Overview of NoSQL Databases

Contents

1. Setting the Scene for NoSQL

- Essential concepts: Relational databases
- Strengths of relational databases
- Limitations of relational databases
- The role of NoSQL databases
- NoSQL databases in the industry

Essential Concepts: Relational Databases

- According to Wiki:

A **relational database** is a digital database whose organization is based on the relational model of data, as proposed by E. F. Codd in 1970.

The **relational model** organizes data into one or more tables of columns and rows, with a unique key identifying each row.

Relationships are a logical connection between different tables, established on the basis of interaction among these tables.

Virtually all relational database systems use **SQL** as the language for querying and maintaining the database.

Strengths of Relational Databases

- Extremely well proven and widely used in the industry
  - E.g. Oracle, SQL Server, MySQL
- Quality of service guarantees
  - Highly efficient, e.g. via indexes, load balancing, etc.
  - Highly available, e.g. via replication, fail-over, etc.
  - Highly secure
  - Transactional

Limitations of Relational Databases

- Not good at storing unstructured or heterogeneous data
  - This kind of data doesn't fit nicely into the structured world of rectangular tables and fixed relationships
- Not ideal for ingressing big data at high velocity
  - It takes time to break the data down into rectangular chunks, so that it can be inserted into table(s) in an RDBMS
- Not good for rapidly evolving (agile) requirements
  - You can't keep changing the database schema all the time!
- Not ideal for scale-out architectures
  - RDBMS aren't really designed for the cloud / commodity storage

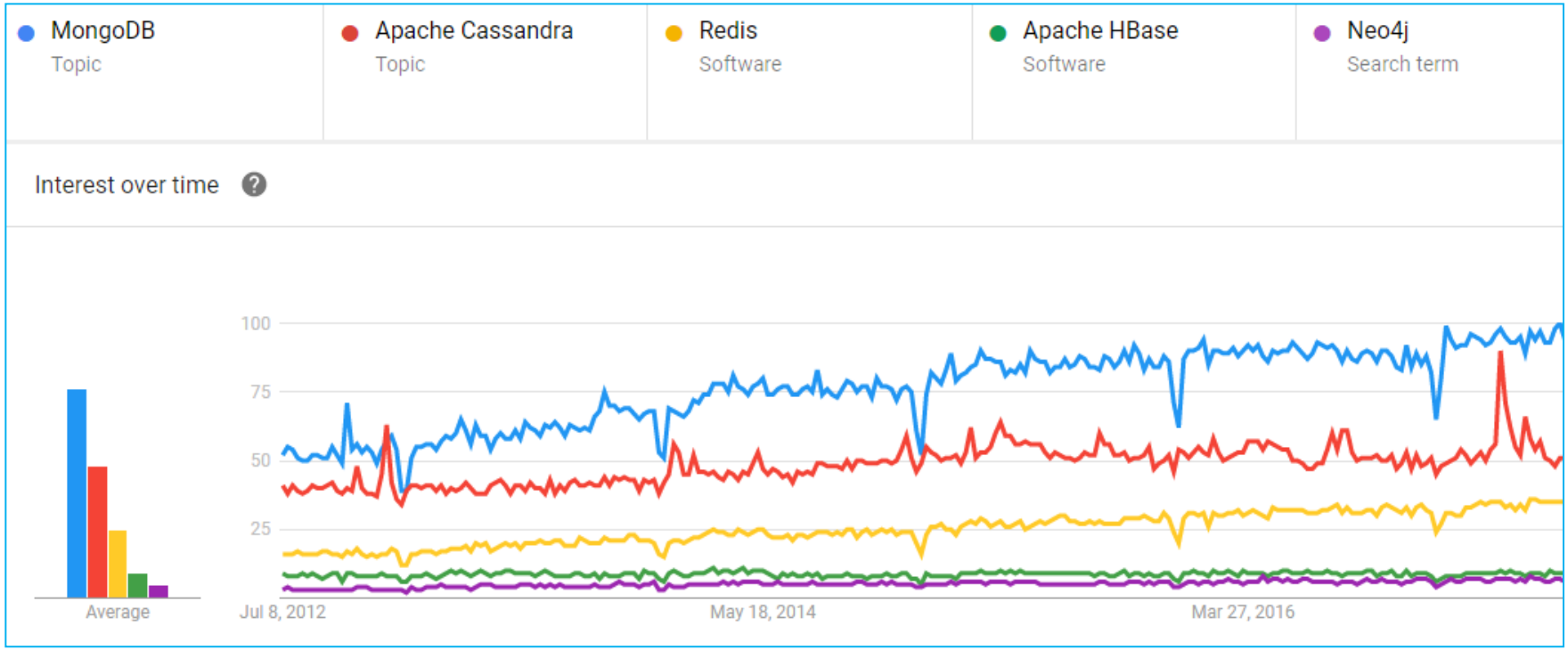
The Role of NoSQL Databases

- NoSQL is a general term to represent non-relational database management systems
  - Encompasses a wide variety of database technologies

- NoSQL databases are designed to address the demands of building contemporary applications
  - Unstructured data
  - Handling big data
  - Data modelling agility
  - Scale-out architecture via auto-sharding, i.e. natively and automatically spread data across any number of servers

NoSQL Databases in the Industry

- MongoDB is the top NoSQL database engine in use today
  - The following chart shows NoSQL usage stats
  - Data is taken from Google Trends, for the period 2012 - 2018



2. Types of NoSQL Databases

- Key-value stores
- Document-oriented databases
- Column-oriented stores
- Graph databases

Key-Value Stores

- Description
  - The simplest type of NoSQL database
  - Each item in the database is stored as a key/value pair
- Accessing data
  - Key lookups, like using a map or dictionary in an OO language
- Examples of key-value stores
  - Redis
  - Oracle NoSQL Database

