

Data Management using mongoDB®



Outline

1. Introduction to MongoDB
2. Insert and Update API
3. Query API
4. Collection Modeling
5. Aggregation Queries
6. Database Scalability Solutions
7. MongoDb Sharding & Replication

Introduction to



mongoDB®

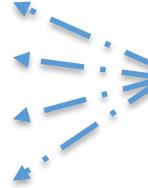


What is MongoDB?

- MongoDB is an open-source **Document Oriented Database**
 - **Uses a document data model:** Stores data as JSON documents (instead of rows and columns as done in a relational database)
 - **Arrange documents in collections** (documents can vary in structure)
 - with dynamic schemas (schemaless)
 - **API to query and manage documents**
- Better alternative data management solution for Web applications compared to using a Relational Database

Document

```
{  
  "isbn" : "123",  
  "title": "Mr Bean and the Forty Thieves",  
  "category": "Fun",  
  "pages": 250  
  "authors": ["Mr Bean", "Juha Dahak"],    ←----- property : array  
  "publisher": {  
    "name": "MrBeanCo",           ←----- property : document  
    "country": "UK"  
  }  
}
```



The diagram illustrates a document as a collection of key-value pairs. It features a central blue arrow pointing right, with several dashed blue arrows branching off from its tip to point to different parts of the JSON object. The text 'property : value' is positioned next to the main arrow, 'property : array' is next to the 'authors' key, and 'property : document' is next to the 'publisher' key.

- **Document = JSON object**
- **Document = set of key-value pairs**
- **Basic unit of data in MongoDB**
- Analogous to **row** in a relational database

Document Data Model

Relational

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

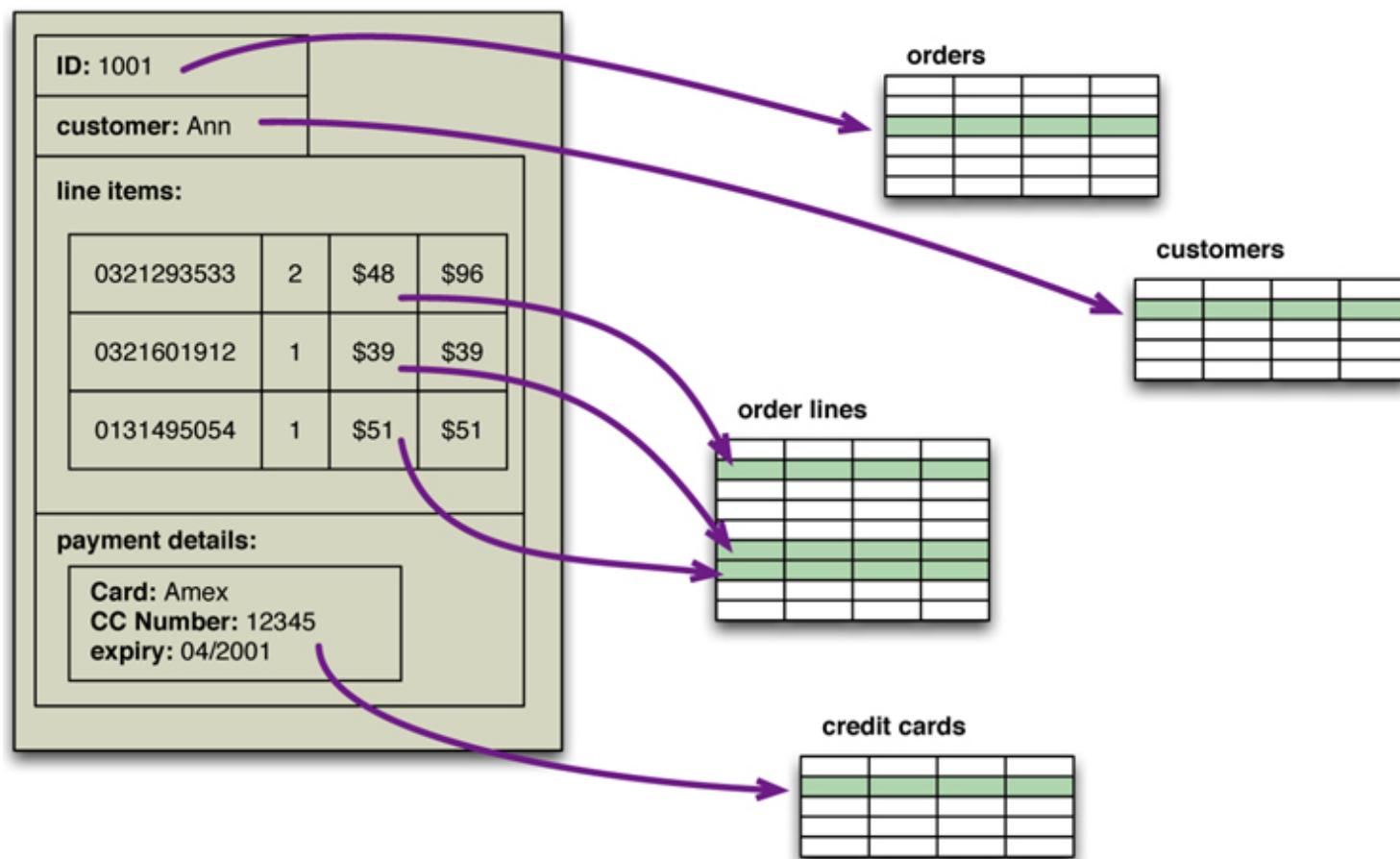
NO RELATION



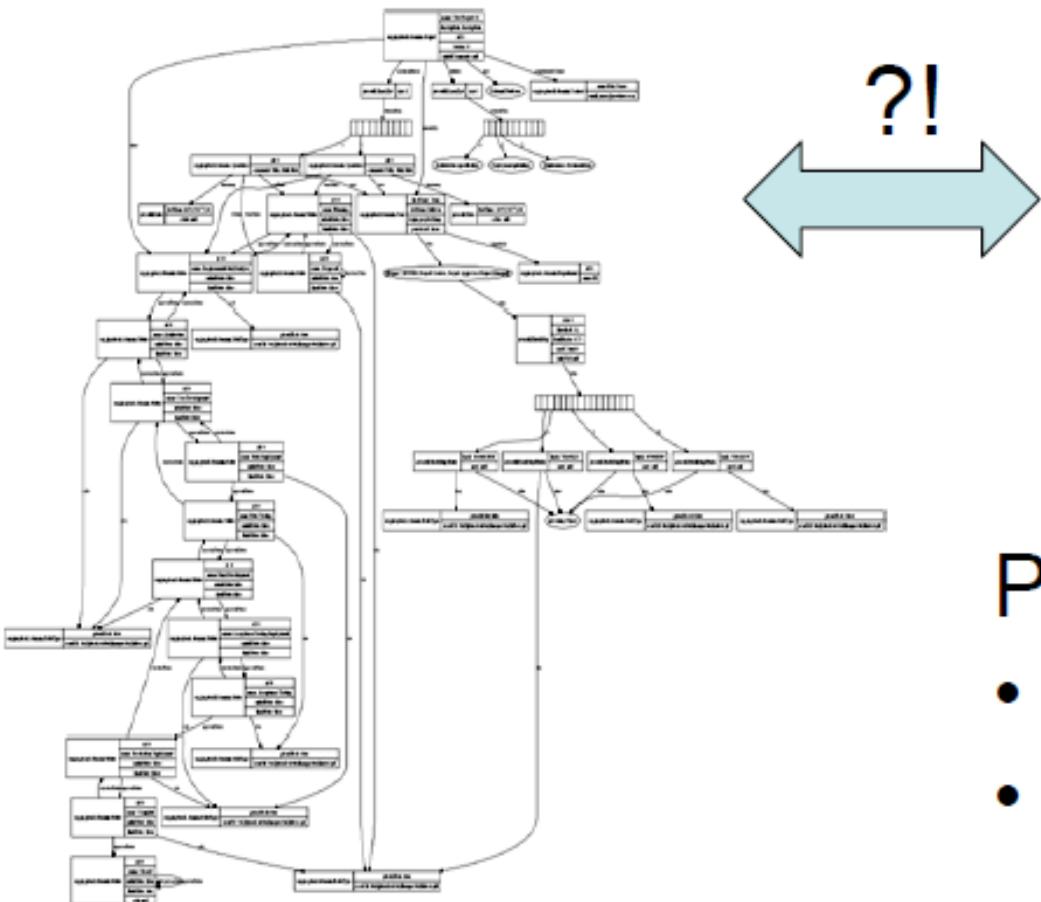
MongoDB

```
{  
  first_name: 'Paul',  
  surname: 'Miller',  
  city: 'London',  
  location:  
    [45.123, 47.232],  
  cars: [  
    { model: 'Bentley',  
      year: 1973,  
      value: 100000, ... },  
    { model: 'Rolls Royce',  
      year: 1965,  
      value: 330000, ... }  
  ]  
}
```

Reduces the Impedance mismatch between the relational model and the in-memory data structures



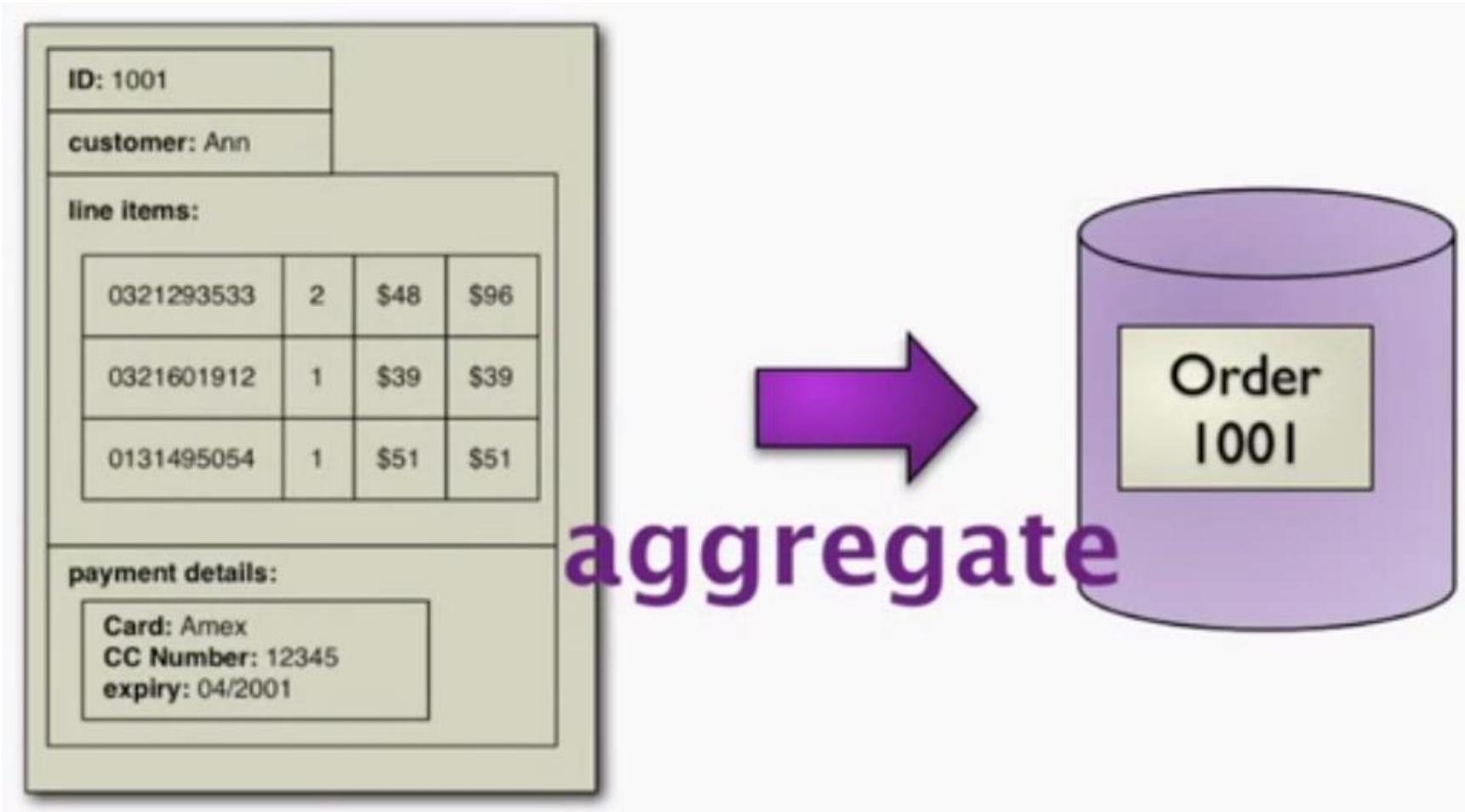
Avoids - Complex object graphs need to be Mapped to the Relational Model



Poor performance

- Many inserts
- Many joins

Aggregate-Oriented



- Aggregate brings **cluster friendliness** as a whole aggregate can be stored in one node of the cluster

Collection

```
{  
  "isbn": "123",  
  "title": "Mr Bean and the Forty Thieves",  
  "authors": ["Mr Bean", "Juha Dahak"],  
  "publisher": {"name": "MrBeanCo", "country": "UK"},  
  "category": "Fun",  
  "pages": 250  
}
```

- **Collection = Group** of documents
- Analogous to **table** in a relational database
- **Does not enforce** a schema
- Documents in a collection usually **have similar purpose** but they may have slightly different schema

Insert and Update API

CRUD



C R U D

- **Create**

- db.collection.insert(<document>)
- db.collection.save(<document>)
- db.collection.update(<query>, <update>, { upsert: true })

- **Read**

- db.collection.find(<query>, <projection>)
- db.collection.findOne(<query>, <projection>)

- **Update**

- db.collection.update(<query>, <update>, <options>)

- **Delete**

- db.collection.remove(<query>, <justOne>)

CRUD Examples

```
> db.user.insert({  
    first: "John",  
    last : "Doe",  
    age: 39  
})
```

```
> db.user.find ()  
{  
    "_id" : ObjectId("51..."),  
    "first" : "John",  
    "last" : "Doe",  
    "age" : 39  
}
```

```
> db.user.update(  
    {"_id" : ObjectId("51...")},  
    {  
        $set: {  
            age: 40,  
            salary: 7000}  
    }  
)
```

```
> db.user.remove({  
    "first": /^J/  
})
```

Examples

In RDBMS

```
CREATE TABLE users (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

