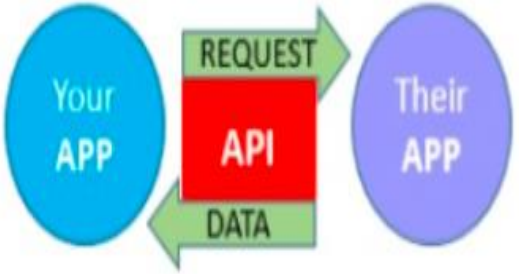
MONGO-DB

Venugopal Shastri

What is NoSql

Schema - Free

Easy – Replication



Simple API

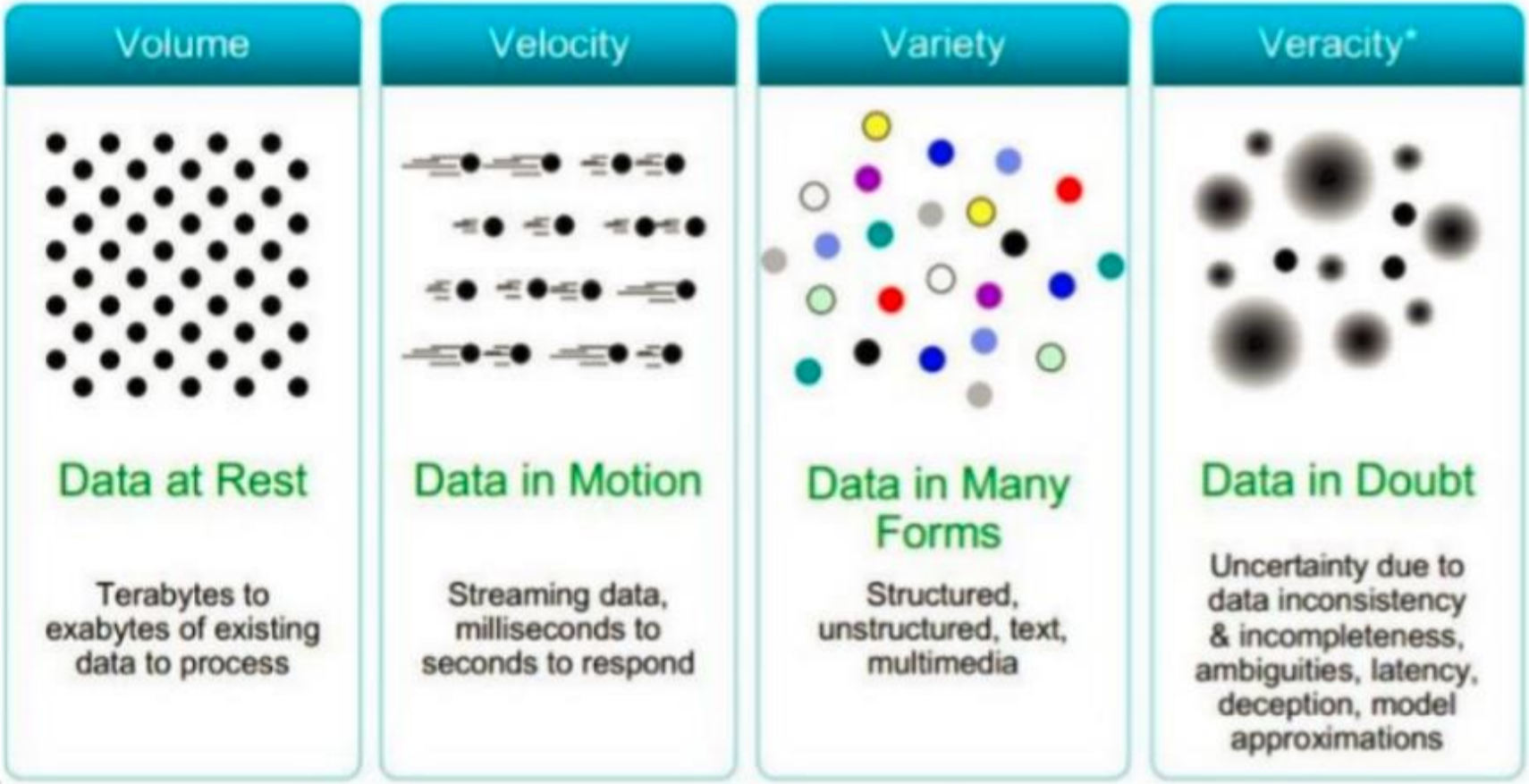
Can Manage Huge Amount of Data

Can be implement on Commodity Hardware's

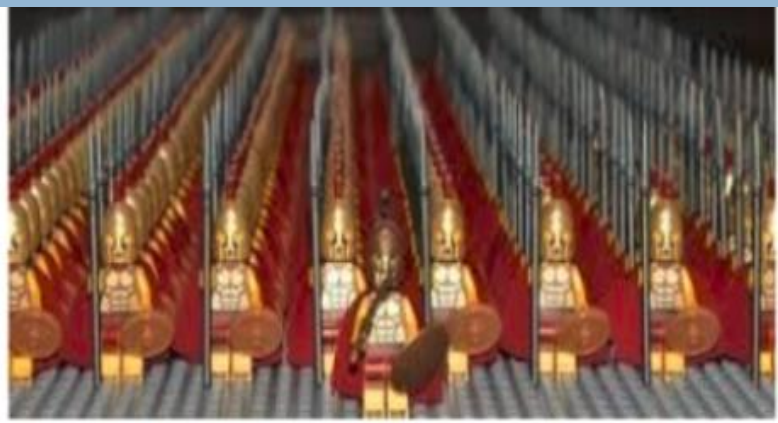
~ 150 No SQL Database are there in Market

Benefits of NoSql

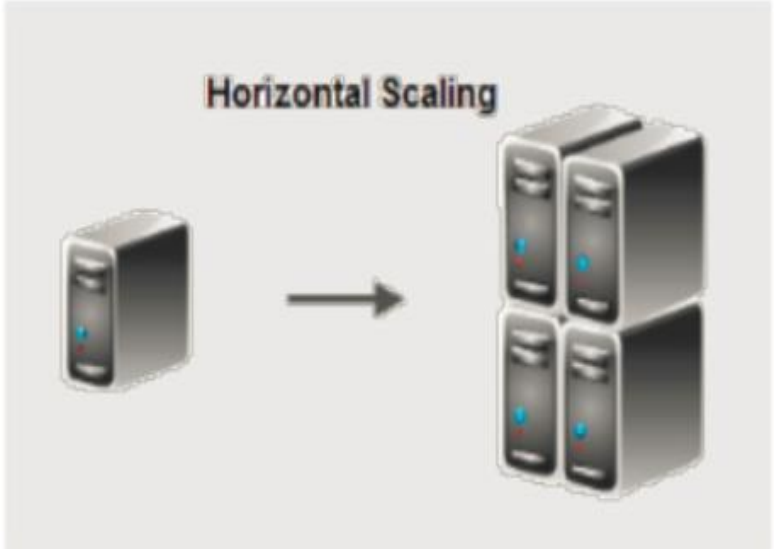
- ✓ Large volumes of structured, semi-structured, and unstructured data

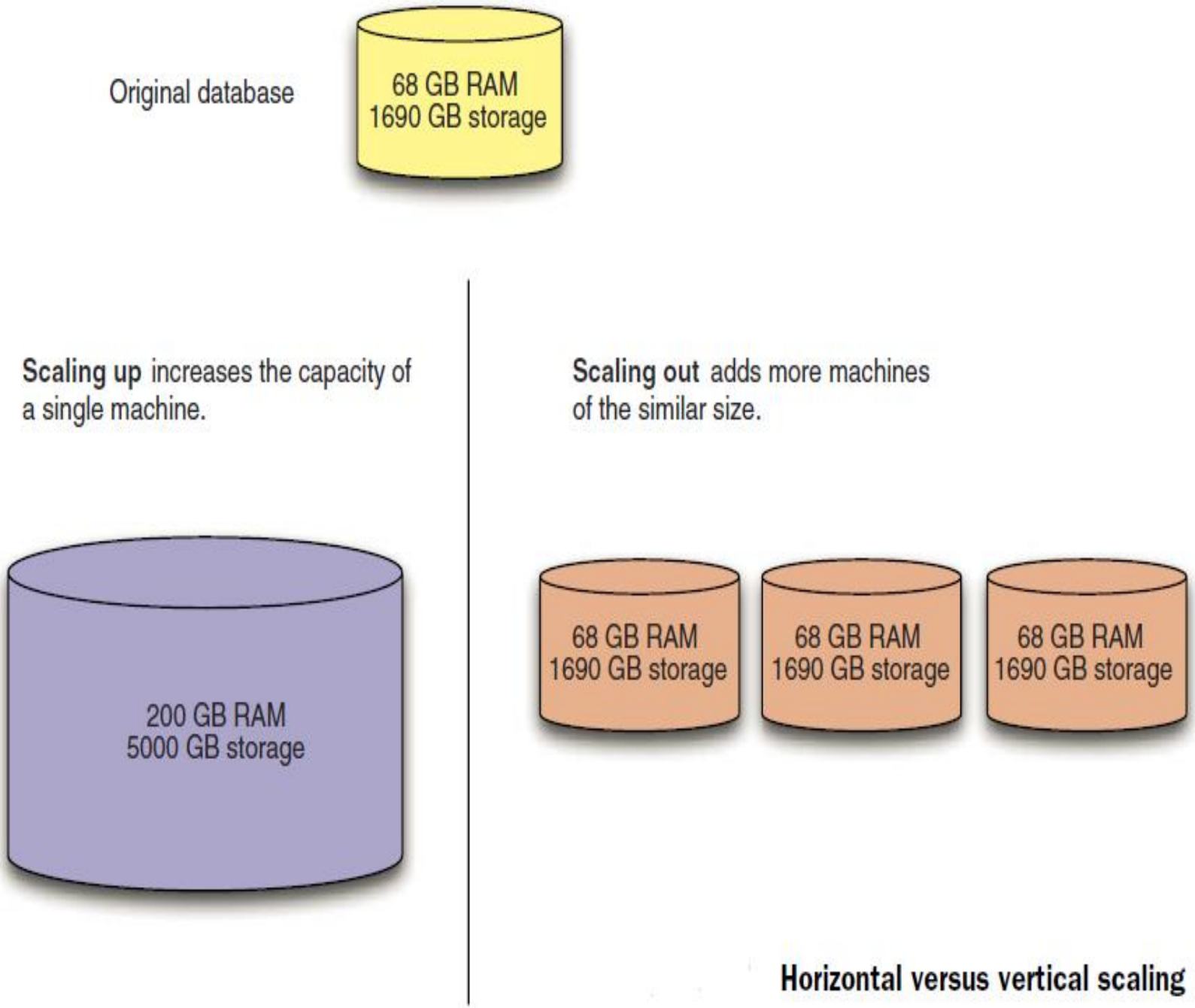


- ✓ Object-oriented programming that is easy to use and flexible



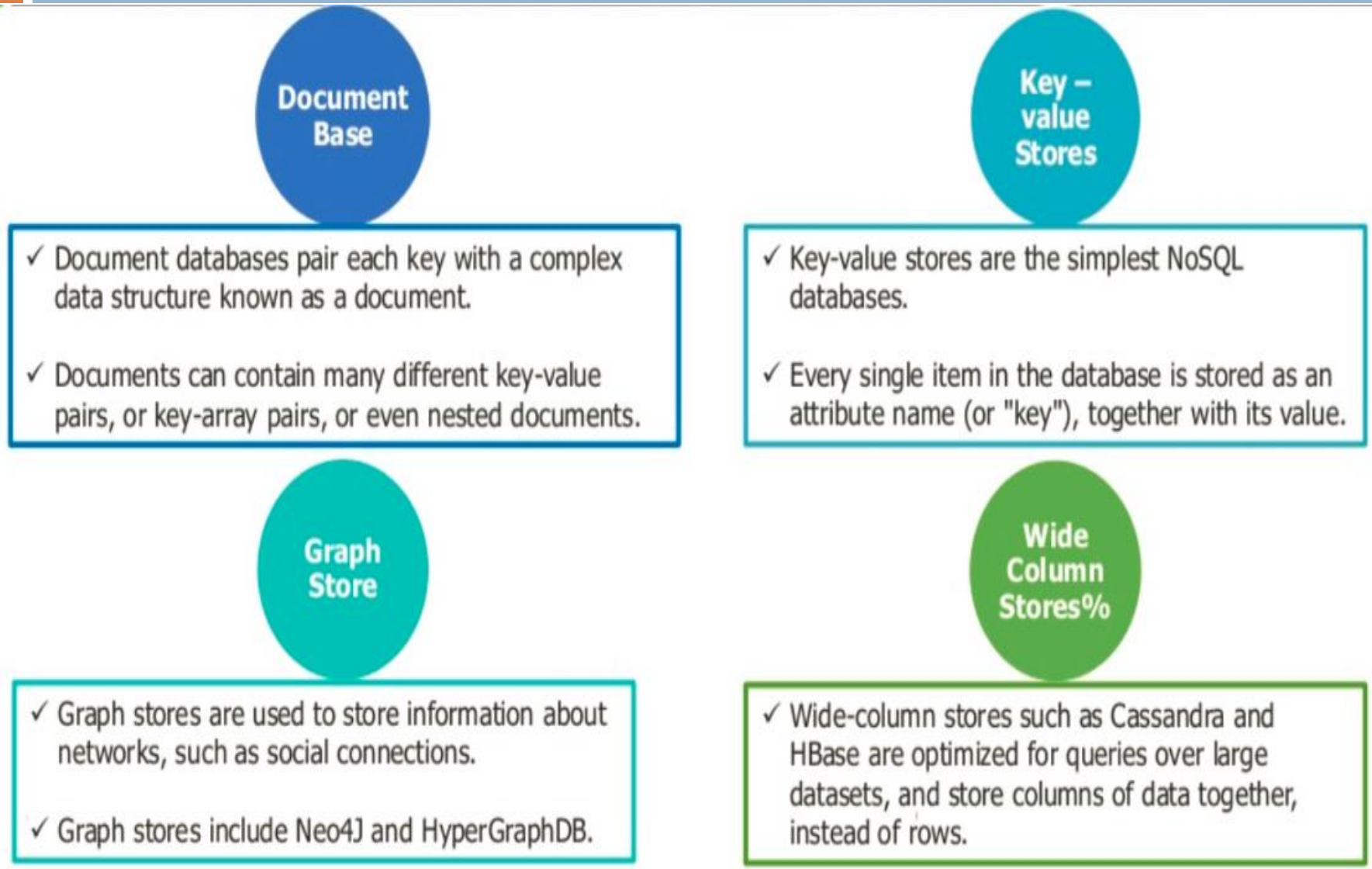
- ✓ Horizontal scaling instead of expensive hardware's