Document-Oriented Databases

- Description
  - Stores documents of any schema
  - Uses encoding formats such as JSON, BSON, YAML, and XML
  - Each document has a unique key
- Accessing data
  - Access documents by unique keys
  - Proprietary APIs to perform CRUD operations
- Examples of document-oriented databases
  - MongoDB
  - MarkLogic

Column-Oriented Stores

- Description
  - Stores data on disk by columns
  - All cells of a column are stored together on disk, which optimises many big-data manipulation scenarios
- Accessing data
  - Proprietary APIs to perform CRUD operations, e.g. Cassandra has the Cassandra Query Language (CQL)
- Examples of column-oriented stores
  - Apache Cassandra
  - HBase

Graph Databases

- Description
  - Intended for data whose relations are well represented as a graph
  - Common use cases include fraud detection, real-time recommendation engines, network and IT operations, etc.
- Accessing data
  - Graph-bases searches
  - APIs available in various standard programming languages such as Java, C++, Scala, and SPARQL
- Examples of graph databases
  - Neo4j
  - ArangoDB

Any Questions?

Overview of MongoDB

Contents

1. Overview of MongoDB

- What is MongoDB?
- Key features of MongoDB
- Hosting vs. local installation
- MongoDB editions

What is MongoDB?

- MongoDB is an open-source document database
  - In MongoDB, a document is a BSON object ("binary JSON")
  - A document contains fieldname/value pairs
  - Values can be simple types, arrays, or nested documents

- Here's an example of a MongoDB document:

```
{
    name: "Sam",
    age: 21,
    skills: [ "Java", "C++", "JavaScript" ],
    additionalInfo: {
        nationality: "UK",
        companyCar: {
            make: 'Bugatti',
            model: 'Chiron'
        }
    }
}
```

Key Features of MongoDB

- High performance
  - Via indexes
- Rich query language for CRUD operations
  - Data aggregation
  - Text search and geospatial queries
- High availability
  - Via automatic failover and data redundancy
- Horizontal scalability across a cluster
  - Via sharding

Hosting vs. Local Installation

- You can get access to a MongoDB instance in the cloud, install your own in the cloud or install locally

- Hosting MongoDB in the cloud…
  - MongoDB Atlas is a cloud-hosted service
  - Allows you to provision, run, monitor and maintain MongoDB
  - Fast, free, and an easy way to get started with MongoDB
  - For details, see https://www.mongodb.com/cloud

- Installing MongoDB on-premise…
  - Some organizations prefer to install MongoDB on their own servers
  - E.g. for historical or governance reasons
  - Many platforms supported, including Unix, Mac, Windows, etc.
  - For details, see https://docs.mongodb.com/manual/installation/

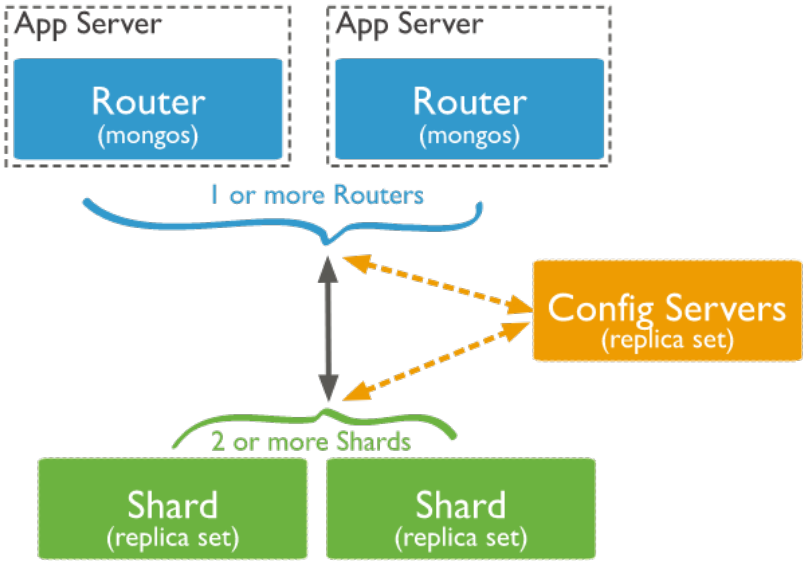
MongoDB Editions

- MongoDB Community Edition
  - Free, standalone NoSQL database engine
  - We'll use this
- MongoDB Enterprise Edition
  - Monthly or annual fee, per server
  - Advanced security features, integration options, production support

Horizontal Vs Vertical Scaling

- One of the core advantages of MongoDB (and many NoSQL databases) is that ***Horizontal Scaling*** is a fundamental assumption
- ***Vertical Scaling*** involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
- ***Horizontal Scaling*** involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required.
- The overall speed or capacity of a single machine may not be high. If each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server.

MongoDB Scaling - Sharding



- shard: Each shard contains a subset of the overall data.
- mongos: The mongos appear to the client as a database, but in fact acts as a query router, ensuring a query goes to the correct shard or shards.
- config servers: Config servers store metadata and configuration settings for the cluster.

MongoDB - Sharding

- Effectively, MongoDB distributes the read and write workload across the shards in a sharded cluster
- Both read **AND** write workloads can be scaled horizontally across the cluster by adding more shards

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

MongoDB – Sharding Vs Replication

- **Sharding** is the MongoDB solution for **Scalability**
- **ReplicaSets** is the MongoDB solution for **Fault Tolerance**
- What happens if the computer that one shard is on has a power supply failure?
- With **ReplicaSets** each shard is automatically duplicated across more than one node. If the node fails, then another node has the same shard data.