```
DROP TABLE users
```

In MongoDB

Either insert the 1st document

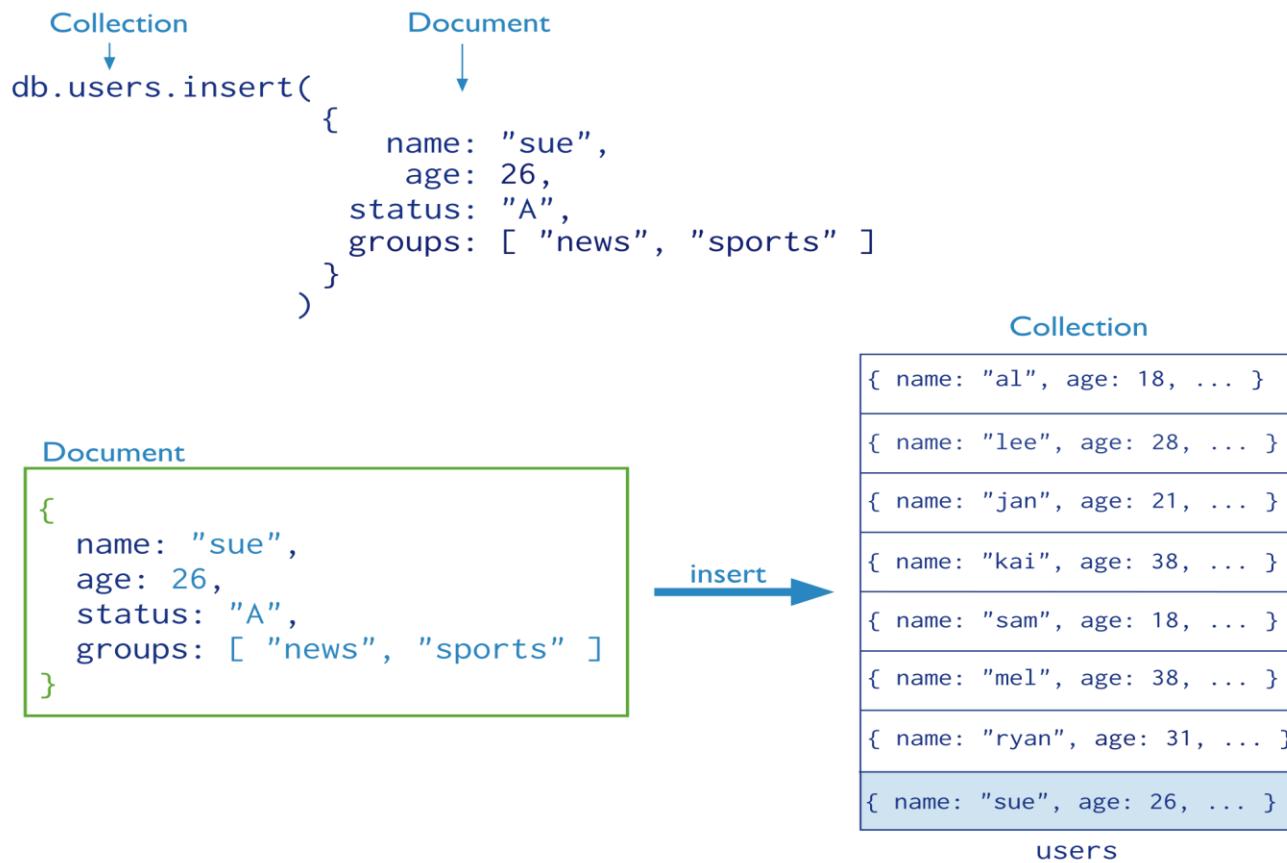
```
db.users.insert( {
    user_id: "abc123",
    age: 55,
    status: "A"
} )
```

Or create “Users” collection explicitly

```
db.createCollection("users")
```

```
db.users.drop()
```

Insertion



- The collection “users” is created automatically if it does not exist

Multi-Document Insertion (Use of Arrays)

```
var mydocuments =  
[  
  {  
    item: "ABC2",  
    details: { model: "14Q3", manufacturer: "M1 Corporation" },  
    stock: [ { size: "M", qty: 50 } ],  
    category: "clothing"  
  },  
  {  
    item: "MNO2",  
    details: { model: "14Q3", manufacturer: "ABC Company" },  
    stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],  
    category: "clothing"  
  },  
  {  
    item: "IJK2",  
    details: { model: "14Q2", manufacturer: "M5 Corporation" },  
    stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],  
    category: "houseware"  
  }  
];
```

```
db.inventory.insert( mydocuments );
```



All the documents
are inserted at once

Multi-Document Insertion (Bulk Operation)

- A temporary object in memory
- Holds your insertions and uploads them at once

There is also *Bulk Ordered* object

```
var bulk = db.inventory.initializeUnorderedBulkOp();

bulk.insert(
  {
    item: "BE10",
    details: { model: "14Q2", manufacturer: "XYZ Company" },
    stock: [ { size: "L", qty: 5 } ],
    category: "clothing"
  }
);
bulk.insert(
  {
    item: "ZYI1",
    details: { model: "14Q1", manufacturer: "ABC Company" },
    stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 } ],
    category: "houseware"
  }
);

bulk.execute();
```

_id column is added automatically

Deletion (Remove Operation)

- You can put condition on any field in the document (even **_id**)

```
db.users.remove(  
  { status: "D" }  
)
```

← collection
← remove criteria

The following diagram shows the same query in SQL:

```
DELETE FROM users  
WHERE status = 'D'
```

← table
← delete criteria

```
db.users.remove()
```



Removes all documents from *users* collection

Update

```
db.users.update(  
    { age: { $gt: 18 } },           ← collection  
    { $set: { status: "A" } },      ← update criteria  
    { multi: true }               ← update action  
)  
                                ← update option
```

Otherwise, it will update only the 1st matching document

Equivalent to in SQL:

UPDATE	users	← table
SET	status = 'A'	← update action
WHERE	age > 18	← update criteria

Update (Cont'd)

```
db.inventory.update(
  { item: "MNO2" },
  {
    $set: {
      category: "apparel",
      details: { model: "14Q3", manufacturer: "XYZ Company" }
    },
    $currentDate: { lastModified: true }
  }
)
```

Two operators

For the document with item equal to "MNO2", use the \$set operator to update the category field and the details field to the specified values and the \$currentDate operator to update the field lastModified with the current date.

Replace a document

New doc

```
db.inventory.update(  
  { item: "BE10" }, ← Query Condition  
  {  
    item: "BE05",  
    stock: [ { size: "S", qty: 20 }, { size: "M", qty: 5 } ],  
    category: "apparel"  
  })
```

For the document having item = “BE10”, replace it with the given document

Insert or Replace

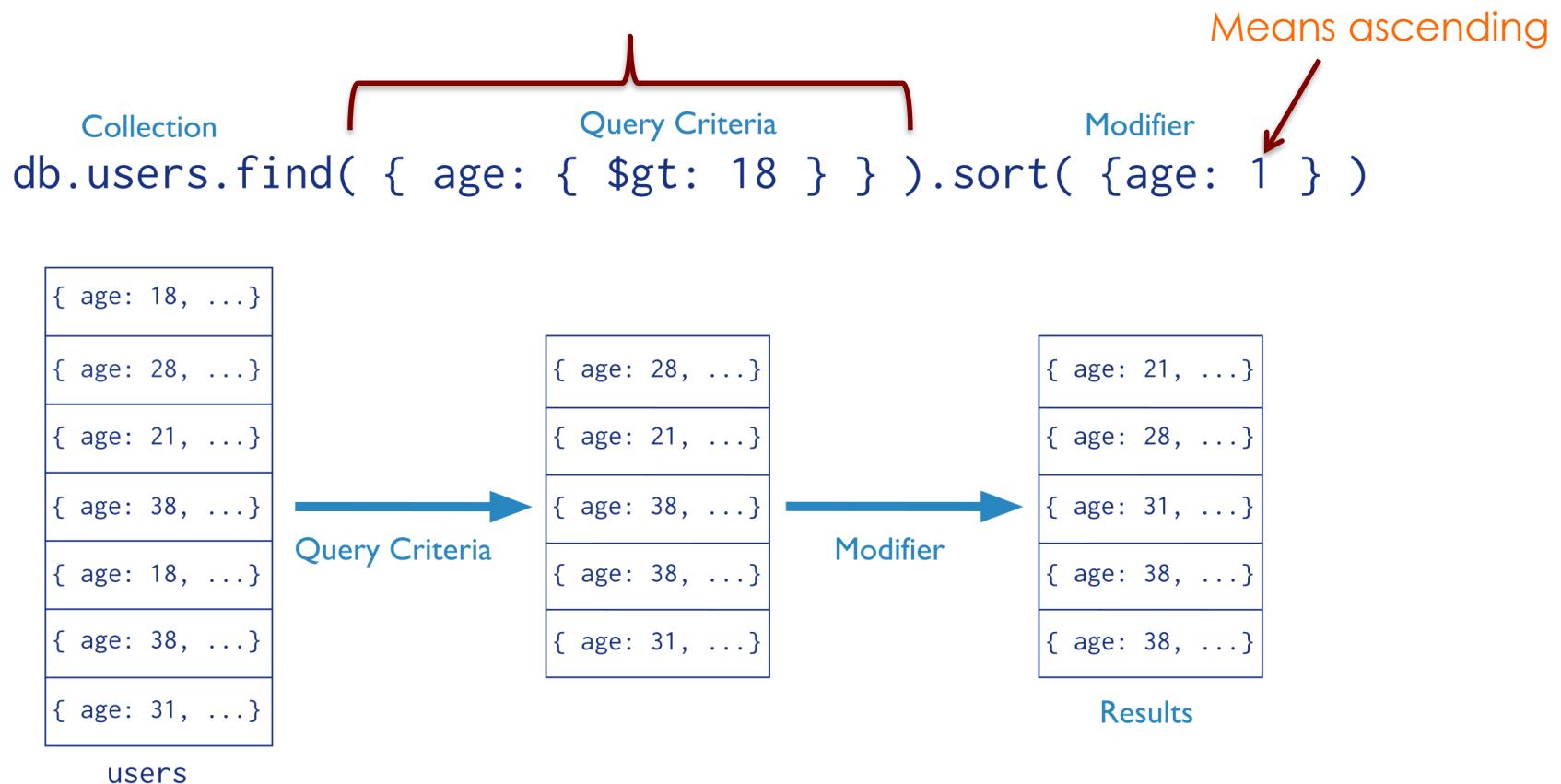
```
db.inventory.update(
  { item: "TBD1" },
  {
    item: "TBD1",
    details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
    stock: [ { "size" : "S", "qty" : 25 } ],
    category: "houseware"
  },
  { upsert: true }
)
```

The **upsert** option

If the document having item = “TBD1” is in the DB, it will be replaced
Otherwise, it will be inserted.

Query API

Find() Operator



Find() + Projection

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
).limit(5)
```

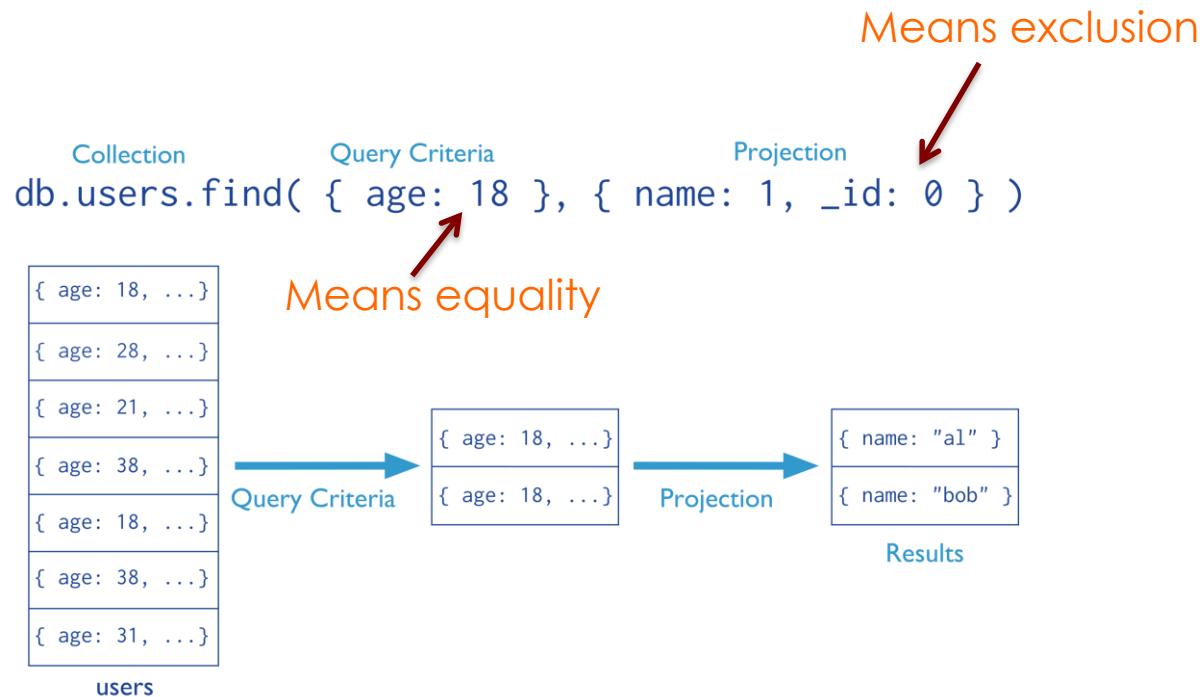
← collection
← query criteria
← projection
← cursor modifier

Means inclusion +
_id is always automatically included

Equivalent to in SQL:

```
SELECT _id, name, address ← projection  
FROM users ← table  
WHERE age > 18 ← select criteria  
LIMIT 5 ← cursor modifier
```

Find(): Exclude Fields



Find() More Examples

Report all documents in the “inventory” collection

```
db.inventory.find()
```

```
db.inventory.find( {} )
```

Equivalent to in SQL:

```
Select *  
From inventory;
```

Report all documents in the “inventory” collection

Where type = ‘food’ or ‘snacks’

```
db.inventory.find(  
  { type: { $in: [ 'food', 'snacks' ] } }  
)
```

Equivalent to in SQL:

```
Select *  
From inventory  
Where type in  
  ('food',  
  'snacks');
```

Find(): AND & OR

AND Semantics

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

OR Semantics

```
db.inventory.find(
  {
    $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
  }
)
```

AND + OR Semantics

```
db.inventory.find(
  {
    type: 'food',
    $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
  }
)
```

Type = 'food' and (qty > 100 or price < 9.95)

Queries Return Cursors

- All queries return the results in a cursor
- If not assigned to a variable ➔ Printed to screen
 - Results are stored in a cursor
 - Many operators on top of that to manipulate the cursor



```
var myCursor = db.inventory.find();

var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;

myCursor.objectsLeftInBatch();
```

Cursor's Methods:

<http://docs.mongodb.org/manual/reference/method/js-cursor/>

Cursor Manipulation

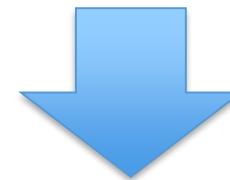
```
var myCursor = db.inventory.find( { type: 'food' } );  
  
myCursor.forEach(printjson);
```

```
var myCursor = db.inventory.find( { type: 'food' } );  
var documentArray = myCursor.toArray();  
var myDocument = documentArray[3];
```

Querying Complex Types

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "isAlive": true,  
  "age": 25,  
  "height_cm": 167.6,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021-3100"  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "office",  
      "number": "646 555-4567"  
    }  
  ],  
  "children": [],  
  "spouse": null  
}
```

Documents can be complex,
E.g.,
(Arrays, embedded documents,
any nesting of these, many
levels)



Queries get complex too !!!

Array Manipulation (Exact Match)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

The operation returns the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Array Manipulation (Search By Element)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Notice: if a document has “ratings” as an Integer field = 5, it will be returned

Array Manipulation (Search By Position)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { 'ratings.0': 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
```

Another Example

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```



```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Embedded Object Matching (Field Matching)

```
{  
  name: "Joe",  
  address: {  
    city: "San Francisco",  
    state: "CA" },  
  likes: [ 'scuba', 'math', 'literature' ]  
}
```

Find the user documents where the address's state
= 'CA'

```
db.persons.find( {"address.state" : "CA"})
```



Using dot notation

Matching Arrays of Embedded Documents

```
{  
  _id: 100,  
  type: "food",  
  item: "xyz",  
  qty: 25,  
  price: 2.5,  
  ratings: [ 5, 8, 9 ],  
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]  
}  
  
{  
  _id: 101,  
  type: "fruit",  
  item: "jkl",  
  qty: 10,  
  price: 4.25,  
  ratings: [ 5, 9 ],  
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]  
}
```

Select all documents where the memos array contains a document written by 'shipping' department

```
db.inventory.find( { 'memos.by': 'shipping' } ) // Returns both documents
```



Means any element in the array

Matching Arrays of Embedded Documents: Multiple Conditions

```
{  
  _id: 100,  
  type: "food",  
  item: "xyz",  
  qty: 25,  
  price: 2.5,  
  ratings: [ 5, 8, 9 ],  
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]  
}  
  
{  
  _id: 101,  
  type: "fruit",  
  item: "jkl",  
  qty: 10,  
  price: 4.25,  
  ratings: [ 5, 9 ],  
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]  
}
```

```
db.inventory.find(  
  {  
    memos:  
      {  
        $elemMatch:  
          {  
            memo: 'on time',  
            by: 'shipping'  
          }  
      }  
  }  
)
```

Select all documents where the memos array contains a document written by 'shipping' department and the content "on time"

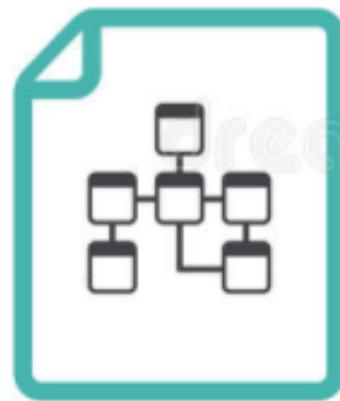
Query Operators: Comparison Op

Name	Description
<code>\$eq</code>	Matches values that are equal to a specified value.
<code>\$gt</code>	Matches values that are greater than a specified value.
<code>\$gte</code>	Matches values that are greater than or equal to a specified value.
<code>\$lt</code>	Matches values that are less than a specified value.
<code>\$lte</code>	Matches values that are less than or equal to a specified value.
<code>\$ne</code>	Matches all values that are not equal to a specified value.
<code>\$in</code>	Matches any of the values specified in an array.
<code>\$nin</code>	Matches none of the values specified in an array.

