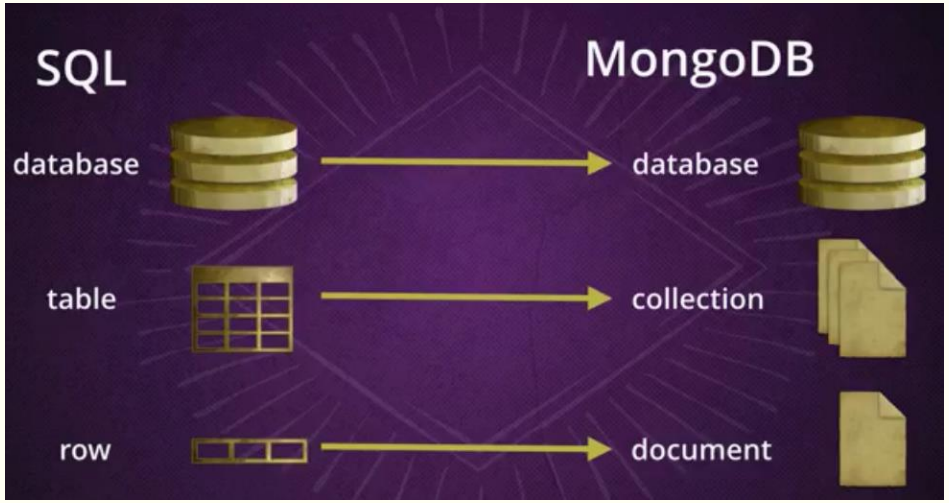
MONGODB

Banuprakash. C
banuprakashc@yahoo.co.in

MongoDB

- Name comes from the word Humongous
- MongoDB is an open-source NoSQL database
  - Databases that generally aren't relational and don't have a query language like SQL
- Document Database
  - MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

```
{
  name: "sue",                          <-- field: value
  age: 26,                              <-- field: value
  status: "A",                          <-- field: value
  groups: [ "news", "sports" ]          <-- field: value
}
```



RDBMS vs MongoDB

Relational

CustomerID	First Name	Last Name	City
0	John	Doe	New York
1	Mark	Smith	San Francisco
2	Jay	Black	Newark
3	Meagan	White	London
4	Edward	Danields	Boston

Order Number	Store ID	Product	Customer ID
10	100	Tablet	0
11	101	Smartphone	0
12	101	Dishwasher	0
13	200	Sofa	1
14	200	Coffee table	1
15	201	Suit	2

MongoDB

```
{    customer_id : 1,
     name : "Mark Smith",
     city : "San Francisco",
     orders: [    {
          order_number : 13,
          store_id : 10,
          date: "2014-01-03",
          products: [
               {SKU: 24578234,
                Qty: 3,
                Unit_price: 350},
               {SKU: 98762345,
                Qty: 1,
                Unit_Price: 110}
               ]
          },
          { <...> }
```