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

2. Getting Started with MongoDB

- Downloading and installing MongoDB
- Starting MongoDB from the Command Line
- Starting MongoDB as a Windows Service
- Using the MongoDB interactive shell
- Using MongoDB Compass

Installing MongoDB For Testing

- You can install MongoDB Community Edition for free
  - For details, see the Annex at the end of this chapter
- We've already installed MongoDB Community Edition
  - See C:\Program Files\MongoDB\Server\3.6

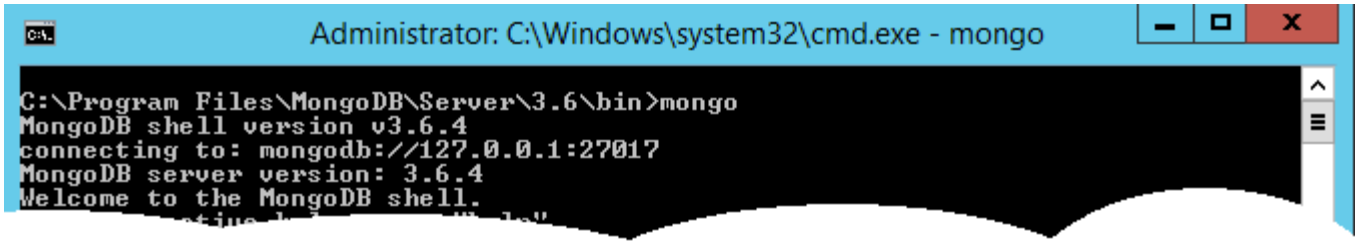
Using the MongoDB Interactive Shell

- You can interact with a running MongoDB instance via the MongoDB interactive shell
  - Enables you to enter simple MongoDB CRUD commands

- To start a MongoDB shell:
  - Open a new Command Prompt window
  - Go to the MongoDB `bin` folder and run the following command

```
mongo
```

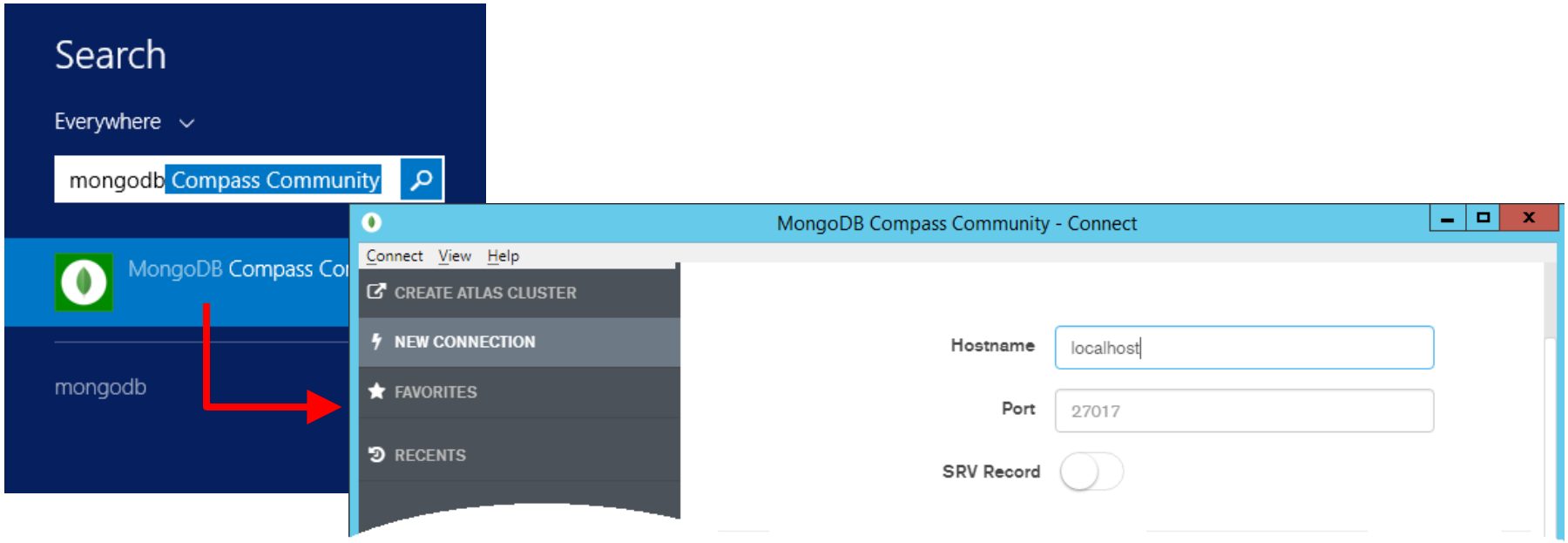
All being well, you'll see the following message:



Using MongoDB Compass (1 of 2)

