NoSQL databases - MongoDB

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>.
- db.items.find({available: true })

Note: **** some commands are deprecated***

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

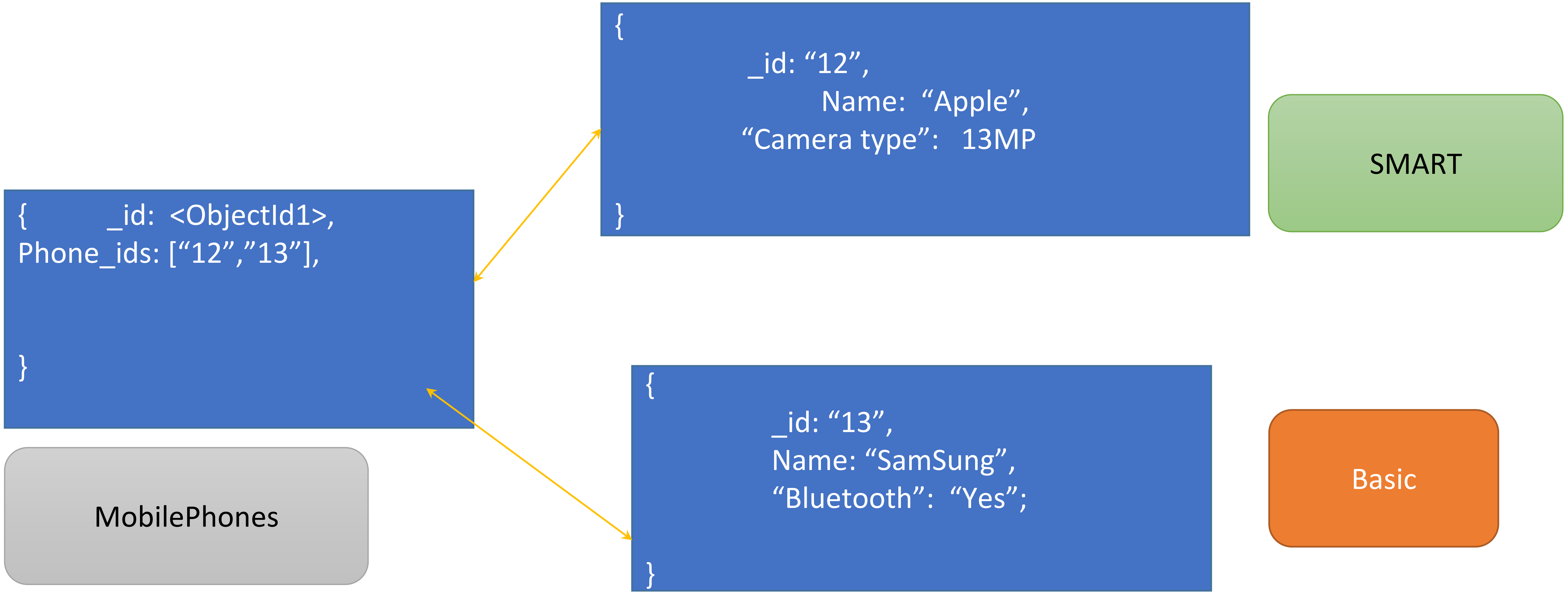
- The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data

- 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



```
{        _id:  <ObjectId1>,
Phone_ids: ["12","13"],


}
```

```
{
        _id: "12",
              Name:  "Apple",
        "Camera type":   13MP

}
```

```
{
        _id: "13",
        Name: "SamSung",
        "Bluetooth":  "Yes";

}
```

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 atomic write operation

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.
- **64-bit floating point number :** All numbers in the shell will be of this type

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"}}

Did you know that

Mongo name has come “**humongous**”

References:

https://www.mongodb.com/docs/manual/core/document/

Thank You