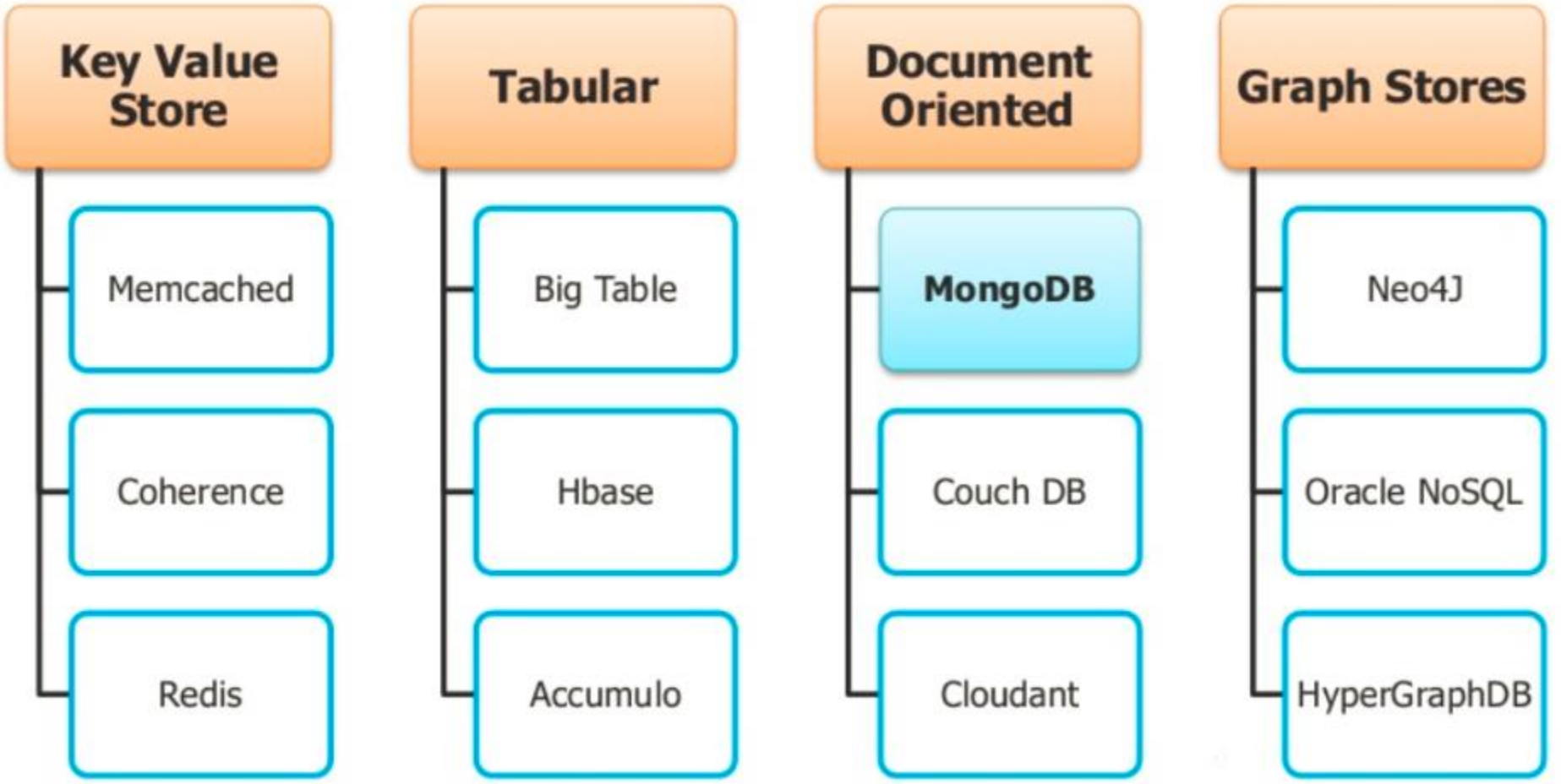


Horizontal versus vertical scaling

Categories of NoSql



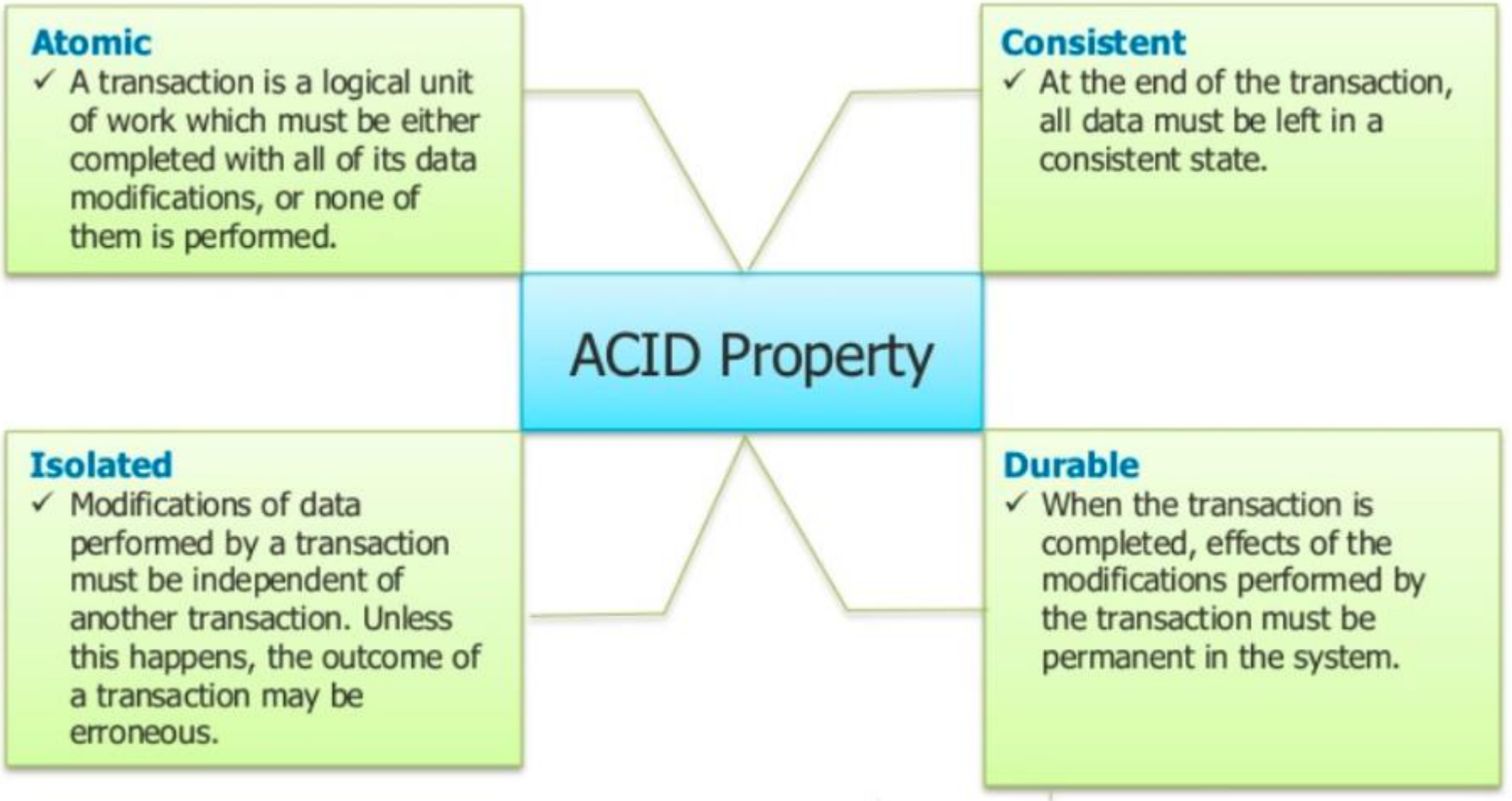
Types of NoSql



Comparison

Entity	SQL Databases	NoSQL Databases
Type	One Type (SQL) with Minor Variation	Many Types (Document, Ke-Value, Tabular, Graph)
Development	1970	2000
Examples	Oracle, MSSQL, DB2 etc.	MongoDB, Cassandra, Hbase, Neo4J
Schemas	Fixed	Dynamic
Scaling	Vertical	Horizontal
Dev Model	Mix	Open Source
Consistency	Follow ACID	Follow BASE

RDBMS (ACID)


Atomic
✓ A transaction is a logical unit of work which must be either completed with all of its data modifications, or none of them is performed.
Consistent
✓ At the end of the transaction, all data must be left in a consistent state.
ACID Property
Isolated
✓ Modifications of data performed by a transaction must be independent of another transaction. Unless this happens, the outcome of a transaction may be erroneous.
Durable
✓ When the transaction is completed, effects of the modifications performed by the transaction must be permanent in the system.

Cap Theorem

CAP theorem states that there are **3 basic requirements** which exist in a special relation when designing applications for a distributed architecture.

Consistency

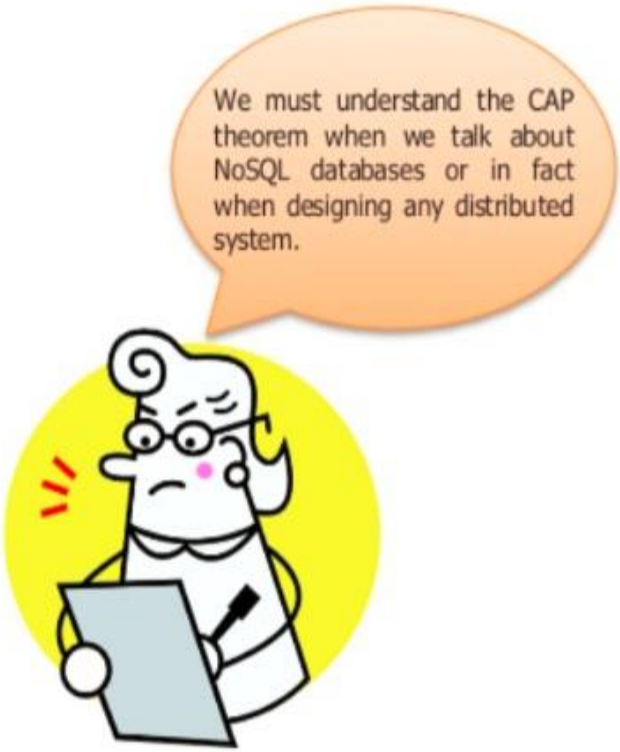
This means that the data in the database remains consistent after the execution of an operation. For example after an update operation all clients see the same data.

Availability

This means that the system is always on (service guarantee availability), no downtime.

Partition Tolerance

This means that the system continues to function even the communication among the servers is unreliable, i.e. the servers may be partitioned into multiple groups that cannot communicate with one another.



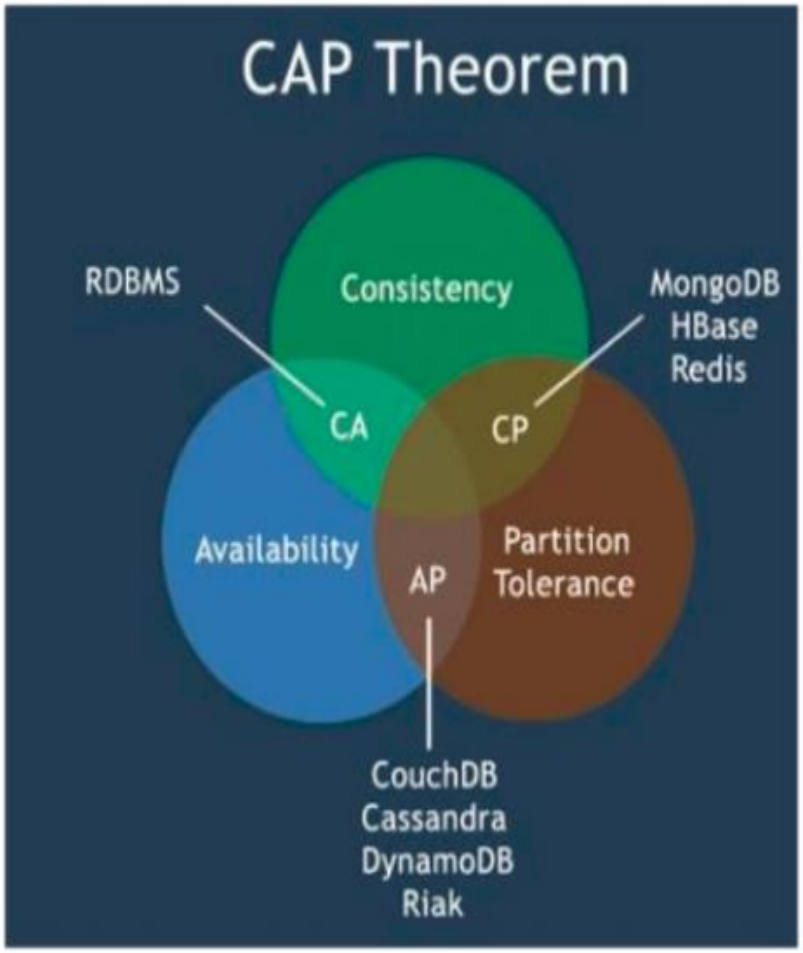
Cap

- ✓ In theoretically it is **impossible** to fulfill all 3 requirements.
- ✓ CAP provides the basic requirements for a distributed system to follow **2 of the 3 requirements.**
- ✓ Therefore all the current NoSQL database follow the different **combinations of the C, A, P** from the CAP theorem.