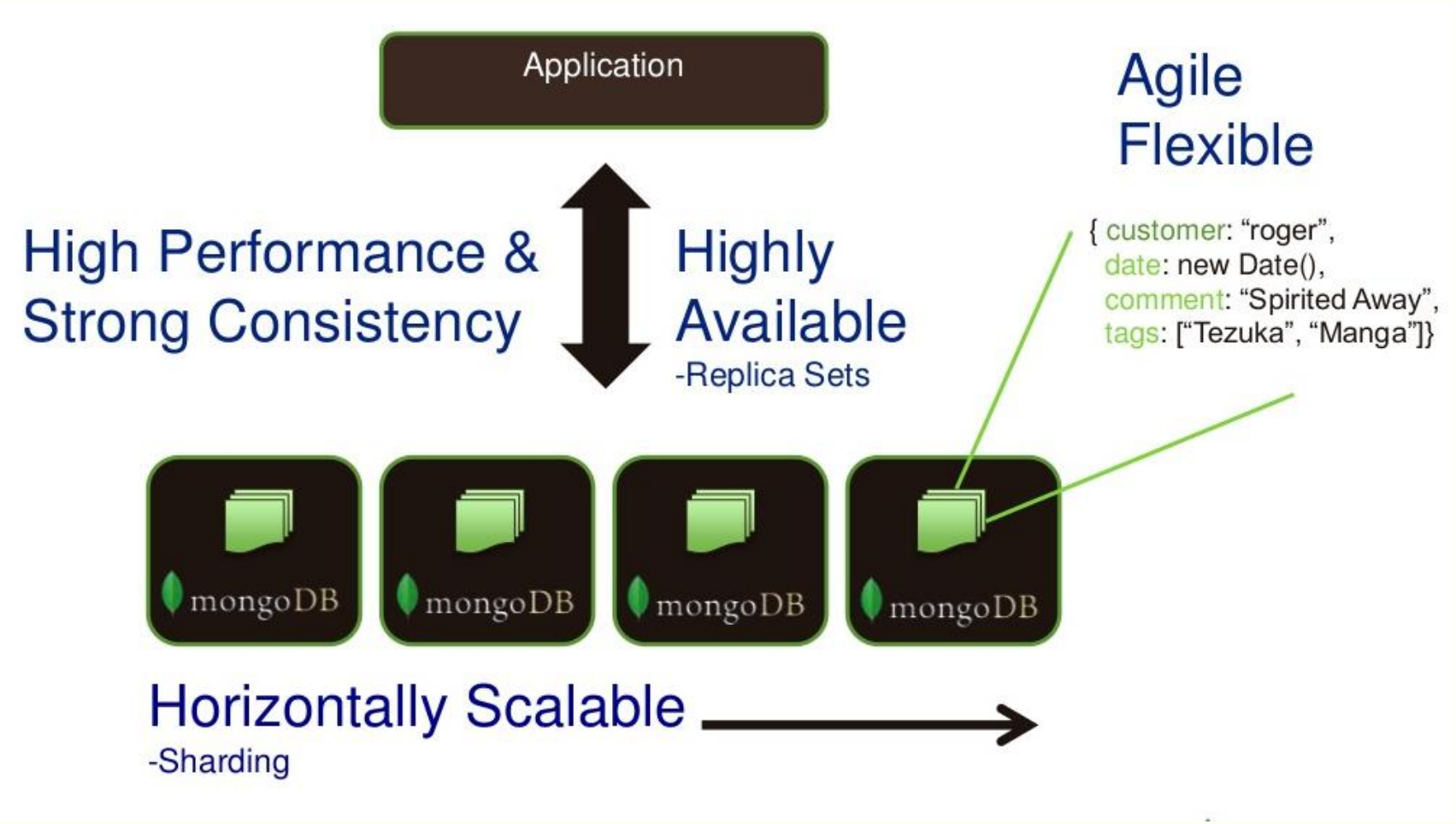
Documents using advantage

- The advantages of using documents are:
  - Documents (i.e. objects) correspond to native data types in many programming languages.
  - Embedded documents and arrays reduce need for expensive joins.
  - Dynamic schema supports fluent polymorphism.

MongoDB Key Features

- High Performance
  - MongoDB provides high performance data persistence. In particular, Support for embedded data models reduces I/O activity on database system.
  - Indexes support faster queries and can include keys from embedded documents and arrays.
- Rich Query Language
  - MongoDB supports a rich query language to support read and write operations as well as:
    - data aggregation
    - Text Search and Geospatial Queries.
- High Availability
  - MongoDB’s replication facility, called replica set, provides:
    - automatic failover and
    - data redundancy.
  - A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

MongoDB Key Features



Databases and Collections

- MongoDB stores BSON documents
- Databases
  - databases hold collections of documents.
  - To select a database to use, in the mongo shell, issue the use <db> statement, as in the following example:
    - use myDB
- Create a Database
  - If a database does not exist, MongoDB creates the database when you first store data for that database. As such, you can switch to a non-existent database and perform the following operation in the mongo shell:
    - use myNewDB
    - db.myNewCollection1.insert({ x: 1 })
    - The insert() operation creates both the database myNewDB and the collection myNewCollection1 if they do not already exist.