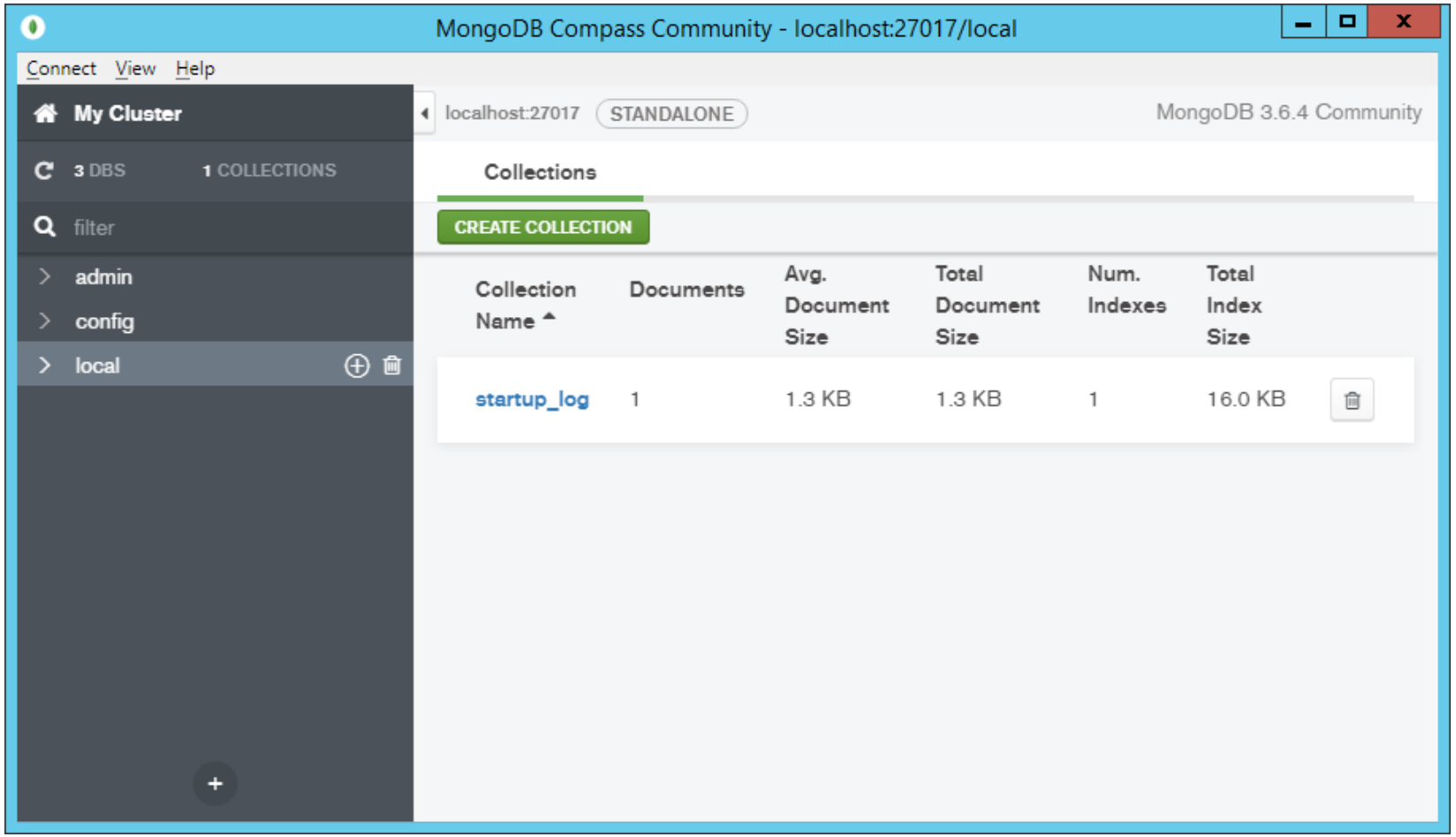
- You can also interact with a running MongoDB instance by using MongoDB Compass
  - This is the official IDE for MongoDB

- Run MongoDB Compass and connect to the MongoDB instance on localhost, port 27017



Using MongoDB Compass (2 of 2)

- MongoDB Compass allows you to explore and manage data in your MongoDB instance



Any Questions?

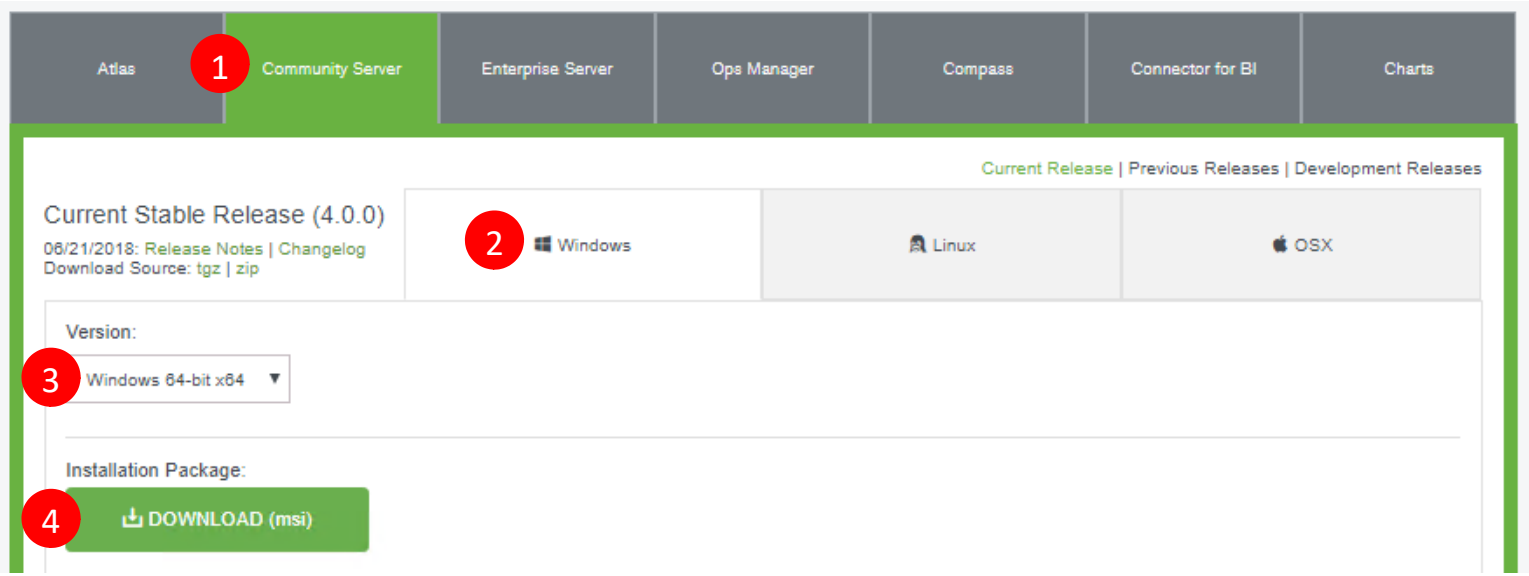
Annex: Installing MongoDB

- Downloading MongoDB for Windows
- Installing MongoDB for Windows
- MongoDB installation options