```
db.inventory.find( { qty: { $gte: 20 } } )
```

```
db.inventory.update(  
  { "carrier.fee": { $gte: 2 } },  
  { $set: { price: 9.99 } }  
)
```

Collection Modeling



Document can have a Complex Structure

Properties

```
{  
    first_name: 'Paul', <---- String  
    surname: 'Miller',  
    cell: 447557505611, <---- Number  
    city: 'London',  
    location: { type: Point,  
                coordinates: [-0.223, 51.52]},  
    Profession: [ 'banking', 'finance', 'trader' ], <---- Array  
    cars: [  
        { model: 'Bentley',  
         year: 1973,  
         value: 100000, ... },  
        { model: 'Rolls Royce',  
         year: 1965,  
         value: 330000, ... }  
    ]  
}
```

Values could be

Geo-Location

Properties can contain an array of sub-documents (JSON objects)

Embedded vs Referenced documents

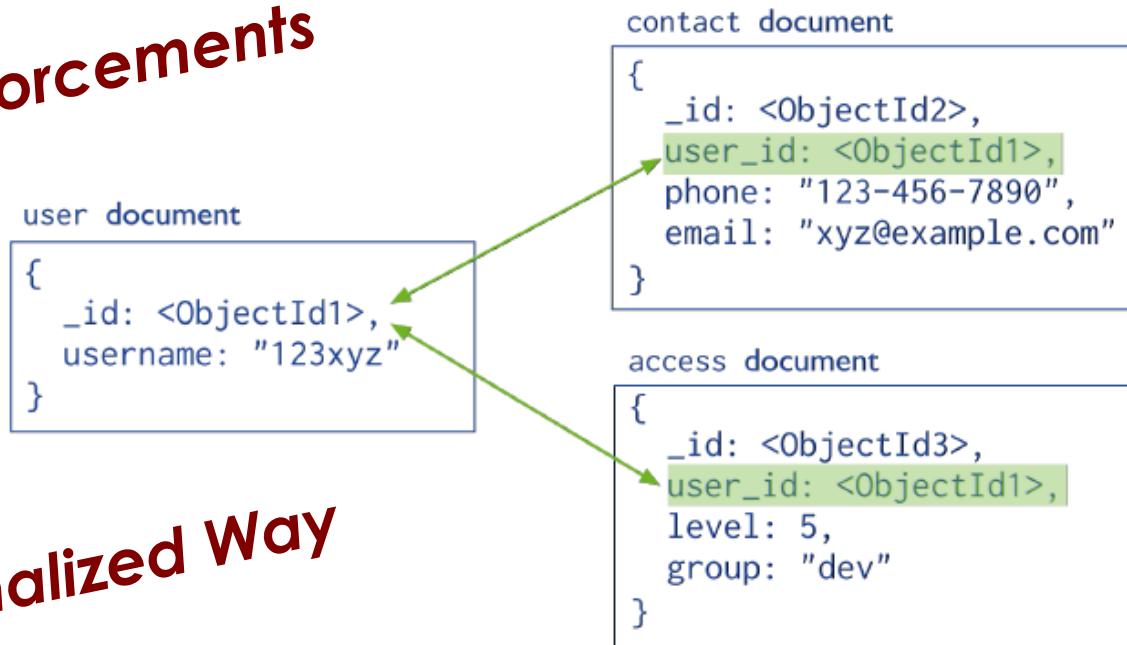
- Modeling multiple collections that reference each other
- In Relational DBs → FK-PK Relationships
- Major design decision when designing a Document Schema is to decide **Embedded** vs **Referenced** subdocuments
- Decision should consider:
 - How the data will be used
 - Size of the document

In MongoDB

- ***Referencing*** between two collections
 - Use Id of one and put in the other
 - Very similar to FK-PK in Relational DBs
 - **Does not come with enforcement mechanism**
- ***Embedding*** between two collections
 - Put the document from one collection inside the other one

Referencing

No Enforcements



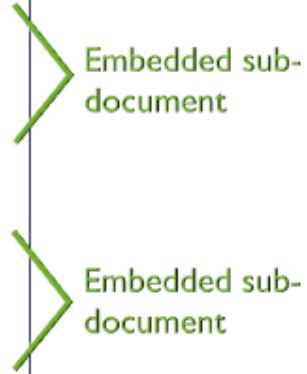
Normalized Way

- Have three collections in the DB: “User”, “Contact”, “Access”
- Link them by `_id` (or any other field(s))

Embedding

De-Normalized Way

```
{  
  _id: <ObjectId1>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```



Embedded sub-document

Embedded sub-document

- Have one collection in DB: “User”
- The others are embedded inside each user’s document

Embedding

- **Advantages**
 - Retrieve all relevant information in a single query/document
 - Avoid implementing joins in application code => fast data retrieval
 - Update related information as a single atomic operation
- **Limitations**
 - Large documents mean more overhead if most fields are not relevant
 - 16 MB document size limit

Referencing

- **Advantages**
 - Smaller documents
 - Less likely to reach 16 MB document limit
 - Infrequently accessed information not accessed on every query
 - No duplication of data
- **Limitations**
 - Two queries required to retrieve information
 - Cannot update related information atomically (in one atomic operation)

1 to 1 Relationships => Better to Embed

Medical Procedures

```
{  
   "_id": 333,  
  "date": "2003-02-09T05:00:00",  
  "hospital": "County Hills",  
  "patient": "John Doe",  
  "physician": "Stephen Smith",  
  "procedure": "Glucose",  
  "result": {  
    "value": 97,  
    "measurement": "mg/dl"  
  }  
}
```

Embed:

- No data duplication
- Data that are read/written together lives together

← Embed – *weak entity*

One to Many Relationships

Patients

```
{  
  _id: 2,  
  first: "Joe",  
  last: "Patient",  
  addr: { ...},  
  procedures: [  
    {  
      id: 12345,  
      date: 2015-02-15,  
      type: "CAT scan",  
      ...},  
    {  
      id: 12346,  
      date: 2015-02-15,  
      type: "blood test",  
      ...}]  
}
```

Embed

OR

Patients

```
{  
  _id: 2,  
  first: "Joe",  
  last: "Patient",  
  addr: { ...},  
  procedures: [12345, 12346]  
}
```

Reference

Procedures

```
{  
  _id: 12345,  
  date: 2015-02-15,  
  type: "CAT scan",  
  ...}  
{  
  _id: 12346,  
  date: 2015-02-15,  
  type: "blood test",  
  ...}
```

One to Many : General Recommendations

- **Embed when:**
 - **One-to-few** (e.g. customer - addresses)
 - **Often queried/updated together** in a single query (e.g., book - chapters)
 - No need to access the embedded object outside the context of the parent object (e.g., order – order items)
 - No additional data duplication introduced
- **Reference when:**
 - **1 to a large number of related items** (e.g. customer orders , book – reviews, video - comments)
 - Related data changes frequently (e.g., video – viewCount)
 - Referenced entity that is used by many others (e.g., session - room)
 - Document size is > 16 MB
 - Subdocument has a large number of infrequently accessed fields

1 to M Example 1

- *"We need to store user information like name, email and their addresses... yes they can have more than one."*

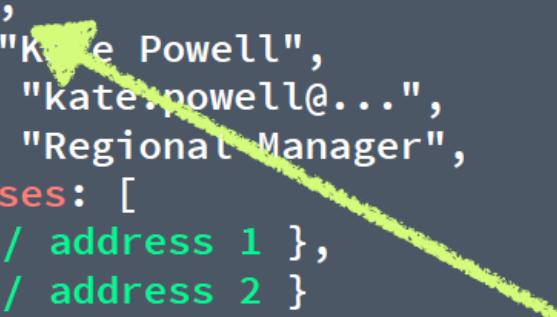
```
{  
  _id: 1,  
  name: "Kate Powell",  
  email: "kate.powell@somedomain.com",  
  title: "Regional Manager",  
  addresses: [  
    { street: "123 Sesame St", city: "Boston" },  
    { street: "123 Evergreen St", city: "New York" }  
  ]  
}
```

One-to-few : embedding is the best design

1 to M Example 2

- "We have to be able to store tasks, assign them to users and track their progress..."

```
> db.user.findOne({_id: 1})      > db.task.findOne({user_id: 1})  
{  
  _id: 1,  
  name: "Kate Powell",  
  email: "kate.powell@...",  
  title: "Regional Manager",  
  addresses: [  
    { // address 1 },  
    { // address 2 }  
  ]  
}  
  
> db.task.findOne({user_id: 1})  
{  
  _id: 5,  
  summary: "Contact sellers",  
  description: "Contact agents  
    to specify our ...",  
  due_date: ISODate(),  
  status: "NOT_STARTED",  
  user_id: 1  
}
```



Referencing is the best design:

- Tasks are unbounded items: initially we do not know how many tasks we are going to have
 - A user can end with thousands of tasks
 - Maximum document size in MongoDB: 16 MB !
- Tasks can be queried without needing to retrieve the user details

Example 3

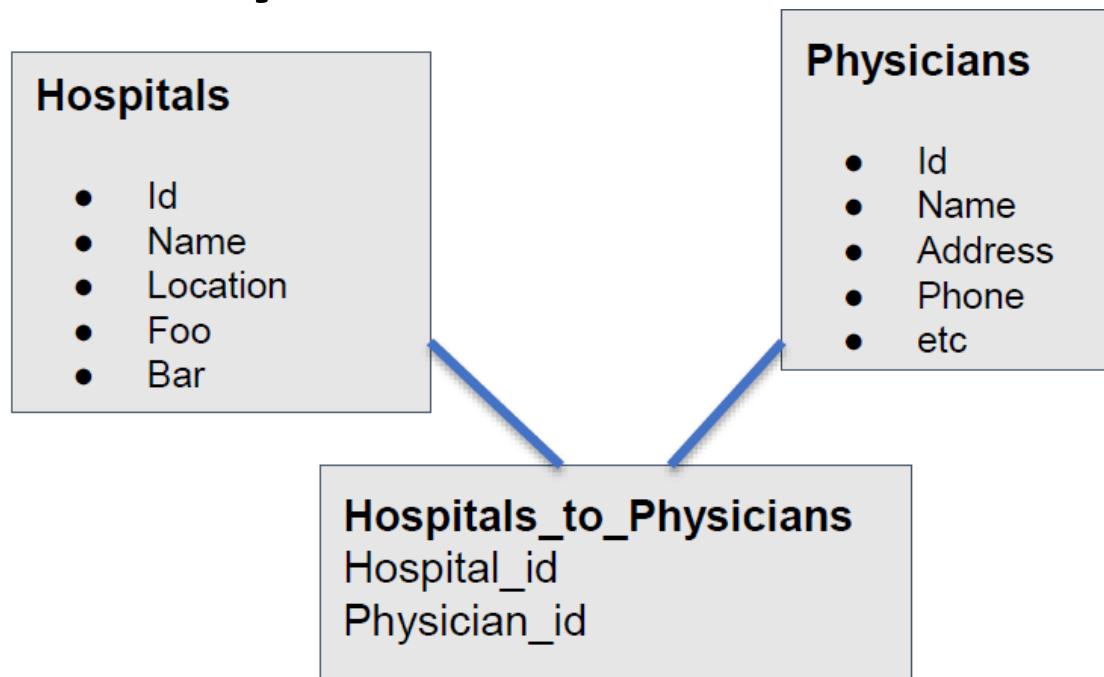
- **One-to-Many “Book”, “Publisher”**
 - A book has one publisher
 - A publisher publishes many books
- **If embed “Publisher” inside “Book”**
 - Repeating publisher info inside each of its books
 - Very hard to update publisher's info
- **If embed “Book” inside “Publisher”**
 - Book becomes an array (many)
 - Frequently update and increases in size

Referencing is better
in this case

Many to Many Relationship

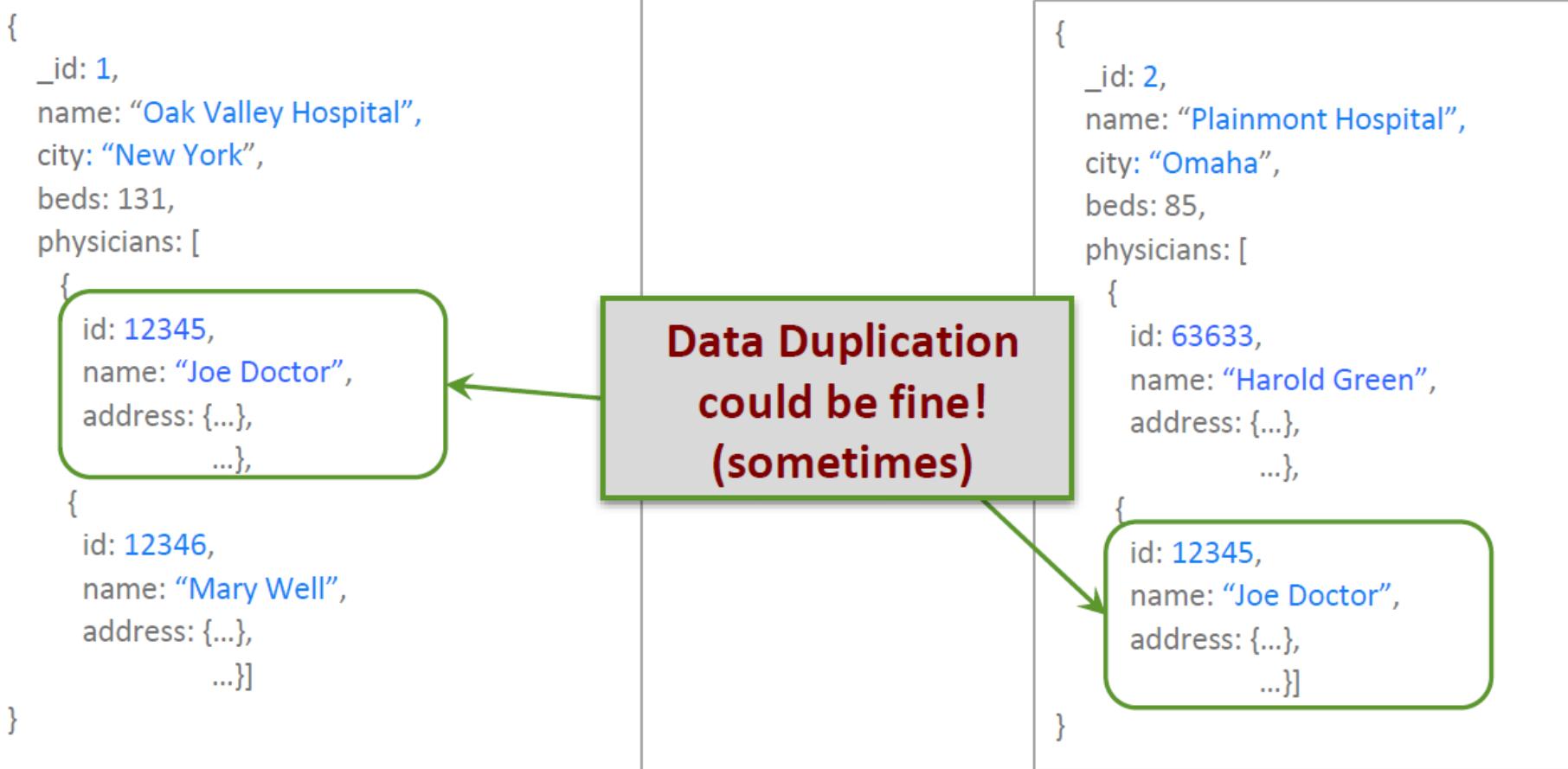
- Like a One-to-Many relationship, you can embed sub-documents or reference them.
- Which approach you take depends on data access patterns and document sizes.