CAP

Here is the brief description of three combinations CA, CP, AP :

- ✓ **CA** - Single site cluster, therefore all nodes are always in contact. When a partition occurs, the system blocks.
- ✓ **CP** - Some data may not be accessible, but the rest is still consistent/accurate.
- ✓ **AP** - System is still available under partitioning, but some of the data returned may be inaccurate.



BASE

A BASE system gives up on consistency.

Basically Available

- ✓ **B**asically **A**vailable indicates that the system **does guarantee** availability, in terms of the CAP theorem.

Soft State

- ✓ **S**oft **S**tate indicates that the state of the system **may change over time**, even without input. This is because of the eventual consistency model.

Eventual Consistency

- ✓ **E**ventual **C**onsistency indicates that the system **will become consistent over time**, given that the system doesn't receive input during that time.

MongoDB Overview

Mongo DB is an Open-source database.

Developed by 10gen, for a wide variety of applications.

It is an agile database that allows schemas to change quickly as applications evolve.

Overview

Scalability, High Performance and Availability.

By leveraging in-memory computing.

MongoDB's native replication and automated failover enable enterprise-grade reliability and operational flexibility.

Challenges

New Apps

New Development Methods

New Data Volumes

New Architectures

New Data Types

What is MongoDB

Open Source

Document Oriented Storage

Object Oriented

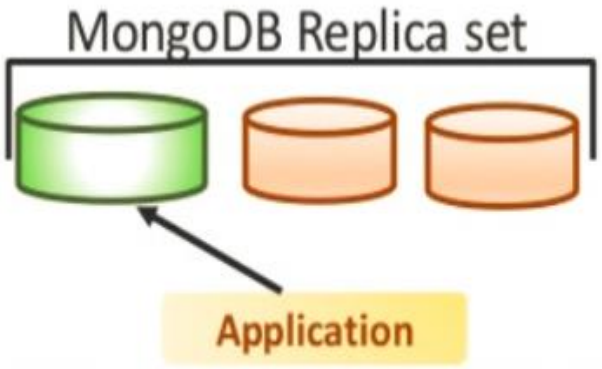
C++

Written in C++

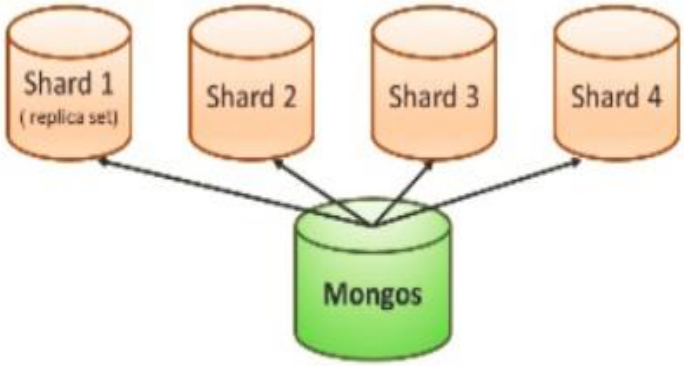
Easy to Use

Full Index Support

What is MongoDB ?

