NoSQL databases and MongoDB

Agenda

- What is NoSQL databases
- Different kinds of NoSQL databases
- What is MongoDB?
- Overview of MongoDB.
- MongoDB's key features
- MongoDB's core server and tools.
- Installing MongoDB
- Use cases and production deployment
- Data Types, Schema Design and Data Modelling

Why NoSQL?

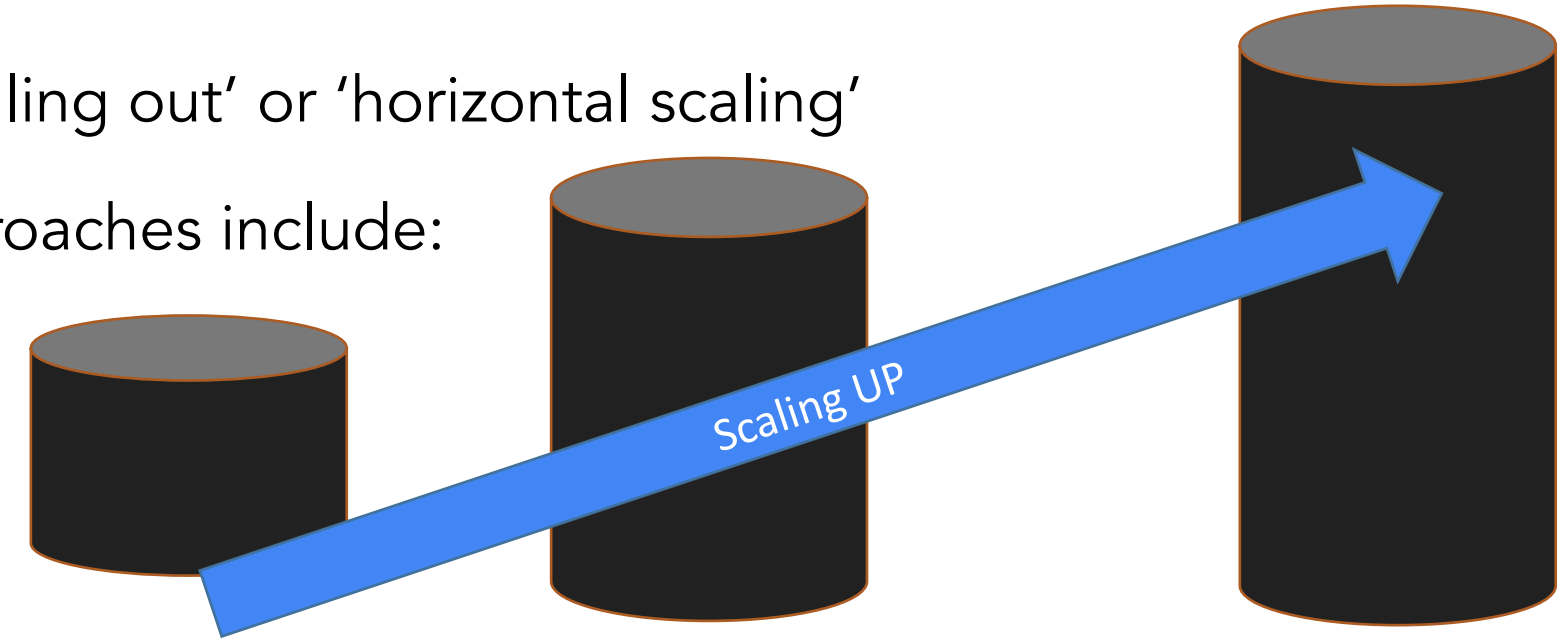
- Relational databases → mainstay of business
- Web-based applications caused spikes
- explosion of social media sites (Facebook, Twitter) with large data needs
- Example of such data : Personal user information, geo location data, social graphs , user generated contents , machine-logging data, sensor generated data etc
- rise of cloud-based solutions such as Amazon S3 (simple storage solution)

Challenges with RDBMS

- Hooking RDBMS to web-based application becomes trouble
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application (ie. Ehcache)
- As datasets grew, the simple memcache/MySQL model (for lower-cost startups) started to become problematic
- Hence, developers look forward to improve existing applications and develop new applications which can meet the needs of Big Data

Issue with Scaling Up

- Not possible to store and query when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as ‘scaling out’ or ‘horizontal scaling’
- Different approaches include:
- Master-slave
- Sharding



Different RDBMS Cluster Approach

Master-Slave

- Master handles all the write request because it must maintain the locks to guarantee roll back in case of failure.
- All reads goes to the replicated slave databases
- Reads from slave may be inconsistent as writes may not have been propagated down

Partition or sharding

- Scales well for both reads and writes
- Not transparent, application needs to be partition-aware
- Can no longer have relationships/joins across partitions
- Loss of referential integrity across shards

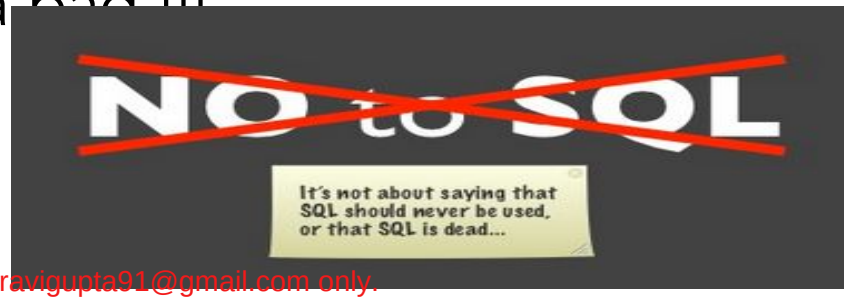
Other ways to scale RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINs, thereby reducing query time
- This involves de-normalizing data
- In-memory databases

What is NoSQL?

