

- CAP
 - Consistency
 - Availability
 - PartitionTolerance
- Pick any two



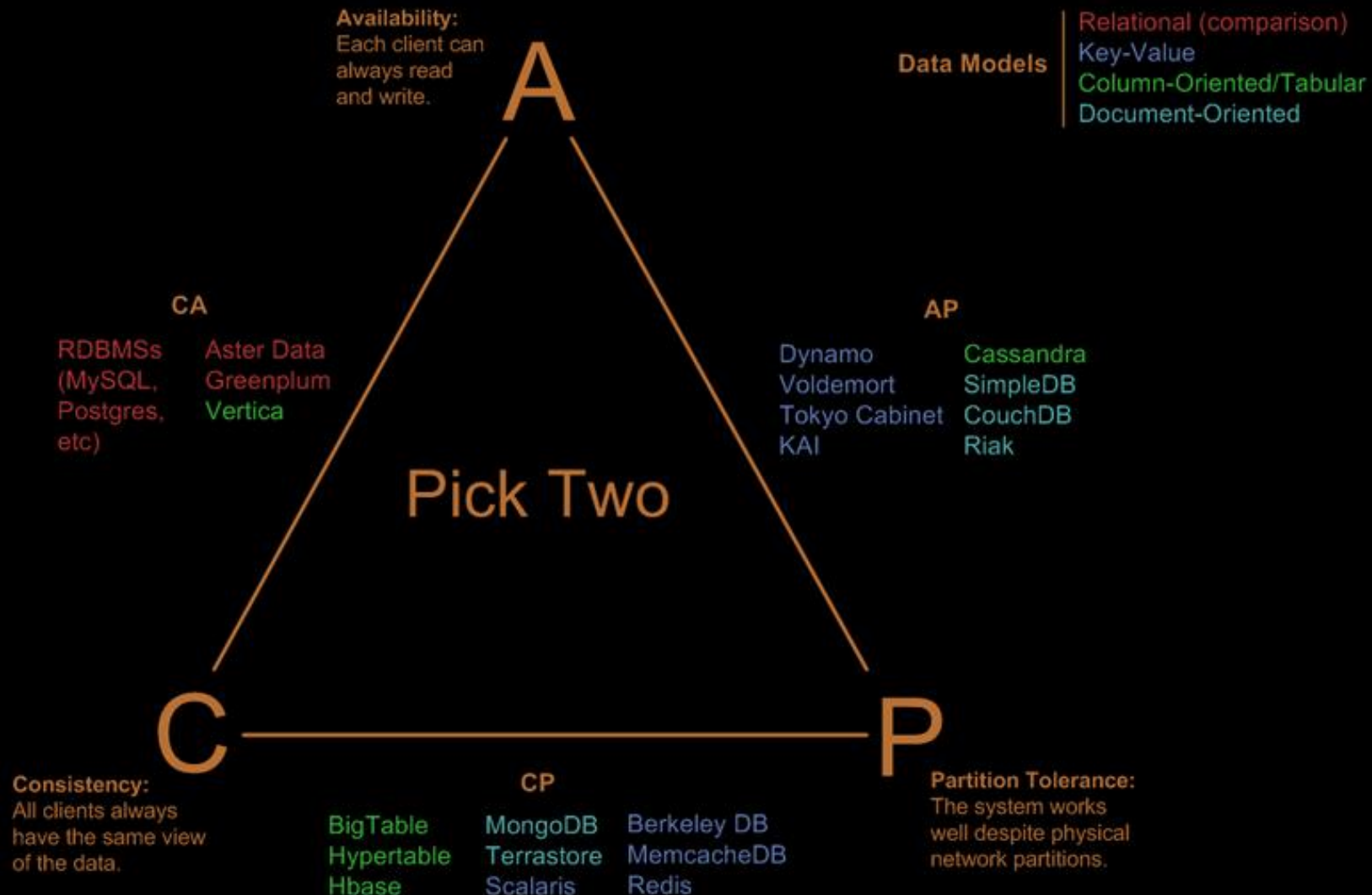
CAP Theorem

- It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
 - Consistency
 - all nodes see the same data at the same time
 - Availability
 - a guarantee that every request receives a response about whether it was successful or failed
 - Partition tolerance
 - the system continues to operate despite arbitrary message loss or failure of part of the system
- A distributed system can satisfy any two of these guarantees at the same time, but not all three.

CAP Theorem

- CAP can be expressed as "If the network is broken, your database won't work"
 - "won't work" = down OR inconsistent
- In RDBMS we do not have P
 - Consistency and Availability are achieved
- In NoSQL we want to have P
 - Need to select either C or A
 - Drop A -> Accept waiting until data is consistent
 - Drop C -> Accept getting inconsistent data sometimes

Visual Guide to NoSQL Systems



ACID vs BASE

- ❑ Scalability and better performance of NoSQL is achieved by sacrificing ACID compatibility.
 - Atomic, Consistent, Isolated, Durable
- ❑ NoSQL is having BASE compatibility instead.
 - Basically Available,
 - Soft state,
 - Eventual consistency

ACID

- Atomicity
 - All of the operations in the transaction will complete, or none will.
- Consistency
 - Transactions never observe or result in inconsistent data.
- Isolation
 - The transaction will behave as if it is the only operation being performed upon the database (i.e. uncommitted transactions are isolated)
- Durability
 - Upon completion of the transaction, the operation will not be reversed (i.e. committed transactions are permanent)

BASE - Basically Available

- Use replication and sharding to reduce the likelihood of data unavailability and use sharding, or partitioning the data among many different storage servers, to make any remaining failures partial.
- The result is a system that is always available, even if subsets of the data become unavailable for short periods of time.