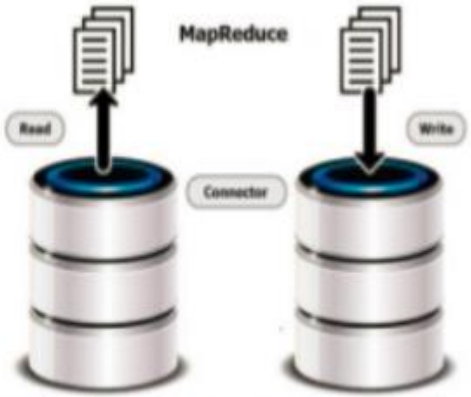


Replication and High Availability



Auto Sharding

Easy Query

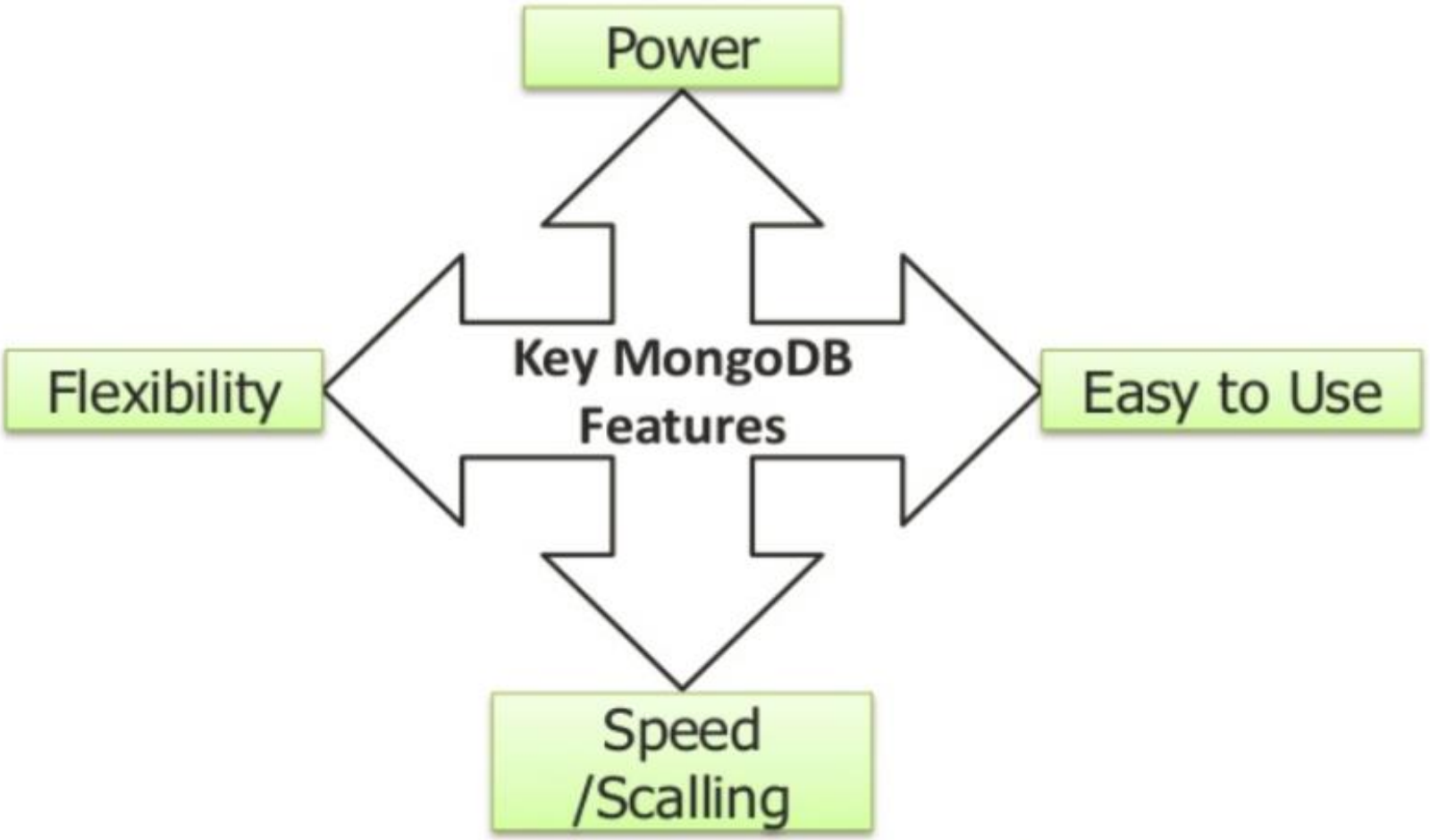


Map Reduce

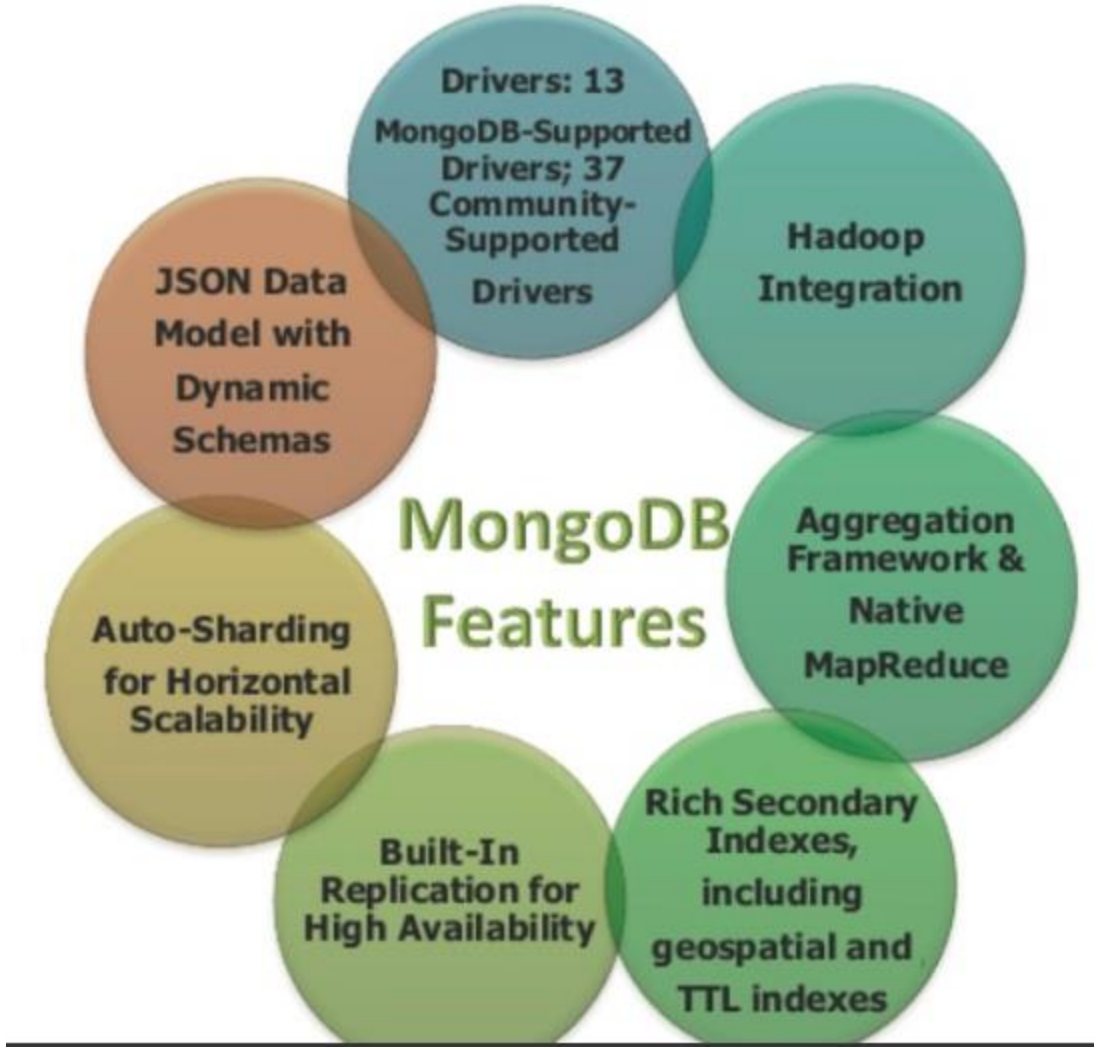
Gird FS

Support from Expert

Key Features Of MongoDB



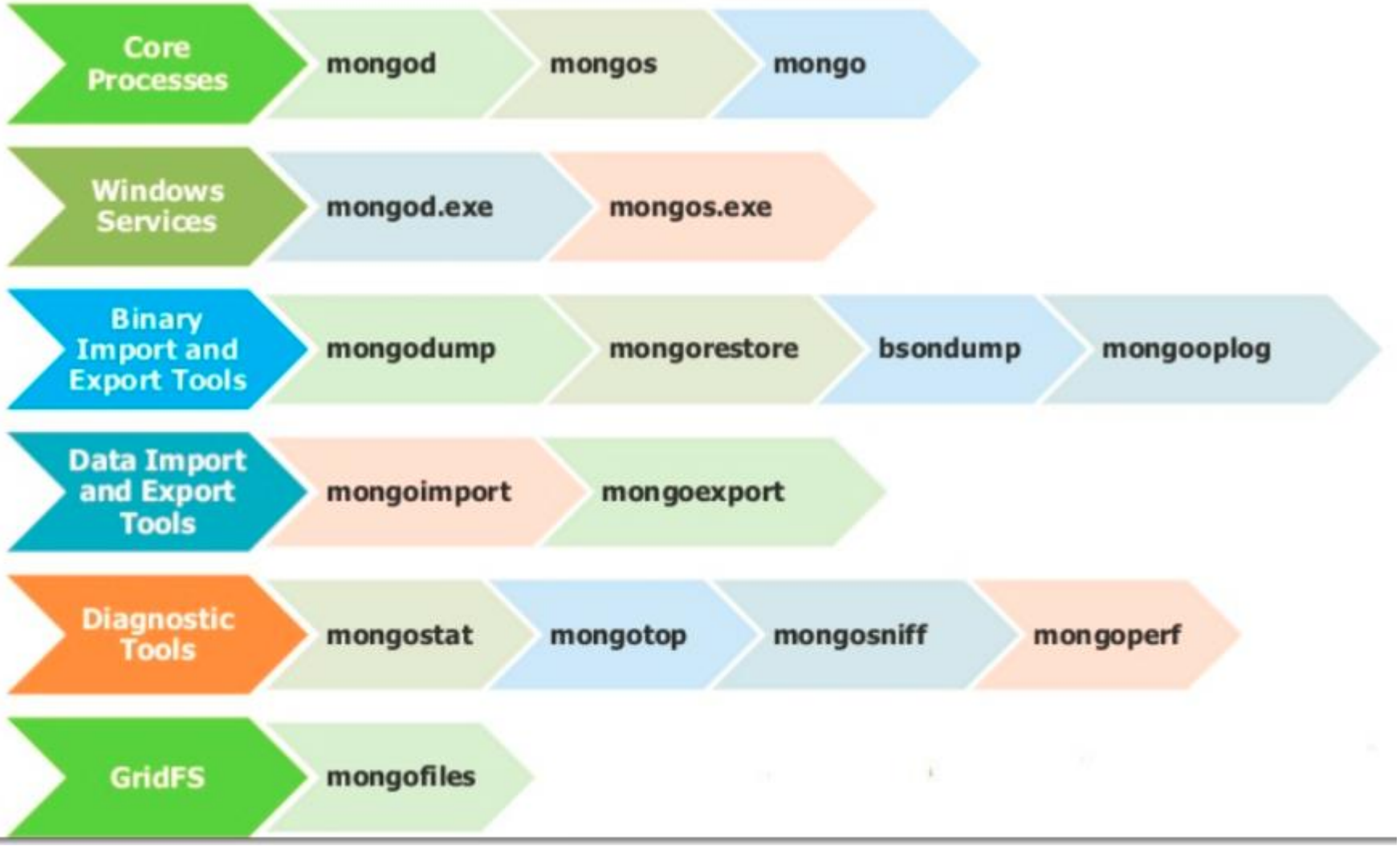
MongoDB Features



MongoDB Package Components

- ✓ mongod
- ✓ mongos
- ✓ mongo
- ✓ mongod.exe
- ✓ mongos.exe
- ✓ mongodump
- ✓ mongorestore
- ✓ bsondump
- ✓ mongooplog
- ✓ mongoimport
- ✓ mongoexport
- ✓ mongostat
- ✓ mongotop
- ✓ mongosniff
- ✓ mongoperf
- ✓ mongofiles

MongoDB Package Tools



MongoDB

- ✓ **Mongod** is the primary daemon process for the MongoDB system.
- ✓ Database is a physical container for collections.
- ✓ Each database gets its own set of files on the file system.
- ✓ A single MongoDB server typically has multiple databases.
- ✓ It handles data requests, manages data format, and performs background management operations.

MongoDB Server

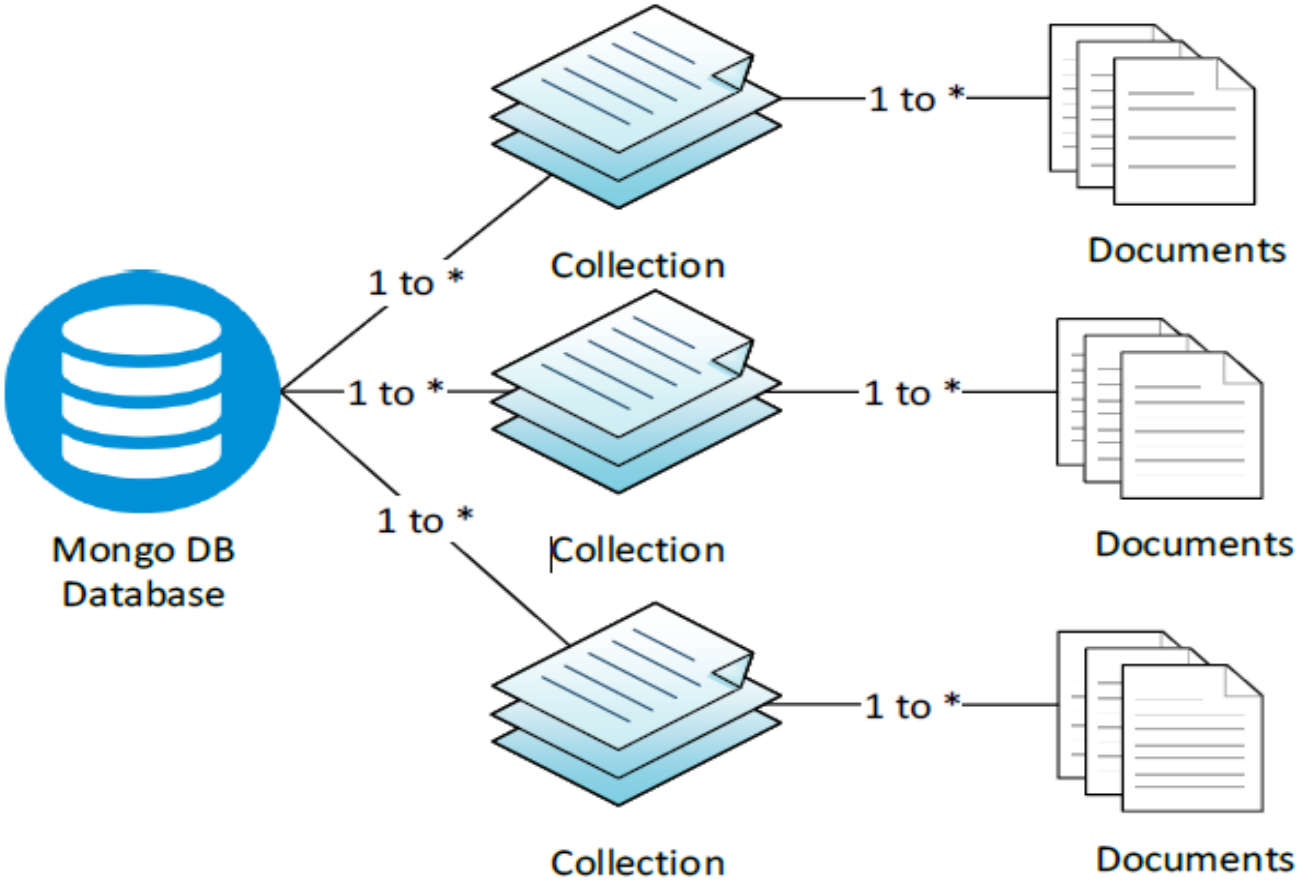
- **Running MongoDB as a standalone application**

```
C:\mongodb\bin>mongod
```

- **Installing MongoDB as a Windows service**

```
C:\mongodb\bin>mongod --logpath c:\data\log\log.log --install
```

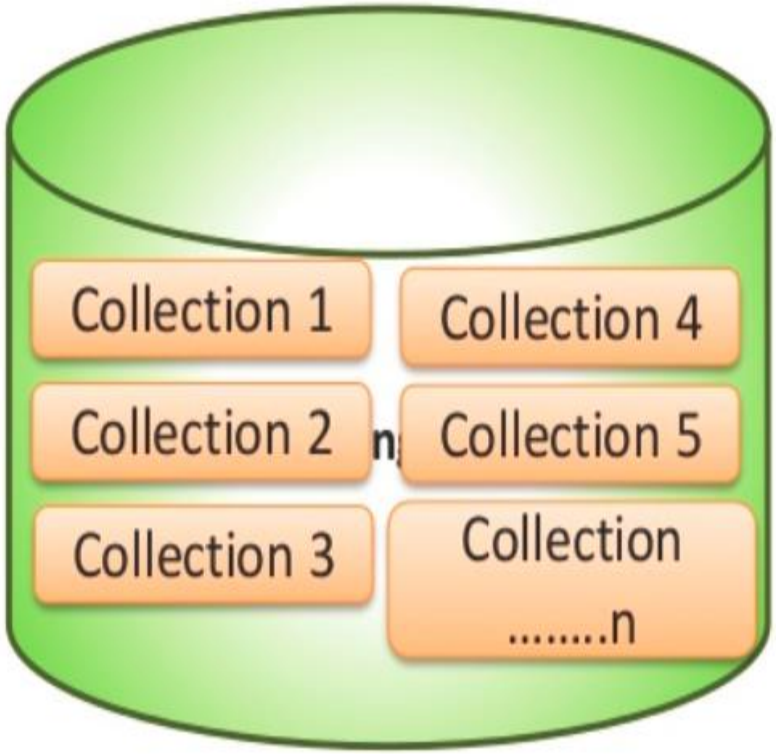
MongoDB Data Structure Org



MongoDB data structure organization.

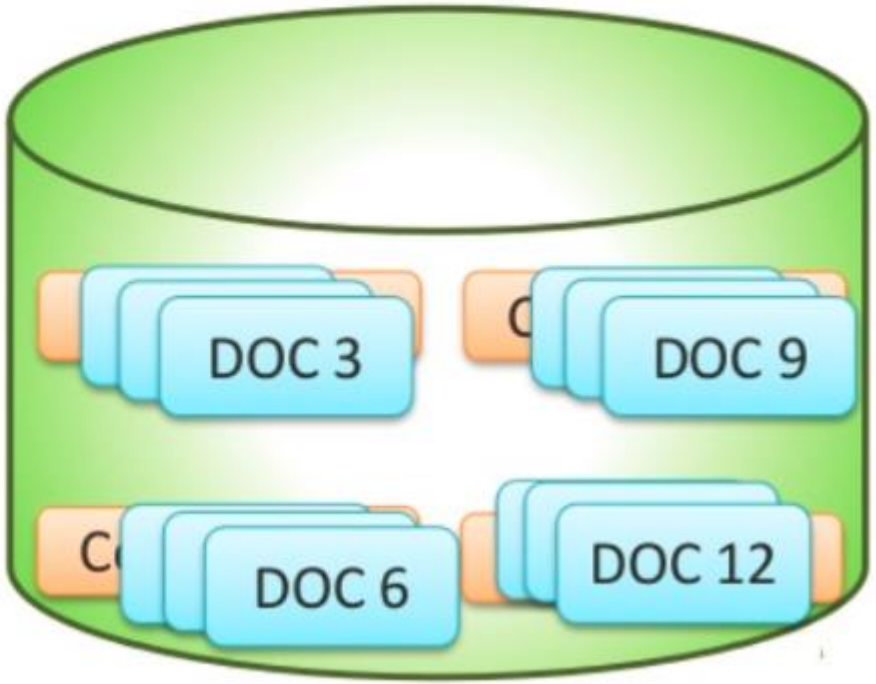
MongoDB Collection

- ✓ Collection is a group of MongoDB documents.
- ✓ It is the equivalent of an RDBMS table.
- ✓ A collection exists within a single database.
- ✓ Collections do not enforce a schema.
- ✓ Documents within a collection can have different fields.
- ✓ Typically, all documents in a collection are of similar or related purpose.

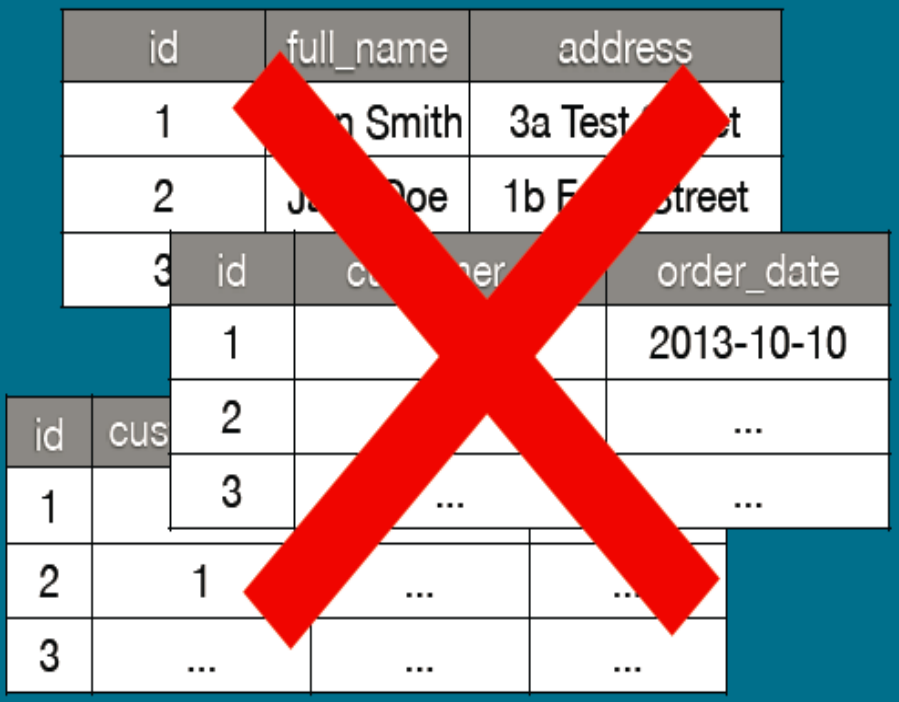


MongoDB Document

- ✓ A document is a set of key-value pairs.
- ✓ Documents have dynamic schema.



