

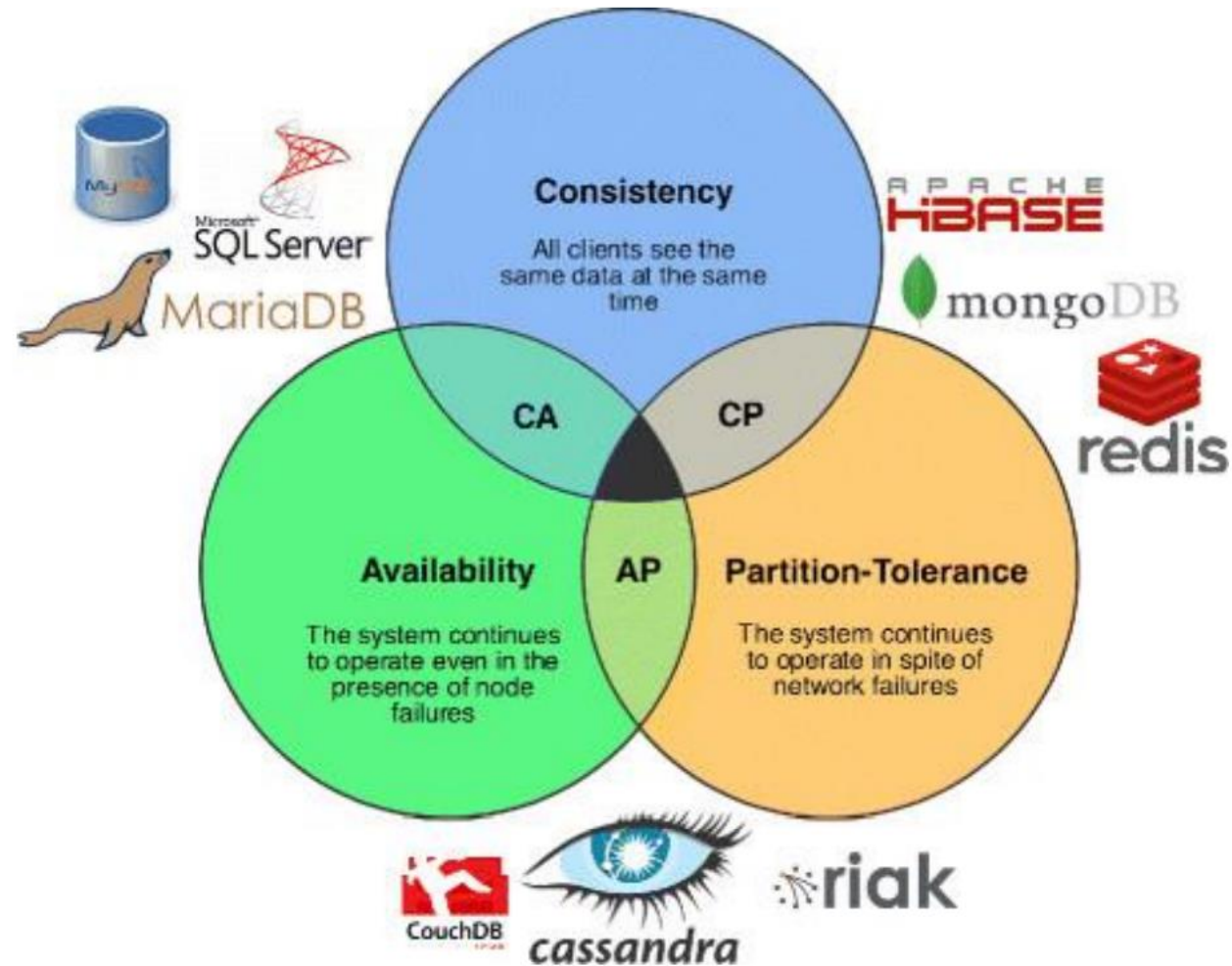


MongoDB - an introduction

Table of Content

Module	Topic
Module 1:	Introduction to NoSQL
Module 2:	MongoDB Basics
Module 3:	MongoDB Queries
Module 4:	Aggregation Framework

CAP Theorem



CAP Theorem

- **Consistency**

A service that is consistent should follow the rule of ordering for updates that spread across all replicas in a cluster - “what you write is what you read”, regardless of location.