BASE and Availability

- The availability of BASE is achieved through supporting partial failures without total system failure.
 - E.g. If users are partitioned across five database servers, BASE design encourages crafting operations in such a way that a user database failure impacts only the 20 percent of the users on that particular host.
- This leads to higher perceived availability of the system. Even though a single node is failing, the interface is still operational.

BASE - Eventually Consistent

- Although applications must deal with instantaneous consistency, NoSQL systems ensure that at some future point in time the data assumes a consistent state.
- In contrast to ACID systems that enforce consistency at transaction commit, NoSQL guarantees consistency only at some undefined future time.

BASE and Consistency

- As DB nodes are added while scaling up, need for synchronization arises
 - If absolute consistency is required, nodes need to communicate when read/write operations are performed on a node
 - Consistency over availability -> bottleneck
- As a trade-off, "eventual consistency" is used
- Consistency is maintained later
 - Numerous approaches for keeping up "distributed consistency" are available
 - MongoDB - auto-sharding+replication cluster with a master server

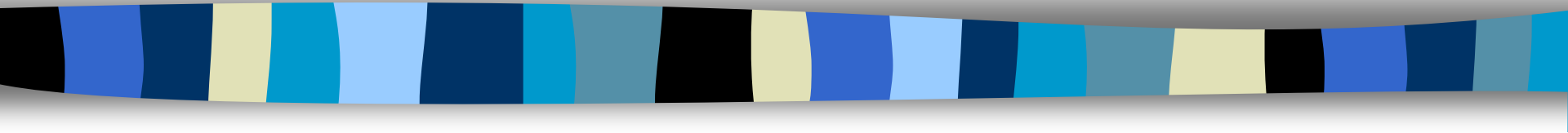
BASE -- Soft State

- While ACID systems assume that data consistency is a hard requirement, NoSQL systems allow data to be inconsistent and relegate designing around such inconsistencies to application developers.
- In other words, soft state indicates that the state of the system may change over time, even without input.
 - This is because of the eventual consistency model

Some NoSQL Challenges

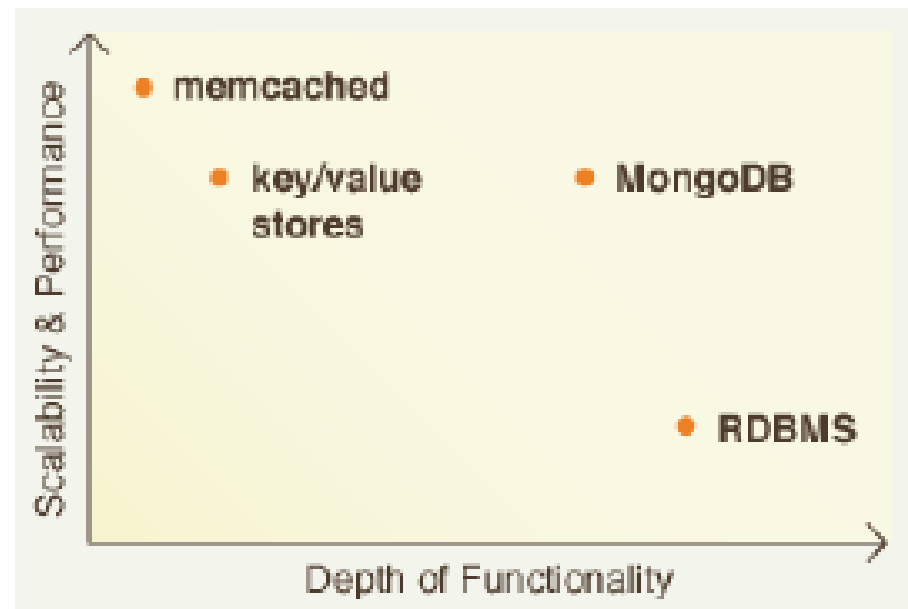
- Lack of maturity
 - Compared to RDBMS
- Lack of commercial support for enterprise users
 - Slowly changing to cloud support
- Lack of support for data analysis
- Maintenance efforts and skills are required --
 - experts are hard to find

Starting with MongoDB



MongoDB

- humongous – huge, monstrous (data)
- Developed by 10gen



Features

- **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 & **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 & fast
 - Eventually-consistent reads can be distributed over replicated servers
- **Rich query language**

Features

- MongoDB is a collection-oriented, schema-free document database.
 - *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.*
 - *Similar to tables without schema.*
- *Schema-free means - database doesn't need to know anything about the structure of the documents that you store in a collection.*
- *Document - 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.*
 - *This data format is "BSON" for "Binary Serialized dOcument Notation"*

BSON

- ❑ BSON is a binary-encoded serialization of JSON-like documents. BSON is designed to be lightweight, traversable, and efficient. BSON, like JSON, supports the embedding of objects and arrays within other objects and arrays
 - Fast scan-ability.
 - Easy manipulation
 - Additional data types.
- ❑ In addition to the basic JSON(string, integer, boolean, double, null, array, and object) , includes date, object id, binary data, regular expression, and code.