Document Database




```
customers = [
   {
      "_id" : ObjectId("5256b399ac46b80084974d9a"),
      "name" : "John Smith",
      "address" : "3a Test Street",
      "orders" [ {
         "order_date": "2013-10-10",
         "order_item": [
            { "product": "Widget"...}
            ...
         ]
         ...
      }]
   },
   {
      "_id" : ObjectId("5256b3a8ac46b80084974d9b"),
      "name" : "Jane Doe",
      "address" : "1b Fake Street"
   }
]
```

RDBMS Anology With MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column/Attribute/Variable	Field
Table Join	Embedded Documents
Database Server and Client	
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

JSON

JavaScript Object Notation

JSON Abbreviation

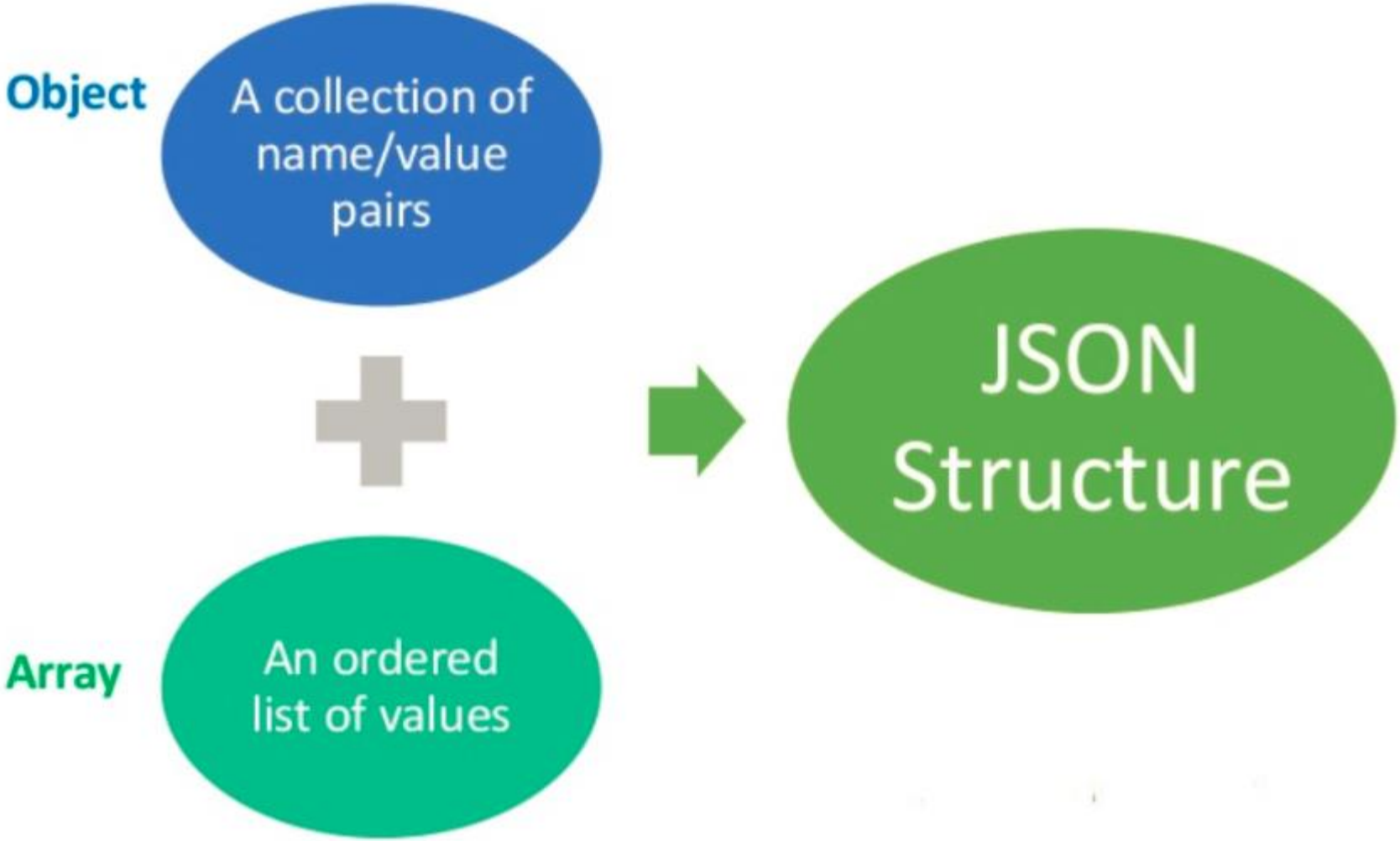
Lightweight data-interchange format

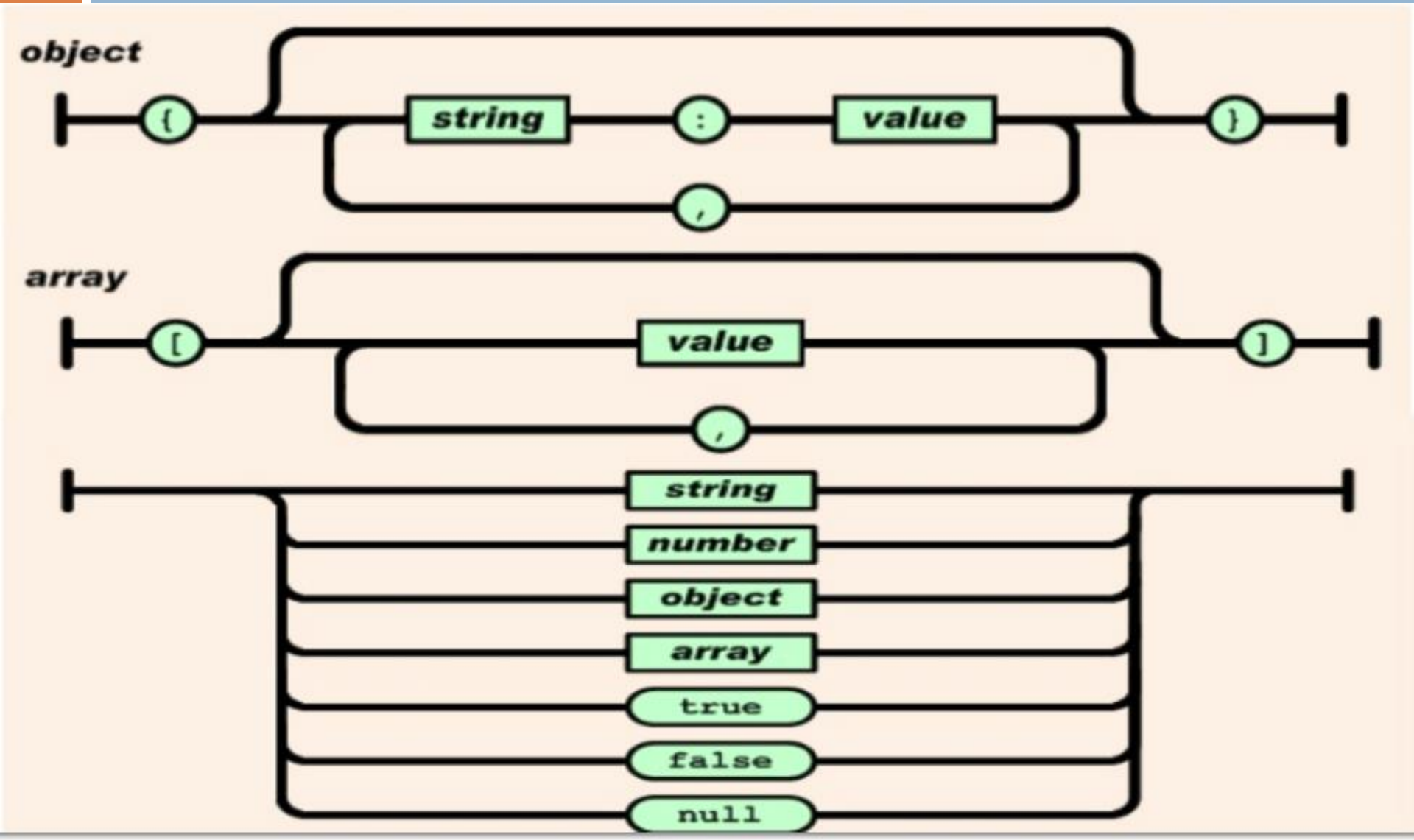
Easy for humans to read and write

Easy for machines to parse and generate

Text format that is completely language independent

JSON


Object
A collection of name/value pairs
Array
An ordered list of values
JSON Structure


object
{
string
:
value
}
,
array
[
value
]
,
string
number
object
array
true
false
null

BSON

Binary JavaScript Object Notation

BJSON Abbreviation

Supports the embedding of documents and arrays within other documents and arrays

Contains extensions that allow representation of data types that are not part of the JSON spec

Easy for machines to parse and generate

Text format that is completely language independent

MongoDB through the -Mongo Shell

- mongo.exe
- An interactive JavaScript interface to the database
- Used to query or manipulate data or perform administrative operations

```
C:\mongodb\bin>mongo --host <remoteServerName> --port 27017 --u <username>
--p <password>
```

MongoDB Storage Engine

- **MongoDB built-in engine: MMAPv1 engine**
- **WiredTiger storage engine**
  - more predictable performance than the related MMAP engine.
- **MongoDB engine (only enterprise edition)**
  - The in-memory storage engine designed to serve ultra-high throughput
- **MongoRock**s
  - MongoDB storage engine based on Facebook's RocksDB-embedded database project.

Databases, collections, and documents

- > use testdb
  - switched to db tutorial
- ***Inserts and queries***
  - db.users.insert({username: “Venu"})
  - db.users.find()
  - db.users.count()

- _ID FIELDS IN MONGODB
  - document’s primary key
  - Every MongoDB document requires an _id

Inseting a Document

`db.<collection>.insert()`	Inserts a document or collection of documents into a collection. Returns a **`BulkWriteResult`** object back to the caller.
`db.<collection>.insertOne()`	New in v3.2. Inserts a single document into a collection.
`db.<collection>.insertMany()`	New in version 3.2. Inserts multiple documents into a collection. Returns a document containing the object IDs and information if the insert is acknowledged.
`db.<collection>.save()`	Updates an existing document or inserts a new document, depending on its **`document`** parameter. Returns a **`WriteResult`** object.

primary key

- The **ObjectId** is a BSON type
- 12 bytes
  - 4-byte value representing the seconds since the Unix epoch
  - 3-byte machine identifier
  - 2-byte process ID
  - 3-byte counter, starting with a random value

Update Documents

- Update the value of an existing field.
- Change the document by adding or removing attributes (fields).
- Replace the document entirely.

`db.<collection>.update()`	Modifies document(s) in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely.
`db.<collection>.updateOne()`	New as of v3.2. Updates one document within a collection.
`db.<collection>.updateMany()`	New as of version 3.2. Updates multiple documents within a collection.

Updating documents

- Update()
- Two Arguments
  - which documents to update
  - how the selected documents should be modified
- db.users.update({username: “venu"}, {$set: {country: “IND"}})
- ***Deleting data***
  - db.foo.remove()
  - db.users.remove({"favorites.cities": "Cheyenne"})
  - db.users.drop()

PASS A QUERY PREDICATE

- db.users.find({username: “Venu"})
- db.users.find({

  ... _id: ObjectId("552e458158cd52bcb257c324"),

  ... username: “Venu“

  ... })

  … indicates that the command takes more than one line

AND – OR Operation

```
> db.users.find({ $and: [
... { _id: ObjectId("552e458158cd52bcb257c324") },
... { username: "smith" }
... ] })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

```
> db.users.find({ $or: [
... { username: "smith" },
... { username: "jones" }
... ]})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

Creating and querying with indexes

```
> for(i = 0; i < 20000; i++) {
    db.numbers.save({num: i});
  }

> db.numbers.find({num: 500})

> db.numbers.find( {num: {"$gt": 19995 }} )

> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
```

```
> db.numbers.find({num: {"$gt": 19995}}).explain("executionStats")
```

```
"cursor" : "BasicCursor",
"isMultiKey" : false,
"n" : 4,
"nscannedObjects" : 20000,
"nscanned" : 20000,
"nscannedObjectsAllPlans" : 20000,
"nscannedAllPlans" : 20000,
"scanAndOrder" : false,
"indexOnly" : false,
"nYields" : 156,
"nChunkSkips" : 0,
"millis" : 8,
"allPlans" : [
        {
                "cursor" : "BasicCursor",
                "isMultiKey" : false,
                "n" : 4,
                "nscannedObjects" : 20000
                "nscanned" : 20000,
                "scanAndOrder" : false,
                "indexOnly" : false,
                "nChunkSkips" : 0
        }
```

```
> db.numbers.createIndex({num: 1})

  db.numbers.getIndexes()
```

```
"cursor" : "BtreeCursor num_1",
"isMultiKey" : false,
"n" : 4,
"nscannedObjects" : 4,
"nscanned" : 4,
"nscannedObjectsAllPlans" : 4,
"nscannedAllPlans" : 4,
"scanAndOrder" : false,
"indexOnly" : false,
"nYields" : 0,
"nChunkSkips" : 0,
"millis" : 0,
"indexBounds" : {
        "num" : [
                [
                        19995,
                        Infinity
                ]
        ]
```