The relational way:



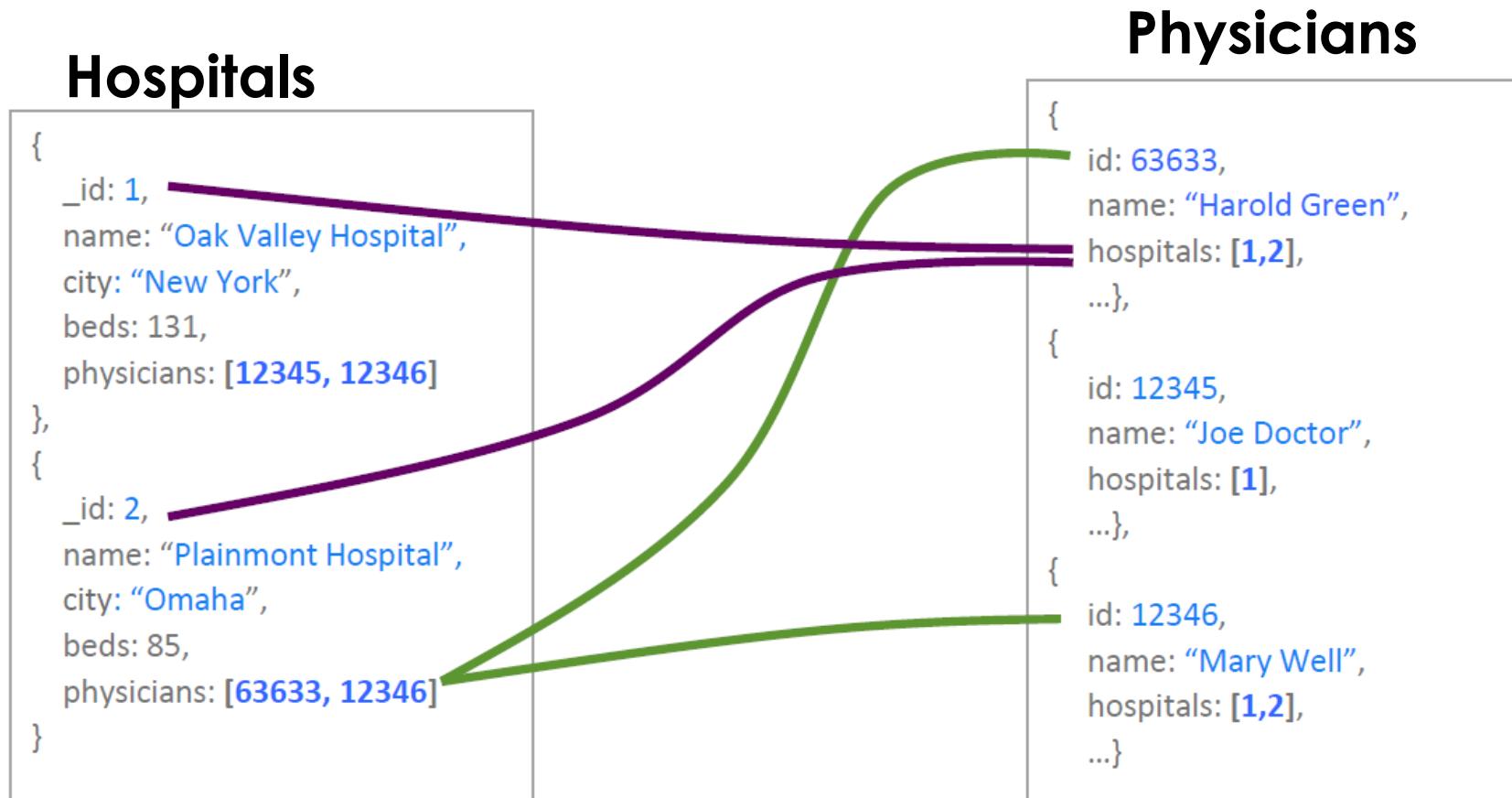
Many to Many Relationship using Embedding

Embedding Physicians in Hospitals collection



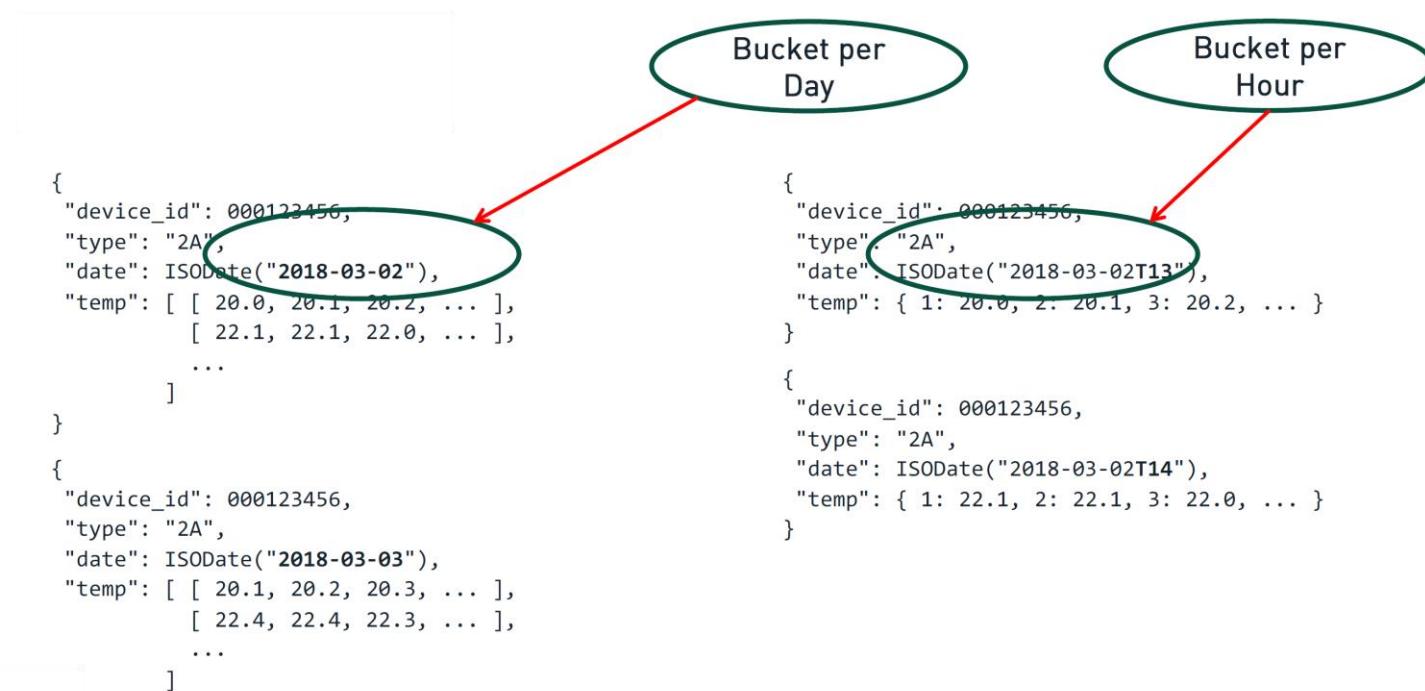
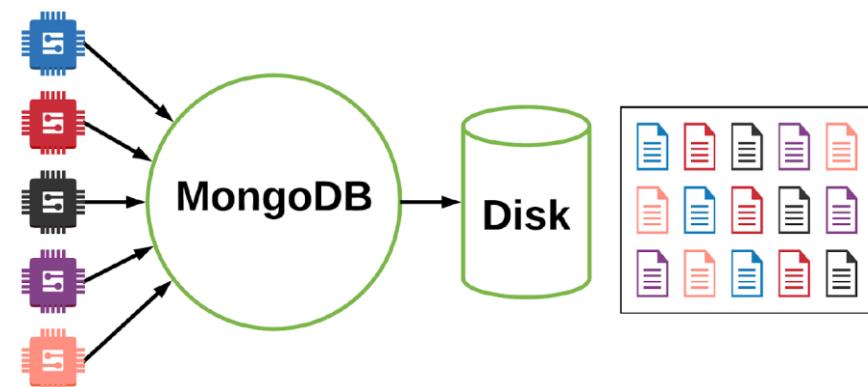
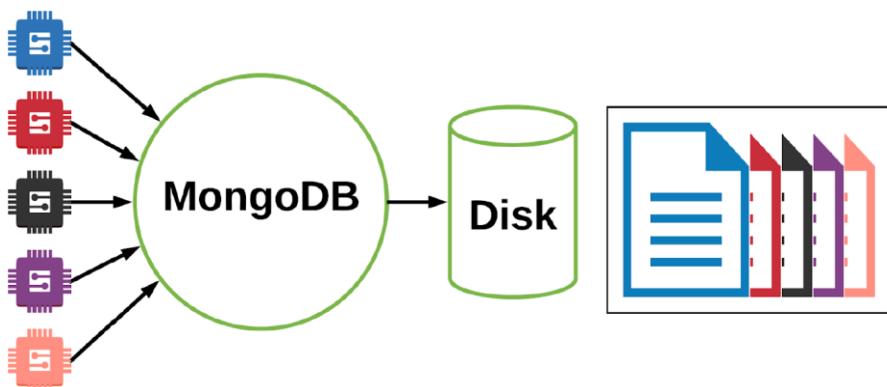
Many to Many Relationship using References

No data duplication. Hence this design is often **recommended**



Note that references can go either direction. There is no primary key/foreign key concept in MongoDB

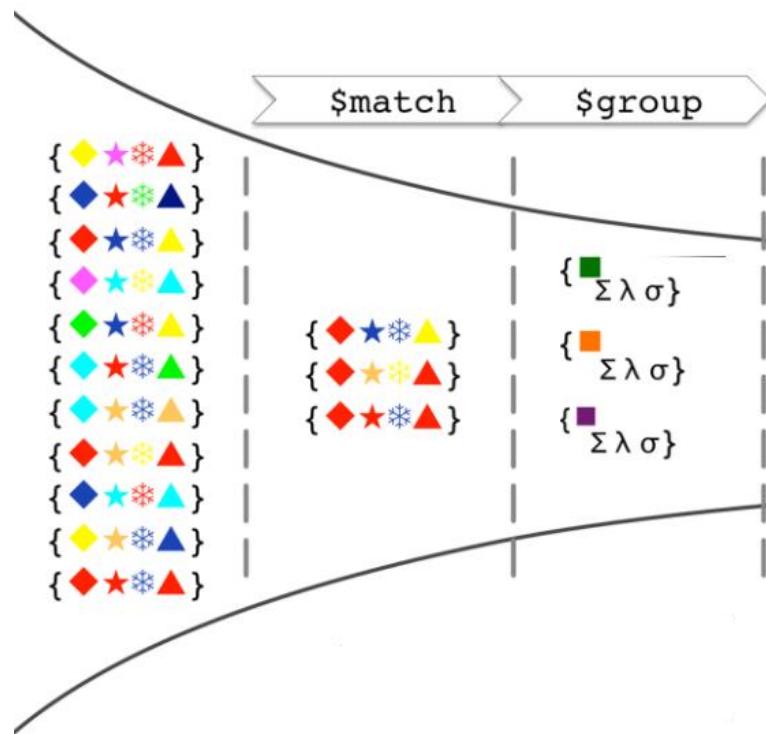
Bucket Pattern

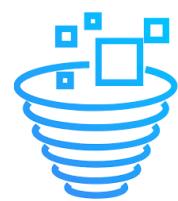


Summary

Type of Relation ->	one-to-one/1-1	one-to-many/1-N	many-to-many/N-N
Document embedded in the parent document	<ul style="list-style-type: none">• one read• no joins	<ul style="list-style-type: none">• one read• no joins	<ul style="list-style-type: none">• one read• no joins• duplication of information
Document referenced in the parent document	<ul style="list-style-type: none">• smaller reads• many reads	<ul style="list-style-type: none">• smaller reads• many reads	<ul style="list-style-type: none">• smaller reads• many reads

Aggregation Queries





Aggregation Queries

- Summarize data typically for reports
- How would we solve this in SQL?

SELECT **GROUP BY** HAVING

- What About MongoDB?

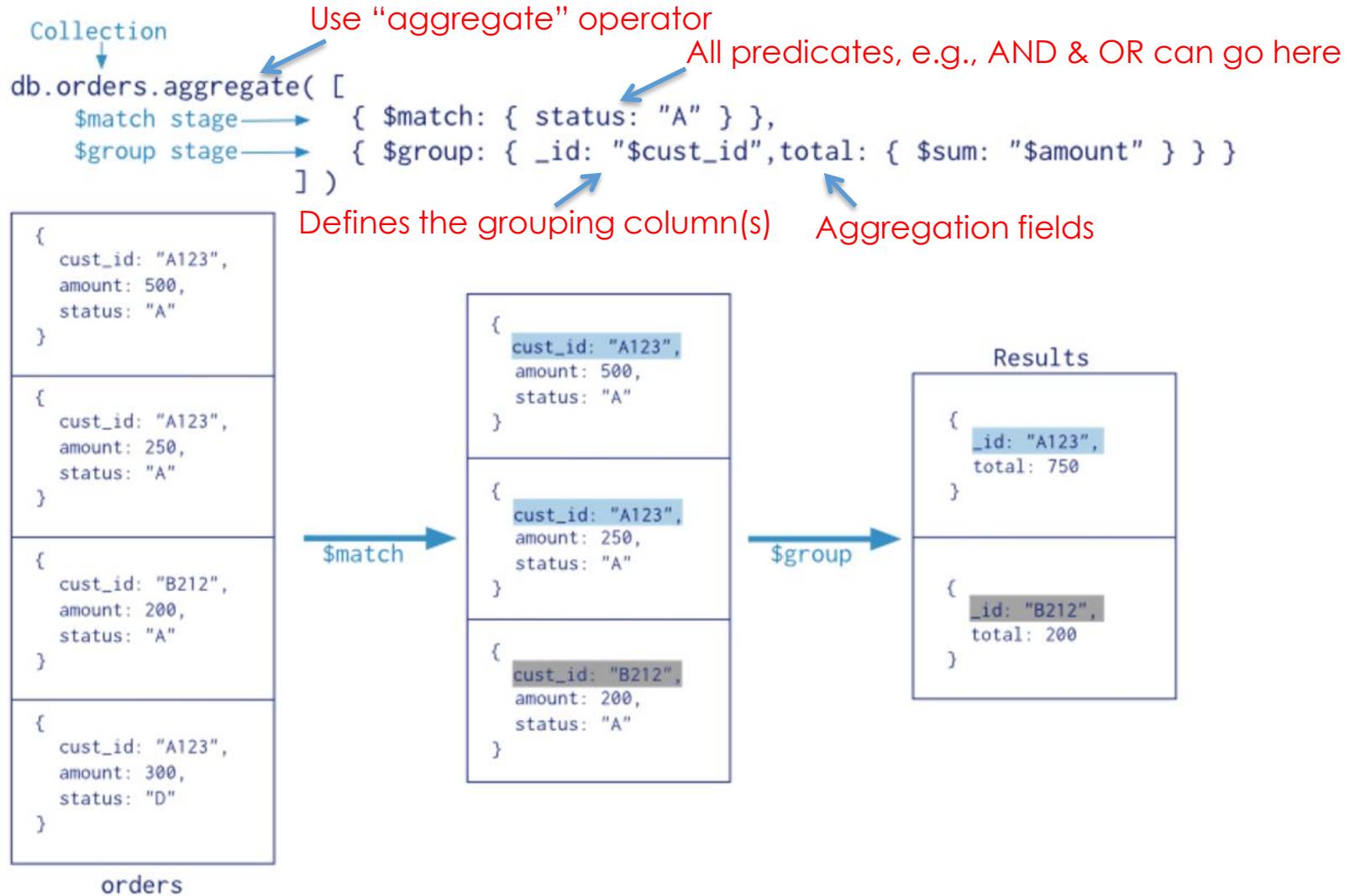
=> **Aggregation Pipeline**



- Pipeline of functions to filter, group, and sort documents
- Operations executed in sequential order
- Output of one stage is used as an input of next

Aggregation Pipeline

- Allows filtering then grouping documents by specific fields



Pipeline Operators

- `$match` Filter documents
- `$group` Summarize documents
- `$sort` Order documents
- `$limit` Limit returned results
- **`$group`** specifies:
 - Properties to group by
 - Computed output properties using `$max`, `$min`, `$avg`, `$sum` ...