Mongo Structure

- A database holds a set of collections
- A collection holds a set of documents

RDBMS		MongoDB
<u>Database</u>	➡	<u>Database</u>
<u>Table, View</u>	➡	<u>Collection</u>
<u>Row</u>	➡	<u>Document (JSON, BSON)</u>
<u>Column</u>	➡	<u>Field</u>
<u>Index</u>	➡	<u>Index</u>
<u>Join</u>	➡	<u>Embedded Document</u>
<u>Foreign Key</u>	➡	<u>Reference</u>
<u>Partition</u>	➡	<u>Shard</u>

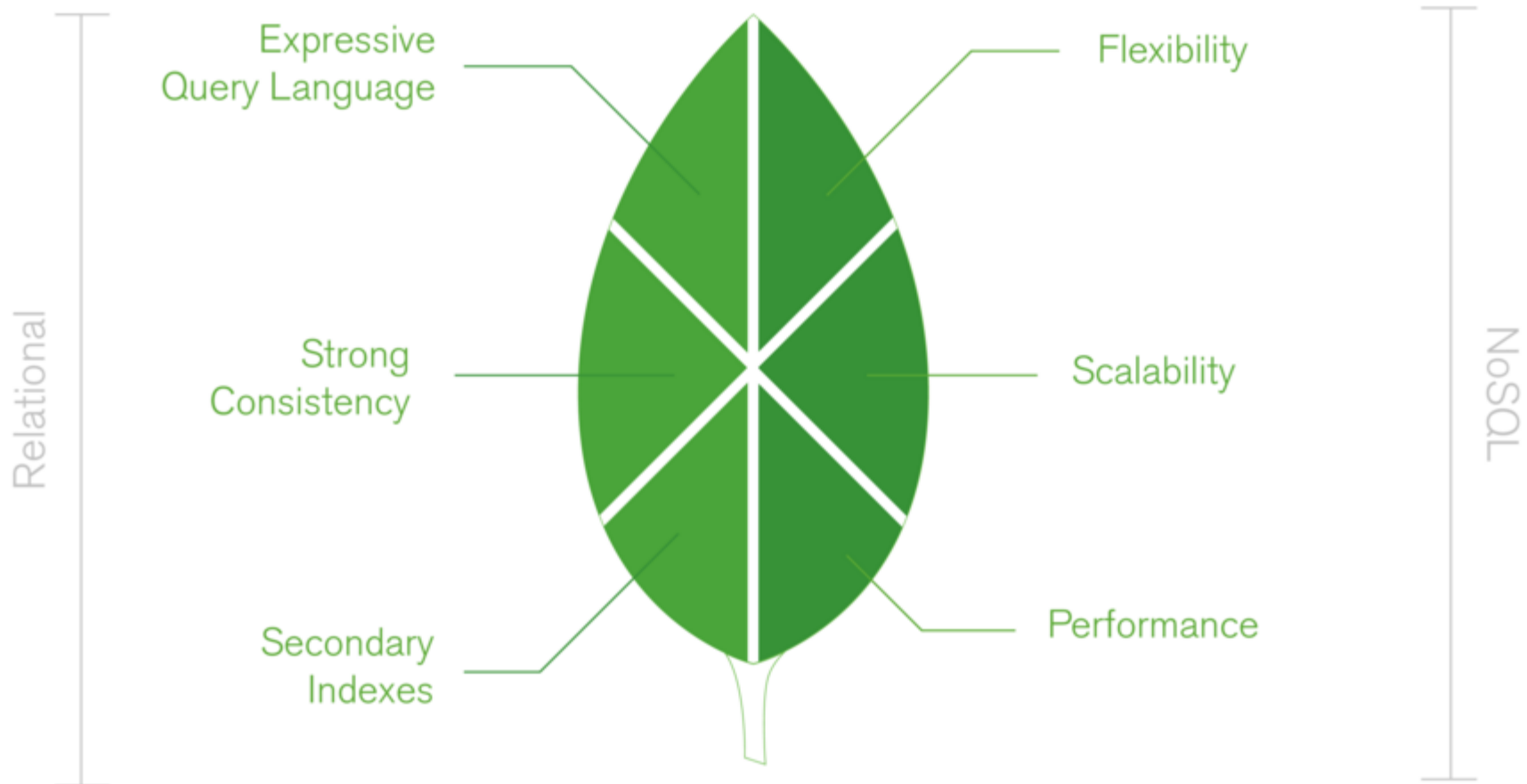
Mongo Architecture



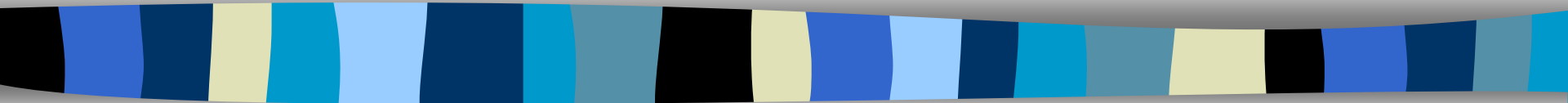
MongoDB's design philosophy

- MongoDB's design philosophy is focused on combining the critical capabilities of relational databases with the innovations of NoSQL technologies.
- "Our vision is to leverage the work that Oracle and others have done over the last 40 years to make relational databases what they are today. Rather than discard decades of proven database maturity, MongoDB is picking up where they left off by combining key relational database capabilities with the work that Internet pioneers have done to address the requirements of modern applications."

MongoDB Nexus Architecture



Installation & Configuration



Download MongoDB Enterprise for Windows.