- **Availability**

A service should be available. There should be a guarantee that every request receives a response about whether it was successful or failed.

CAP Theorem

- **Partition Tolerance**

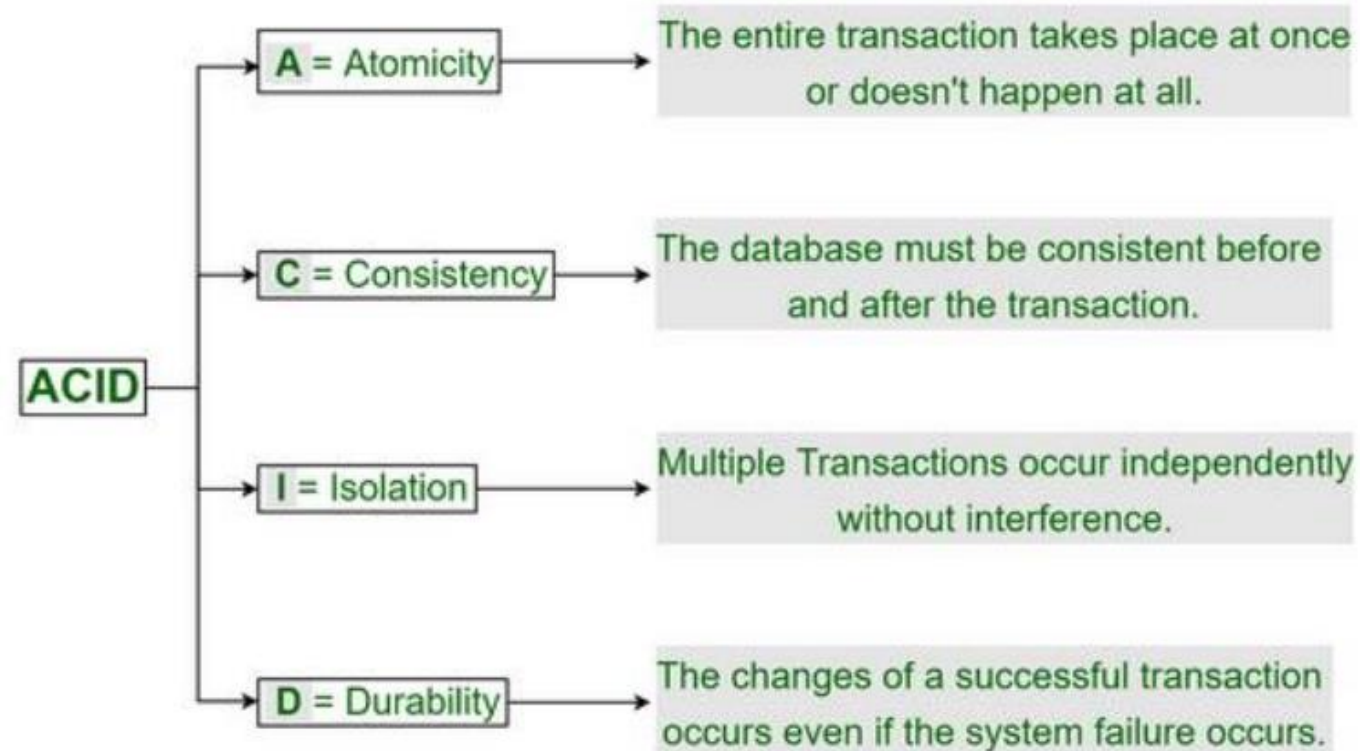
The system continues to operate despite arbitrary message loss or failure of part of the system.

CAP Theorem Statement:

Though its desirable to have Consistency, High-Availability and Partition-tolerance in every system, unfortunately no system can achieve all three at the same time.

ACID properties

To ensure the integrity of data during a transaction, the database system maintains essential properties known as ACID properties.



BASE Model

- **Basic Availability:**

Data will be available even in case of multiple failures. This is possible by spreading the data across many storage systems with a high degree of replication.