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }

{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }

{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }

{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }

{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
```

For each day, get the:

- TotalPrice ↪ Sum (Price * Quantity)
- average quantity
- Count

```
db.sales.aggregate([
    {$group : {$_id : { month: { $month: "$date" },
                        day: { $dayOfMonth: "$date" },
                        year: { $year: "$date" } },
               totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
               averageQuantity: { $avg: "$quantity" },
               count: { $sum: 1 }
             }})
])
```

Group By...Having

- In MongoDB → \$match operator after the \$group

```
{ "_id": "10280",
  "country": "USA",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [ -74.016323, 40.710537]
}
```

Select state, sum(pop)
From collection
Where country = "USA"
Group By state
Having sum(pop) > 10,000,000;

Group By...Having

- In MongoDB → \$match operator after the \$group

```
{ "_id": "10280",
  "country": "USA",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [ -74.016323, 40.710537]
}
```

Select state, sum(pop)
From collection
Where country = "USA"
Group By state
Having sum(pop) > 10,000,000;

Get USA states having total population > 10,000,000

```
db.zipcodes.aggregate( [
  { $match: { country: "USA" } },
  { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
  { $match: { totalPop: { $gt: 10*1000*1000 } } }
] )
```

\$group Examples

- Return average GPA for all students

```
Student.aggregate([
  {
    "$group" : {"_id" : null,
    "avgGPA" : {"$avg" : "$gpa"}}
  }])
```

- Return total completed Credit Hours per student