Downloading MongoDB for Windows

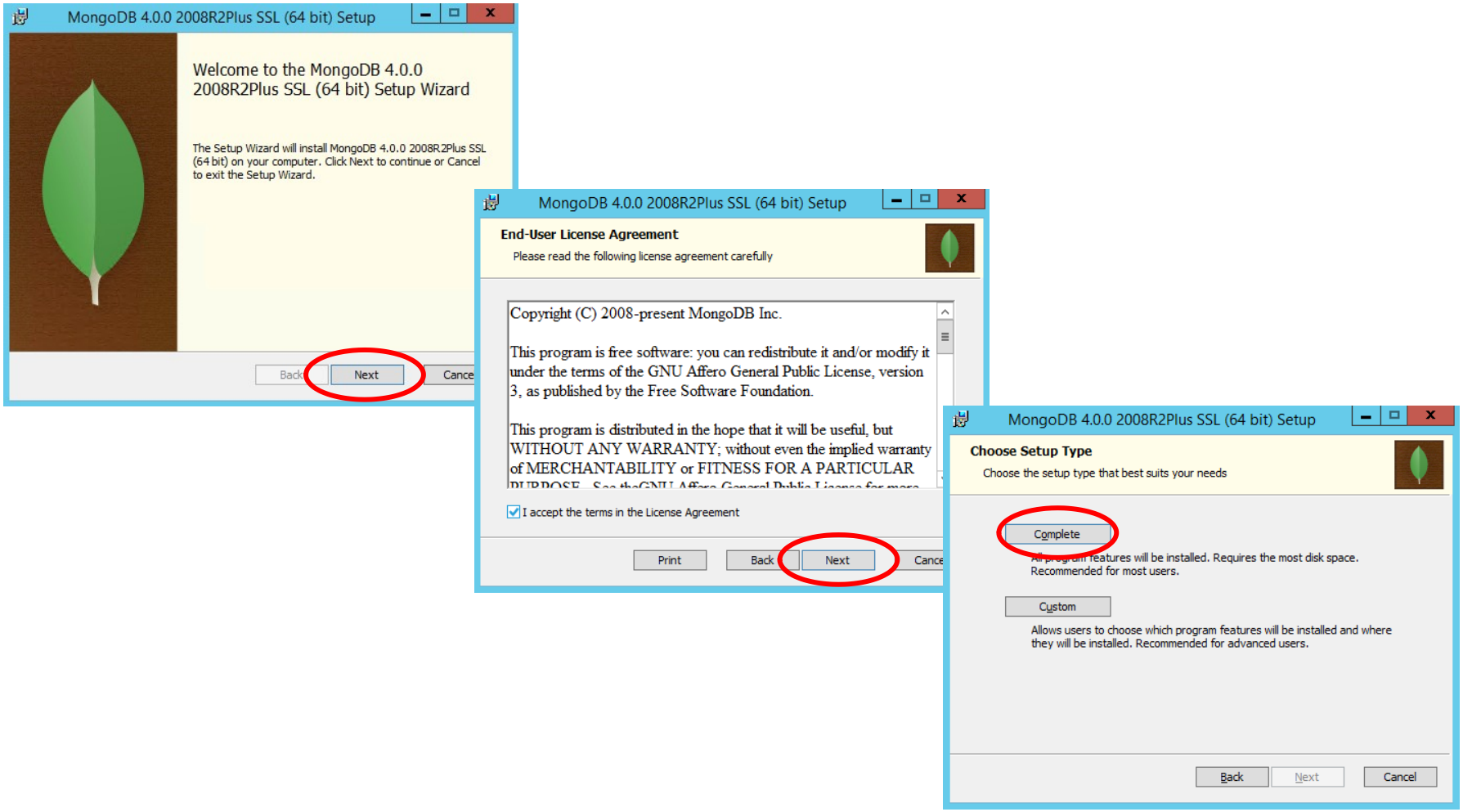
- This section shows how to install MongoDB Community Edition on Windows…
  - Requires Windows Server 2008 R2, Windows Vista, or later

- Go to the download page for MongoDB Community Edition
  - https://www.mongodb.com/download-center#community
  - Select the Windows 64-bit installation



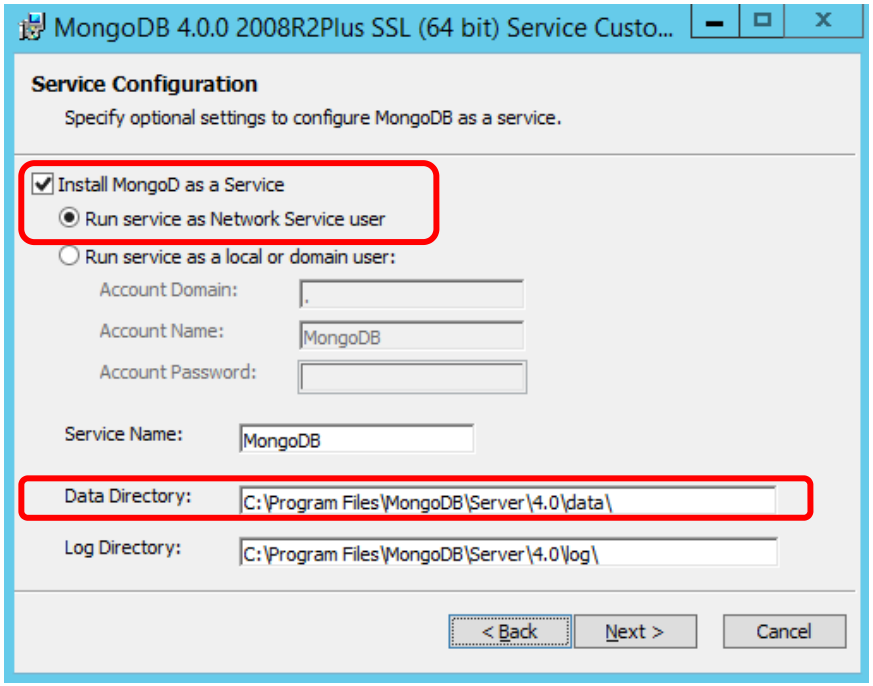
Installing MongoDB for Windows

- When the MongoDB msi has downloaded, run it



MongoDB Installation Options (1 of 3)