Field Names

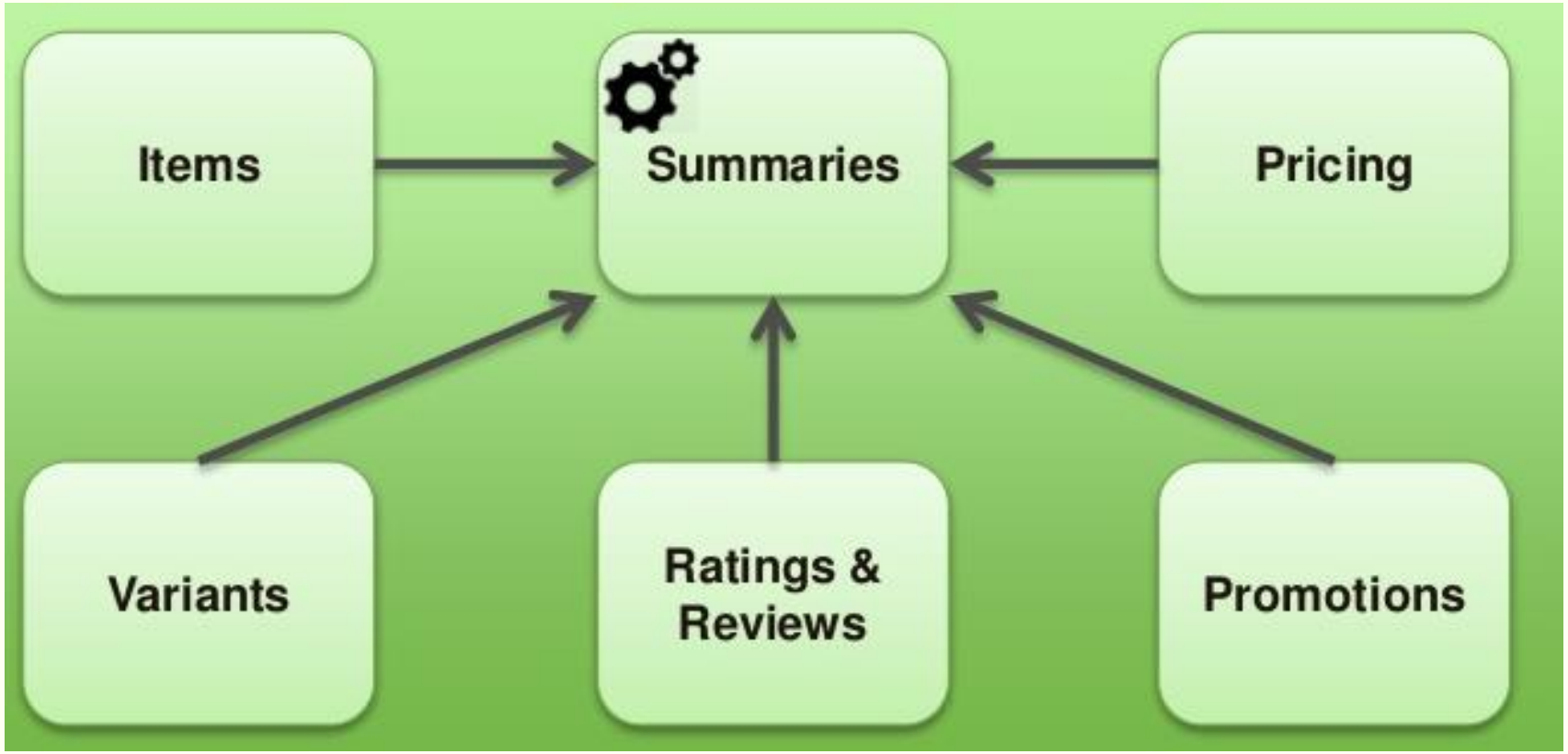
- Field names are strings.
- Documents have the following restrictions on field names:
  - The field name _id is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names cannot start with the dollar sign ($) character.
  - The field names cannot contain the dot (.) character.
  - The field names cannot contain the null character.

Starting the Shell

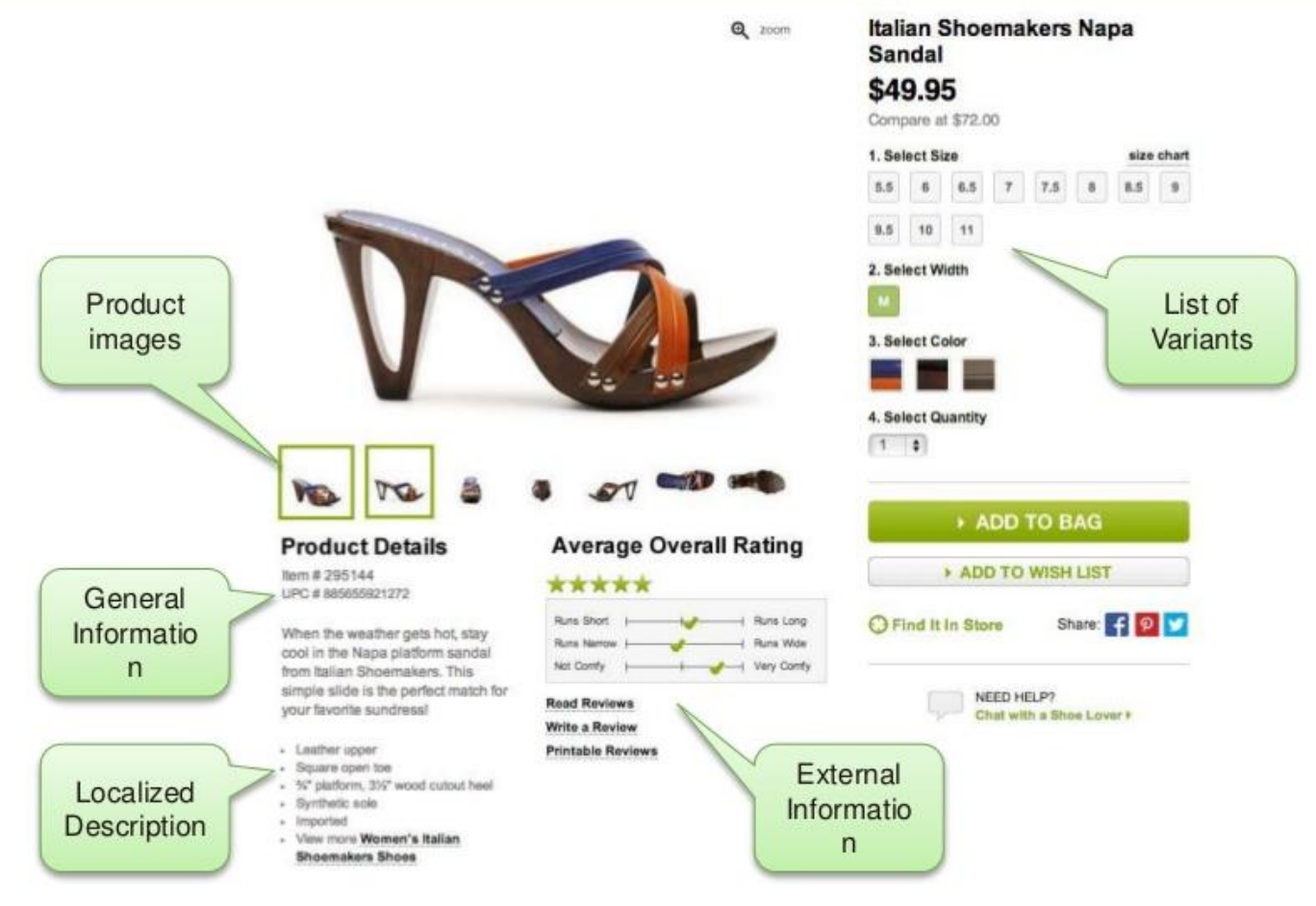
- We can access MongoDB through the terminal application.