- **Soft state:**

The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

- **Eventual consistency:**

The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

ACID vs. BASE


Sr. No.	ACID (used in RDBMS)	BASE (used in NoSQL)
1.	Strong consistency	Weak consistency (Stale data OK)
2.	Isolation	Last write wins
3.	Transaction	Program managed
4.	Robust database	Simple database
5.	Consistency & Availability (CA)	Available & Partition-tolerant (AP) OR Consistency & Partition-tolerant (CP)

RDBMS vs NoSQL

Sr. No.	RDBMS	NoSQL
1.	Handles Limited Data Volumes	Handles Huge Data Volumes
2.	Vertically scaled (Scale-in)	Horizontally scaled (Scale-out)
3.	SQL is used as query language	No declarative query language
4.	Predefined Schema	Schema less
5.	Supports relational data and its relationships are stored in separate tables	Supports unstructured and unpredictable data
6.	Based on ACID model	Based on BASE model
7.	Transaction Management is strong	Transaction Management is weak



When to use NoSQL?

1. You need to handle extremely large data sets.
 2. You need extremely fast in-memory data.
 3. You need Schema less & de-normalized database.
 4. You want to handle database in Object oriented fashion.
- 

Types of NoSQL databases

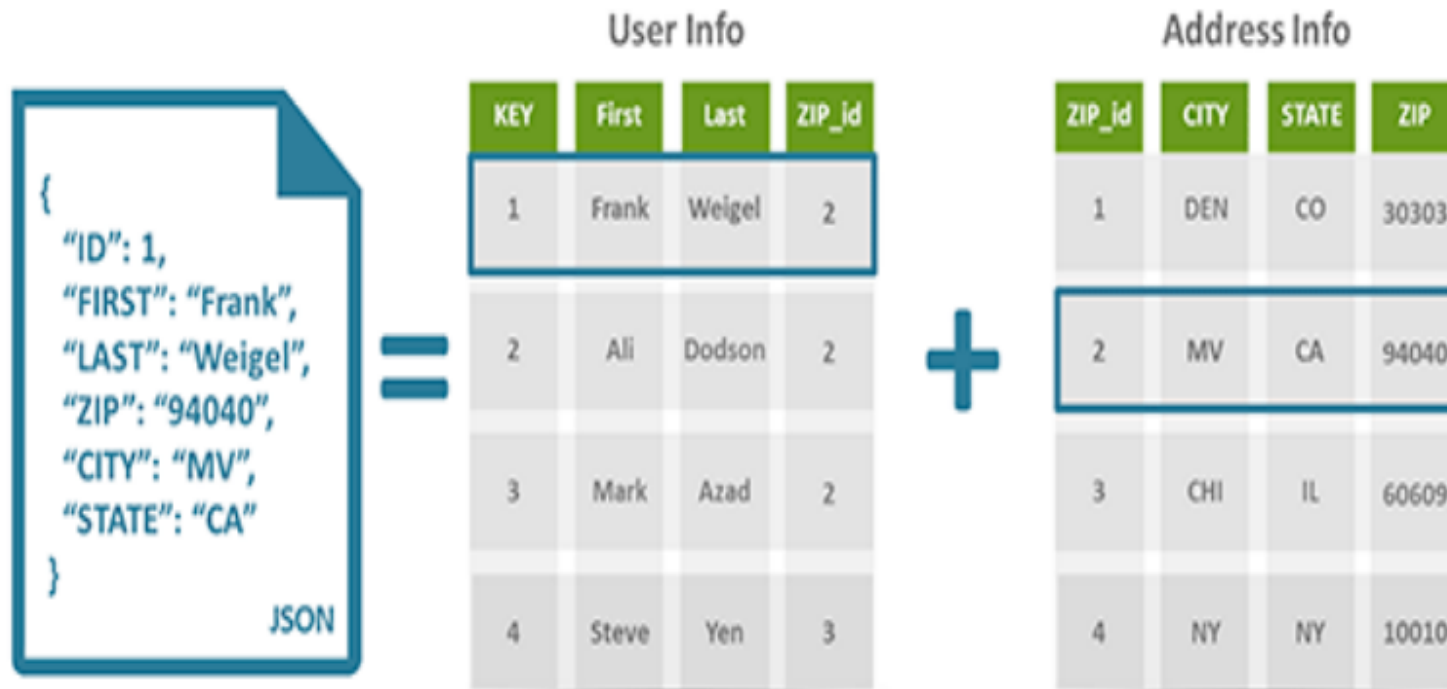
- **Document DBs**
 - **MongoDB**, CouchDB, ...
- **Graph DBs**
 - Neo4j, FlockDB...
- **Column oriented DBs**
 - HBase, Cassandra, BigTable...
- **Key-Value DBs**
 - Memcache, MemcacheDB, Redis, Voldemort, Dynamo...