Stands for Not Only SQL

- The term NOSQL was given by Carl Strozzi in 1998 to name his file-based database
- It was again re-introduced by Eric Evans when an event was organized to discuss open source distributed databases
- Eric clarified later that NoSQL does not mean "No to SQL"
- NoSQL means seeking alternatives for the relational databases specially for those use cases for which RDBMS are a bad fit



Features of NoSQL

- non-relation based , no primary and foreign key relationship
- Either don’t require schema or provide flexible schema
- data are replicated to multiple nodes, data are partitioned too
- down nodes easily replaced
- no single point of failure
- horizontal scalable
- open-source software
- massive write performance
- fast key-value access

Types of NoSQL databases

1. Key-value

   Example: DynamoDB, Voldermort, Scalaris

2. Document-based

   Example: MongoDB, CouchDB

3. Column-based

   Example: BigTable, Cassandra, Hbase

4. Graph-based

   Example: Neo4J, InfoGrid

- "No-schema" is a common characteristics of most NOSQL storage systems
- Provide "flexible" data types

Benefits of NoSQL databases

- High Scalability : Ability to execute more and more queries and store more and more data without having any upper limit.
- High Availability : Ability to run Read/Write queries even when some servers are down

Disadvantage of NoSQL databases

- Don’t fully support relational features
- Normalization can’t be used, so No JOIN Queries
- no referential integrity constraints
- Non-Availability of SQL query language
- More programing is needed to work with these DBs
- NoSQL don’t follow and provide ACID properties
- They provide fewer guarantees by following CAP theorem
- No easy integration with applications that needs JDBC or ODBC drivers

Who are users of NoSQL

- All the e-commerce companies such as Flipkart, Amazon, Walmart etc use
- NoSQL database for storing huge volume of data and large amount of request from user.
- All the Cab aggregator companies such as OLA and UBER
- The mobile app companies like Kobo and Playtika
- Consumer appliances companies such as LG, Samsung etc use NoSQL for IOT use cases
- NOSQL has been used by some of the mobile gaming companies like, electronic arts, zynga and tencent for Social Gaming use cases

Tradeoff in Cluster Databases

PES UNIVERSITY

ACID

A DBMS is expected to support "ACID transactions," processes that are:
Atomicity: either the whole operation is performed, or none is
Consistency: Whenever clients reads the data it will be consistent
Isolation: one operation or transaction at a time by holding locks
Durability: once data is committed, DBs will safely keep this data even in the case of process crash

CAP

Consistency: all the nodes on cluster has the same copies
Availability: cluster always accepts reads and writes even if some nodes are down
Partition tolerance: guaranteed properties are maintained even if cluster gets divided into 2 or more partitions because of network failures.

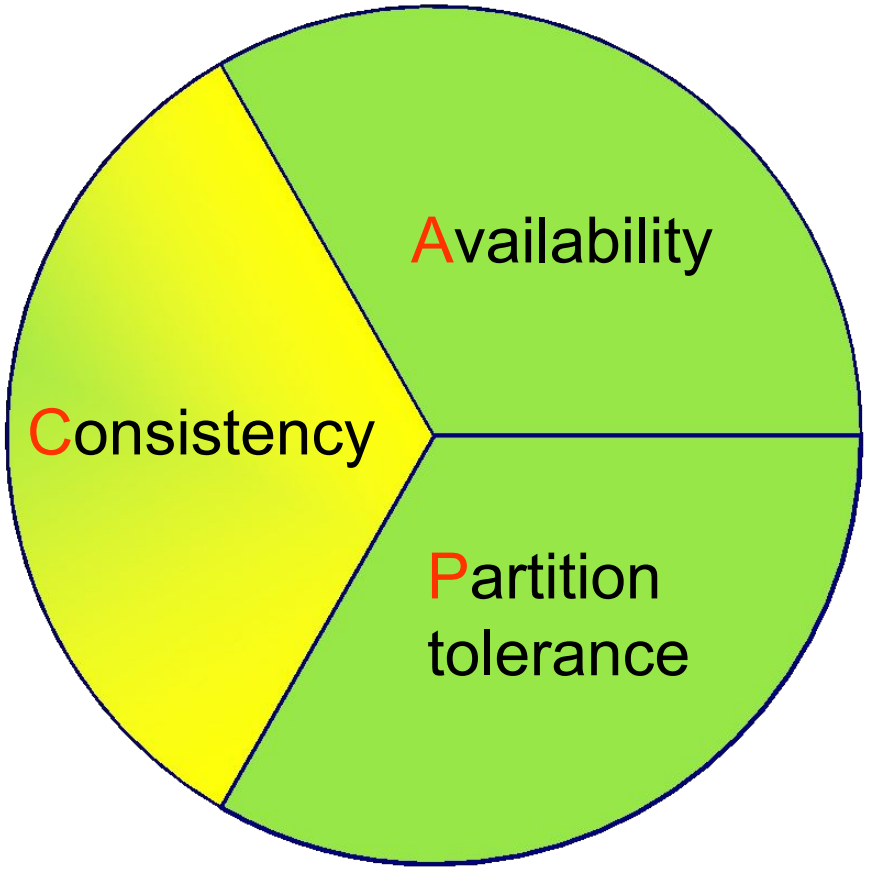
CAP Theorem

Brewer's CAP Theorem:

- For any distributed cluster, it is "impossible" to guarantee simultaneously all of these three properties
- You can have at most two of these three properties for any distributed shared data system
- Large Cluster will "partition" at some point due to network failures :

  If database is designed to provide partition tolerance, then Database by default will either provide C(Consistency) or A(Availability) but not the both .