Merchandising Architecture

- MongoDB Store



Merchandising Product page



Merchandising Item Model

```
> db.item.findOne()
{   _id: "301671", // main item id
    department: "Shoes",
    category: "Shoes/Women/Pumps",
    brand: "Guess",
    thumbnail: "http://cdn…/pump.jpg",
    image: "http://cdn…/pump1.jpg", // larger version of thumbnail
    title: "Evening Platform Pumps",
    description: "Those evening platform pumps put the perfect
finishing touches on your most glamourous night-on-the-town
outfit",
    shortDescription: "Evening Platform Pumps",
    style: "Designer",
    type: "Platform",
    rating: 4.5, // user rating
    lastUpdated: Date("2014/04/01"), // last update time
    … }
```

Item Data Model

- This type of simple data model allows us to easily query for items based on the most demanded criteria. For example, using db.collection.findOne, which will return a single document that satisfies a query:
- Get item by ID
  - db.definition.findOne({_id:”301671”})
- Get items for a set of product IDs
  - db.definition.findOne({_id:{$in:[”301671”,”452318”]}})
- Get items by category prefix
  - db.definition.findOne({category:/^Shoes\/Women/})
- When performed on properly indexed documents, MongoDB is able to provide high throughput and low latency for these types of queries.

Merchandising Variant Model

- Our item data model above only captures a small amount of the data about each catalog item. So what about all of the available item variations we may need to retrieve, such as size and color?

```
> db.variant.findOne()
{
    _id: "730223104376", // the sku
    itemId: "301671", // references item id
    thumbnail: "http://cdn…/pump-red.jpg", // variant
specific
    image: "http://cdn…/pump-red.jpg",
    size: 6.0,
    color: "Red",
    width: "B",
    heelHeight: 5.0,
    lastUpdated: Date("2014/04/01"), // last update time
    …
}
```

Variant Data Model

- This data model allows us to do fast lookups of specific item variants by their SKU (stock keeping unit) number:
- db.variation.find({_id:"9328484736 2823"})
- As well as all variants for a specific item by querying on the itemId attribute:
- db.variation.find({itemId:"30671"}).sort({_id:1})
- In this way, we maintain fast queries on both our primary item for displaying in our catalog, as well as every variant for when the user requests a more specific product view.

CRUD operations

- Insert Methods
- MongoDB provides the following methods for inserting documents into a collection:
  - db.collection.insertOne()
  - db.collection.insertMany()
  - db.collection.insert()

```
db.users.insertOne (
   {
      name: “Rahul",       age: 19,     status: "P"
   }
)
```

The method returns a document with the status of the operation:

```
{
   "acknowledged" : true,    "insertedId" : ObjectId("5742045ecacf0ba0c3fa82b0")
}
```

CRUD operations

Insert Methods

db.collection.insertMany() inserts multiple documents into a collection.

```
db.users.insertMany(
   [
     { name: "Raj", age: 42, status: "A", },
     { name: "Karthik", age: 22, status: "A", },
     { name: "Swetha", age: 34, status: "D", }
   ]
)
```