```
StudentCourse.aggregate([
  {
    "$group" : {"_id" : studentId,
    "completedCHs" : {"$sum" : "$CourseCH"}}
  }])
```



JONGO

Query in **Java** as in **Mongo** shell

<https://jongo.org/>

Database Scalability Solutions

What is scalability?

*A service is said to be scalable if when we **increase the resources** in a system, it **results in increased performance** in a manner proportional to resources added.*

Werner Vogels CTO - Amazon.com

Scale up



Scale out

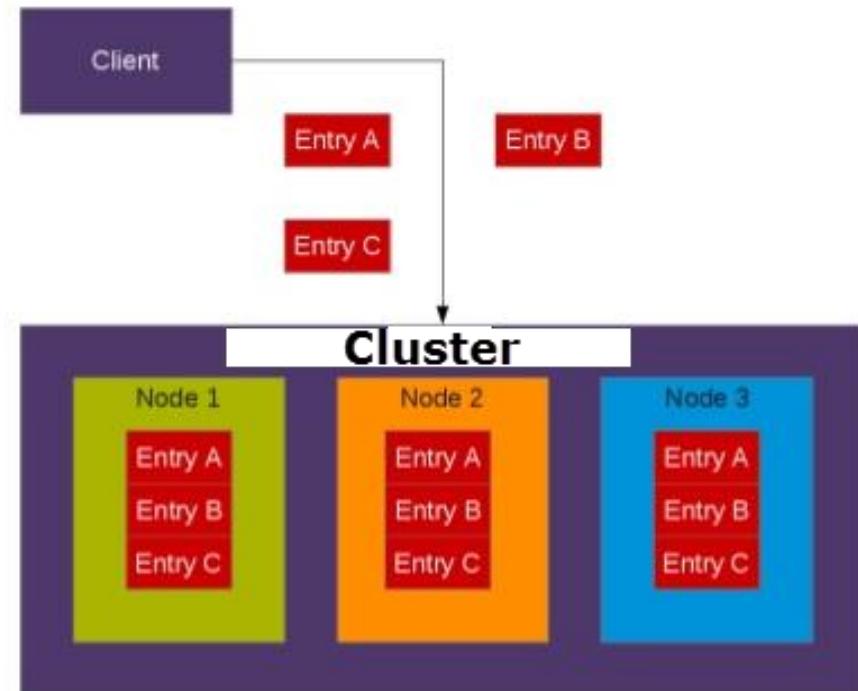
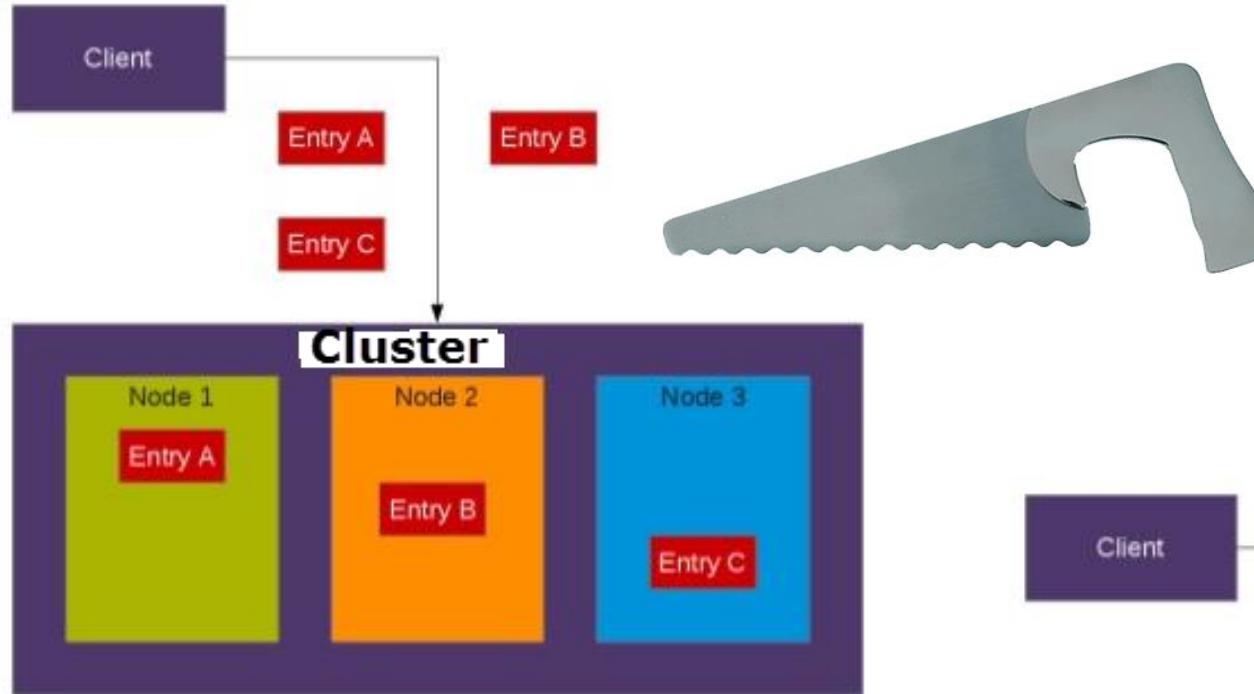


Solutions to address scalability

Two styles of **distributing data** to enhance scalability:

- **Sharding** (aka Data Partitioning) distributes different data across multiple servers
 - each server acts as the single source for a subset of data
- **Replication** copies data across multiple servers
 - each bit of data can be found in multiple places
 - Replication is good for performance & reliability
 - Key challenge = keeping replicas up-to-date

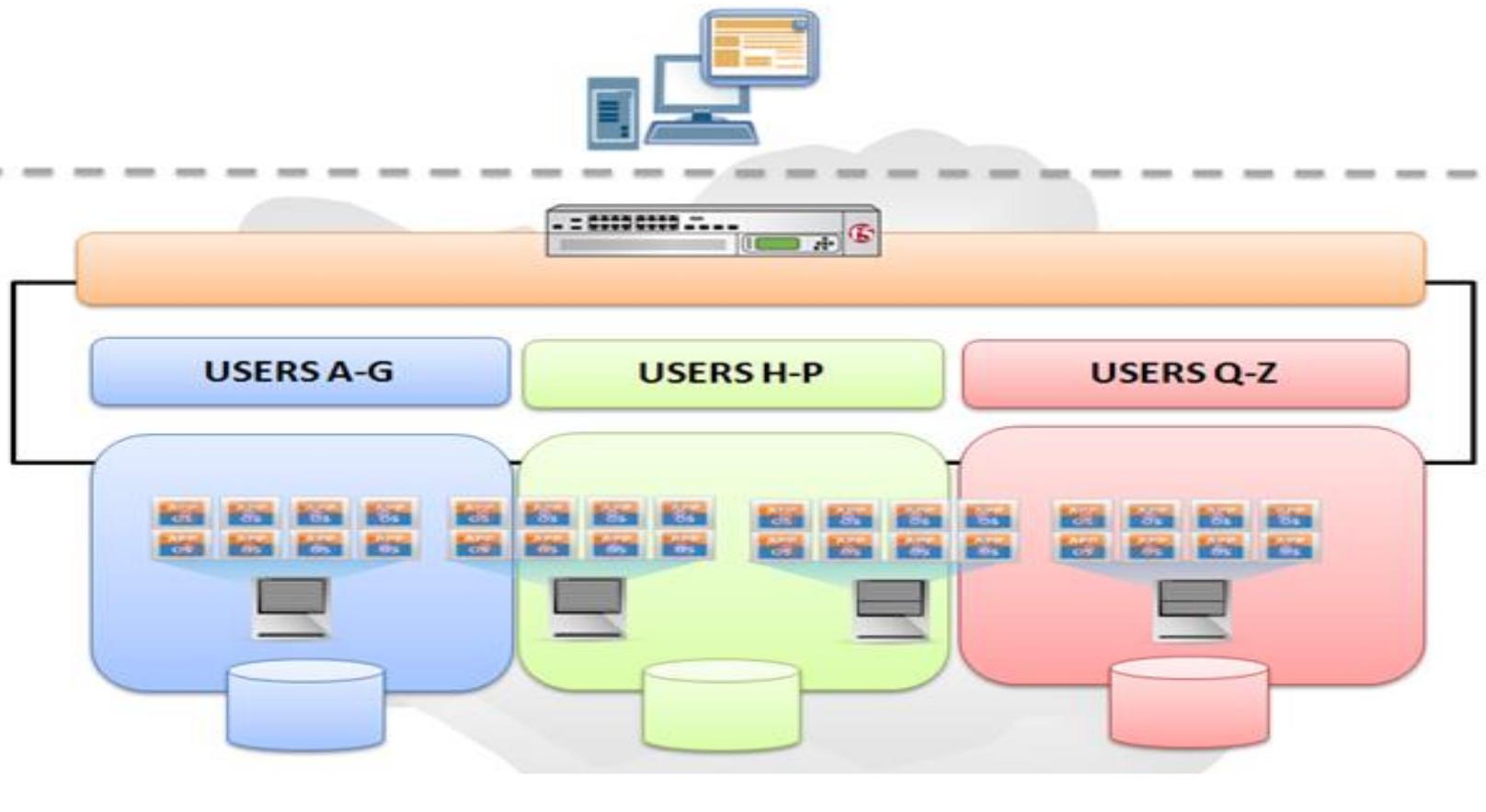
Sharding vs. Replication



What is Sharding?

- **Problem:** one database can't handle all the data
 - Too big, poor performance, needs geo distribution, ...
- **Solution:** split data across multiple databases
 - One **Logical Database**, multiple **Physical Databases**
 - *Scales well for both reads and writes*
- Each Physical Database Node is a **Shard**

Database Sharding



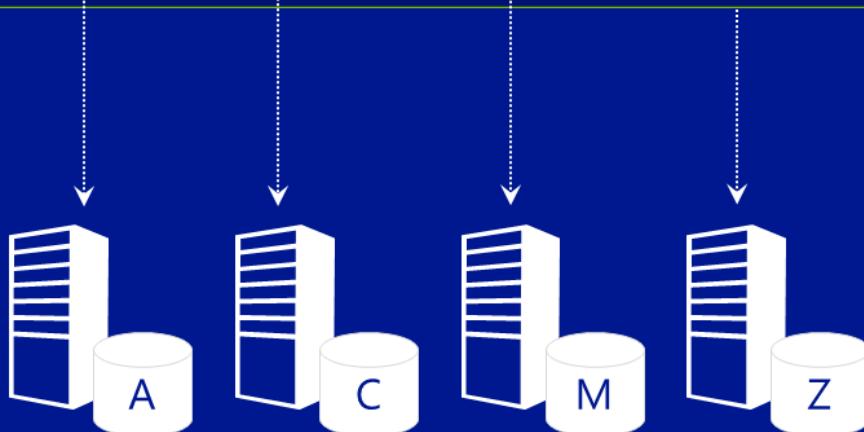
COURTESY F5 NETWORKS
(CC BY-ND)

All shards have same schema

Horizontal Partitioning

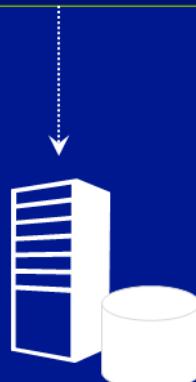
- Horizontal or Range Based Sharding: the data is split based on the value ranges of a particular attribute such as the last name.

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contoso.com	3kb	3MB
Sue	Charles	suec@contoso.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contoso.com	3kb	3MB



Vertical Partitioning

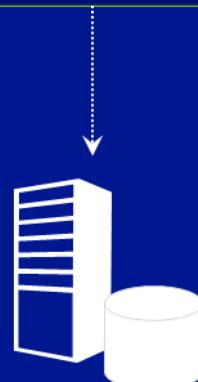
First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.com	3kb	3MB



RDMS Server



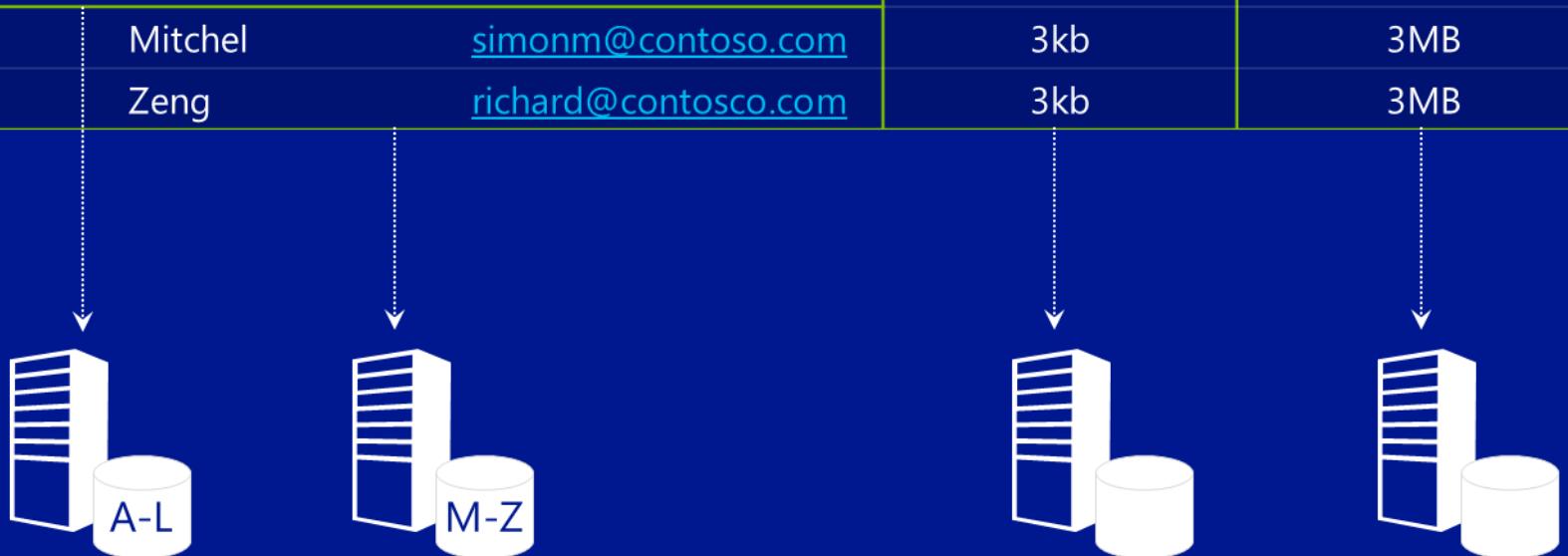
BLOBS



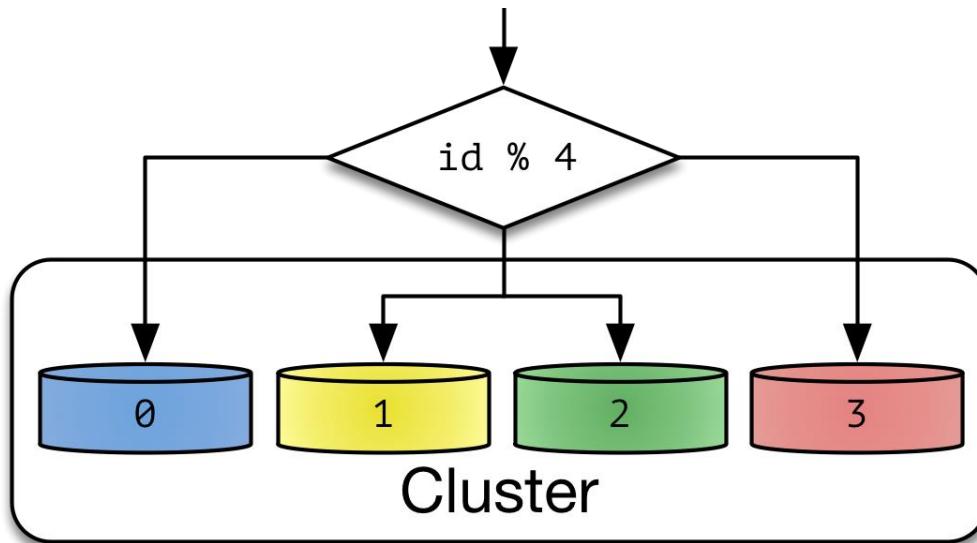
BLOBs

Hybrid Portioning

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.com	3kb	3MB

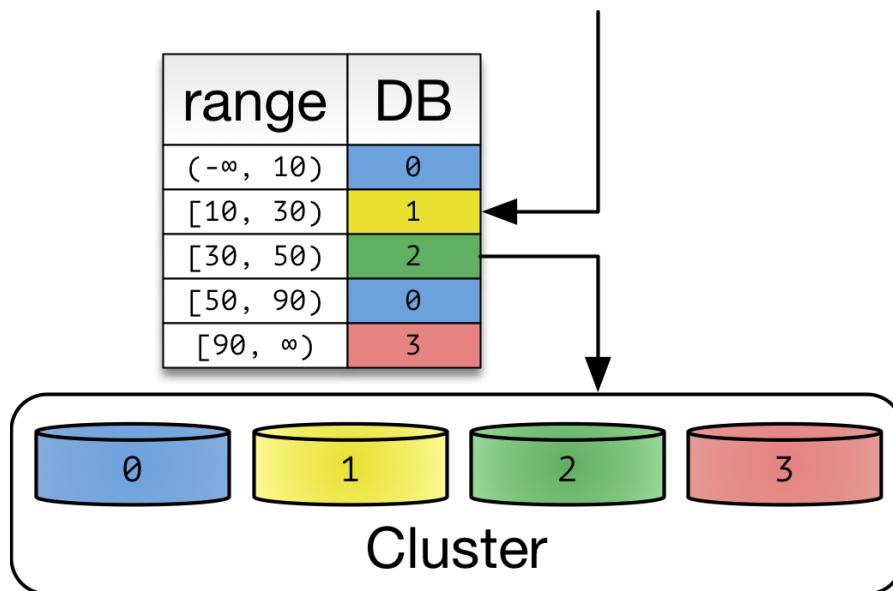


Algorithmic versus Dynamic.



$\text{hashValue} = \text{hashFunction(key)}$
 $\text{serverIndex} = \text{hashValue \% number_of_servers}$

- Algorithmic sharding:
 $\text{hash(key)} \rightarrow \text{Node}_{\text{id}}$



- Dynamic sharding: uses a **locator service** to map key ranges to nodes

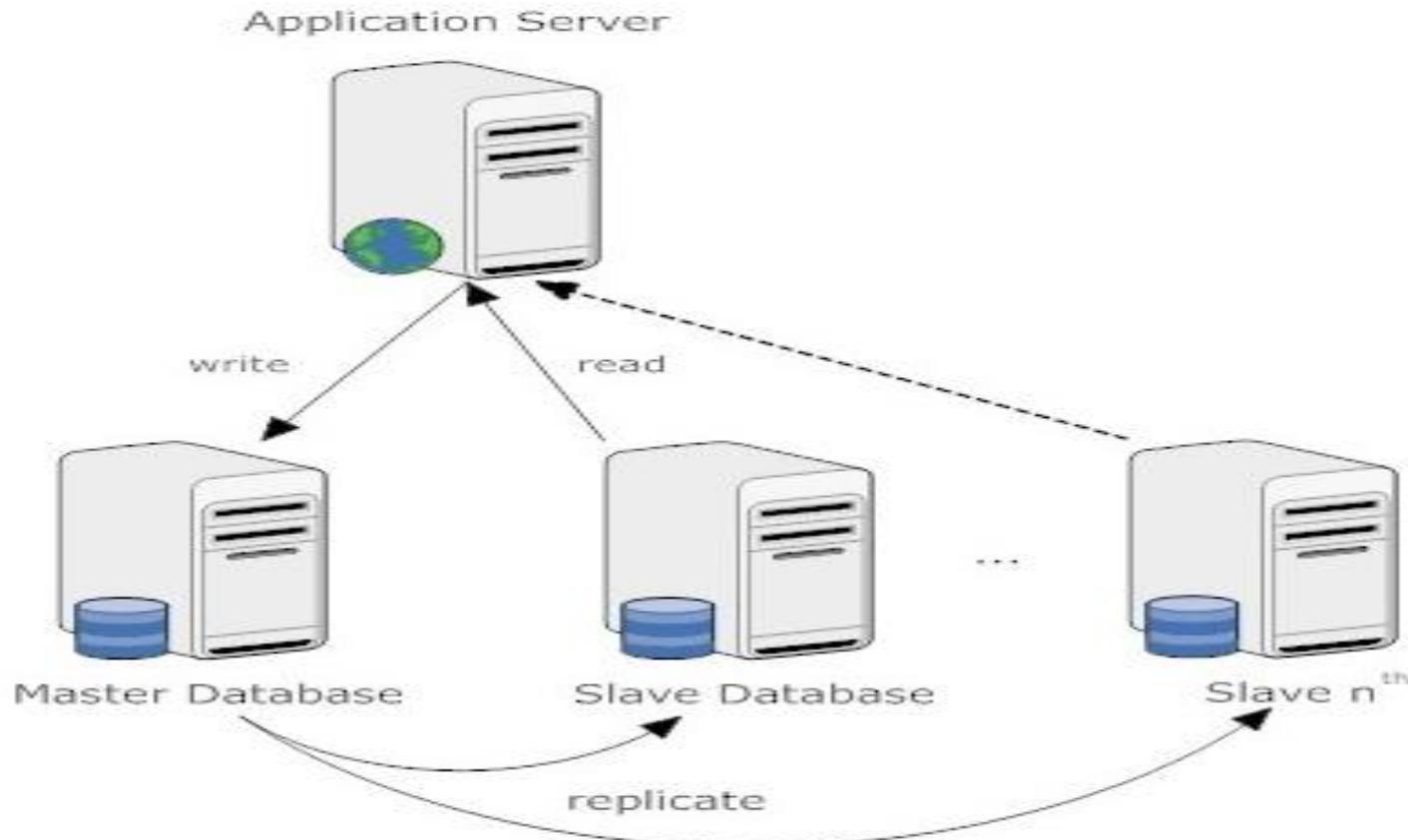
Sharding Challenges

- For most RDBMS, sharding is NOT transparent, the application needs to be **partition-aware**
 - Sharding is largely managed outside RDBMS
 - Recent version of RDBMS may provide limited support for sharding
- Joins become too expensive
- **Loss of referential integrity across shards. So the constraints are moved away from database and are part of the application**
- Re-balancing or Re-Sharding is hard
 - What to do when data do not fit in one shard
- Deciding on a partition key/plan is hard
 - May generate hotspots

Sharding is Difficult

- What defines a shard? (Where to put stuff?)
 - Example – use country of origin: customer_us, customer_fr, customer_cn, customer_ie, ...
 - Use same approach to find records
 - May generate hotspots
- What happens if a shard gets too big?
 - Rebalancing shards can get complex
- Query / join / transact **across** shards

Master-Slave Replication



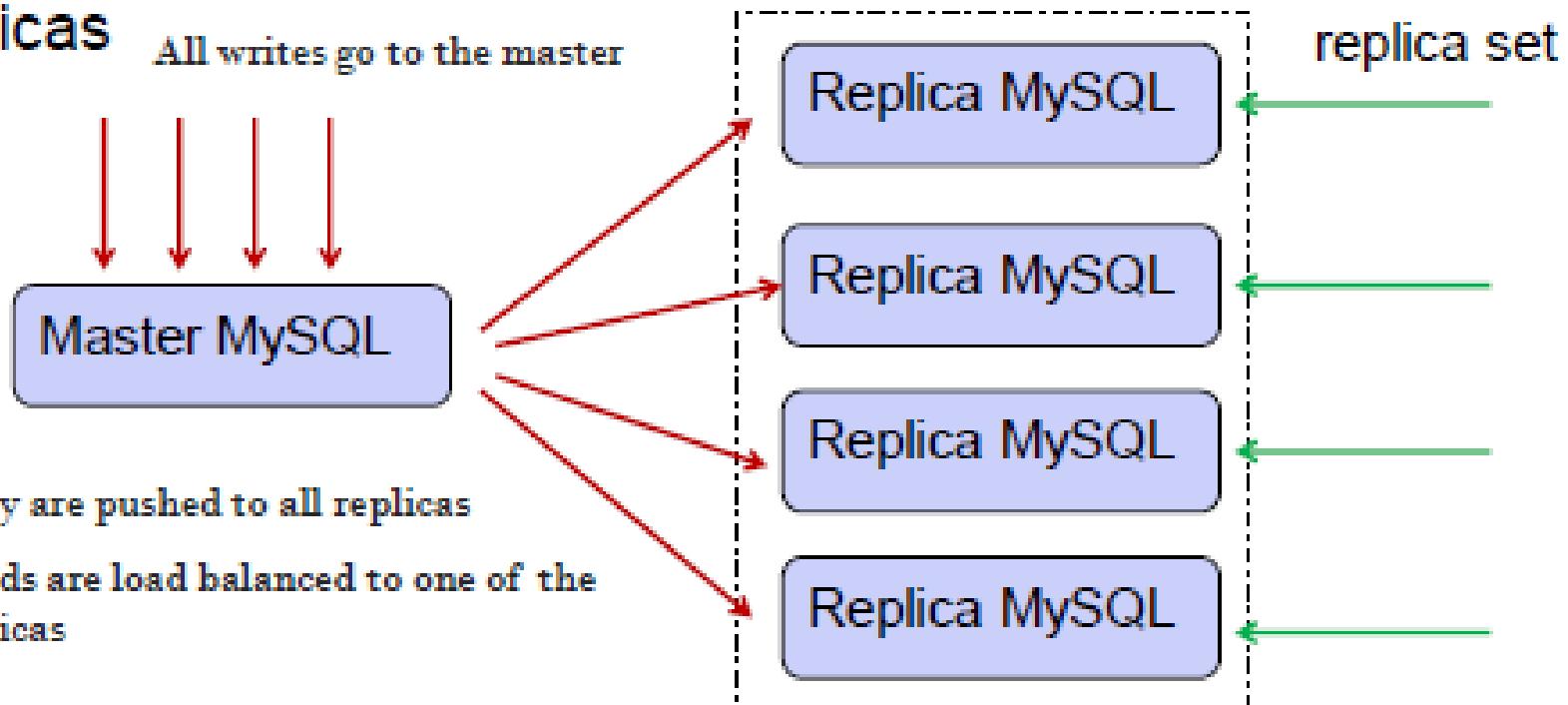
- Replication increases resilience and availability but introduces new class of **consistency** problems

Master-Slave Replication

- Makes one node the **authoritative copy** that **handles writes** while slaves synchronize with the master and may handle reads
 - Reads **may be inconsistent** as writes may not have been propagated to slaves
 - **Large data sets** can pose problems as master needs to **duplicate data to slaves**

Master-Slave Replication Example

- Example: Wikipedia has one Master database and many replicas



<http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf>