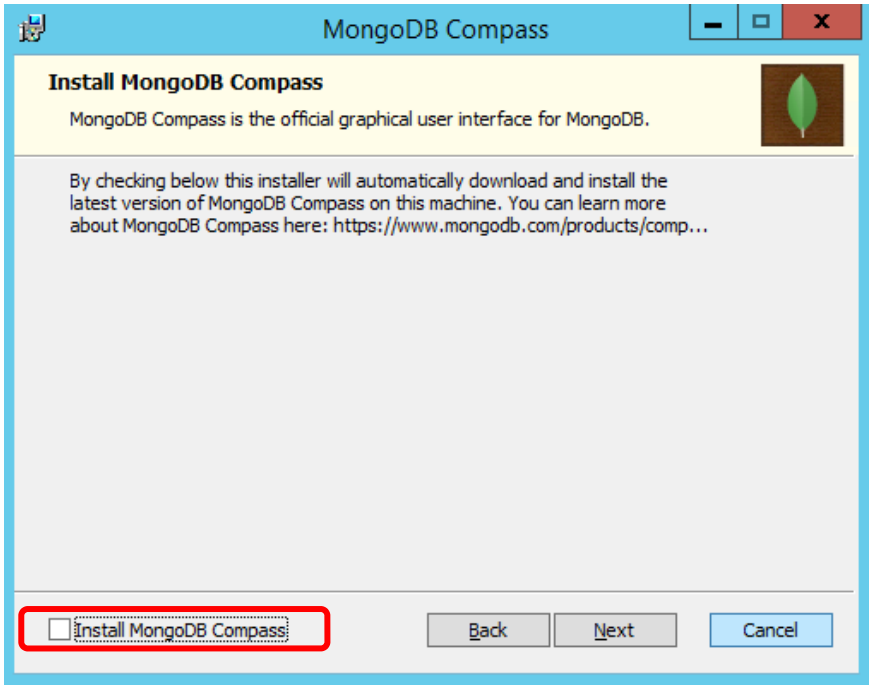
- It's possible to install MongoDB as a Windows Service
  - MongoDB starts automatically when the machine boots up
- MongoDB requires a data directory to store all data
  - You can accept the default location, or specify a different location



MongoDB Installation Options (2 of 3)

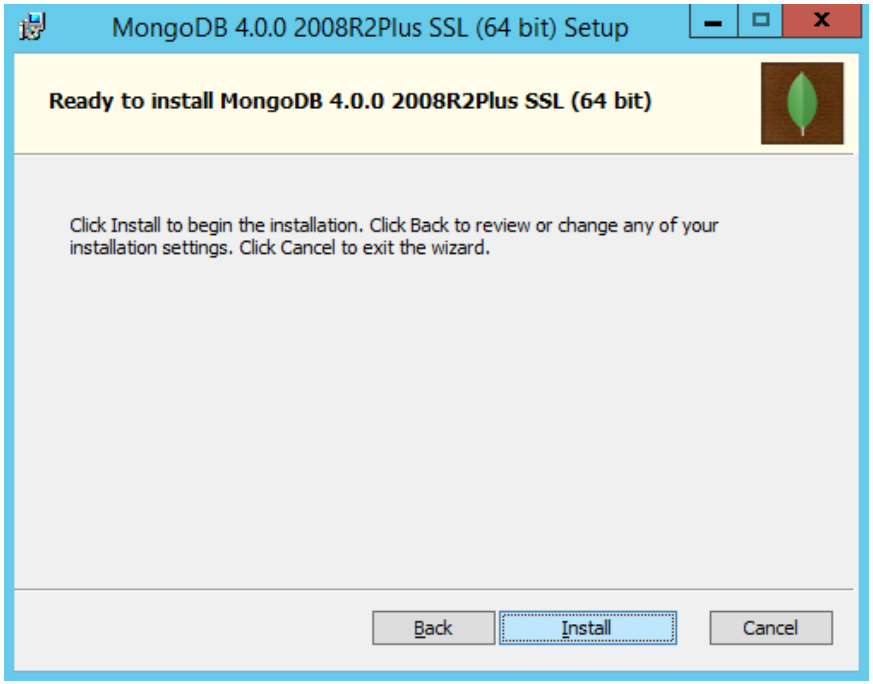
- You can choose to install MongoDB Compass
  - The official IDE for managing MongoDB

- Deselect this option if Compass is already installed, or if you want to install it later



MongoDB Installation Options (3 of 3)

- Proceed to begin the installation



Understanding the MongoDB API

Contents

1. MongoDB Documents and Collections

- Overview
- MongoDB documents
- Field types
- Accessing fields in a document
- MongoDB collections
- Creating a collection

Overview

- MongoDB provides a simple-to-use API that allows you to perform CRUD operations on NoSQL data
  - Create (insert) documents into a collection
  - Read (find) documents in a collection
  - Update existing documents in a collection
  - Delete exitsing documents in a collection

- The MongoDB API is available in several languages, including:
  - JavaScript (via the MongoDB Shell - see this chapter)
  - Python (via PyMongo - see chapter 4)
  - C# (via MongoDB NuGet packages - see Chapter 5)