CRUD operations

- Insert Methods

```
db.users.insert (
  {
    name: "Rahul",
    age: 19,
    status: "P"
  }
)
```

```
db.users.insert(
   [
     { name: "bob", age: 42, status: "A", },
     { name: "ahn", age: 22, status: "A", },
     { name: "xi", age: 34, status: "D", }
   ]
)
```

The operation returns a WriteResult object with the status of the operation. A successful insert of the document returns the following WriteResult object:

WriteResult({ "nInserted" : 1 })

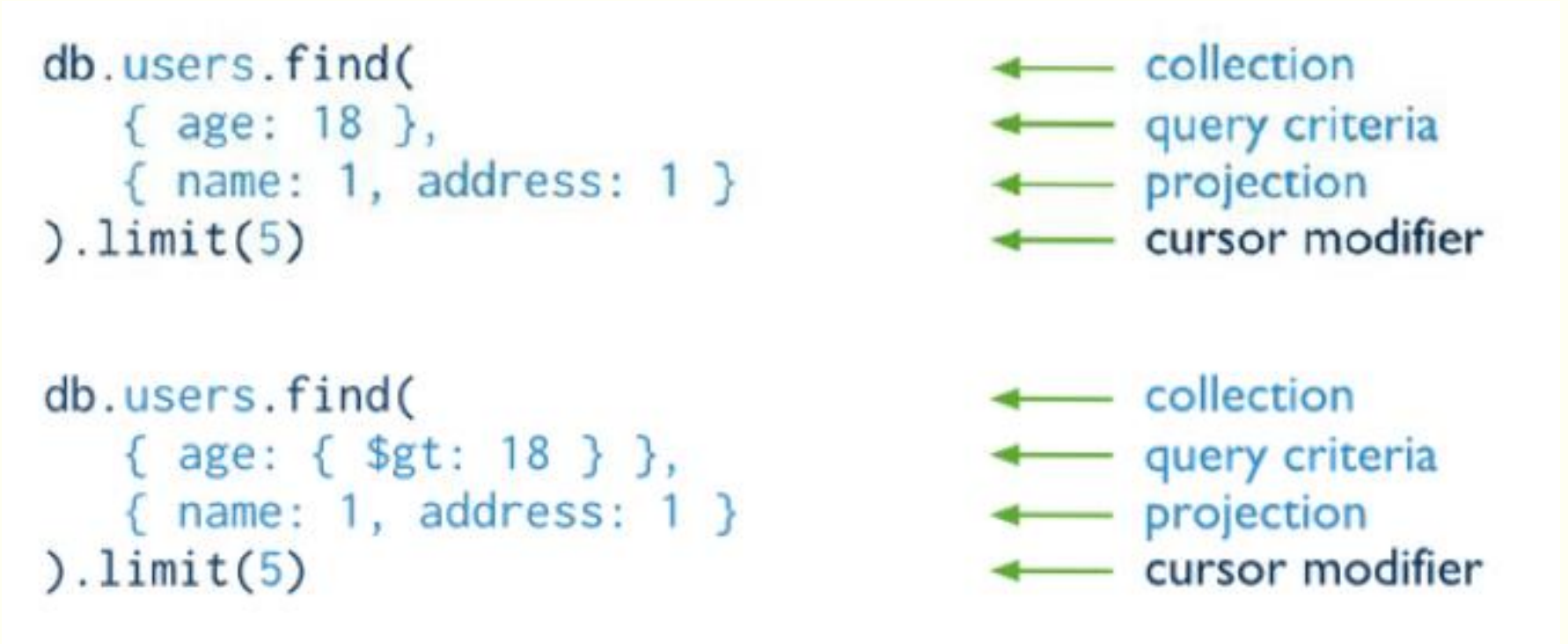
The nInserted field specifies the number of documents inserted. If the operation encounters an error, the WriteResult object will contain the error information.

CRUD operations

- Query Documents
- The db.collection.find() method returns a cursor to the matching documents.
  - db.customers.find();
- Specify Query Filter Conditions
  - db.customers.find({gender:'female'});

CRUD operations

- Query Documents



CRUD operations

- Ordering the query results

- Use the sort() method to achieve it
  - db.customers.find().sort({firstName:1});

Query operators

Name	Description
$gt	Matches values that are greater than the value specified in the query.
$gte	Matches values that are equal to or greater than the value specified in the query.
$in	Matches any of the values that exist in an array specified in the query.
$lt	Matches values that are less than the value specified in the query.
$lte	Matches values that are less than or equal to the value specified in the query.
$ne	Matches all values that are not equal to the value specified in the query.
$nin	Matches values that **do not** exist in an array specified to the query.

More operators

Name	Description
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$exists	Matches documents that have the specified field.
$type	Selects documents if a field is of the specified type.