CAP Theorem?



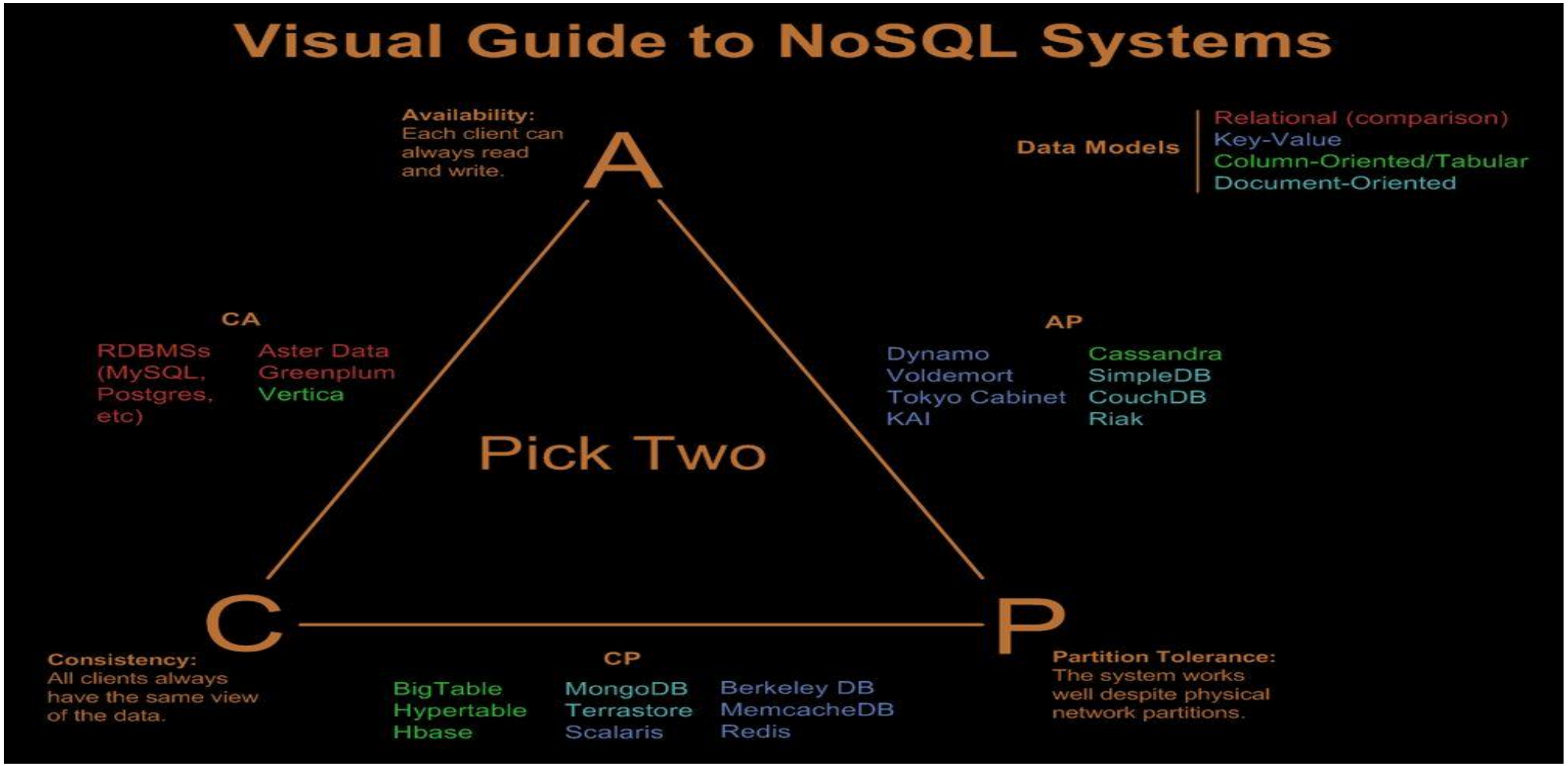
Consistency

All client always have the same view of the data

2 types of consistency:

1. Strong consistency – ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability)
2. Weak consistency – BASE (**B**asically **A**vailable **S**oft-state **E**ventual consistency)

CAP



Key Value Databases

- Key-value stores are the simplest NoSQL databases.
- They are like a dictionary, stores every item as an attribute name (or "key"), together with its value.
- Examples of key-value stores are Riak and Voldemort.
- Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.
- Key-value NoSQL databases are ideal for database for lookup queries with extremely quick and optimized retrieval

Document Databases

- {Name:“Michael", Address:“FlatNo 112,Waterfront Aparment ,NYK,USA”,Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3", "Kirsten: "1", "Otis: "3", Richard: "1”} Phones: [“123-456-7890”, “234-567-8963”] }

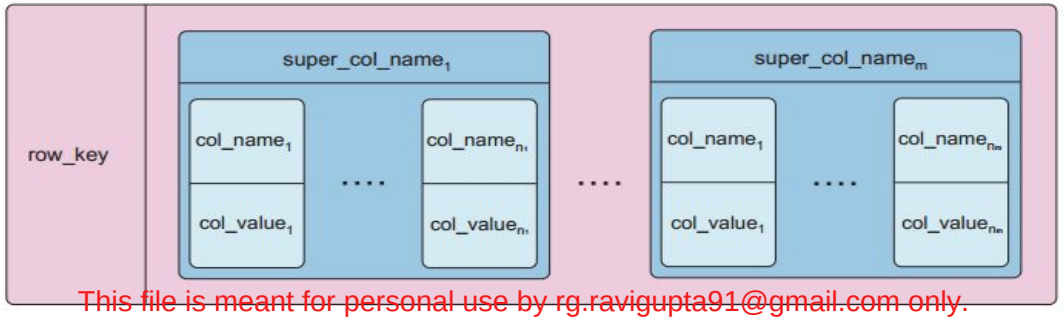
- Example: MongoDb,CouchBase etc

Columnar Databases

- Based on Google's BigTable paper, they are like column oriented relational databases (store data in column order)
- Tables similarly to RDBMS, but handle semi-structured