MongoDB Documents

- A MongoDB document is a BSON object
  - BSON is effectively binary JSON - see http://bsonspec.org/
  - Max document size is 16MB
  - https://www.w3schools.com/js/js_json_datatypes.asp

- MongoDB documents contain fieldname/value pairs

```
{
    field1: value1,
    field2: value2,
    …
    fieldN: valueN
}
```

- Miscellaneous notes:
  - Field names are strings
  - Each document has a special field named `_id` (primary key)
  - MongoDB preserves the ordering of fields (`_id` is always first)

Field Types

- A field can be:
  - Any BSON type
  - An array, document, or array of documents

- Example:

```
var emp1 = {
    _id:    ObjectId("21aa914e0405a59ce30a94a2"),      // Unique ID for this object.
    name:   { first: "Ola", last: "Nordmann" },        // Embedded document.
    dob:    new Date('Jul 2, 1997'),                  // Date object.
    langs:  [ "Norwegian", "Swedish", "English" ],    // Array of strings.
    views:  NumberLong(1250000)                       // 64-bit long integer.
}
```

- Note:
  - BSON has many more standard data types than JSON
  - See https://docs.mongodb.com/manual/reference/bson-types/

Accessing Fields in a Document

- To access a field in a document:
  - Use dot notation
- To access an element in an array:
  - Use [] notation and specify a zero-based index
- Examples:

```
emp1.name          // { "first" : "Ola", "last" : "Nordmann" }

emp1.name.first    // Ola

emp1.langs         // [ "Norwegian", "Swedish", "English" ]

emp1.langs[0]      // Norwegian
```

MongoDB Collections

- MongoDB stores documents in collections
  - MongoDB collections are analogous to tables in a RDBMS
- By default, documents in a collection don't have to have the same schema
  - This is one of the attractions of NoSQL databases
  - You can specify document validation rules if you like (v3.2+)

Creating a Collection

- You can explicitly create a collection
  - Via `db.createCollection()`
  - Useful if you want to specify creational options

```
db.createCollection("log", {
    capped: true,
    size: 20000,
    max: 500
})
```

- If you don't want to set any options for a collection, you don't need to create the collection explicitly
  - Just start inserting documents into the collection
  - MongoDB creates the collection if it doesn't already exist
  - See next section for details

2. CRUD Operations

- Creating documents
- Reading documents
- Updating documents
- Deleting documents
- Additional useful collection operations

Creating Documents

- To create documents in a collection, call:
  - `insertOne()` – insert a single document into a collection
  - `insertMany()` – insert an array of documents into a collection

```
db.people.insertOne(
    { name: "Jayne", age: 52, gender: "F" }
)
```

```
db.people.insertMany([
    { name: "Thomas", age: 20, gender: "M" },
    { name: "Emily",  age: 20, gender: "F", favTeam: "Swans" }
])
```

- Notes:
  - MongoDB creates the collection if it doesn't already exist
  - MongoDB generates unique `_id` fields if not specified
  - Documents don't have to have the same schema

Reading All Documents

- To read documents in a collection, call:
  - `find()` – find some or all documents in the collection
- If you call `find()` without any parameters, it returns all the documents in the collection
  - Analogous to `SELECT *` in SQL
- Example
  - Find all documents in the `people` collection

```
db.people.find()
```

```
{ "_id" : ObjectId("59632bc5f7a43f30a38c3599"), "name" : "Jayne", "age" : 52, "gender" : "F" }
{ "_id" : ObjectId("59632bccf7a43f30a38c359a"), "name" : "Thomas", "age" : 20, "gender" : "M" }
{ "_id" : ObjectId("59632bccf7a43f30a38c359b"), "name" : "Emily", "age" : 20, "gender" : "F", "favTeam" : "Swans" }
```

Reading Selective Documents

- You can pass a query filter document into `find()`
  - Specify the conditions that determine which documents to select
  - Analogous to `WHERE` in SQL

```
{
    field1: value1,
    field2: { operator: value },
    …
}
```

- Here are some of the query operators you can use:
  - $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin
  - $and, $or, $nor, $not
  - $exists, $type
  - $mod, $regex, $text, $where
  - For full details about these query operators and more, see https://docs.mongodb.com/manual/reference/operator/query/

Reading Selective Documents - Examples 1

- Explain the following queries:

```
db.people.find({
    name: 'Jayne'
})
```

```
db.people.find({
    age: { $gte: 20 }
})
```

```
db.people.find({
    age: { $gte: 20 },
    age: { $lte: 30 }
})
```

```
db.people.find({
    $or: [
        { age: { $lt: 20 } },
        { age: { $gt: 30 } }
    ]
})
```

Reading Selective Documents - Examples 2

- How about this one:

```
db.people.find({

    name: /^J/,

    gender: 'F',

    $or: [
        { age: { $lt: 20 } },
        { age: { $gt: 30 } }
    ]
})
```

Reading Selective Fields

- By default, `find()` returns all fields in a document
  - Analogous to `SELECT *` in SQL

- You can pass a projection document into `find()`
  - Specify the fields to include/exclude in the result documents
  - To specify fields to include, set fields to 1
  - To specify fields to exclude, set fields to 0

```
{
    fieldToInclude: 1,
    anotherFieldToInclude: 1,
    …
}
```

```
{
    fieldToExclude: 0,
    anotherFieldToExclude: 0,
    …
}
```

- Note:
  - The `_id` field is always returned, by default

Reading Selective Fields - Examples

- Explain the following queries:

```
db.people.find(
    { name: 'Jayne' },
    { age: 1, gender: 1 }
)
```

```
db.people.find(
    { name: 'Jayne' },
    { age: 1, gender: 1, _id: 0 }
)
```

```
db.people.find(
    { name: 'Jayne' },
    { name: 0 }
)
```

```
db.people.find(
    { name: 'Jayne' },
    { name: 0, _id: 0 }
)
```

Updating Documents

- To update existing documents in a collection, call:
  - `updateOne()` – update a single document in a collection
  - `updateMany()` – update an array of documents in a collection
  - `replaceOne()` – replace a single document in a collection

- For `updateOne()` and `updateMany()`, pass 3 params:
  - Filter, same as for `find()`
  - Update to perform (e.g. `$set`, `$unset`, etc.)
  - Options object:
    - `upsert` – If true, will cause an insert if no matching document found
    - `writeConcern` – Details about how to perform the "write" operation
    - `collation` – Language-specific rules for string comparison (locale etc.)