CRUD operations

- Query Documents
- Specify Conditions Using Query Operators
  - The following example retrieves all documents from the users collection where status equals either "P" or "D":

```
db.users.find( { status: { $in: [ "P", "D" ] } } )
```

- Specify AND Conditions

```
db.users.find( { status: "A", age: { $lt: 30 } } )
```

- Specify OR condition

```
db.users.find(
  {
    $or: [ { status: "A" }, { age: { $lt: 30 } } ]
  }
)
```

Using $exists operator

- Get Customers who have placed orders
  - db.customers.find({"orders":{$exists:1} });
- Get Customers who have not placed orders
  - db.customers.find({"orders":{$exists:0} });
- Get count of customers who have placed orders
  - db.customers.find({"orders":{$exists:1} }).count();
- Get count of customers who have placed more than 1 order
  - db.customers.find({"orders.1":{$exists:1} }).count();

Query

- Projection
  - db.customers.find({},{firstName:1, gender:1});
- Aggregate
  - db.collection.aggregate({GROUP_OPTIONS, HAVING_OPTIONS})

```
> db.sales.aggregate( { $group: { _id: "$category", sales: {$sum: 1} } });
{ "_id" : "Beverages", "sales" : 46 }
{ "_id" : "Dairy Products", "sales" : 38 }
{ "_id" : "Condiments", "sales" : 39 }
{ "_id" : "Seafood", "sales" : 45 }
{ "_id" : "Confections", "sales" : 48 }
{ "_id" : "Grains/Cereals", "sales" : 28 }
{ "_id" : "Produce", "sales" : 19 }
{ "_id" : "Meat/Poultry", "sales" : 23 }
```

Query

```
> db.sales.aggregate(
        {
                $group: {  _id: "$category",
                     salesCount: {$sum: 1},
                    salesTotal: {$sum: "$sales"} }
        },
        { $match: { salesCount: { $gte: 40}}} );

{ "_id" : "Beverages", "salesCount" : 46, "salesTotal" : 102074.29000000001 }
{ "_id" : "Seafood", "salesCount" : 45, "salesTotal" : 65544.18999999999 }
{ "_id" : "Confections", "salesCount" : 48, "salesTotal" : 80894.11000000002 }
```

Query [$group operator: $avg, $first, $last, $max, $min, $push, $sum]

```
> db.sales.aggregate( { $group: { _id: "$category", sales: {$max: "$sales"} } });
{ "_id" : "Beverages", "sales" : 25127.36 }
{ "_id" : "Dairy Products", "sales" : 11959.75 }
{ "_id" : "Condiments", "sales" : 3857.41 }
{ "_id" : "Seafood", "sales" : 7100 }
{ "_id" : "Confections", "sales" : 6014.6 }
{ "_id" : "Grains/Cereals", "sales" : 9868.6 }
{ "_id" : "Produce", "sales" : 11898.5 }
{ "_id" : "Meat/Poultry", "sales" : 14037.79 }

> db.sales.aggregate( { $group: { _id: "$category", sales: {$min: "$sales"} } });
{ "_id" : "Beverages", "sales" : 42 }
{ "_id" : "Dairy Products", "sales" : 99.5 }
{ "_id" : "Condiments", "sales" : 85.4 }
{ "_id" : "Seafood", "sales" : 60 }
{ "_id" : "Confections", "sales" : 68.85 }
{ "_id" : "Grains/Cereals", "sales" : 87.75 }
{ "_id" : "Produce", "sales" : 128 }
{ "_id" : "Meat/Poultry", "sales" : 490.21 }
```

