MongoDB Architecture
Data Model

• Stores data in form of BSON (binary JavaScript Object Notation) documents

```json
{
    name: "travis",
    salary: 30000,
    designation: "Computer Scientist",
    teams: [ "front-end", "database" ]
}
```

• Group of related documents with a shared common index is a collection
Query all employee names with salary greater than 18000 sorted in ascending order

db.users.find({salary:{$gt:18000}, {name:1}}).sort({salary:1})
Insert

Insert a row entry for new employee Sally

db.users.insert({
    name: "sally",
    salary: 15000,
    designation: "MTS",
    teams: [ "cluster-management" ]
})
Update

All employees with salary greater than 18000 get a designation of Executive

```
    db.users.update(
        Update Criteria           {salary:{$gt:18000}},
        Update Action            {$set: {designation: “Manager”}},
        Update Option            {multi: true}
    )
```

Multi option allows multiple document update
Delete

Remove all employees who earn less than 10000

```
db.users.remove(
  Remove Criteria
  {salary:{$lt:10000}},
)
```

Can accept a flag to limit the number of document removal
Typical MongoDB Deployment

- **Shards**: mongod servers store the data
- **Multiple shard servers** form a replica set
- **Replica set** maintain same replica of data
- **Routers**: mongos interfaces with clients and routers operations to appropriate shards
- **Config**: Stores collection level metadata.
Read Preference

• Determine where to route read operation
• Default is primary. Possible options are secondary, primary-preferred, etc.
• Helps reduce latency, improve throughput
• Reads from secondary may fetch stale data
Write Concern

• Determines the guarantee that MongoDB provides on the success of a write operation
• Default is *acknowledged*. Others are unacknowledged, replica-acknowledged, etc
• For the default case, primary replicas acknowledge the success of a write operation
• Weaker write concern implies faster write time
Write operation performance

• Indexing: Every write needs to update every index associated with the collection

• Document Growth: When document grows beyond the current allocation, it is relocated on disk

• Hardware

• Journaling: Write-ahead logging to an on-disk journal for durability
Partition

• Shard Key: Single or compound field in schema used for data partitioning

• Partitions are called *chunks*. Two strategies:
  – Range based: Shard Key Values are partitioned into ranges

Total Key Space for x

```
Chunk 1  Chunk 2  Chunk 3  Chunk 4  Chunk 5  Chunk 6
```

```
x:minkey  x:-100  x:-50  x:0  x:50  x:100  x:maxkey
```
Partition

- Hash based: Hash of shard key values are partitioned into ranges

- Range Queries are efficient for the first strategy
- Hash Scheme leads to better data balancing
Balancing

- Splitting: Background process which splits when a chunks grows beyond a threshold
- Balancing: Migrates chunks among shards if there is an uneven distribution
Replication

- **Primary**
- **Secondary**

Connections:
- Replication from Primary to Secondary
- Replication from Secondary to Primary
- Write from Primary to Secondary
- Read from Secondary to Primary
- Heartbeat between Secondary nodes
Replication

- Oplog based data sync up
- Leader Election protocol elects a master
- Arbiters are mongod servers which do not maintain data but vote
Consistency

• Strongly Consistent: Read Preference is Master
• Eventually Consistent: Read Preference is Slave
• CAP Theorem: Under partition, MongoDB becomes write unavailable thereby ensuring consistency
Performance

• 30 – 50x faster than Sql Server 2008 for writes[1]
• At least 3x faster for reads[1]
• MongoDB 2.2.2 offers slower throughput for different YCSB workloads compared to Cassandra[2]

Demo
Insert

Insert a row entry for new employee Sally

use records     -- Creates a database

db.employee.insert({
    name: "Sally",
    salary: 15000,
    designation: "MTS",
    teams: "cluster-management"
})

Also can use save instead of insert
Bulk Load

- `salary` = [10000, 5000, 8000, 2000];
- `designation` = [“MTS”, “Computer Scientist”, “Manager”, “Director”];
- `teams` = [“cluster-management”, “human-resource”, “backend”, “ui”];

```javascript
for(var i=0; i<10000; i++){
    name = people[Math.floor(Math.random() * people.length)];
    salary = salary[Math.floor(Math.random() * salary.length)];
    designation = designation[Math.floor(Math.random() * designation.length)];
    teams = teams[Math.floor(Math.random() * teams.length)];
    db.employee.save({"name":name, salary:salary, "designation": designation, "teams":teams});
}
```
Query

- `db.users.find()`
- `db.users.find({name: "Sally"})`
- `var cursor = db.users.find({salary: {$in: [5000, 2000]}})`
- Use `next()` to access the rest of the records
Query

• `db.users.find({name: "Steve", salary: {$lt: 3000} })`

• `db.inventory.find( { $or: [ { name: "Bill" }, { salary: { $gt: 9000 } } ] } )`

• Find records of all managers who earn more than 5000
Aggregation Commands

- db.users.count()
- db.users.find({name: "Steve"}).count()
- db.users.find({name: "Steve"}).skip(10)
- db.users.find({name: "Steve"}).limit(10)
Modify/Remove

- `db.users.update( { designation : "Manager" }, { $inc : { salary : 1000 } } )`
- `db.users.update( { designation : "Manager" }, { $inc : { salary : 1000 } }, { multi: true } )`
- `db.users.remove( { name : "Sally" } )`