- Version of windows
 - wmic os get caption
 - wmic os get osarchitecture
- Download supporting version
 - <https://www.mongodb.org/downloads#production>

Install MongoDB Enterprise for Windows

- Attended Vs Unattended Installation
- Unattended Installation
 - `msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.2.3-signed.msi ^
INSTALLLOCATION="C:\mongodb" ^
ADDLOCAL="all"`



Component Set for ADDLOCAL

Component Set	Binaries
Server	mongod.exe
Router	mongos.exe
Client	mongo.exe
MonitoringTools	mongostat.exe, mongotop.exe
ImportExportTools	mongodump.exe, mongorestore.exe, mongoexport.exe, mongoimport.exe
MiscellaneousTools	bsondump.exe, mongofiles.exe, mongooplog.exe, mongoperf.exe

Run MongoDB Enterprise

- To Start Server (on cmd prompt)
 - set up the MongoDB environment
 - md \data\db (data directory)
 - Start server
 - C:\mongodb\bin\mongod.exe --dbpath
d:\test\mongodb\data
 - Permanently set dbpath in configuration file

Run MongoDB Enterprise

- Configure a Windows Service for MongoDB Enterprise
 - **Open an Administrator command prompt.**
 - **Create directories.** 
 - **Create a configuration file.** 
 - create a file at C:\mongodb\mongod.cfg that specifies both systemLog.path and storage path

Starting with MongoDB

- Install
- Set up the MongoDB environment
 - Run server
 - `mongod --dbpath /dbpath/`
 - Run Client
 - Mongo
 - On mongo
 - use admin (where admin is name of database)

Configuration File (YAML format)

□ --configFile

```
systemLog:
  destination: file
  path: "/var/log/mongodb/mongod.log"
  logAppend: true
storage:
  journal:
    enabled: true
processManagement:
  fork: true
net:
  bindIp: 127.0.0.1
  port: 27017
setParameter:
  enableLocalhostAuthBypass: false
```

Insert

The following operation inserts a new document into the users collection. The new document has four fields name, age, and status, and an `_id` field. MongoDB always adds the `_id` field to the new document if that field does not exist.

```
db.users.insert (  ← collection
{
  name: "sue",      ← field: value
  age: 26,           ← field: value
  status: "A"        ← field: value
}                  } document
)
```

The following diagram shows the same query in SQL:

```
INSERT INTO users      ← table
      ( name, age, status ) ← columns
VALUES ( "sue", 26, "A" ) ← values/row
```

Update

`db.collection.update()` method modifies existing documents in a collection. Can accept query criteria to determine which documents to update as well as an options document that affects its behavior, such as the `multi` option to update multiple documents.

`save()` method replaces the existing document with new doc.

```
db.users.update(  
  { age: { $gt: 18 } },  
  { $set: { status: "A" } },  
  { multi: true }  
)
```

← collection
← update criteria
← update action
← update option

The following diagram shows the same query in SQL:

```
UPDATE users  
SET status = 'A'  
WHERE age > 18
```

← table
← update action
← update criteria

Remove

`db.collection.remove()` method deletes documents from a collection. The `db.collection.remove()` method accepts a query criteria to determine which documents to remove

```
db.users.remove(  
  { status: "D" }  
)
```

← collection
← remove criteria

The following diagram shows the same query in SQL:

```
DELETE FROM users  
WHERE status = 'D'
```

← table
← delete criteria

Find

Read operations, or queries, retrieve data stored in the database. In MongoDB, queries select documents from a single collection.

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

The next diagram shows the same query in SQL:

```
SELECT _id, name, address  
FROM users  
WHERE age > 18  
LIMIT 5
```

← projection
← table
← select criteria
← cursor modifier

Query Behavior

- **MongoDB queries exhibit the following behavior:**
 - All queries in MongoDB address a single collection.
 - You can modify the query to impose limits, skips, and sort orders.
 - The order of documents returned by a query is not defined unless you specify a sort().
 - Operations that modify existing documents (i.e. updates) use the same query syntax as queries to select documents to update.
 - In aggregation pipeline, the \$match pipeline stage provides access to MongoDB queries.
 - MongoDB provides a `db.collection.findOne()` method as a special case of `find()` that returns a single document.

RECAP

□ Create

- `db.collection.insert(<document>)`
- `db.collection.save(<document>)`
- `db.collection.update(<query>, <update>, { upsert: true })`

□ Read

- `db.collection.find(<query>, <projection>)`
- `db.collection.findOne(<query>, <projection>)`

□ Update

- `db.collection.update(<query>, <update>, <options>)`

□ Delete

- `db.collection.remove(<query>, <justOne>)`

Common Commands



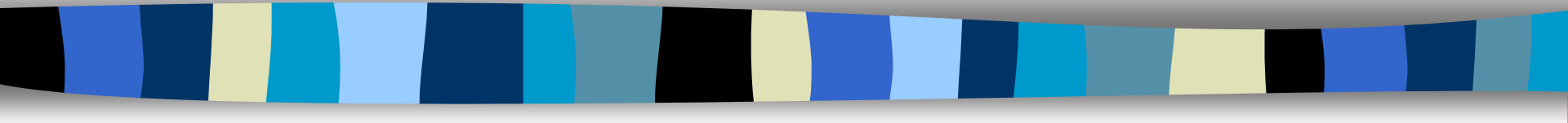
Database Related Commands

- ❑ use - create database.
 - The command will create a new database, if it doesn't exist otherwise it will return the existing database.
- ❑ db – to check your currently selected database
- ❑ show dbs – to show your databases list
- ❑ db.dropDatabase() - to drop existing database.