Document based DBs

The data which is a collection of key value pairs is compressed as a document store quite similar to a key-value store. However, the only difference is that the values stored (referred as “documents”) provide some structure and encoding of the managed data. XML, JSON, BSON (Binary JSON) are some common standard encodings.

Document based DBs continue...



Advantages of MongoDB

- **Simplicity**

Mongo adopts storage in JSON format makes it simple database rather than adding complexities that come with relational databases.

- **Data Replication and Reliability**

MongoDB allow users to replicate data on multiple mirrored servers which ensures data reliability. In case a server crashes, its mirror is still available and database processing remains unaltered.

Advantages of MongoDB

- **NoSQL Queries**

Mongo JSON Based document oriented queries are extremely fast as compared to traditional sql queries.

- **Schema Free Migrations**

In MongoDB, schema is defined by the code. Hence, in case of database migrations, no schema compatibility issue arises.

- **Open Source**

MongoDB is a database server that is open source and customizable according to the requirements of the organization.

MongoDB Terminologies

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded documents
Primary key	Primary key supplied by MongoDB itself

MongoDB datatypes

- Double
- String
- Object
- Array
- Binary Data
- ObjectId
- Boolean
- Date
- Null
- Integer
- Timestamp

Mongo queries

1. Create new database: `>use xordb`
2. Find the current database: `>db`
3. List down all databases: `>show dbs`
4. Delete database: `>use xordb THEN >db.dropDatabase()`
5. Create collection: `>db.createCollection("orders")`

Mongo queries continued...

List all collection from database: `>show collections`

Drop collection: `>db.COLLECTION_NAME.drop()`

Insert document: `>db.COLLECTION_NAME.insert({ name: 'Chairs',
quantity: 35, price: 5000})`

Display all documents within collection:
`>db.COLLECTION_NAME.find().pretty()`

Mongo queries continued...

Find documents based upon filter criteria:

```
>db.Orders.find({name: 'Bag Purchase'}).pretty()
```

```
>db.Orders.find({price: {$lt: 60000}}).pretty()
```

```
>db.Orders.find({$or: [ { price: {$lt: 60000}}, {name: 'Bag Purchase'}  
] }).pretty()
```

Update document:

```
>db.Orders.update({name: 'Car Purchase'}, {$set: {name: 'Car sale'  
}})
```

Mongo queries continued...

Save document:

```
>db.Orders.save({  
    "_id": ObjectId("565bbac2c95843b6ef06c00a"),  
    "name": "Handbag purchase", "price": 2300 })
```

Delete document:

```
>db.Orders.remove({ name: "Handbag purchase" })  
>db.Orders.remove({ name: "Handbag purchase" }, 1)  
>db.Orders.remove()
```

Mongo queries continued...

Projection in document:

```
>db.Orders.find({name: 'Laptop Purchase'}, {price: 1}).pretty()
```

Limiting documents:

```
>db.Orders.find().limit(2).pretty()
```

Skipping documents:

```
>db.Orders.find().limit(1).skip(1).pretty()
```

Sorting documents:

```
>db.Orders.find().sort({price: 1}).pretty()
```

MongoDB Aggregation Framework

Sometimes Mongo developer wants to analyze and crunch it in interesting ways. Here we can use aggregation facility provided by Mongo.

Aggregation groups values from multiple documents together and can perform a variety of operations on the grouped data to return a single result.

Aggregation Framework

The aggregation framework lets you transform and combine documents in a collection. Basically, you build a pipeline that processes a stream of documents through several building blocks: filtering, projecting, grouping, sorting, limiting, and skipping.

Take an example of collection 'Magazine' having following fields:

```
{  
  "name" : "India Today",  
  "price" : 150,  
  "author" : "Ivan"  
}
```


Query example

List down the first 3 authors based upon number of magazines they have published. In order to write Mongo query for this, lets first categorize the steps:

- Project the authors out of each magazine document.
- Group the authors by name, counting the number of occurrences.
- Sort the authors by the occurrence count, descending.
- Limit results to the first three.

Query example continued...