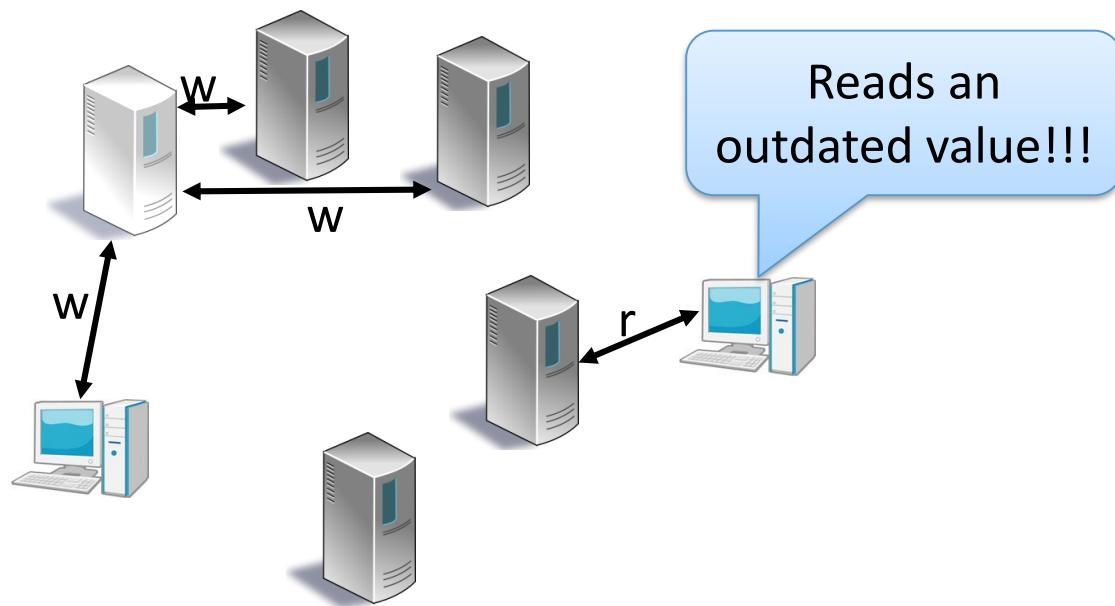
Master-Slave Replication Implications

- When the master dies
 - One of the replica can be elected as the new master
- Some read may return old data if the latest value has not been pushed from the master
- It is possible to let Master handle read request for data requiring **strong consistency**
- Relatively easy to setup in most RDBMS

Problems with Master-Slave Replication

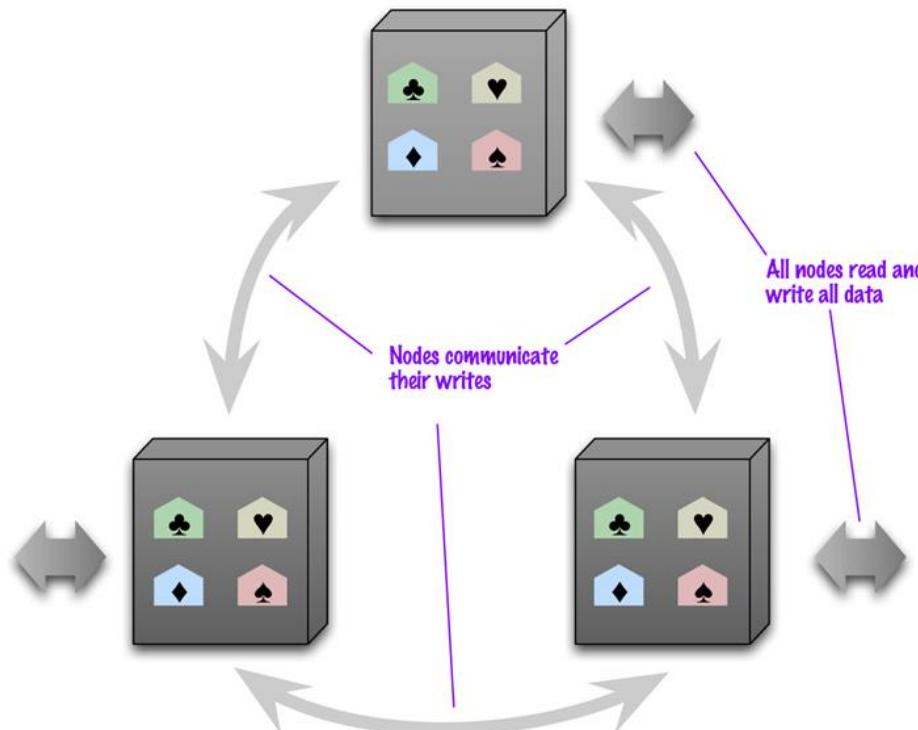
- If the clients can only send read requests to the Master, the system stalls if the Master fails
- However, if the clients can also send read requests to the other servers, the clients may not have a **consistent view**

- Master-slave replication helps with read scalability but doesn't help with scalability of writes



Peer-to-Peer Replication

- *Peer-to-peer replication* allows writes to any node; the nodes coordinate to synchronize their copies of the data.



Replication: consistency of copies

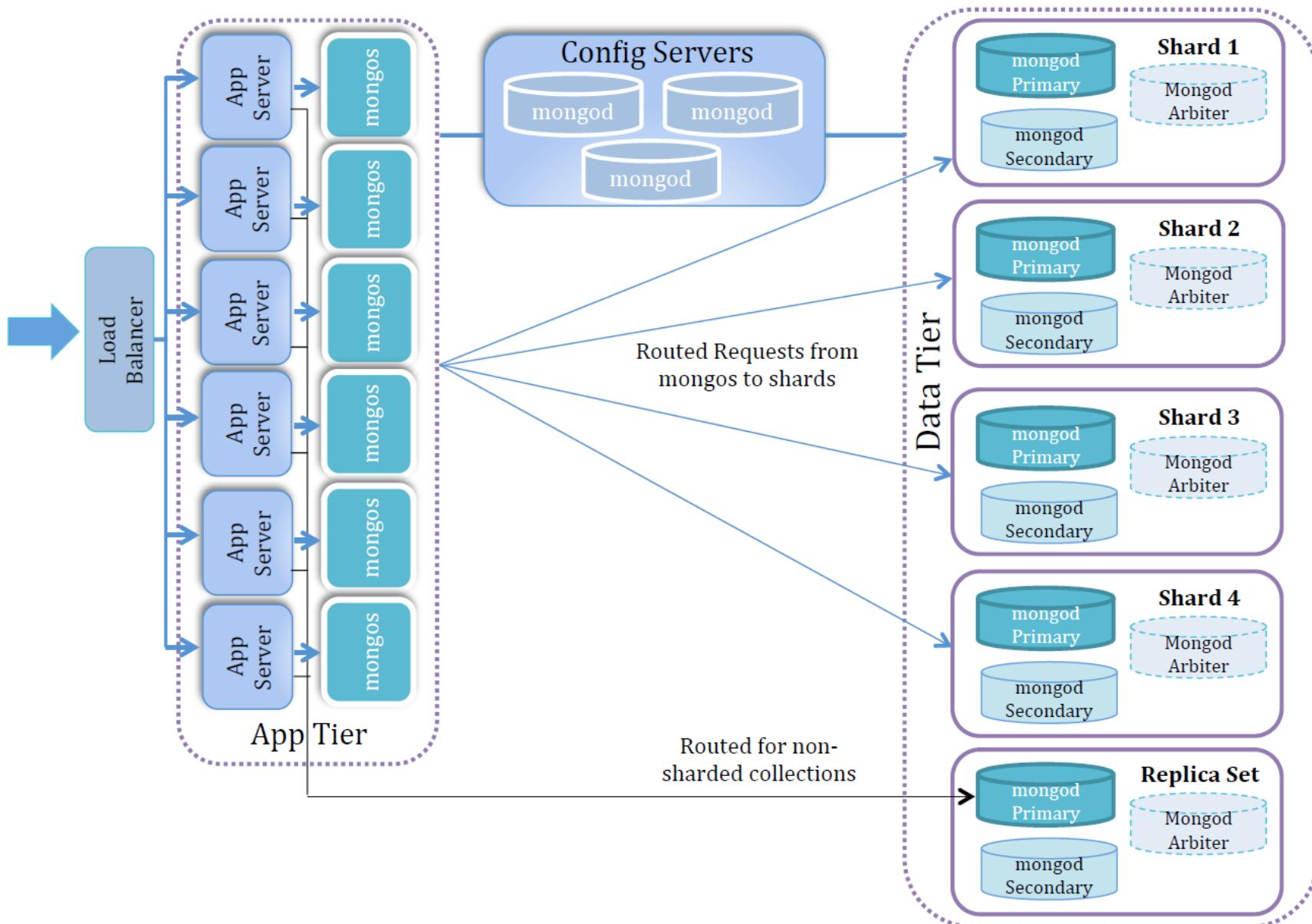
- **Synchronous:** All copies of a modified data item must be updated before the modifying transaction commits
 - copies are consistent
- **Asynchronous:** Copies of a modified data item are asynchronously updated
 - copies may be inconsistent over periods of time.

Summary

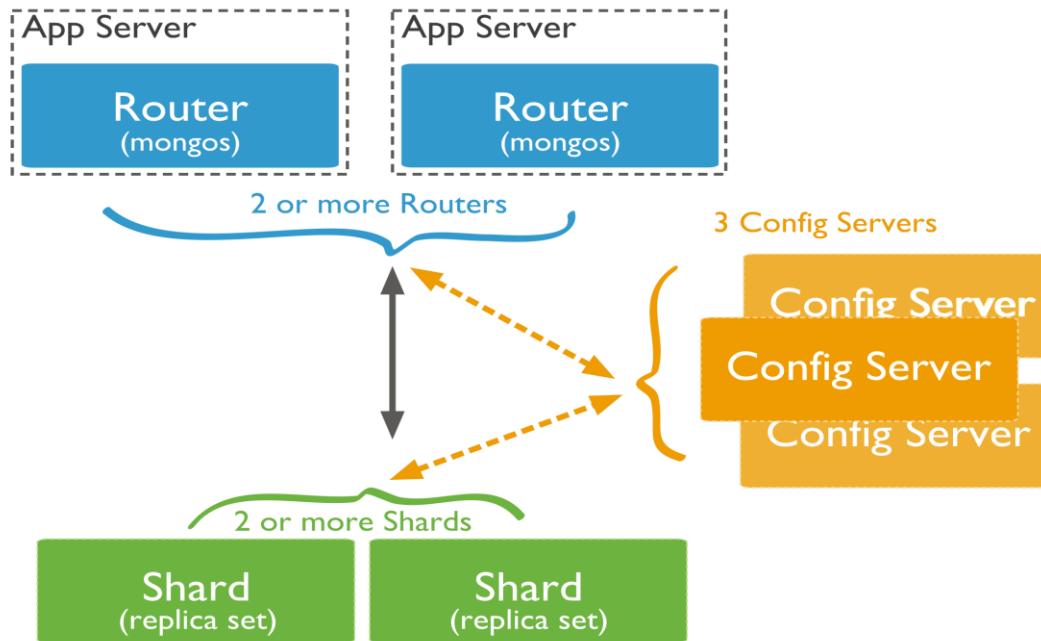
- Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single point of failure.
- Replication and sharding are strategies that are often combined.

MongoDb Sharding & Replication

MongoDB Scalability Architecture



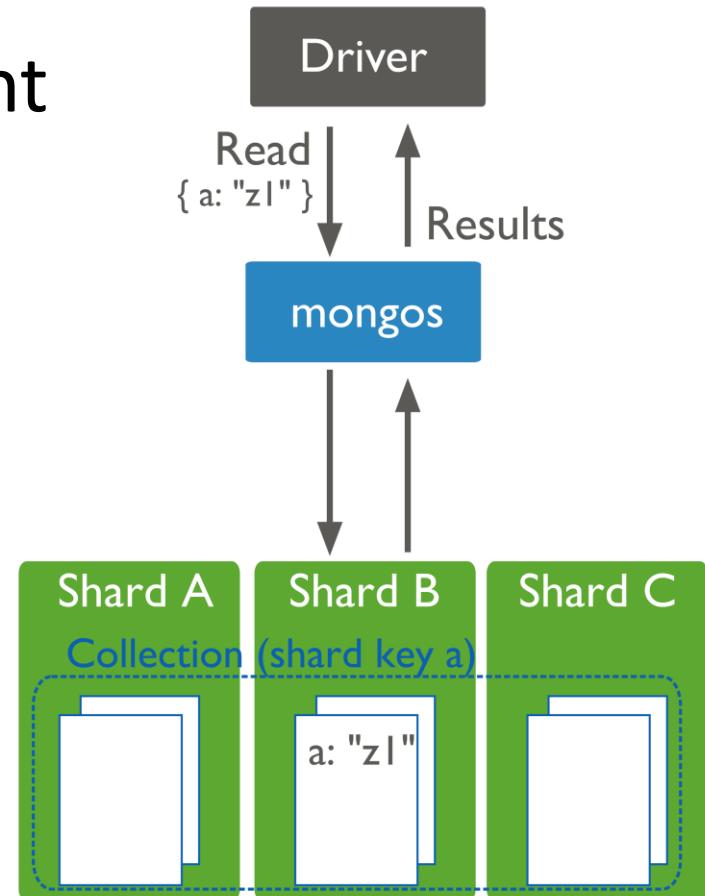
MongoDB Sharded Cluster



- Shard: storing data, can be replicated (replica set)
- Config Server: Storing metadata info (e.g., list of chunks on every shard and the ranges that define the chunks)
- Router: Accepts and routes client's queries & update operations

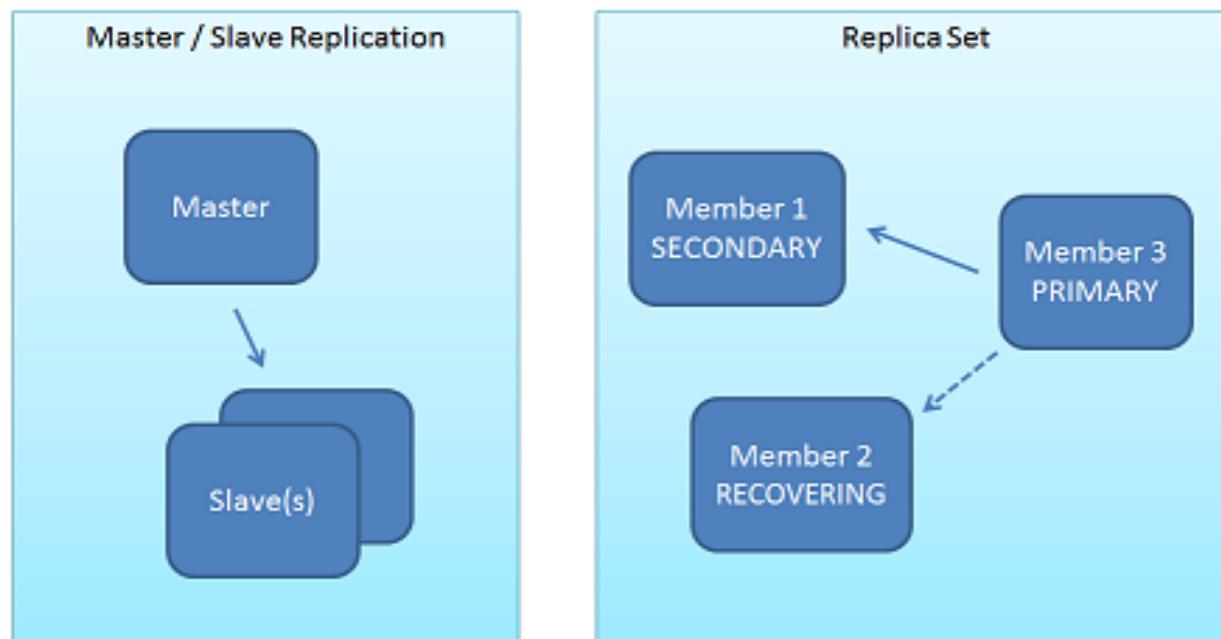
Routing Operations to Shards

- Read/write operations are sent from client to mongos
- Mongos routes them to the appropriate shard(s)



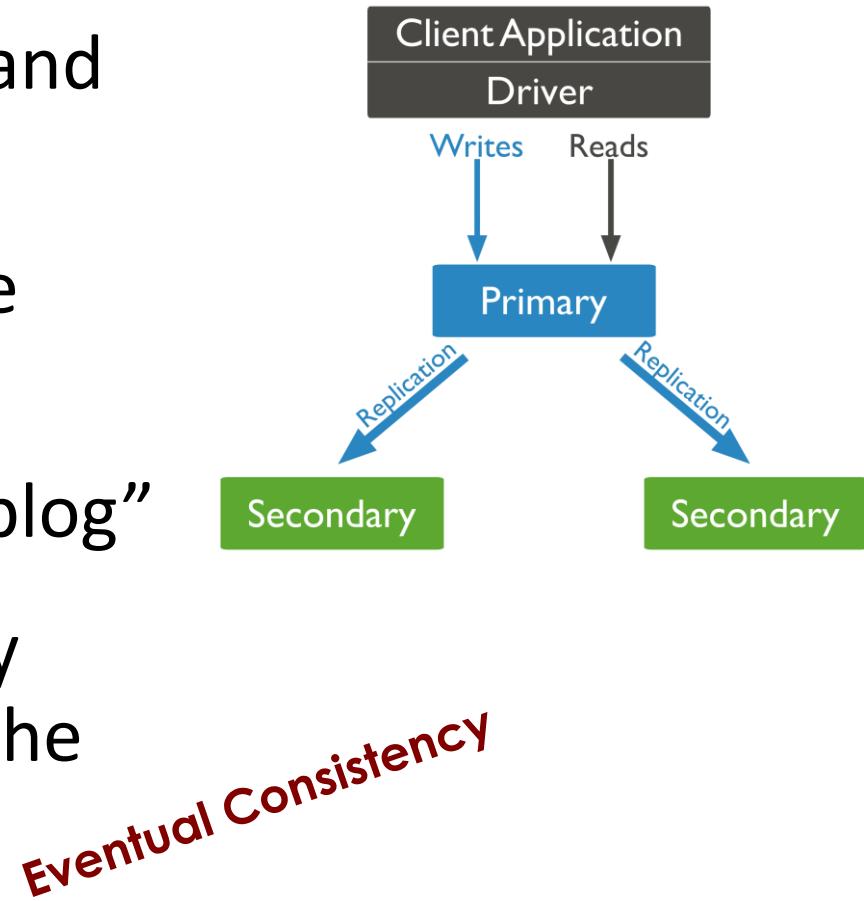
Replication

- **Replica Set**
 - Similar in concept to Master-Slave architecture



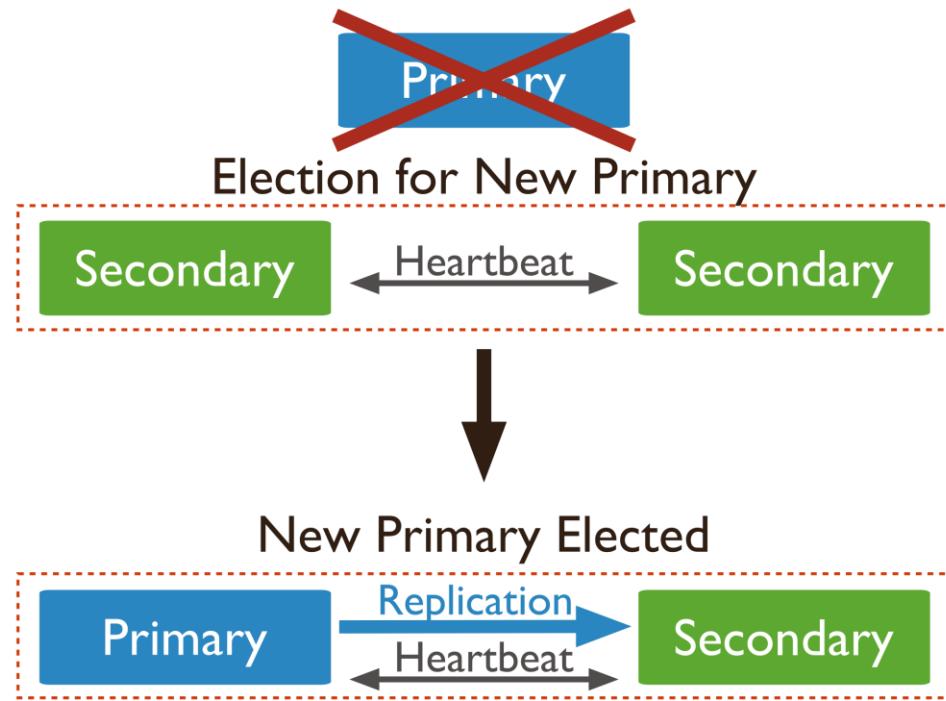
Replica Set

- Goal: Improved Availability, Fault Tolerance, Load Balancing
- Consists of one “**Primary**” and multiple “**Secondary**”
- All write ops must go to the primary
- Primary maintains a log “oplog”
- Secondary sites periodically read & apply the log from the primary node

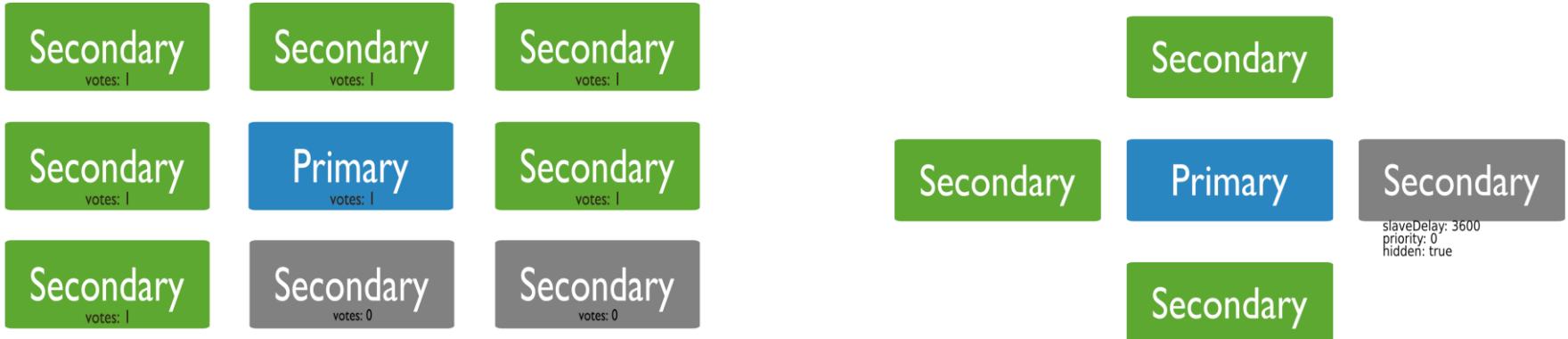


Election when Primary Fails

- Based on majority voting
- Number of members should be odd
- During election, no writes are accepted



Configuring Secondary Nodes

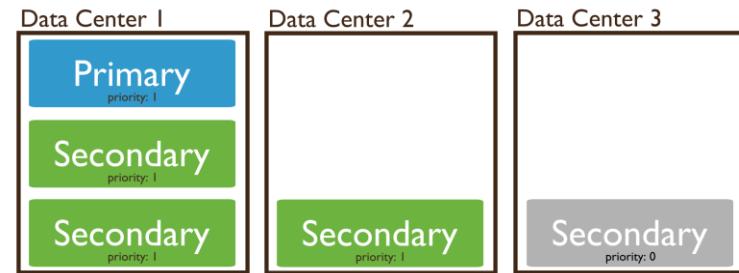


- Number of secondaries
- Priority = 0 ← cannot be elected as primary
- Hidden = True ← Cannot serve client operations
- SlaveDelay = m ← waits m msec before getting the updates from the primary site

Configuring Secondary Nodes

- **Priority = 0**

- Cannot be primary
- Cannot accept write
- Still has data & accept reads
- May want some data centers not to accept write ops



- **Hidden = True**

- Implies Priority = 0
- But also cannot accept reads from clients
- Good for dedicated offline tasks, e.g., reporting

- **SlaveDelay = m**

- Should be Hidden = True
- Good to recover from bad transactions

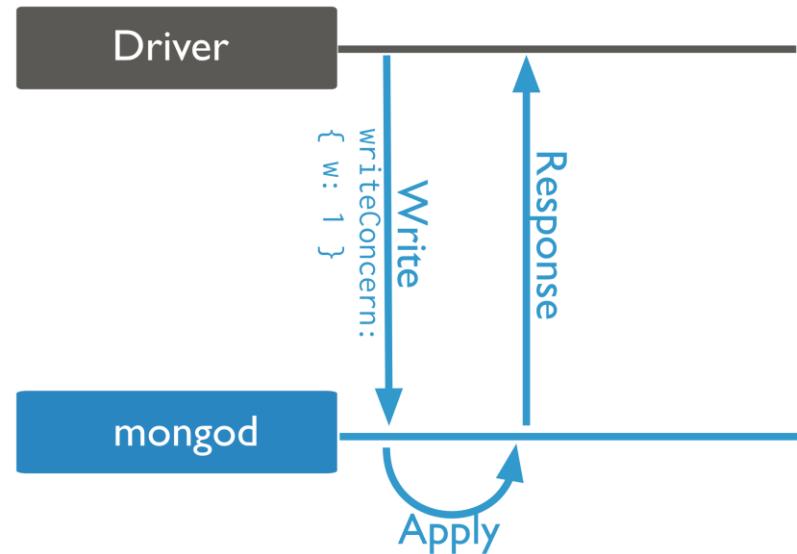
Writing/Reading: Default Behavior

- **Write**

- All writes go to the primary
- A write is accepted once the primary accept op. (in memory)
- Secondaries are not updated yet

- **Read**

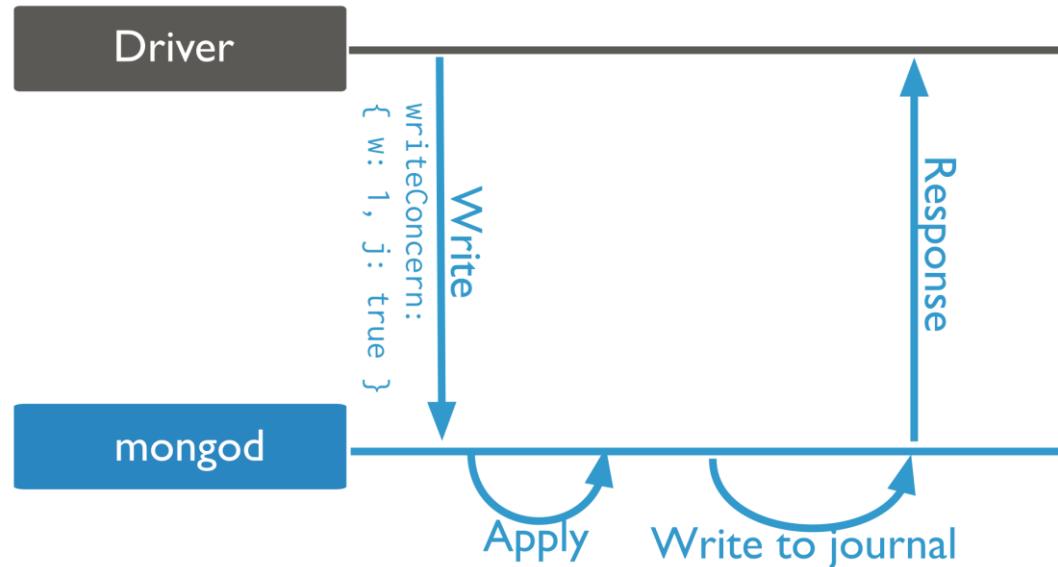
- All reads go to the primary
- Ensures *Strict Consistency*



Accepted data can be lost

In this case Secondaries are mostly for
Availability & Fault Tolerance

Journaling: Persistent Data



- As before, but a write is accepted only after written to a log on disk
- Still on the primary site
- Accepted data become persistent

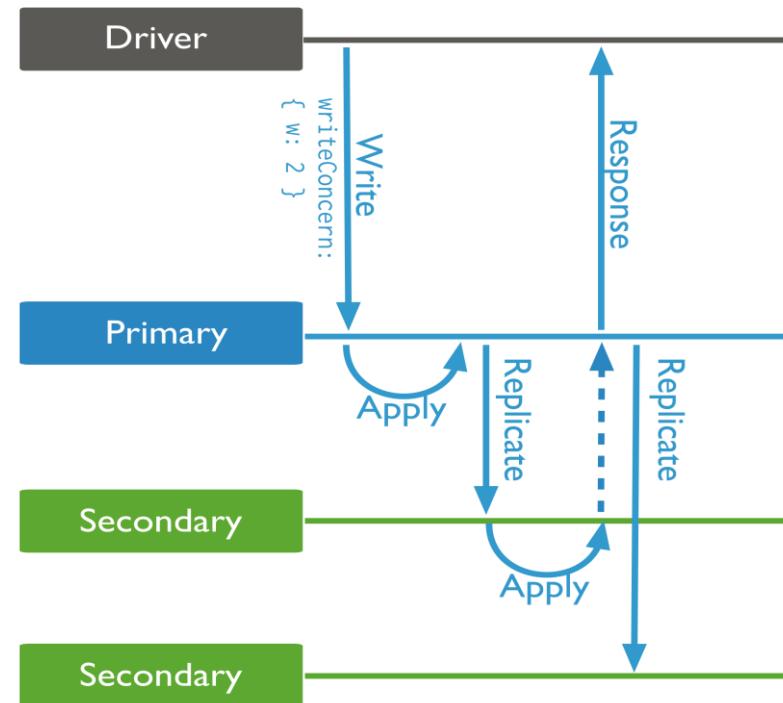
Higher Consistency For Reads

- **Option 1: Read From Primary**

- Keep writing as is
- Enforce the read from Primary
- → Strict Consistency

- **Option 2: Expensive Write**

- Write is not accepted until m secondaries are also updated