Collections

- Collections are containers for structurally or conceptually similar documents
- db.createCollection("users")
- db.createCollection("users", {size: 20000})
- db.products.renameCollection("store_products")
- Standard Collections
- Capped Collections
- TTL COLLECTIONS

Capped Collection

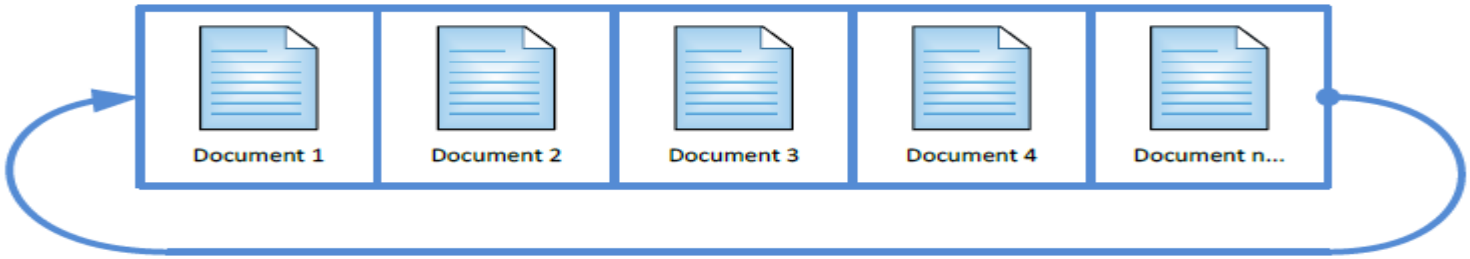
- fixed-size, circular collections
- *Fixed-size refers to the fact that there is a predefined (configurable) limit on the maximum number of items this table will support.*
- *Circular refers to the fact that once the maximum amount is reached, the oldest of the items gets deleted to make room to the new one.*
- collection itself preserves the order in which the items get inserted
- we cannot remove a document or Update Document

Capped Collection Usage

- Logging (for example, the latest activity performed on the website).
- Caching (preserving the latest items).
- Acting as a queue: Capped collection might be also used to act as a queue where the first-in-first-out logic applies.

```
db.createCollection("LogCollection", { capped:true, size:10000, max:1000}) ;
```

- **Capped**: True sets the type of the collection as capped (the default is false).
- **Size**: Sets the maximum size in bytes for this particular collection.
- **Max**: Specifies the maximum number of documents allowed for the given collection.



TTL Collection

- Expire documents from a collection after a certain amount of time has passed
- implemented using a special kind of index
- TTL index on _id is not allowed
- TTL indexes with capped collections are not allowed
- Compound TTL indexes are not allowed

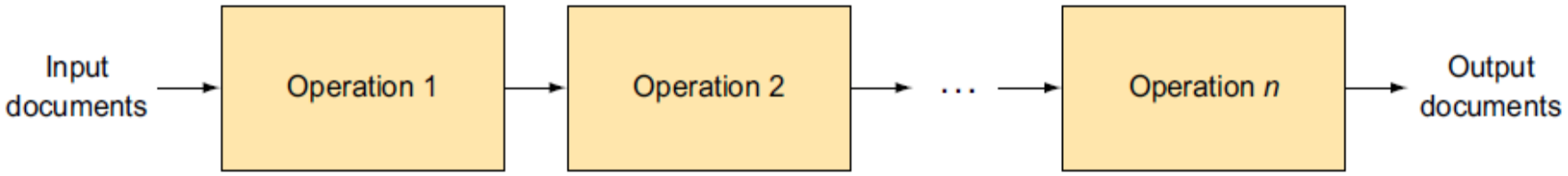
```
> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})

> db.reviews.insert({
      time_field: new Date(),
      ...
    })
```

SYSTEM COLLECTIONS

- system.namespaces
- system.indexes

Aggregation framework is MongoDB's

- advanced query language
- transform and combine data from multiple documents to generate new information
- A call to the aggregation framework defines a pipeline



Aggregation Pipeline Operations

- $project—Specify fields to be placed in the output document (projected).
- $match—Select documents to be processed, similar to find().
- $limit—Limit the number of documents to be passed to the next step.
- $skip—Skip a specified number of documents.
- $unwind—Expand an array, generating one output document for each array entry.
- $group—Group documents by a specified key.
- $sort—Sort documents.
- $geoNear—Select documents near a geospatial location.
- $out—Write the results of the pipeline to a collection (new in v2.6).
- $redact—Control access to certain data (new in v2.6).

```
db.users.aggregate([
    {$match: {username: 'kbanker',
              hashed_password: 'bd1cfa194c3a603e7186780824b04419'}},
    {$project: {first_name:1, last_name:1}}
])
```

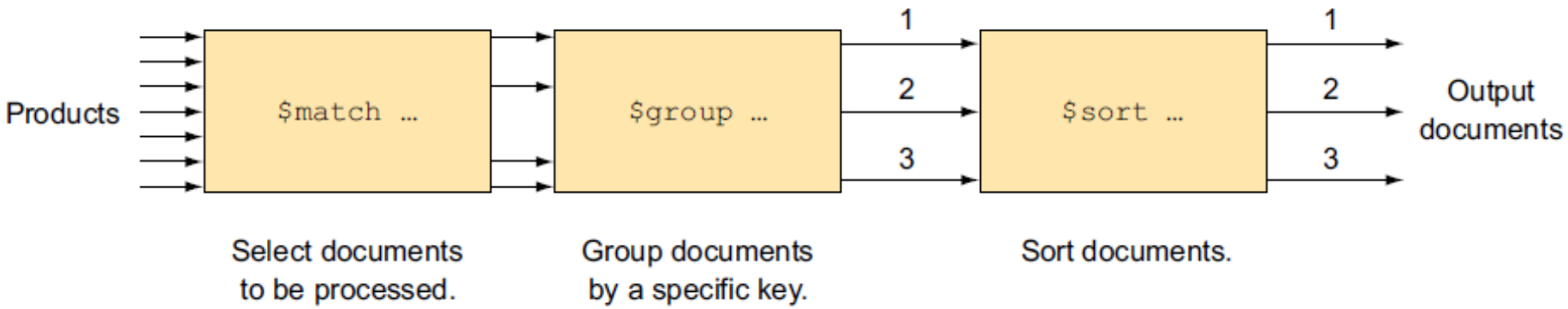
Project pipeline operator that returns first name and last name

```
db.orders.aggregate([
    {$project: {user_id:1, line_items:1}},
    {$unwind: '$line_items'},
    {$group: {_id: {user_id:'$user_id'},
              purchasedItems: {$push: '$line_items'}}}
]).toArray();
```

$push function adds object to purchasedItems array

```
reviews2 = db.reviews.aggregate([
    {$match: {'product_id': product['_id']}},
    {$skip : (page_number - 1) * 12},
    {$limit: 12},
    {$sort:  {'helpful_votes': -1}}
]).toArray();
```

Example aggregation framework pipeline



SQL command	Aggregation framework operator
SELECT	$project
	$group functions: $sum, $min, $avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	$unwind
WHERE	$match
GROUP BY	$group
HAVING	$match

Examples

```
product  = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

```
db.reviews.aggregate([
 {$group : { _id:'$product_id',
             count:{$sum:1} }}
]);
```

Group the input documents by product_id.

Count the number of reviews for each product.

```
product  = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
 {$match : { product_id: product['_id']} },
 {$group : { _id:'$product_id',
             count:{$sum:1} }}
]).next();
```

Select only a single product.

Return the first document in the results.

Examples

```
product  = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary  = db.reviews.aggregate([
    {$match : {'product_id': product['_id']}},
    {$group : { _id:'$product_id',
        average:{$avg:'$rating'},
        count: {$sum:1}}}
]).next();
```

Calculate the average rating for a product.

```
countsByRating = db.reviews.aggregate([
 {$match : {'product_id': product['_id']}},
 {$group : { _id:'$rating',
             count:{$sum:1}}}
]).toArray();
```

Select product

Group by value of rating: '$rating'

Count number of reviews for each rating

Convert resulting cursor to an array

```
db.mainCategorySummary.remove({});

db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]).forEach(function(doc){
    var category = db.categories.findOne({_id:doc._id});
    if (category !== null) {
        doc.category_name = category.name;
    }
    else {
        doc.category_name = 'not found';
    }
    db.mainCategorySummary.insert(doc);
})
```

Remove existing documents from mainCategorySummary collection

Read category for a result

You aren't guaranteed the category actually exists!

Insert combined result into your summary collection

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}},
    {$out : 'mainCategorySummary'}
])
```

Save pipeline results to collection mainCategorySummary

```
db.products.aggregate([
    {$project : {category_ids:1}},
    {$unwind : '$category_ids'},
    {$group : { _id:'$category_ids',
                count:{$sum:1}}},
    {$out : 'countsByCategory'}
]);
```

Pass only the array of category IDs to the next step. The _id attribute is passed by default.

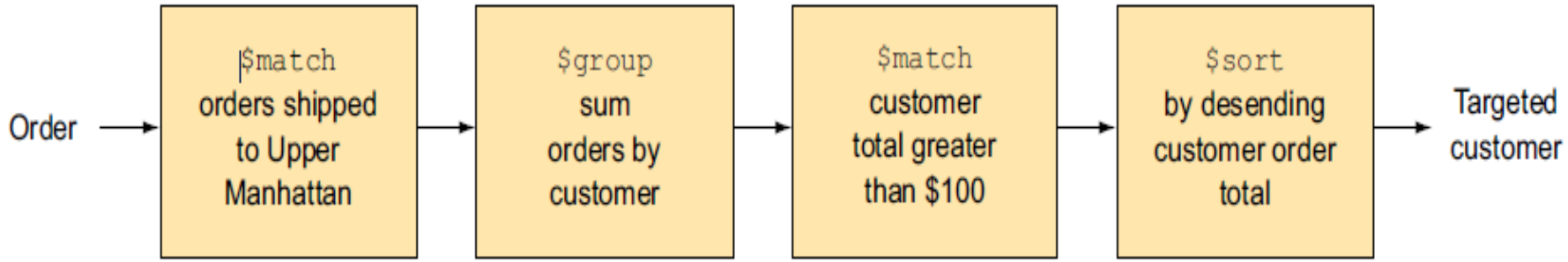
Create an output document for every array entry in category_ids.

$out writes aggregation results to the named collection countsByCategory.

```
db.orders.aggregate([
      {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
      {$group: {
            _id: {year : {$year :'$purchase_data'},
                  month: {$month :'$purchase_data'}},
            count: {$sum:1},
            total: {$sum:'$sub_total'}}},
      {$sort: {_id:-1}}
]);
```

SUMMARIZING SALES BY YEAR AND MONTH

```
db.orders.aggregate([
      {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
      {$group: {
            _id: {year : {$year :'$purchase_data'},
                  month: {$month :'$purchase_data'}},
            count: {$sum:1},
            total: {$sum:'$sub_total'}}},
      {$sort: {_id:-1}}
]);
```




```
upperManhattanOrders = {'shipping_address.zip': {$gte: 10019, $lt: 10040}};

sumByUserId = {_id: '$user_id',
               total: {$sum:'$sub_total'}, };

orderTotalLarge = {total: {$gt:10000}};

sortTotalDesc = {total: -1};
```

```
db.orders.aggregate([
    {$match: upperManhattanOrders},
    {$group: sumByUserId},
    {$match: orderTotalLarge},
    {$sort: sortTotalDesc}
]);
```

1. Project the authors out of each article document.
2. Group the authors by name, counting the number of occurrences.
3. Sort the authors by the occurrence count, descending.
4. Limit results to the first five.

1. `{"$project" : {"author" : 1}}`
2. `{"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}`
3. `{"$sort" : {"count" : -1}}`
4. `{"$limit" : 5}`

```
> db.articles.aggregate({"$project" : {"author" : 1}},
... {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}},
... {"$sort" : {"count" : -1}},
... {"$limit" : 5})
```

Aggregation pipeline options

- db.collection.aggregate(pipeline,additionalOptions)
- explain()—Runs the pipeline and returns only pipeline process details
- allowDiskUse—Uses disk for intermediate results
- cursor—Specifies initial batch size
- {explain:true, allowDiskUse:true, cursor: {batchSize: n} }

Aggregation in C#

- building the pipeline
- Each operation in the pipeline will make modifications to the data: the operations can for example filter, group and project the data
- pipeline is a collection of BsonDocument object
- Each document represents one operation.

```
var match = new BsonDocument
    {
        {
            "$match",
            new BsonDocument
                {
                    {"name", "Tom"}
                }
        }
    };

    var pipeline= PipelineDefinition<BsonDocument,BsonDocument>.Create(match);
    var result = users.Aggregate<BsonDocument>(pipeline).ToList() ;
```

Aggregation by pre-filtering the data to be grouped

```
var aggregate = collection.Aggregate()
    .Match(Builders<Movie>.Filter.Where(x => x.Name.Contains("Godfather")))
    .Group(new BsonDocument
    {
        {"_id", "$year"},
        {"count", new BsonDocument("$sum", 1)}
    });

var results = aggregate.ToList();
```

Understanding aggregation pipeline performance

- Try to reduce the number and size of documents as early as possible in your pipeline.
- Indexes can only be used by $match and $sort operations and can greatly speed up these operations.
- You can't use an index after your pipeline uses an operator other than $match or $sort.