- In order to project the author from magazine, we use \$project operator:

```
{"$project" : {"author" : 1}}
```
- Grouping by author name & count number of authors within collection, we use \$group operator:

```
{"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}
```
- Sorting based number of upon magazines published by an author, we use \$sort:

```
{"$sort" : {"count" : -1}}
```
- Show only limited number of records, use \$limit:

```
{"$limit" : 3}
```

Query example continued...

Final query:

```
db.magazine.aggregate(  
    {"$project" : {"author" : 1}},  
    {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}},  
    {"$sort" : {"count" : -1}},  
    {"$limit" : 2}  
)
```




Pipeline operations

The aggregation framework extensively uses pipeline operations.

Here one operator generate document & it becomes input to another operator & so on.. Finally the last operator returns the result to the client.

The operators can be combined in any order & repeated many times as necessary.



Pipeline operators

\$project

```
{ "$project" : { "author" : "$author" } }
```

\$group

```
{ "$group": { "_id": "$author", "count": { "$sum": 1 } } }
```

\$sort

```
{ "$sort": { count: -1 } }
```

\$limit

```
{ "$limit": 2 }
```

\$skip

```
{ $skip : 5 }
```

\$match

```
{ $match : { author : "Ivan" } }
```

\$unwind

```
{ $unwind : "$prices" }
```

\$group operators

\$sum

```
"$group" : { "totalRevenue" : { "$sum" : "$revenue" } }
```

\$avg

```
"$group" : { "averageRevenue" : { "$avg" : "$revenue" } }
```

\$max, \$min

```
"$group" : { "costlyBook" : { "$max" : "$bookPrice" } }
```

\$first, \$last

```
$group: { _id: "$item", firstSalesDate: { $first: "$date" } } }
```

\$addToSet

```
$group: { _id: { day: { $dayOfYear: "$date" }, itemsSold: { $addToSet: "$item" } } }
```

\$push

```
$group: { _id: { day: { $dayOfYear: "$date" }, itemsSold: { $push: { item: "$item",  
quantity: "$quantity" } } } }
```

Pipeline expressions

Pipeline expressions allow us to perform more powerful operations in aggregation framework like manipulating numeric values, playing with date fields, performing operations on strings, adding various logical conditions etc.

Pipeline expressions are divided into following types:

- Mathematical expressions
- Date expressions
- String expressions
- Logical expressions

Mathematical expressions

Mathematical expressions let you to manipulate numeric values.

\$add

```
{ $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
```

\$subtract

```
{ $project: { item: 1, dateDifference: { $subtract: [ "$date", 5 * 60 * 1000 ] } } }
```

\$multiply

```
{ $project: { date: 1, item: 1, total: { $multiply: [ "$price", "$quantity" ] } } }
```

\$divide

```
{ $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }
```

\$mod

```
{ $project: { remainder: { $mod: [ "$hours", "$tasks" ] } } }
```


Date expressions

Using date expressions, we can perform several operations on date field. For example, in order to find month for a date:

```
month: { $month: "$date" }
```

\$dayOfYear

\$dayOfMonth

\$dayOfWeek

\$year

\$month

\$week

\$hour

\$minute

\$second

\$milisecond

\$dateToString

String expressions

Spring expressions can be used to perform basic operations on string.

`$substr`

```
{ "$substr" : ["$firstName", 0, 1] }
```

`$concat`

```
{ "$concat" : [ "$firstName", "$lastName" ] }
```

`$toLowerCase`

```
{ "$toLowerCase" : "$firstName" }
```

`$toUpperCase`

```
{ "$toUpperCase" : "$firstName" }
```

Logical expressions

Logical expressions are used to perform conditional operations.

\$cmp

isHighQuantity: { **\$cmp**: ["\$qty", 250] }

\$strcasecmp

isNameMatching: { \$strcasecmp: ["\$firstName", "TOM"] }

\$eq, \$ne, \$gt, \$gte, \$lt, \$lte

isQuantity250: { **\$eq**: ["\$qty", 250] }

\$and, \$or

result: { **\$or**: [{ \$gt: ["\$qty", 250] }, { \$lt: ["\$qty", 200] }] }

\$not

result: { **\$not**: [{ \$gt: ["\$qty", 250] }] }

\$cond

discount: { **\$cond**: { if: { \$gte: ["\$qty", 250] }, then: 30, else: 20 } }

\$ifNull

description: { **\$ifNull**: ["\$description", "Unspecified"] }



Thank You all



Presented by



Ashwini Patil