- `replaceOne()` is the same, except the 2nd param is the replacement object

Updating Documents - Examples 1

- Explain the following updates:

```
db.people.updateOne(
    { name: 'Jayne' },
    { $set: { name: 'JAYNE', favTeam: 'Swans' }  }
)
```

```
db.people.updateMany(
    {},
    { $inc: { age: 1 } }
)
```

```
db.people.updateMany(
    {},
    { $rename: { favTeam: 'favouriteTeam' } }
)
```

```
db.people.updateMany(
    {},
    { $currentDate: {
        datestamp: { $type: 'date' },
        timestamp: { $type: 'timestamp' }
      }
    }
)
```

Updating Documents - Examples 2

- Explain the following replacement:

```
db.people.replaceOne(
    { name: 'JAYNE' },
    { name: 'Jayne', age: 52, gender: 'F' }
)
```

Deleting Documents

- To delete documents in a collection, call:
  - `deleteOne()` – delete a single document in a collection
  - `deleteMany()` – delete an array of documents in a collection

- For both these methods, pass 2 params:
  - Filter, same as for `find()`
  - Options object:
    - `writeConcern` – Details about how to perform the "write" operation
    - `collation` – Language-specific rules for string comparison (locale etc.)

Deleting Documents - Examples

- Explain the following deletions:

```
db.people.deleteOne(
    { name: 'Wilfried' }
)
```

```
db.people.deleteMany(
    { favouriteTeam: 'Cardiff' }
)
```

```
db.people.deleteMany(
    { overdraft: { $exists: true } }
)
```

Additional Useful Collection Operations

- MongoDB has various other useful collection operations available, including:
  - `aggregate()`
  - `bulkWrite()`
  - `count()`, `totalSize()`, `explain()`, `distinct()`,
  - `createIndex()`, `dropIndex()`, `reIndex()`,
  - `findAndModify()`, `findAndReplace()`, `findAndDelete()`
  - `mapReduce()`
  - `remove()`

- For full details, see:
  - http://docs.mongodb.com/manual/reference/method/js-collection/

3. Aggregation Operations

- Overview
- Scenario
- Aggregation framework
- Map-reduce
- Single-purpose aggregation operations

Overview

- Aggregation operations allow you to process data records and return computed results
  - Very useful way to analyze large data sets, potentially from multiple databases
- MongoDB has 3 ways to perform aggregation operations:
  - Aggregation framework
  - Map-reduce
  - Single-purpose aggregation methods
- We'll explore each technique in this section

Scenario

- To illustrate aggregate operations, we'll use the following sample data set
  - This is the data set that appears in the MongoDB docs online

```
db.orders.insertMany([
    { cust_id: "A123", amount: 500, status: "A" },
    { cust_id: "A123", amount: 250, status: "A" },
    { cust_id: "B212", amount: 200, status: "A" },
    { cust_id: "A123", amount: 300, status: "D" }
])
```

Aggregation Framework (1 of 2)

- MongoDB's aggregation framework is based on the concept of a processing pipeline
  - You pass the data through a series of operations in a pipeline
  - You end up with an aggregated result

- The pipeline typically includes operations such as:
  - Filter the records, to keep just the ones we're interested in
  - Perform a transformation operation on remaining records
  - Perform a sort, or calculate an average value, etc. etc.

- The aggregation framework is the preferred way to do data aggregation in MongoDB
  - Efficient, because it uses native operations within MongoDB

Aggregation Framework (2 of 2)

- This example has two stages:
  - `$match` stage
  - `$group` stage

```
db.orders.aggregate([
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
])
```

```
{ "_id" : "B212", "total" : 200 }
{ "_id" : "A123", "total" : 750 }
```

- For details, see:
  - http://docs.mongodb.com/manual/core/aggregation-pipeline/

Map-Reduce (1 of 2)

- MongoDB provides map-reduce operations…
- Phase 1 is a "map" stage
  - Processes each document and emits a transformed result
- Phase 2 is a "reduce" stage
  - Combines the output of the map operation
- Optionally, map-reduce can have a finalize stage
  - Makes final modifications on the result
- Map-reduce uses JavaScript functions
  - Slower than the native ops in the aggregation framework
  - But more flexible

Map-Reduce (2 of 2)

- This example is equivalent to the aggregation framework example earlier

```
db.orders.mapReduce(

    function() { emit(this.cust_id, this.amount); },     // Map op on each document.

    function(key, values) { return Array.sum(values) },  // Reduce operation.

    {
        query: { status: "A" },  // Query (we only want records with status "A").
        out: "order_totals"      // Output table (will contain the final results).
    }
)
```

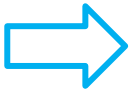
```
db.order_totals.find()
```

```
{ "_id" : "A123", "value" : 750 }
{ "_id" : "B212", "value" : 200 }
```

Single-Purpose Aggregation Operations

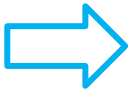
- MongoDB provides various single-purpose aggregation operations in `db.collection`, such as:

```
db.orders.distinct("cust_id")
```



```
[ "A123", "B212" ]
```

```
db.orders.count()
```



```
4
```

- For details, see:
  - https://docs.mongodb.com/manual/reference/method/js-collection/

Any Questions?