Data model:

- Collection of Column Families
- Column family = (key, value) where value = set of **related** columns (standard, super)
- indexed by *row key*, *column key* and *timestamp*



Columnar Databases

- One column family can have variable numbers of columns
- Cells within a column family are sorted “physically”
- Very sparse, most cells have null values

Comparison: RDBMS vs column-based NOSQL

Query on multiple tables

RDBMS: must fetch data from several places on disk and glue together

Column-based NOSQL: only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation □ data locality)

Example : Hbase, Cassandra, HyperTable etc

Graph Databases

- Focus on modeling the structure of data (interconnectivity)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))

Data model:

(Property Graph) nodes and edges

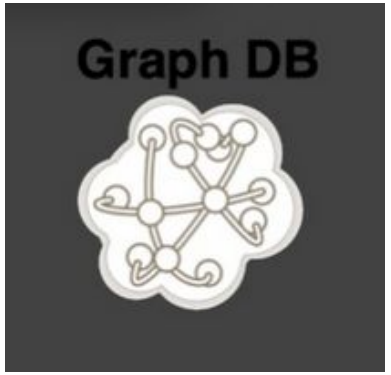
Nodes may have properties (including ID)

Edges may have labels or roles

- Single-step vs path expressions vs full recursion

Example:

Neo4j, FlockDB, Pregel, InfoGrid …



What is MongoDB?

- MongoDB is a document-oriented database
- MongoDB replaces the concept of a "row" with a more flexible model, the "document."
- MongoDB stores data in the form of BSON(binary form of JSON)
- Document-oriented approach allows to store complex hierarchical relationships with a single record in the form of nested JSON
- This approach suits application developers of modern object-oriented languages as Java, Java Script based framework(AngularJS, RectJS), C++ etc
- MongoDB is also schema-free: it is not necessary to define collection attributes before writing data into it.
- This provide developers a lot of flexibility to handle evolving data models.

MongoDB Usecases

- Personalization
- Mobile
- Internet of things
- Real time Analytics
- Web Applications
- Content Management
- Catalog
- Single View

What is JSON?

- JSON : JSON (JavaScript Object Notation) is a lightweight data-interchange format.It is easy for humans to read and write.

- JSON is a text format and language independent, It also uses object concept that are familiar to programmers of different object oriented languages such as Java,C#,C++ etc
- JSON supports all the basic data types you'd expect: numbers, strings, and boolean values, as well as arrays and hashes

- Document databases such as MongoDB use JSON documents in order to store records, just as tables and rows store records in a relational database

- A JSON database returns query results that can be easily parsed, with little or no transformation, directly by JavaScript and most popular programming languages – reducing the amount of logic you need to build into your application layer

Example of JSON

```
{
    "_id" : 1,
    "name" : { “GreatLearning"},
    "customers" : [ “Genpact", “Accenture", “Wipro", “Infosys" ],
    “courses" : [
            {       "name"   : “Data Science with SAS",
               "domain" : “Statistics and Analytics "
            },
            { "name"   : “Big Data Specialization",
              "domain" : “Hadoop eco-system analytics"
            }
    ]
}
```

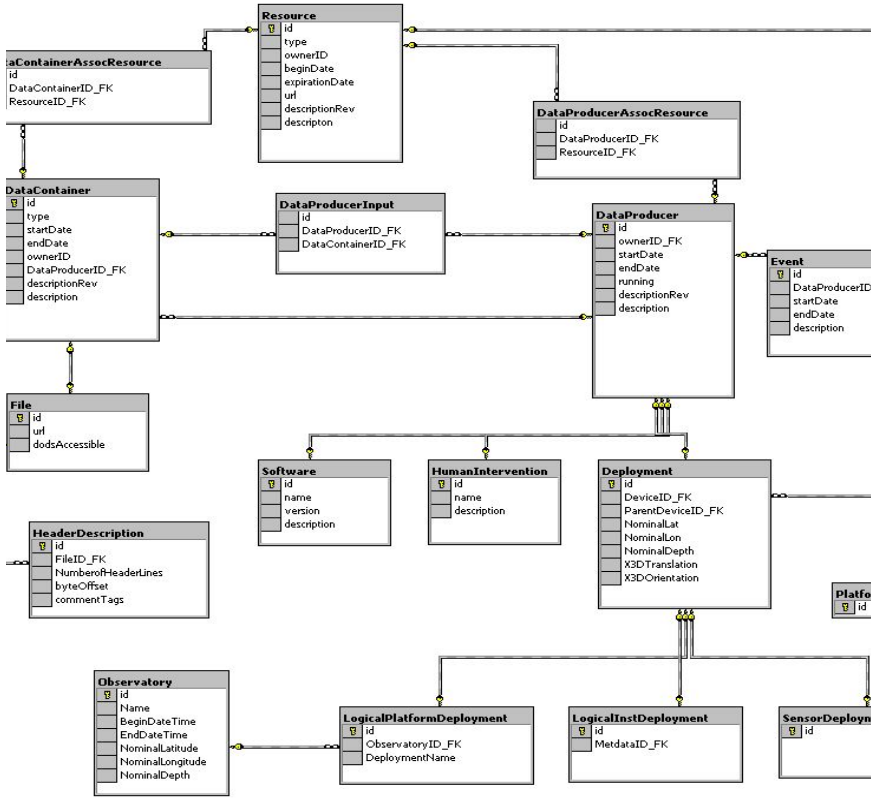
Why MongoDB stores data as BSON

- MongoDB stores data or document in binary-encoded format called BSON.
- BSON is a binary-encoded serialization of JSON-like documents.
- BSON supports the embedding of documents and arrays within other documents and arrays
- BSON also supports additional data types that are not part of the JSON spec such as BinData and Date data type
- BSON is designed to be lightweight , traversable and efficient