Collection Related Commands

- ❑ `db.createCollection(name, options)`
 - used to create collection with specified name and options (Optional -Specify options about memory size and indexing)
 - Options include capped, autoIndexID, size(for capped=true), max(for capped=true)
- ❑ `show collections`
- ❑ `db.collection.drop()`

Data Modeling



Document Structure

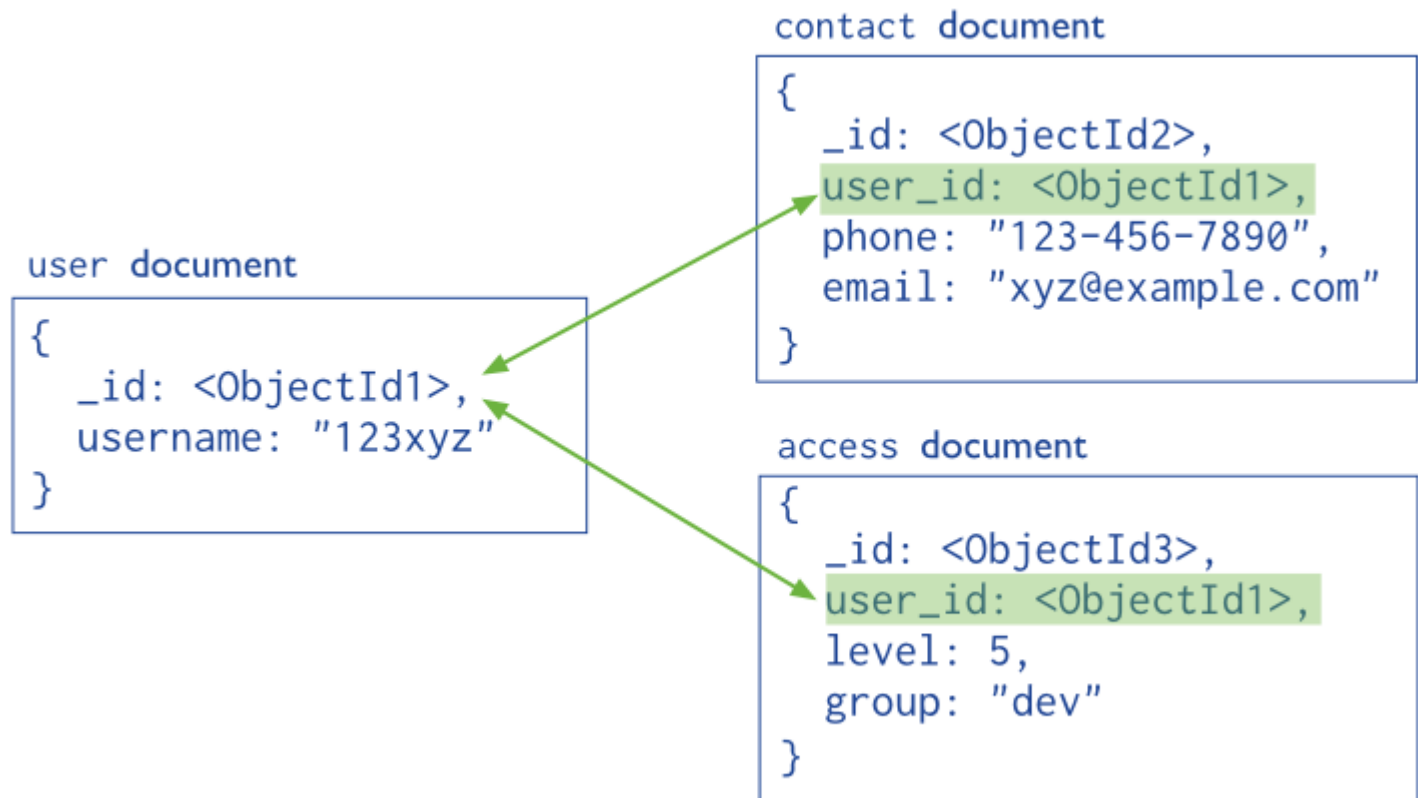
- The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data.
- Two options to represent relationships
 - *references*
 - *embedded documents*

□

References

- References store the relationships between data by including links or *references* from one document to another.
- Applications can resolve these references to access the related data. Broadly, these are *normalized* data models.

References



Embedded Data

- ❑ Embedded documents capture relationships between data by storing related data in a single document structure.
- ❑ MongoDB documents make it possible to embed document structures in a field or array within a document.
- ❑ These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

Use References

- + more flexibility than embedding
- client-side applications must issue follow-up queries to resolve references.
- normalized data models can require more round trips to the server.
- When to use
 - when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
 - to represent more complex many-to-many relationships.
 - to model large hierarchical data sets.

Manual Vs Auto References

- **\$ref:**
 - This field specifies the collection of the referenced document
- **\$id:**
 - This field specifies the `_id` field of the referenced document
- **\$db:**
 - This is an optional field and contains name of the database in which the referenced document lies

```
{
  "_id": ObjectId("53402597d8524
    26020000002"),
  "address":
  {
    "$ref": "address_home",
    "$id":
      ObjectId("534009e4d852427
        820000002"),
    "$db": "adiffdb"
  },
  "contact": "987654321" }
```

Embedded Data

```
{  
  _id: <ObjectId>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```

Embedded sub-document

Embedded sub-document

Use embedded data models

- + better performance for read operations
- + request and retrieve related data in simple db op
- + update related data in single atomic write op
- documents grow after creation
- can impact write performance & fragmentation
- When to use
 - you have “contains” relationships between entities
 - you have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents.

One-One

- Embedded Documents
- E.g. Emp -> address

```
{ _id: "joe",  
  name: "Joe  
  Bookreader"  
}  
{ patron_id: "joe",  
  street: "123 Fake  
  Street",  
  city: "Faketon", state:  
  "MA",  
  zip: "12345" }
```

One-to-Many - Embedded

- With the embedded data model, your application can retrieve the complete patron information with one query.

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


One-to-Many - Referenced

```
{
  _id: "oreilly",
  name: "O'Reilly Media", ..
}
{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",...,
  publisher_id: "oreilly"
}
```

Tree Structures



```
db.categories.insert( { _id:
    "MongoDB", parent:
    "Databases" } )
db.categories.insert( { _id:
    "dbm", parent: "Databases" }
)
db.categories.insert( { _id:
    "Databases", parent:
    "Programming" } ..
```

To Retrieve