Connecting to the database (C#)

```
mongodb://[username:password@]host1[:port1][,hostN[:portN]]
[/[database][?options]]
```

```
string connectionString = "mongodb://localhost:27017";

MongoClient client = new MongoClient(connectionString);
```

Authentication

```
string dbName = "ecommlight";
string userName = "some_user";
string password = "pwd";

var credentials = MongoCredential.CreateCredential(dbName, userName, password);

MongoClientSettings clientSettings = new MongoClientSettings()
{
    Credentials = new[] { credentials },
    Server = new MongoServerAddress("localhost", 27017)
};

MongoClient client = new MongoClient(clientSettings);

Console.WriteLine("Connected as {0}", userName);
```

Referencing a database

Server > Database > Collection > Document > Data

```
IMongoDatabase database = client.GetDatabase(dbName);
```

```
using (var cursor = await client.ListDatabasesAsync())
{
    await cursor.ForEachAsync(d => Console.WriteLine(d.ToString()));
}
```

```
var databases = client.ListDatabases().ToList();
databases.ForEach(d => Console.WriteLine(d.GetElement("name").Value));
```

```
client.DropDatabase(databaseName);
```

```
await client.DropDatabaseAsync(databaseName);
```

Working with collections

- **CreateCollection** and **CreateCollectionAsync**: Creates a new collection if not already available.
- **ListCollections** and **ListCollectionsAsync**: Lists the already available collections on the database.
- **DropCollection** and **DropCollectionAsync**: Deletes (drops) a collection from the given database.

```
//create a new collection.
database.CreateCollection(collectionName);
```

```
var collectionsList = database.ListCollections();
```

```
foreach (var collection in collectionsList.ToList())
{
    Console.WriteLine(collection.ToString());
}
```

BulkWrite

- BulkWrite and BulkWriteAsync
  - Uses WriteModel
    - InsertOneModel
    - DeleteOneModel
    - DeleteManyModel
    - UpdateOneModel
    - UpdateManyModelModel
    - ReplaceOneModel

Object Mapping

- Using CustomAttributes
- BsonClassMap

```
BsonClassMap.RegisterClassMap<Movie>(movie =>
{
    movie.MapIdProperty(p => p.MovieId)
            .SetIdGenerator(new StringObjectIdGenerator());
    movie.MapProperty(p => p.Name).SetElementName("name");
    movie.MapProperty(p => p.Director).SetElementName("director

    movie.MapProperty(p => p.Year).SetElementName("year");
    movie.MapProperty(p => p.Actors).SetElementName("actors");
    movie.UnmapProperty(p => p.Age);
    movie.MapExtraElementsMember(p => p.Metadata);
    movie.SetIgnoreExtraElements(true);
});
```

Find (Query) Data in C#

```
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);
    var collection = db.GetCollection<BsonDocument>(collName);
    var filter = new BsonDocument();
    int count = 0;
    using (var cursor = await collection.FindAsync<BsonDocument>(filter))
    {
        while (await cursor.MoveNextAsync())
        {
            var batch = cursor.Current;
            foreach (var document in batch)
            {
                var movieName = document.GetElement("name").Value.ToString(
);
                Console.WriteLine("Movie Name: {0}", movieName);
                count++;
            }
        }
    }
```

FilterDefinitionBuilder

```
/* Filter to retrieve movies where the name equals to "The Godfather" */
var expresssionFilter = Builders<Movie>.Filter.Eq(x => x.Name, "The Godfath
er");

/* Filter to retrieve movies where the name equals to "The Godfather"
 * by using BsonDocument notation */
var bsonFilter = Builders<BsonDocument>.Filter.Eq("name", "The Godfather");
```

```
/* find movies where the name is "The Godfather" OR "The Seven Samurai" */
var filter = Builders<Movie>.Filter.Or(new[]
{
    new ExpressionFilterDefinition<Movie>(x => x.Name == "The Godfather"),
    new ExpressionFilterDefinition<Movie>(x => x.Name == "The Seven Samurai
")
});
```

```
var collection = db.GetCollection<Movie>(collName);

var movies = collection.Find(x => x.Name == "The Godfather");
```

```
var filter = Builders<BsonDocument>.Filter.Lt("age", 25);
var filter = Builders<Student>.Filter.Lt(student => student.Age, 25);
```

```
var builder = Builders<BsonDocument>.Filter;
var filter = builder.Lt("Age", 40) & builder.Eq("FirstName", "Peter");
```

Projecting data

- Return just a subset of data for a given query.
  - This is called a projection of data
- Main entry point for the projections is the **Builders**

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
       .Include("name")
       .Include("year")
       .Exclude("_id");

var data = collection.Find(new BsonDocument())
           .Project<BsonDocument>(projection)
           .ToList();

foreach (var item in data)
{
    Console.WriteLine("Item retrieved {0}", item.ToString());
}
```

Projection defined as strongly typed Object

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
        .Include(x => x.Name)
        .Include(x => x.Year)
        .Exclude(x => x.MovieId);

var data = await collection.Find(new BsonDocument())
           .Project<Movie>(projection)
           .ToList();
```

Async version of the strongly typed projection definition

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
        .Include(x => x.Name)
        .Include(x => x.Year)
        .Exclude(x => x.MovieId);

var options = new FindOptions<Movie, BsonDocument>
{
    Projection = projection
};

var cursor = await collection.FindAsync(new BsonDocument(), options);
var data = cursor.ToList();

foreach (var item in data)
{
    Console.WriteLine("Item retrieved {0}", item.ToString());
}
```

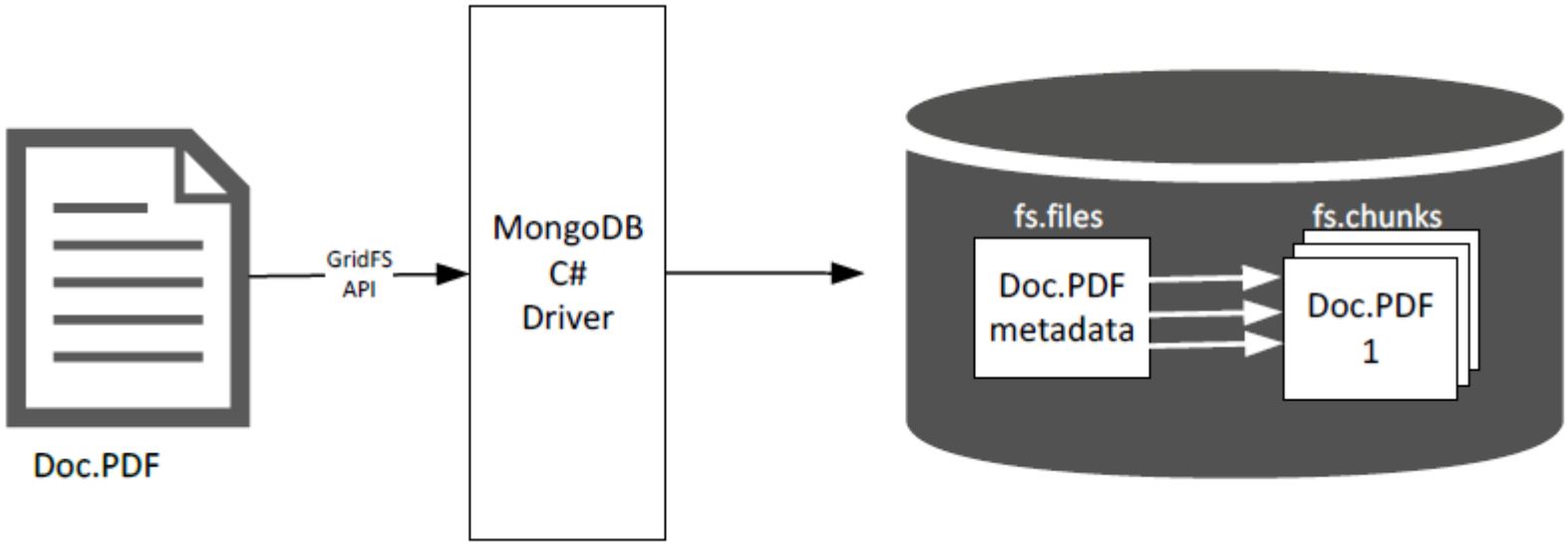
Binary Data (File Handling) in C#

- **GridFS** is a MongoDB specification and a way of storing binary information larger than the maximum document size
- GridFS It is kind of a file system to store files
- GridFS divides a file into parts called ***chunks***
  - Each chunk is a separate document
  - 255 kilobytes of data.
- When the file is downloaded (retrieved) from GridFS, **the original content is reassembled.**

GridFS

- Two Built in Collections
  - Fs.files
    - store the file's metadata
  - fs.chunks
    - Store the chunks
    - each chunk is identified by its unique **_id**
- The **fs.files** acts as a **parent document**. The **files_id** assigned to a chunk holds a reference to its parent

GridFs


Doc.PDF
GridFS
API
MongoDB
C#
Driver
fs.files
fs.chunks
Doc.PDF
metadata
Doc.PDF
1

GridFs – C#

- Use **GridFSBucket** in order to interact with the underlying GridFS system
- Install the **MongoDB.Driver.GridFS** package from NuGet.
  - Install-Package MongoDB.Driver.GridFS -Version 2.5.0
- Uploading files
  - specifying the **location** on the disk
  - submitting the data to a **Stream object that the driver supplies.**

Uploading File

- >db.fs.files.find()
- >db.fs.chunks.find()

```
public static void UploadFile() {

    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionStrin
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    byte[] source = File.ReadAllBytes("..//..//sample.pdf");
    ObjectId id = bucket.UploadFromBytes("sample.pdf", source);
    Console.WriteLine(id.ToString()); }
}
```

Command Prompt - mongo

```
> db.fs.files.find().pretty();
{
        "_id" : ObjectId("5881457bc6d11b21c0dded80"),
        "length" : NumberLong(77123),
        "chunkSize" : 261120,
        "uploadDate" : ISODate("2017-01-19T23:02:20.141Z"),
        "md5" : "97a36af46c74151b55378c02055f796b",
        "filename" : "sample.pdf"
}
>
> db.fs.chunks.find().pretty();
{
        "_id" : ObjectId("5881457cc6d11b21c0dded81"),
        "files_id" : ObjectId("5881457bc6d11b21c0dded80"),
        "n" : 0,
        "data" : BinData(0,"JVBERi0xLjQNJeLjz9MNCjYgMCBvYmogPDwvTGluZWFyaXplZCAxL0wgNzcxMjMvTyA4L0U
gNzI5MDcvTiAxL1QgNzY5NTcvSCBbIDg5NiAyMDNdPj4NZW5kb2JqDSAgICAgICAgICAgICAgICAgICAgICAgICAgICAgDQp4cmVmDQo2IDMwDQ
owMDAwMDAwMDE2IDAwMDAwIG4NCjAwMDAwMDEwOTkgMDAwMDAgbg0KMDAwMDAwMTE3NSAwMDAwMCBuDQowMDAwMDAxMzU3IDAwM
DAwIG4NCjAwMDAwMDE0NzMgMDAwMDAgbg0KMDAwMDAwMTYwNyAwMDAwMCBuDQowMDAwMDAxODkwIDAwMDAwIG4NCjAwMDAwMDIw
MTkgMDAwMDAgbg0KMDAwMDAwMjM5NSAwMDAwMCBuDQowMDAwMDAzNDU1IDAwMDAwIG4NCjAwMDAwMDQ0NzEgMDAwMDAgbg0KMDA
wMDAwNTM1MSAwMDAwMCBuDQowMDAwMDA2MzMzIDAwMDAwIG4NCjAwMDAwMDczOTkgMDAwMDAgbg0KMDAwMDAwODM4NCAwMDAwMC
BuDQowMDAwMDA5NDEwIDAwMDAwIG4NCjAwMDAwMTA0MTYgMDAwMDAgbg0KMDAwMDAwMDAyMjY0OCAwMDAwMCBuDQowMDAwMDIyOTAwI
DAwMDAwIG4NCjAwMDAwMjMwODYgMDAwMDAgbg0KMDAwMDAyMzM3MCAwMDAwMCBuDQowMDAwMDM3OTgxIDAwMDAwIG4NCjAwMDAw
MzgyMzQgMDAwMDAgbg0KMDAwMDA1MTU1NiAwMDAwMCBuDQowMDAwMDUxODAyIDAwMDAwIG4NCjAwMDAwNTE5ODMgMDAwMDAgbg0
KMDAwMDA1MjI2OCAwMDAwMCBuDQowMDAwMDcyNTg0IDAwMDAwIG4NCjAwMDAwNzI4MzEgMDAwMDAgbg0KMDAwMDAwMDg5NiAwMD
AwMCBuDQp0cmFpbGVyDQo8PC9TaXplIDM2L1ByZXYgNzY5NDcvUm9vdCA3IDAgUi9lEWzw2Q0UwNTExNkQxM
zc1QUECMkEyRDgxMzU5QjQ3MUM2QT48N0E5QUJEMjY4NTM3MEM0MEI1QDU3QTk5NTMzNUEzRDc+XT4+DQpzdGFydHhyZWYNCjAN
```

Uploading files from a stream