MongoDB Structure

RDBMS		MongoDB
Database		Database
Table, View		Collection
Row or Record		Document (JSON, BSON)
Column		Field
Index		Index
Join		Embedded Document
Foreign Key		Reference
Partition		Shard

MongoDB is easy to use



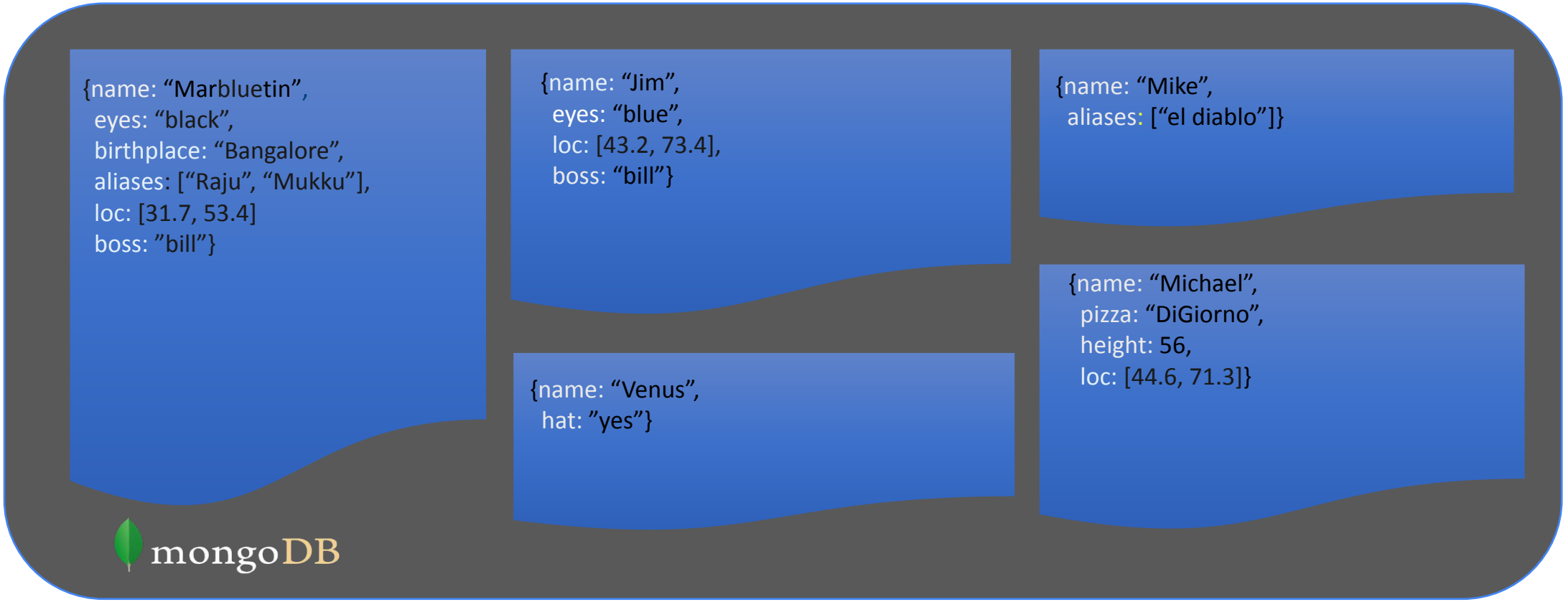
RDBMS ER Diagram

```
{
  title: 'MongoDB',
  contributors: [
    { name: 'Mukesh Kumar',
      email:
'mukesh@greatlearning.com' },
    { name: 'Laxman L',
      email: 'laxman@gmail.com' }
  ],
  model: {
    relational: false,
    awesome: true
  }
}
```

MongoDB's nested model

Schema Free

- MongoDB does not need any pre-defined data schema
- Every document could have different data!



How to scale database?

- Applications data are growing at an incredible pace
- Due to Rapid pace of digitalization ,Advances in sensor technology, increases in available bandwidth, and the popularity of smart devices
- where even small scale applications need to store more data than many databases were meant to handle
- A terabyte of data, once considered huge data, is now commonplace
- Application developers face a difficult decision: how should they scale their databases to make their application scalable

MongoDB is designed for scale out

- MongoDB is designed to scale out from the beginning.
- Using Sharding, it can split up data across multiple servers
- Sharding in MongoDB, can balance data and load across a cluster, redistributing documents automatically
- This helps developers to focus on programming the application, not scaling it
- When they need more capacity, they can just add new shards to the cluster

MongoDB Features

- Ad hoc queries
- Secondary Indexes
- Replication
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Aggregation
- Capped Collection

Collection and Database

- A collection is a group of documents.
- Collections are schema-free, documents within a single collection can have different attributes or fields
- However, Keeping different kinds of documents within the same collection can be a nightmare for developers and admins.
- Grouping similar documents together in the same collection is recommended for application development
- MongoDB groups collections into databases. A MongoDB database or cluster can host several databases, each of which can be thought of as completely independent.
- A good rule of thumb is to store all data for a single application in the same database