```
db.categories.findOne( { _id:
    "MongoDB" } ).parent
```

Queries in detail



Basic find

- ❑ `find()` – returns all records
- ❑ `pretty()` - display the results in a formatted way

RDBMS Where Clause Equivalents

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>: <value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>: {\$lt: <value>}}	db.mycol.find({"likes": {\$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>: {\$lte: <value>}}	db.mycol.find({"likes": {\$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>: {\$gt: <value>}}	db.mycol.find({"likes": {\$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>: {\$gte: <value>}}	db.mycol.find({"likes": {\$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>: {\$ne: <value>}}	db.mycol.find({"likes": {\$ne:50}}).pretty()	where likes != 50

And/Or

□ AND

- `db.mycol.find({key1:value1, key2:value2})`

□ OR

- `db.mycol.find({ $or: [{key1: value1}, {key2:value2}] }).pretty()`

□ Using AND and OR together

- `db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"}, {"title": "MongoDB Overview"}]}).pretty()`
 - where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')

Find - Projections

- Find accepts second optional parameter that is list of fields that you want to retrieve. By default it displays all fields of a document.
 - To limit this you need to set list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the field.
 - `db.mycol.find({}, {"title":1, _id:0})`

Find – Pagination

- `limit()`
 - To limit the number of retrieved records
 - Default is to retrieve all elements
- `skip()`
 - Starting point of retrieved records
 - Default is zero

Find - Sort

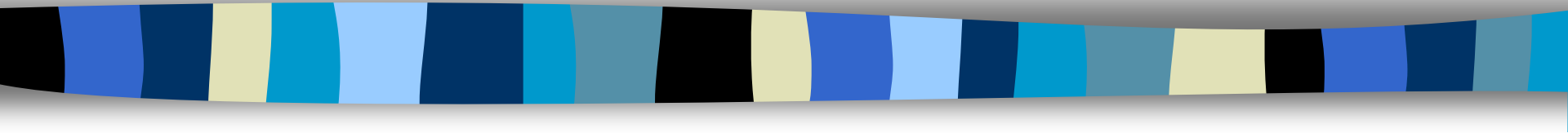
□ **sort()**

- accepts a document containing list of fields along with their sorting order. To specify sorting order 1 (ascending, default) and -1 (descending) are used.
- `db.mycol.find({}, {"title":1, _id:0}).sort({"title":-1})`

Aggregation

- Aggregation operations group values from multiple documents together, and can perform a variety of operations on grouped data to return single result.
 - Similar to sql count(*) with group by
- E.g. `db.c.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}]`)
- `$sum`, `$avg`, `$min`, `$max`, `$first`, `$last`
- `$push` - Inserts the value to an array in the resulting documents
 - `db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}]`)
- `$addToSet` - Inserts the value to an array in the resulting document but does not create duplicates
 - `db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}]`)

Indexing



Indexing

- Indexes are special data structures, that store a small portion of the data set in an easy to traverse form.
- The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

Index Types

- Default `_id` Index
 - The `_id` index is *unique* and prevents clients from inserting two documents with the same value for the `_id` field.
- Single Field Indexes
- Compound Indexes
 - user-defined indexes on multiple fields
- Multikey Indexes
 - content stored in arrays
- Text Indexes
 - supports searching for string content in a collection

ensureIndex

- To create an index
 - `db.people.createIndex({ zipcode: 1}, {background: true})`
- Check documentation to see other properties

Check Query Performance

- Run Query
 - `db.dept.find({'loc':'Pune'}).explain("executionStats")`
 - Observe – docs-examined and returned
- Create index
 - `db.dept.createIndex({loc:1})`
- Run Query

Map Reduce

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results.
- In this operation, MongoDB applies the *map* phase to each input document (that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

MapReduce Command

```
db.collection.mapReduce  
( function()  
  {emit(key,value);}, //map  
  function  
  function(key,values)  
    {return reduceFunction},  
  { //reduce function  
    out: collection,  
    query: document,  
    sort: document,  
    limit: number } )
```

- map is a javascript function that maps a value with a key and emits a key-value pair
- reduce is a javascript function that reduces or groups all the documents having the same key
- out - the location of the map-reduce query result
- query - optional selection criteria for selecting documents
- sort - optional sort criteria
- limit - optional maximum number of documents to be returned

MapReduce

```
db.emp.mapReduce(  
  function() { emit(this.deptno, this.salary); },  
  function(key, values) {return Array.sum(values)}, {  
    out:"post_total"  })  
db.post_total.find()
```

Collection
↓

```
db.orders.mapReduce(  
  map    → function() { emit( this.cust_id, this.amount ); },  
  reduce → function(key, values) { return Array.sum( values ) },  
  {  
    query → { status: "A" },  
    output → "order_totals"  
  }  
)
```



A **F**ast **AND** **S**teady Approach

Security



Security

- ❑ Maintaining a secure MongoDB deployment requires administrators to implement controls to ensure that users and applications have access to only the data that they require. MongoDB provides features that allow administrators to implement these controls and restrictions for any MongoDB deployment.

Enable Access Control and Enforce Authentication

- ❑ Enable Security for Mongo
 - Mongod –auth
- ❑ Create Admin User
 - use admin
 - var user = { "user" : "admin", "pwd" : "manager",
roles : [{ "role" : "userAdminAnyDatabase", "db" :
"admin" }] }
 - db.createUser(user);
- ❑ Connect with the Administrator user
 - mongo -u username -p password --
authenticationDatabase admin

Enable Access Control and Enforce Authentication

- Create Application User (read/write)