```
public static void UploadFileFromAStream()
{
    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionString);
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    Stream stream = File.Open("..//..//sample.pdf", FileMode.Open);
    var options = new GridFSUploadOptions()
    {
        Metadata = new BsonDocument() {
                            { "author", "Venu" },
                            { "year", 2017 }
        }
    };
    var id = bucket.UploadFromStream("newsample.pdf", stream, options);
    Console.WriteLine(id.ToString());

}
```

Downloading files

- Download as **Byte Array**
- Receiving back a **Stream object from the driver**

`DownloadAsBytes` `DownloadAsBytesAsync`	Downloads a file stored in GridFS and returns it as a byte array.
`DownloadAsBytesByName` `DownloadAsBytesByNameAsync`	Downloads a file stored in GridFS and returns it as a byte array.
`DownloadToStream` `DownloadToStreamAsync`	Downloads a file stored in GridFS and writes the contents to a stream.
`DownloadToStreamByName` `DownloadToStreamByNameAsync`	Downloads a file stored in GridFS and writes the contents to a stream.

DownloadAsBytes

```
public static async Task DownloadFile()
{
    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionString);
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    var filter = Builders<GridFSFileInfo<ObjectId>>.Filter.Eq(x => x.Filename,"sample.pdf");
    var searchResult = await bucket.FindAsync(filter);
    var fileEntry = searchResult.FirstOrDefault();
    byte[] content = await bucket.DownloadAsBytesAsync(fileEntry.Id);
    File.WriteAllBytes("..//..//sample2.pdf", content);

}
```

DownloadBytes By Name

```
public static async Task DownloadFileByName()
{
    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionString);
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    byte[] content = await bucket.DownloadAsBytesByNameAsync("sample.pdf");
    File.WriteAllBytes("..//..//sampleclone.pdf", content);


}
```

Download to a stream

```
public static async Task DownloadFileToStream()
{
    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionString);
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    var filter = Builders<GridFSFileInfo<ObjectId>>.Filter.Eq(x => x.Filename, "newsample.pdf");
    var searchResult = await bucket.FindAsync(filter);
    var fileEntry = searchResult.FirstOrDefault();
    var file = "..//..//ssamplestream.pdf";
    using (Stream fs = new FileStream(file, FileMode.CreateNew, FileAccess.Write)) {
        await bucket.DownloadToStreamAsync(fileEntry.Id, fs); fs.Close();
    }
}
```

Download to a stream By Name

```
public static async Task DownloadFileToStreamByName()
{
    string connectionString = "mongodb://localhost:27017";
    mongo.MongoClient _client = new mongo.MongoClient(connectionString);
    var database = _client.GetDatabase("employees");
    IGridFSBucket bucket = new GridFSBucket(database);
    var file = "..//..//samplenamestream.pdf";
    using (Stream fs = new FileStream(file, FileMode.CreateNew, FileAccess.Write))
    {
        await bucket.DownloadToStreamByNameAsync("newsample.pdf", fs);
        fs.Close();
    }
}
```

Back Up and Restore

```
mongodump -h localhost --db mydb -o c:\backup
```

Where:

- **-h** represents the host where the MongoDB runs.
- **--db** represents the database to be backed up.
- **-o** contains the output folder.

```
mongorestore -h localhost --db mydb --drop c:\backup
```

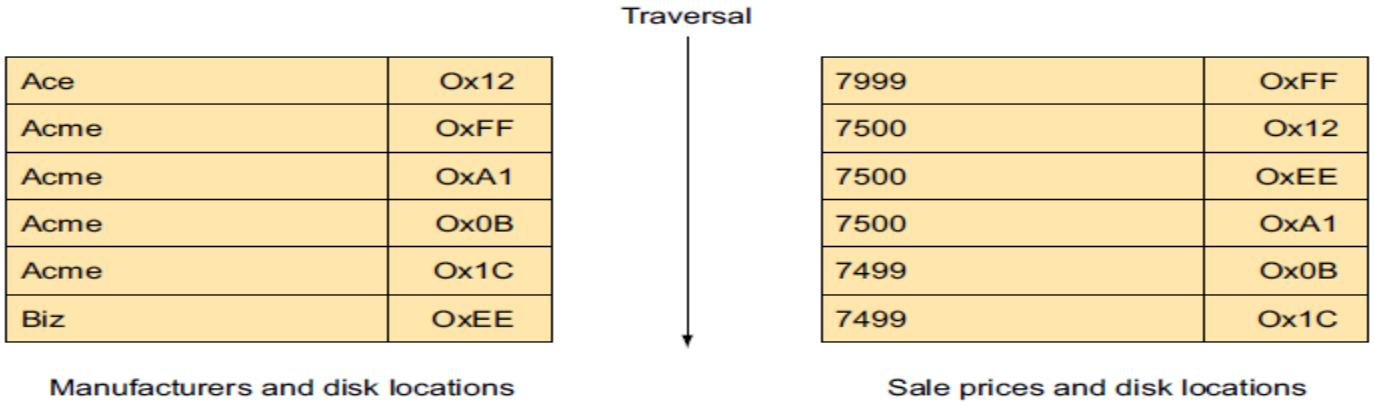
- **-h** represents the host where the MongoDB runs.
- **--db** represents the database to be restored.
- **--drop** contains the information of dropping the collection before recreating it.

Indexing and Query Optimization

- INDEXING RULES
- Indexes significantly reduce the amount of work required to fetch documents.
- Without the proper indexes, the only way to satisfy a query is to scan all documents linearly until the query conditions are met.
- Only one single-key index will be used to resolve a query
- if you have a compound index on a-b, then a second index on a alone will be redundant, but not one on b.
- The order of keys in a compound index matters

Index Types

- single-key index
  - Each entry in the index corresponds to a single value from each of the documents indexed.
- COMPOUND-KEY INDEXES
  - query on more than one attribute
  - A compound index is a single index where each entry is composed of more than one key



```
db.products.find({
    'details.manufacturer': 'Acme',
    'pricing.sale': {
      $lt: 7500
    }
  })
```

7999 – Acme	OxFF
7500 – Ace	OxEE
7500 – Acme	Ox12
7500 – Biz	OxA1
7499 – Acme	Ox0B
7499 – Acme	Ox1C

Prices and manufacturers, with disk locations

As a general rule, a query where one term demands an exact match and another specifies a range requires a compound index where the range key comes second

Ace – 8000	Ox12
Acme – 7999	OxFF
Acme – 7500	OxA1
Acme – 7499	Ox0B
Acme – 7499	Ox1C
Biz – 8999	OxEE

Manufacturers and prices, with disk locations

Indexing in MongoDb

- UNIQUE INDEXES
  - Unique indexes enforce uniqueness across all their entries
  - db.users.createIndex({username: 1}, {unique: true})
- Sparse Indexes
  - db.ensureIndex({"email" : 1}, {"unique" : **true, "sparse" : true}**)
- MULTIKEY INDEXES
  - db.values.createIndex({open: 1, close: 1})
- DEFRAGMENTING
  - db.values.reIndex();

Sparse Indexes

- unique indexes count null as a value
  - Collection cannot have a unique index with more than one document missing the key
- How to enforce the unique index only if the key exists
- Sparse indexes do not necessarily have to be unique
  - If you have a field that may or may not exist but must be unique when it does

USING THE PROFILER

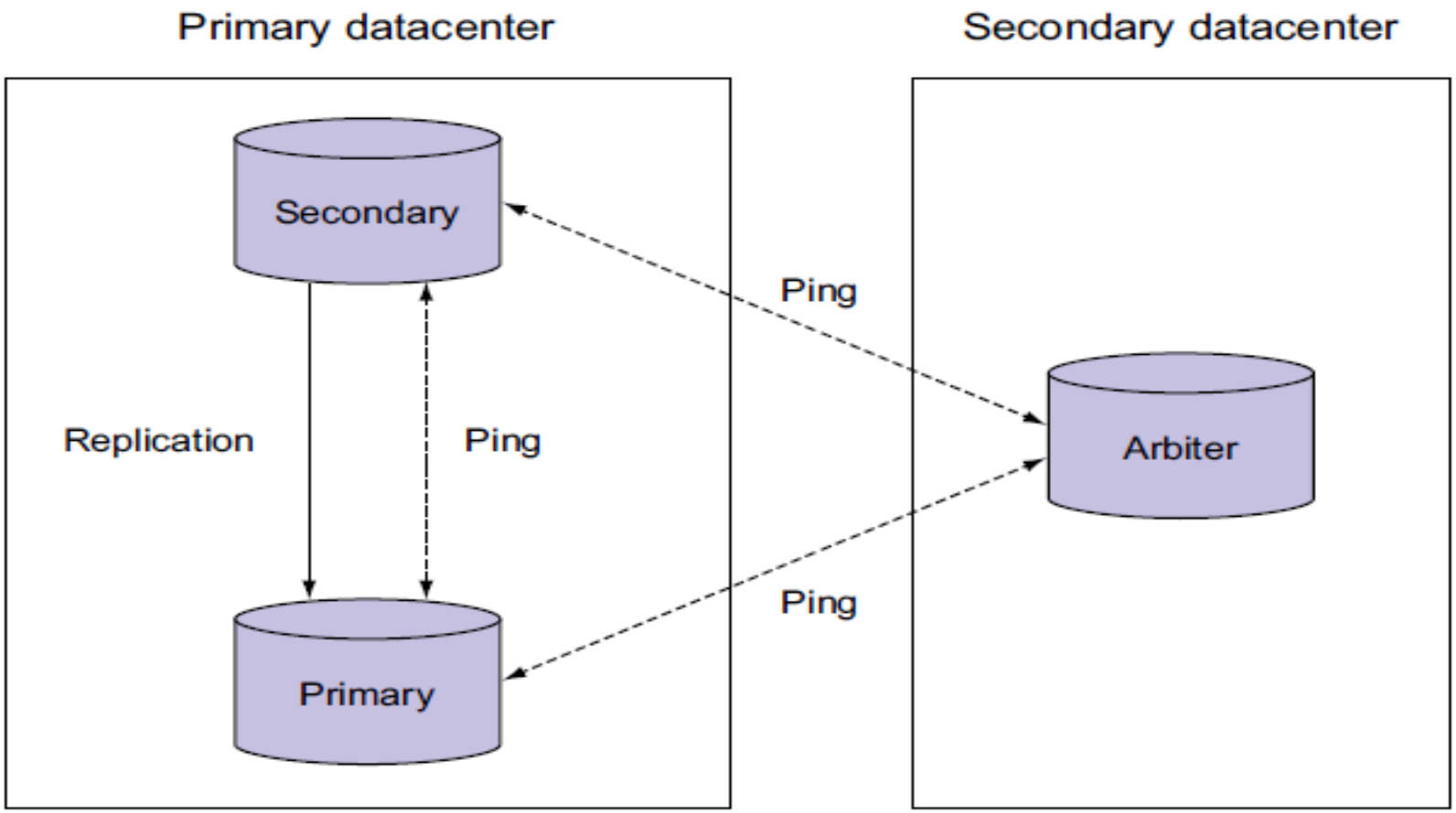
- db.setProfilingLevel(2)
- db.setProfilingLevel(1, 50)
- Profiling Results
  - stored in a special capped collection called system.profile
  - The system.profile collection is allocated 128 KB
  - db.system.profile.find({millis: {$gt: 150}})
  - db.system.profile.find().sort({$natural: -1}).limit(5).pretty()

Summary

- Query optimization is always application-specific
- Make a habit of profiling and explaining your queries
- Indexes are incredibly useful but carry a cost—they make writes slower
- MongoDB generally uses only one index in a query, so queries on multiple fields require compound indexes to be efficient
- Order matters when you declare compound indexes

- You should plan for, and avoid, expensive queries. Use MongoDB's explain command, its expensive query logs, and its profiler to discover queries that should be optimized.
- Optimize queries by reducing the number of documents scanned. The explain command is immensely useful for discovering what a query is doing; use it as a guide for optimization

Replication



Summary

- We recommend that every production deployment of MongoDB where data protection is critical should use a replica set. Failing that, frequent backups are especially essential.
- A replica set should include at least three members, though one of these can be an arbiter.
- Data isn't considered committed until it has been written to a majority of replica set members. In a failure scenario, if a majority of members remain they'll continue to accept writes. Writes that haven't reached a majority of members in this situation will be placed in the rollback data directory and must be handled manually.
- If a replica set secondary is down for a period of time, and the changes made to the database don't fit into MongoDB's oplog, this node will be unable to catch up and must be resynced from scratch. To avoid this, try to minimize the downtime of your secondaries.
- .

- The driver's write concern controls how many nodes must be written to before returning. Increase this value to increase durability. For real durability, we recommend you set it to a majority of members to avoid rollback scenarios, though his approach carries a latency cost.
- MongoDB give you fine-grained controls over how reads and writes behave in more complex replica sets using read preferences and tagging. Use theseoptions to optimize the performance of your replica set, especially if you have set members in multiple datacenters

MongoDB Limits

- Max document size: 16 MB
- Max document nesting level: 100 (documents inside documents inside documents…)
- Namespace is limited to ~123 chars
- Index field can't contain more than 1024 bytes
- Max 64 indexes per collection
- Max 31 fields in a compound index
- On windows, mongod can't store more than 4 TB of data (8 TB without journal)
- Max 12 nodes in a replica set
- Max 7 voting nodes in a replica set
- You can't refer db object in $where functions.
- Database names are case-sensitive (even on case-insensitive file systems)
- Forbidden characters in database names: linux - /. “, windows - /. “*<>:|?
- Forbidden characters in collection names: $ sign, “system.” prefix
- Forbidden characters in field names: .$
- Hashed index can't be unique
- Max connection number is hardcoded to 20k.