Insert Operation

- When we perform an insert, the application driver in the application code converts the data structure into BSON then sends to the database

- database accepts BSON and checks for an "_id" key and validate that the document's size does not exceed 16MB

- All the data validation will be performed by application Drivers at the client side

Retrieval Operation

- db.collection.find() method retrieves documents from a collection & it returns a cursor to the retrieved documents.
- The method takes both the query criteria and projections and returns a cursor to the matching documents.
- We can change the query to impose limits, skips, and sort orders
- The order of documents returned by a query is random unless we specify a sort().
- db.items.find({available: true },{item:1,}).limit(5)

Retrieval Operation

- findOne() : Find the first record in the document

- pretty() method can be used to display the output in a formatted manner
- db.collection.find().pretty()

- db.collection.find() or db.collection.find({}) selects all documents in the collection:

- Specify Equality Condition : use the query document { <field>: <value> } to select all documents that contain the <field> with the specified <value>.

Query Operators

- Specify Conditions Using Query Operators: A query document can use the query operators to specify conditions in a MongoDB query.

- If you have more than one possible value to match for a single key, use an array of criteria with "$in".

- db.items.find({available : { $in: [true, false] } })

- Specify AND Conditions: A compound query can specify conditions for multiple fields
- db.items.find({available: true, soldQty: { $lt: 900 } })

OR, AND Operators

- Specify OR Conditions
- Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction
- db.items.find({ $or: [{soldQty : { $gt: 500 } }, { available:true }] }).

- Specify AND as well as OR Conditions
- db.items.find({ available:true,$or: [{soldQty : { $gt: 200 } }, {item: “Book” }]})

- "$not" is a metaconditional: it can be applied on top of any other criteria.
- > db.items.find({“_id”:{"$not":{"$mod":[4,1]}}})

Regular expression

- Regular Expressions: Regular expressions are useful for flexible string matching

- For example, if we want to find all items whose whose value starts with “Pe” i.e Pen and Pencil
- db.items.find({item:/pe/i})

- If we want to match not only various capitalizations of Pen, but also for Pencil
- db.items.find({item: /Pen?/i})

Limit , Skip and Sorting

- To set a limit, chain the limit function onto your call to find

  db.c.find().limit(3)

- If we want to skip the first three matching documents and return the rest of the matches

  db.c.find().skip(3)

- Sort takes an object: a set of key/value pairs where the keys are key names and the values are the sort directions.
- Sort direction can be 1 (ascending) or -1 (descending).

  db.c.find().sort({username : 1, age : -1})

Update

- db.collection.update() method modifies existing documents in a collection
- db.collection.update() method can accept query criteria to determine which documents to update
- db.collection.update() method either updates specific fields in the existing document or replaces the document
- By default, the db.collection.update() method updates a single document
- To change a field value, MongoDB provides update operators, such as $set to modify values.
- The following updates the model field within the embedded details document.
- db.inventory.update({ item: "ABC1" },{ $set: { "details.model": "14Q2" }},{multi:true})

Upsert

- An upsert is a special type of update. If no document is found that matches the update criteria, a new document will be created by combining the criteria and update documents.

- db.items.update({“item" : “Bag"}, {"$inc" : {“soldQty" : 1}}, {upsert:true})
- Updates, by default, update only the first document found that matches the criteria
- To modify all of the documents matching the criteria, we need to pass true as the fourth parameter to update.
- db.users.update({item: "10/22/1978"},{$set : {gift : "Happy Birthday!"}}, false, true)

Remove Operation

- db.courses.remove() removes all the documents from the collection
- This doesn't remove the collection and any indexes created on it.
- The remove function optionally takes a query document as a parameter
- db.items.remove({"item" : "Bag")
- By default, db.collection.remove() method removes all documents that match its query
- However, the method can accept a flag to limit the delete operation to a single document
- db.items.remove({" item" : "Bag},1)
- db.courses.drop() will drop whole collection and indexes created on it.

Schema Design

- By-default, MongoDB's do not enforce any kind of structure on the documents of the collection
- This flexibility helps the mapping of documents to an entity or an object.
- Different document in the collection can have different keys and types
- In general, however, the documents in a collection share a similar structure.
- We should data balance the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns while doing data modelling.
- The application usage pattern of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself matters most in designing data model.