```
db.createUser({ "user" : "u1", "pwd" : "p1",  
  roles : [ {"role" : "readWrite", "db" : "tmp"  
    }  ]});
```
- `mongo -u u1 -p p1 --
authenticationDatabase tmp
– Insert/update`

Enable Access Control and Enforce Authentication

- Create a reporting user (Read Only)
db.createUser(user = {"user" : "r1", "pwd" :
"p1", roles : [{ "role" : "read",
"db" : "tmp"
}]});
- Connect using reporting user
 - Run queries
 - Run insert/update

A decorative vertical bar on the left side of the slide, composed of numerous horizontal stripes in various shades of blue, teal, yellow, and black.

MongoDB

from code

.Net

□ Visual Studio

- New Console Project
- Tools -> NuGet Package Manager
 - Package Manager Console
 - Install-Package MongoDB.Driver

Establish Connection

using MongoDB.Driver;

```
var client = new  
MongoClient("mongodb://localhost:27017"  
);  
var db =  
client.GetDatabase("csharpcode");  
Console.WriteLine("after db" + db);
```

Insert a document

- IMongoCollection<TDocument> Interface
 - InsertOne
 - InsertOneAsync(TDocument, InsertOneOptions, CancellationToken)
 - ReplaceOne
 - UpdateMany
 - UpdateOneAsync
 - DeleteOne
 - DeleteMany
 - FindAsync<TProjection>
 - FindSync<TProjection>

Linq

□ using MongoDB.Driver.Linq;

```
var query =  
    from e in collection.AsQueryable<Emp>()  
    where e.ename == "Saloni"  
    select e;
```