- Query on Embedded Documents

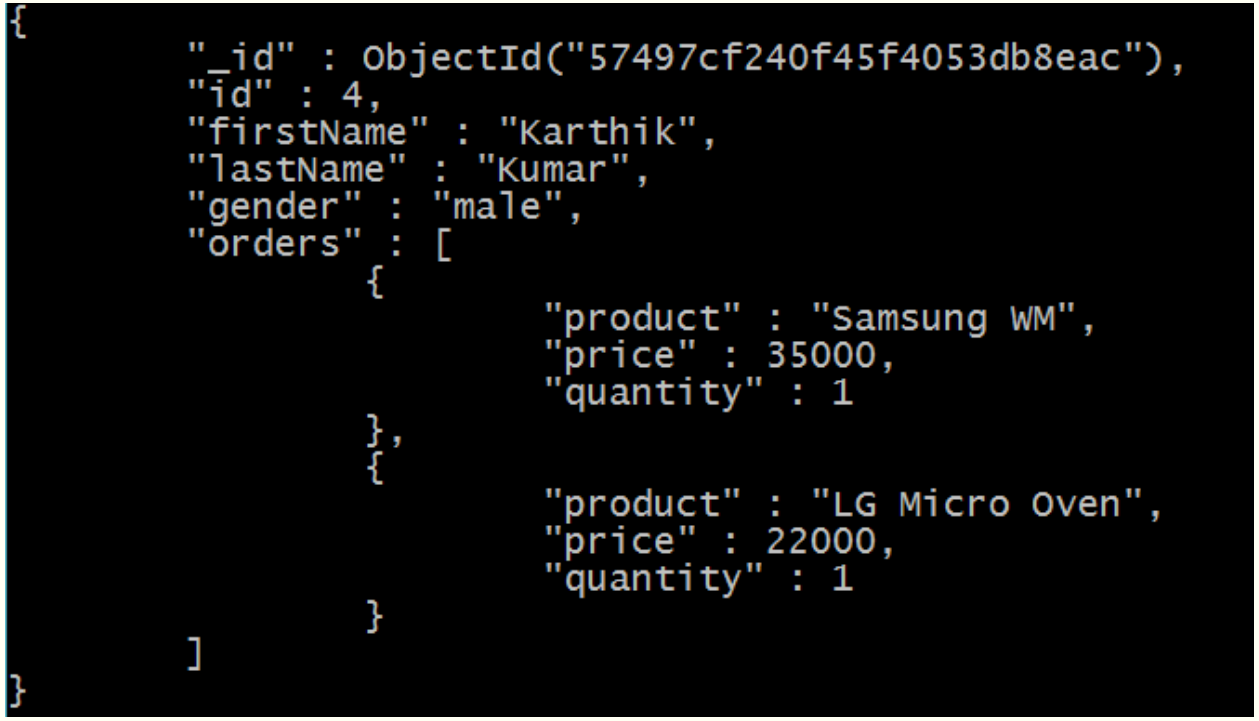
- Exact Match on the Embedded Document

```
db.users.find({
favorites:
{ artist: "Picasso", food: "pizza" }
} )
```

```
{
  _id: 6,
  name: "abc",
  age: 43,
  type: 1,
  status: "A",
  favorites: { food: "pizza", artist: "Picasso" },
  finished: [ 18, 12 ],
  badges: [ "black", "blue" ],
  points: [
     { points: 78, bonus: 8 },
     { points: 57, bonus: 7 }
  ]
}
```

CRUD operations

- Array of Embedded Documents
- db.customers.find({ 'orders.product' :'LG Micro Oven'}).pretty();



Map Reduce

- MongoDB provides map-reduce operations to perform aggregation.
- In general, map-reduce operations have two phases:
  - a *map* stage that processes each document and *emits* one or more objects for each input document, and *reduce* phase that combines the output of the map operation.
- db.collection.mapReduce(mapFunction, reduceFunction, options);

Example

```
> db.customers.mapReduce(
        function() { emit(this.gender,this.firstName);}, /* map */
        function(key,values) { return values.join(); }, /* reduce */
        {out:{inline:true}});
{
        "results" : [
                {
                        "_id" : "female",
                        "value" : "Anitha,Sunita"
                },
                {
                        "_id" : "male",
                        "value" : "Rajesh,Suresh,Karthik"
                }
        ],
        "timeMillis" : 19,
        "counts" : {
                "input" : 5,
                "emit" : 5,
                "reduce" : 2,
                "output" : 2
        },
        "ok" : 1
}
```

Indexes

- Indexes support the efficient execution of queries in MongoDB.
- Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement.
- If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.
- Indexes are special data structures that store a small portion of the collection’s data set in an easy to traverse form.
- The index stores the value of a specific field or set of fields, ordered by the value of the field.

Indexes

- Default _id Index
  - MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index on the _id field.
- Create an Index
  - To create an index, use db.collection.createIndex()
- **Single field index:**
  - db.records.createIndex({ score: 1 })
  - A value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order
  - The created index will support queries that select on the field score, such as the following:
    - db.records.find({ score: 2 })
    - db.records.find({ score: { $gt: 10 } })

Indexes

- Create an Index on an Embedded Field

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

- The following operation creates an index on the location.state field:
  - db.records.createIndex({ "location.state": 1 })
- The created index will support queries that select on the field location.state, such as the following:
  - db.records.find({ "location.state": "CA" })
  - db.records.find({ "location.city": "Albany", "location.state": "NY" })

Indexes

- Create a Compound Index
  - The following operation creates an ascending index on the item and stock fields:
    - db.products.createIndex({ "item": 1, "stock": 1 })
  - The index supports queries on the item field as well as both item and stock fields:
    - db.products.find({ item: "Banana" })
    - db.products.find({ item: "Banana", stock: { gt: 5 } })

Indexes

- Remove Indexes
  - Remove a Specific Index
  - To remove an index, use the db.collection.dropIndex() method.
    - db.records.dropIndex({ score: 1 })

- Remove All Indexes
  - db.collection.dropIndexes() to remove all indexes, except for the _id index from a collection.