Reference data model

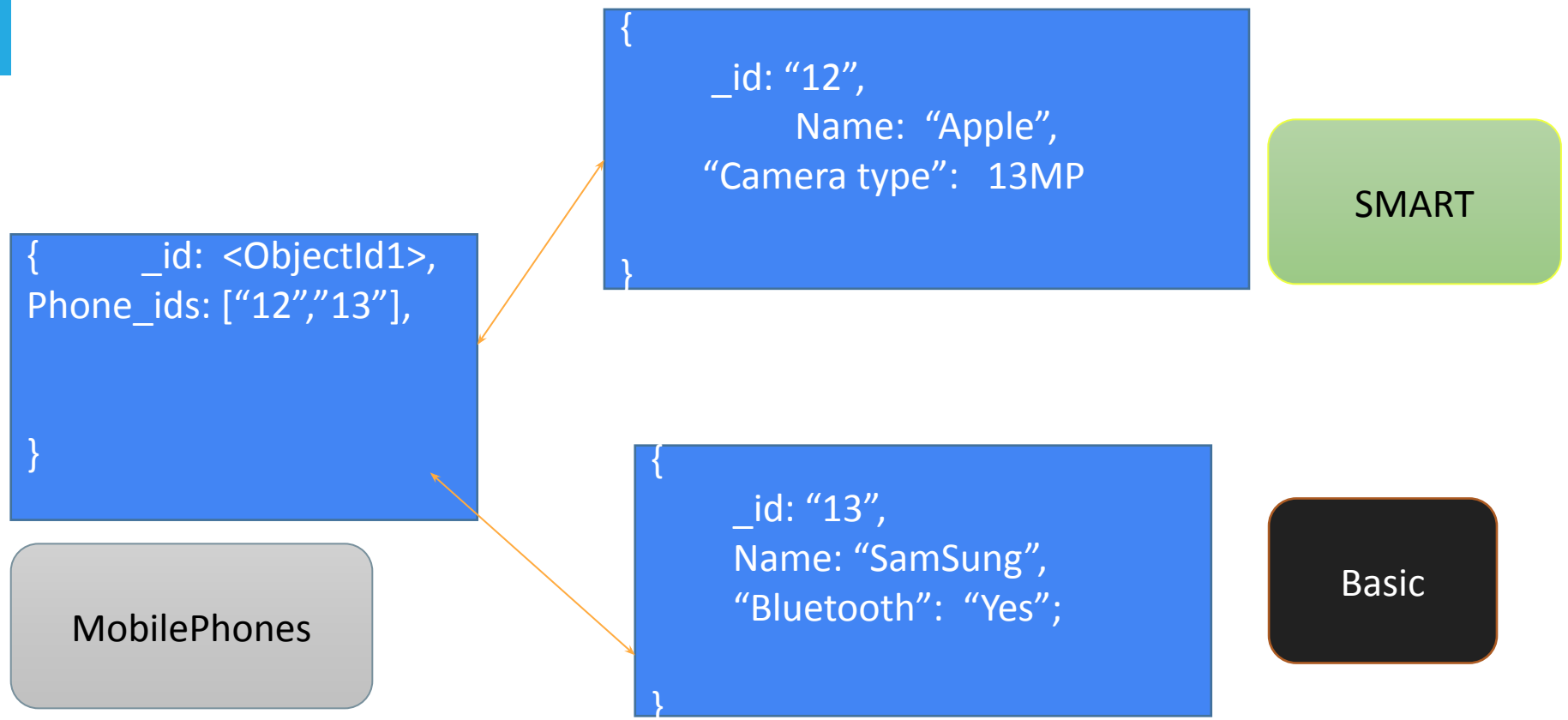
- The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data

- There are two tools that allow applications to represent these relationships: references and 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.

Reference data model example



Embedded data model

- 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
- Grouping documents of the same kind together in the same collection allows for data locality
- In general, use embedded data models when one-to-one, 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
- In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation.
- Embedded data models make it possible to update related data in a single

Embedded data model

```
{
     productdetail: [
    {
        _id: 12,
        Name:  “Apple”,
        Price: “45,000”

    },
     {
        _id: 13,
        Name: “SamSung”,
        Price: “40,000”
    }
]
}
```

Data Types

- Null : Null can be used to represent both a null value and a nonexistent field:{"x”: null}

- Undefined : Undefined can be used in documents as well (JavaScript has distinct types for null and undefined):
- {"x" : undefined}

- Boolean: There is a boolean type, which will be used for the values 'true' and 'false’: {"x" : true}

- 32-bit integer : This cannot be represented on the shell.
- 64-bit integer : Again, the shell cannot represent these.

Data Types

- maximum value: BSON contains a special type representing the largest possible value.
- minimum value : BSON contains a special type representing the smallest possible value.
- ObjectId : These values consists of 12-bytes
- String: BSON strings are UTF-8.
- Symbol: This type is not supported in the Mongo shell
- Timestamps: BSON has a special timestamp type for internal MongoDB use and is not associated with the regular Date type
- Date : BSON Date is a 64-bit integer that represents as the Unix epoch (Jan 1, 1970)
- regular expression: Documents can contain regular expressions as value of the field using JavaScript's regular expression syntax: {"x" : /learn/i}

Data Types

- Code : Documents can also contain JavaScript code as field value:

  {“y" : function() { /* java script code ... */ }}

- Binary data: Binary data is a string of arbitrary bytes.
- Array: Sets or lists can be represented as arrays:
- {“courses" : [“Big Data Specialist", “Data Science with R", “HR Analytics"]}
- embedded document
- Documents can contain nested documents, embedded as values in a parent or outer document: {"course_duration" : {" Big Data Specialist " : “72 Hrs"}}

PES UNIVERSITY

Mongo name has come “**humongous**”

Secondary Indexes

- Secondary indexes in MongoDB are implemented as B-trees.
- B-tree indexes are optimized for a variety of queries, including range scans and queries with sort clauses
- By permitting multiple secondary indexes, MongoDB allows users to optimize for a wide variety of queries
- With MongoDB, you can create up to 64 indexes per collection
- ascending, descending, unique, compound-key, and even geospatial indexes are supported

Different Indexes Types

- MongoDB provide support for different kinds of ndexes to support specific types of data and queries.