```
foreach (var employee in query)  
{  
    Console.WriteLine(employee.ToString());  
}
```

Replication & Sharding



Replication

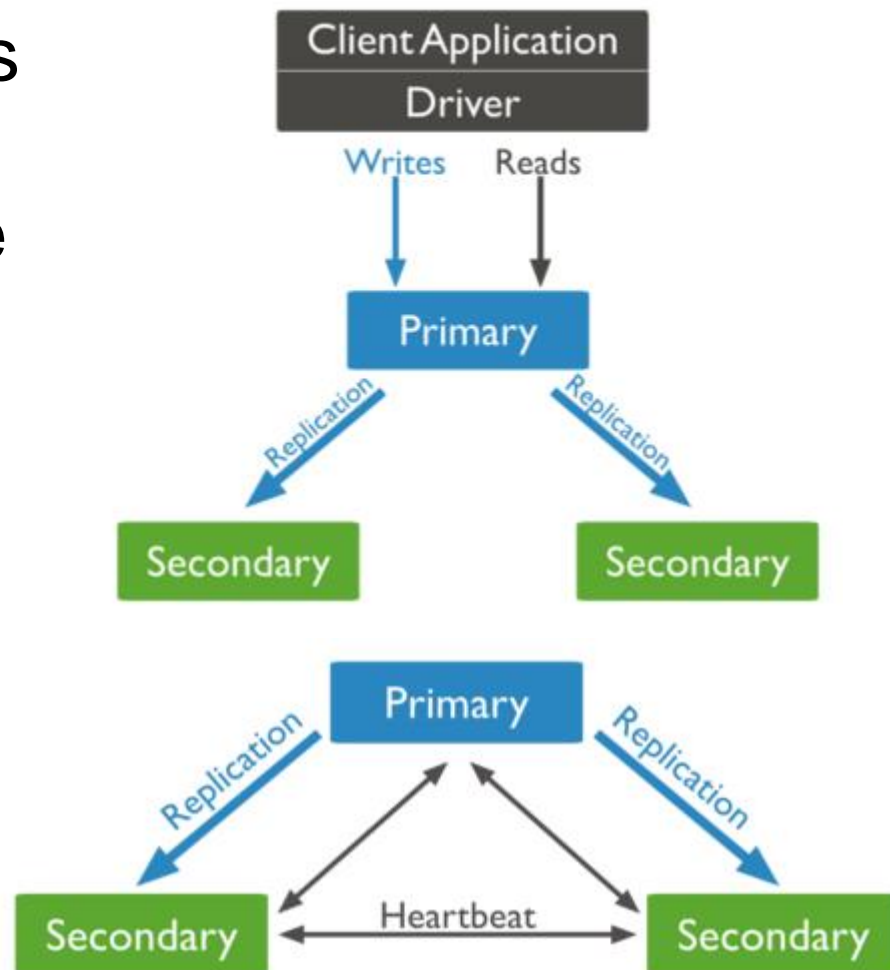
- The process of synchronizing data across multiple servers.
- Redundancy and Data Availability
- Provides increased read capacity as clients can send read operations to different servers.
- Option to maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Replication in MongoDB

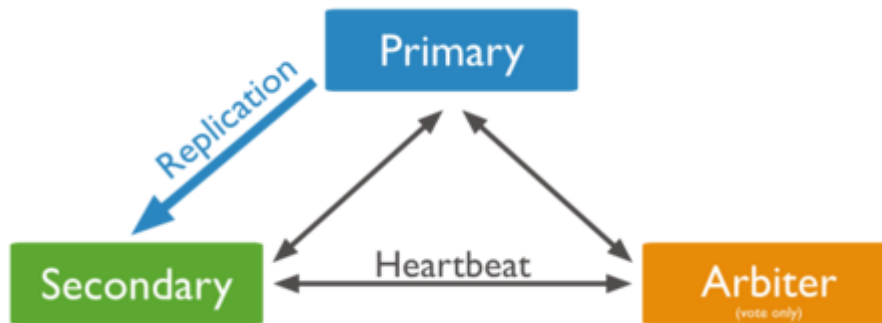
- A replica set is a group of mongod instances that maintain the same data set. A replica set contains several data bearing nodes and optionally one arbiter node. Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.

Replication in MongoDB

- Primary node receives all write operations.
- A replica set can have only one primary capable of writes
- Primary may change
- Secondaries replicate the primary's oplog and apply the ops to their data sets



Arbiter



- ❑ Arbiter - may add an extra instance to a replica set as an Arbiter
- ❑ Arbiters do not maintain a data set.
- ❑ Purpose is to maintain a quorum in a replica set by responding to heartbeat and election requests by other replica set members.
- ❑ Cheaper resource cost
- ❑ If your replica set has an even number of members, add an arbiter to obtain a majority of votes in an election for primary.

Sharding

- ❑ Sharding is a method for storing data across multiple machines.
- ❑ Purpose of Sharding
 - Database systems with large data sets and high throughput applications can challenge the capacity of a single server.
 - High query rates can exhaust CPU capacity of the server.
 - Larger data sets exceed the storage capacity of a single machine.

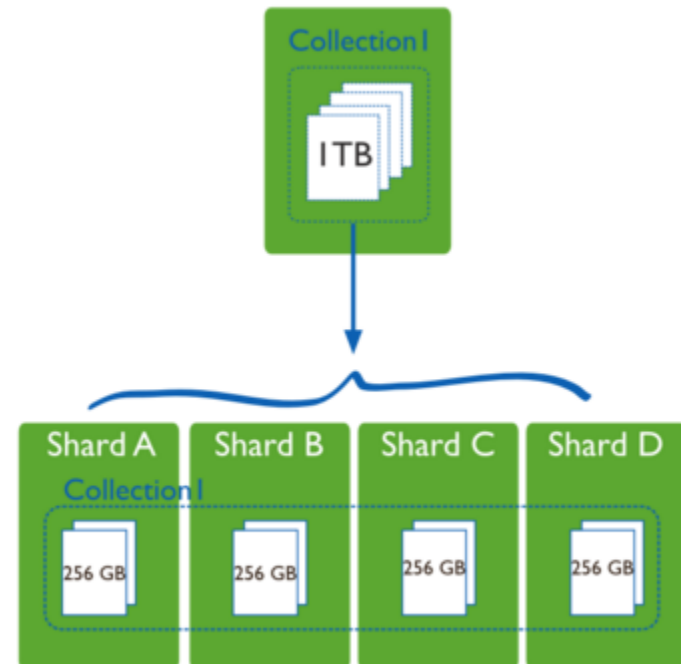
Vertical Scaling & Sharding.

□ 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.
- Cloud-based providers may only allow users to provision smaller instances. As a result there is a practical maximum capability for vertical scaling.

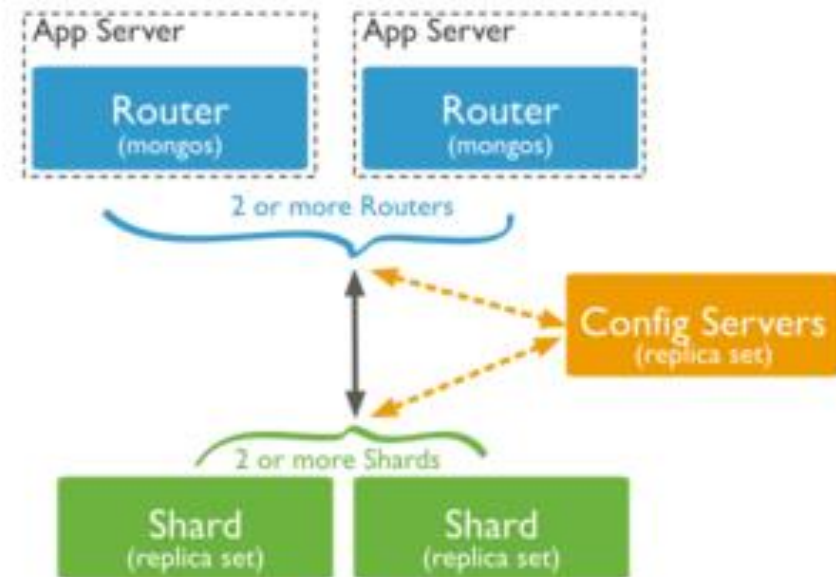
Sharding

- ❑ Sharding or *horizontal scaling*, by contrast, divides the data set and distributes the data over multiple servers, or **shards**.
- ❑ Each shard is an independent database, and collectively, the shards make up a single logical database.



Sharding in MongoDB

- components:
 - shards,
 - query routers
 - config servers



Components

- Shards
 - Store the data.
 - To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set.
- Query Routers (or mongos instances)
 - Interface with client applications and direct operations to the appropriate shard or shards.
 - may have multiple for load balancing
- 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.

Data Partitioning

- Distributes data/shards at the collection level
- Partitions a collection's data by the shard key
- Shard Key
 - To shard a collection, select a shard key.
 - Either an indexed field or an indexed compound field that exists in every document in collection.
 - Divides the shard key values into chunks and distributes the chunks evenly across the shards.
- Types
 - Range Based Sharding
 - Hash Based Sharding

QUESTION / ANSWERS



Questions

- ❑ what is sharding concept? .difference between sharding and replication?
- ❑ what is GridFS? how it works? how it is different than HDFS in big data?
- ❑ what are the 5 scenarios Mongo dB is recommended?
- ❑ what are the 5 scenarios where Mongo dB is not preferred instead use RDBMS?
- ❑ what are the 5 scenarios that both Mongo dB, RDBMS has to be used in same project and store same data in both? what is the plus we get in this scenario?

THANKING YOU !