Two Phase Commits

- Operations on a single document are always atomic with MongoDB databases;
- Operations that involve multiple documents, which are often referred to as “multi-document transactions”, are not atomic.
- When executing a transaction composed of sequential operations, certain issues arise, such as:
  - Atomicity: if one operation fails, the previous operation within the transaction must “rollback” to the previous state (i.e. the “nothing,” in “all or nothing”).
  - Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

Two Phase Commits

The examples use following two collections:

1. A collection named accounts to store account information.
2. A collection named transactions to store information on the fund transfer transactions.

Initialize collection with A and B Accounts

Below is mongo code for insert document in accounts collection.

```
db.accounts.insert(
   [
     { _id: "A", balance: 1000, pendingTransactions: [] },
     { _id: "B", balance: 1000, pendingTransactions: [] }
   ]
)
```

Two Phase Commits

Initialize Transfer Record Insert records to transaction collection to perform transfer of money.

The document in transaction collection include following fields.

```
db.transactions.insert(
{		_id: 1,
		source: "A",
		destination: "B",
		value: 100,
		state: "initial",
		lastModified: new Date()
})
```

Two Phase Commits

Transfer Funds Between Accounts Using Two-Phase Commit:

1) Retrieve the transaction to start.

```
var t = db.transactions.findOne( { state: "initial" } )
```

```
{ "_id" : 1, "source" : "A", "destination" : "B", "value" : 100, "state" : "initial",
"lastModified" : ISODate("2014-07-11T20:39:26.345Z") }
```

2) Update transaction state to pending.

```
db.transactions.update({ _id: t._id, state: "initial" },
{
        $set: { state: "pending" },
        $currentDate: { lastModified: true }
})
```

Two Phase Commits

3) Apply the transaction to both accounts.

```
db.accounts.update(
    { _id: t.source, pendingTransactions: { $ne: t._id } },
    { $inc: { balance: -t.value }, $push: { pendingTransactions: t._id } })


db.accounts.update(
    { _id: t.destination, pendingTransactions: { $ne: t._id } },
    { $inc: { balance: t.value }, $push: { pendingTransactions: t._id } })
```

Two Phase Commits

4) Update transaction state to applied

```
db.transactions.update( { _id: t._id, state: "pending" },
{
        $set: { state: "applied" },
        $currentDate: { lastModified: true }
})
```

Two Phase Commits

5) Update both accounts list of pending transactions.

```
db.accounts.update(
    { _id: t.source, pendingTransactions: t._id },
    { $pull: { pendingTransactions: t._id } })

db.accounts.update(
    { _id: t.destination, pendingTransactions: t._id },
    { $pull: { pendingTransactions: t._id } })
```

6)Update transaction state to done.

```
db.transactions.update(
    { _id: t._id, state: "applied" },
    {  $set: { state: "done" },  $currentDate: { lastModified: true }
})
```

Two Phase Commits: Recovering from Failure Scenarios

- Transactions in Pending State
  - To recover from failures that occur after step “Update transaction state to pending.” but before “Update transaction state to applied.” step, retrieve from the transactions collection a pending transaction for recovery:
  - var dateThreshold = new Date();
  - dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);
  - var t = db.transactions.findOne({ state: "pending", lastModified: { $lt: dateThreshold } });
  - And resume from step “Apply the transaction to both accounts.“

Two Phase Commits: Recovering from Failure Scenarios

- Transactions in Applied State
  - To recover from failures that occur after step “Update transaction state to applied.” but before “Update transaction state to done.” step, retrieve from the transactions collection an applied transaction for recovery:

  - var dateThreshold = new Date();
  - dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);
  - var t = db.transactions.findOne({ state: "applied", lastModified: { $lt: dateThreshold } });

  - And resume from “Update both accounts’ list of pending transactions.“

Two Phase Commits: Rollback

- Transactions in Applied State
  - After the “Update transaction state to applied.” step, you should not roll back the transaction.
  - Instead, complete that transaction and create a new transaction to reverse the transaction by switching the values in the source and the destination fields.

1. Update transaction state to cancelling.

Update the transaction state from pending to cancelling.

```
db.transactions.update(
  { _id: t._id, state: "pending" },
  {
    $set: { state: "canceling" },
    $currentDate: { lastModified: true }
  } )
```

Two Phase Commits: Rollback

2. Undo the transaction on both accounts.

```
db.accounts.update( { _id: t.destination, pendingTransactions: t._id },
  {      $inc: { balance: -t.value },
         $pull: { pendingTransactions: t._id }
  })
db.accounts.update( { _id: t.source, pendingTransactions: t._id },
  {      $inc: { balance: t.value},
         $pull: { pendingTransactions: t._id }
  })
```

Two Phase Commits: Rollback

3. Update transaction state to canceled

To finish the rollback, update the transaction state from canceling to cancelled.

```
db.transactions.update( { _id: t._id, state: "canceling" },
  {     $set: { state: "cancelled" },
            $currentDate: { lastModified: true }
  })
```