- Default _id
- Single Field
- Compound Index
- Multikey Index
- Geospatial Index
- Text Indexes
- Hashed Indexes

Single field Index

- MongoDB provides complete support for indexes on any field in a collection of documents
- By default, all collections have an index on the _id field
- Applications and users may add additional indexes to support important queries and operations.
- The following command creates an index on the name field for the users collection

db.users.createIndex({ “name" : 1 })

Compound Indexes

- MongoDB supports creating compound indexes by combing multiple fields
- Compound indexes can support queries that match on multiple fields.
- db.productsInfo.createIndex({ "itemName": 1, “noOfItems": 1 })
- The order of the fields in a compound index is very important.
- Documents sorted first by the values of the first field and, within each value of the first field, sorted by values of the second field
- MongoDB imposes a limit of 31 fields for any compound index

Multikey Indexes

- Multikey index is supported on a field that holds an array value
- MongoDB creates an index key for each element in the array
- Multikey indexes helps running efficient queries against array fields

- Multikey indexes can be generated over arrays that hold both scalar values (e.g. strings, numbers) and nested documents.
- db.collectionName.createIndex({ <field>: < 1 or -1 > })

- if any indexed field is an array then MongoDB automatically creates a multikey index

Hashed Indexes

- Hashed indexes stores hash values of the indexed field.
- Hash Index can’t be created on multi-key (i.e. arrays) field
- The hashing function flattens embedded documents and generate the hash value
- MongoDB can use the hashed index to support equality queries

- Hashed indexes do not support range queries

- Hashed indexes support sharding a collection using a hashed shard key
- Using a hashed shard key to shard a collection ensures a more even distribution of data.
- db.items.createIndex({ item: "hashed" })

TTL(Time to Live) Indexes

- TTL indexes are special single-field indexes to automatically remove documents from a collection after a certain amount of time.
- It is useful for certain types of data such as application and server logs, and session information that only need to persist in a database for a finite amount of time.

- Use the db.collection.createIndex() method with the expireAfterSeconds option on a field whose value is either a date or an array that contains date values

- db.eventlog.createIndex({ "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 })
- TTL indexes are a single-field indexes. Compound indexes do not support TTL

Unique Indexes

- Using unique index, MongoDB rejects all documents that contain a duplicate value for the indexed field
- db.items.createIndex({ “item": 1 }, { unique: true })

- If you use the unique constraint on a compound index, then MongoDB will enforce uniqueness on the combination of values rather than the individual value for any or all values of the key.
- It stores a null value for the document, If a document does not have a value for the indexed field in a unique index
- Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field
- we can also combine the unique constraint with the sparse index to filter these null values from the unique index and avoid the error

Text Indexes

- A MongoDB provides text indexes to support text search of string content in documents of a collection
- Text indexes can be created on any field whose value is a string or an array of string elements
- To perform queries that access the text index, use the $text query operator.
- To create a text index on a field “customer_name” that contains a string or an array of string elements of the customer_info collection:
- db.customer_info.createIndex({”customer_name”: “text”})

- To perform the text search

  db.customer_info.find({$text:{$search:”John”}})

- A collection can have at most one text index.

Multi Languages Text Search

Supported Languages and Stop Words

- Mongo DB supports text search for various languages and drop language-specific stop words
- For English language it uses simple language-specific suffix stemming and drop stop words like “the”, “an”, “a”, “and”, etc.)
- db.quotes.createIndex({ content : "text" }, default_language: "spanish" })

- Text Search Languages: Danish,dutch,English,finish, French, German,Italian, Norwegian, Portuguese, Romanian, Russian, Spanish. Swedish,Turkish, Hungarian, Portuguese

Geospatial Indexes

- MongoDB provides a special type of index for coordinate plane queries, called a geospatial index
- db.collection.createIndex({ <location field> : "2dsphere" })

Geospatial Query Operators

- Inclusion: MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the $geoWithin operator.
- Intersection : MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the $geoIntersects operator.
- Proximity MongoDB can query for the points nearest to another point. Proximity queries use the $near operator.

Geospatial $geoWithin Query

- The $geoWithin operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

db.<collection>.find({ <location field> : { $geoWithin :{ $geometry :{ type : "Polygon" ,coordinates : [<coordinates>]} } } })

- The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

db.places.find({ loc :{ $geoWithin :{ $geometry :{ type : "Polygon" ,coordinates : [[[0 , 0] ,[3 , 6] ,[6 , 1] ,[0 , 0]]]} } } })

Geospatial Proximity Query

- Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a 2dsphere index
- To query for proximity to a GeoJSON point, use either the $near operator or geoNear command. Distance is in meters.

  db.<collection>.find({ <location field> :{ $near :{ $geometry :{ type : "Point" ,coordinates : [<longitude> , <latitude>] } ,$maxDistance : <distance in meters>} } })

- The geoNear command uses the following syntax:

  db.runCommand({ geoNear : <collection> ,near : { type : "Point" ,coordinates: [<longitude>, <latitude>] } ,spherical : true })

- To select all grid coordinates in a “spherical cap” on a sphere, use $geoWithin with the $centerSphere operator.
- db.<collection>.find({ <location field> :{ $geoWithin :{ $centerSphere :[[<x>, <y>] , <radius>] }} }

Sparse Indexes

- Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value.
- It ignores and skips over those document which don't have the indexed field.
- The index is called as "sparse" because it does not include all documents of a collection.
- db.sales.createIndex({ "sales_id": 1 }, { sparse: true })

- 2d (geospatial) and text indexes are sparse by Default

- Sparse and unique Properties: An index that has both sparse and unique properties defined,prevents collection from having documents with duplicate values for a field but allows multiple documents that omit the key

Index Creation

- By default, creating an index blocks all other operations by holding X exclusive lock on that collection
- The Collection that holds the collection is unavailable for read or write operations until the index build completes.
- For potentially long running index building operations, consider the creating the indexes in the background
- db.people.createIndex({ zipcode: 1}, {background: true})

- By default, background is false for building MongoDB indexes.
- You can combine the background option with other options, as in the following:
- db.people.createIndex({ zipcode: 1}, {background: true, sparse: true })

Removing an Index

- To remove an index from a collection use the dropIndex() method

  db.accounts.dropIndex({ "tax-id": 1 })

- You can also use the db.collection.dropIndexes() to remove all indexes, except for the _id index

  db.items.dropIndexes()

References:

https://docs.mongodb.com/manual/tutorial/

Thank You