```
db.products.insert(  
    { item: "envelopes", qty : 100, type: "Clasp" },  
    { writeConcern: { w: 2, wtimeout: 5000 } }  
)
```

Read Modes

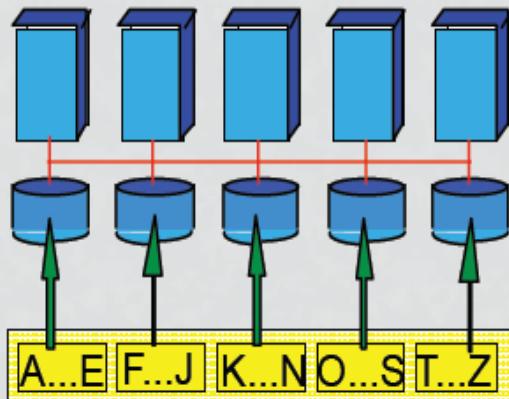
- Primary
- PrimaryPreferred
- Secondary
- SecondaryPreferred
- Nearest

Sharding

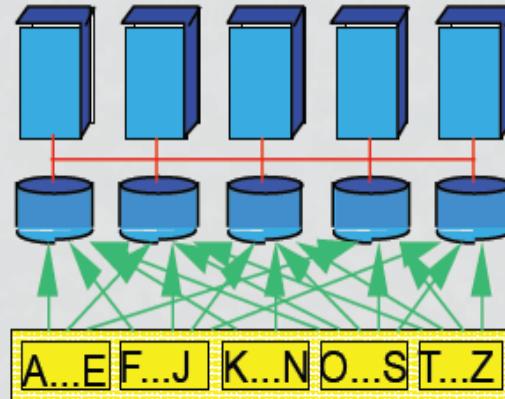
- Partitioning the data across many machine
- Similar Concept in RDBMS
- Orthogonal to “Replication”

To partition a relation R over m machines

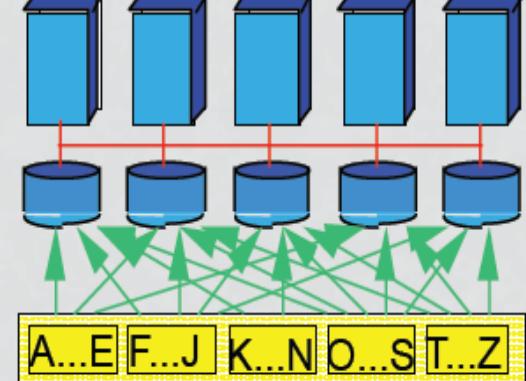
Range partitioning



Hash-based partitioning

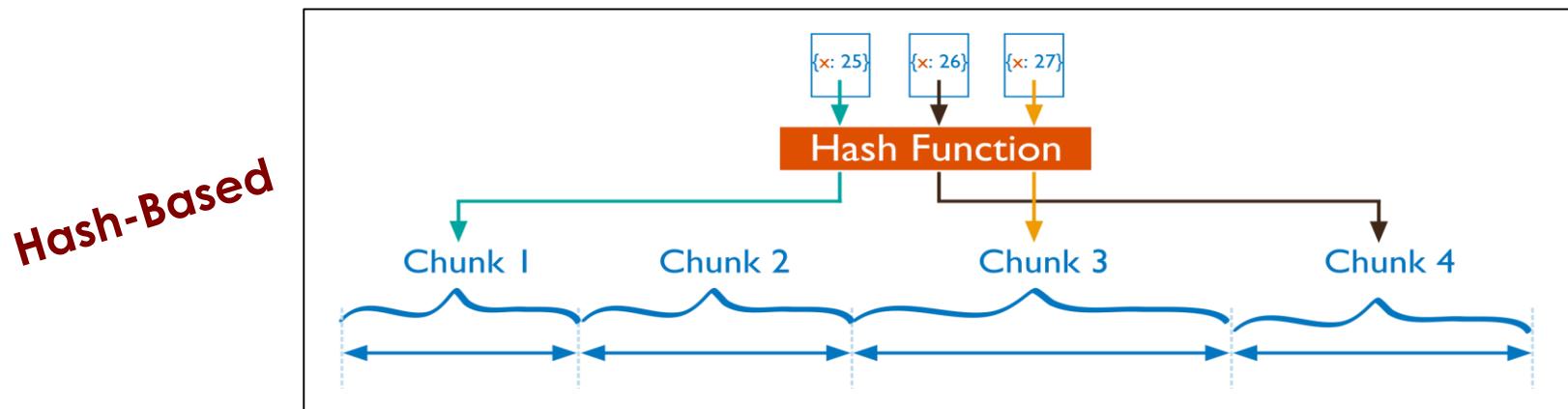
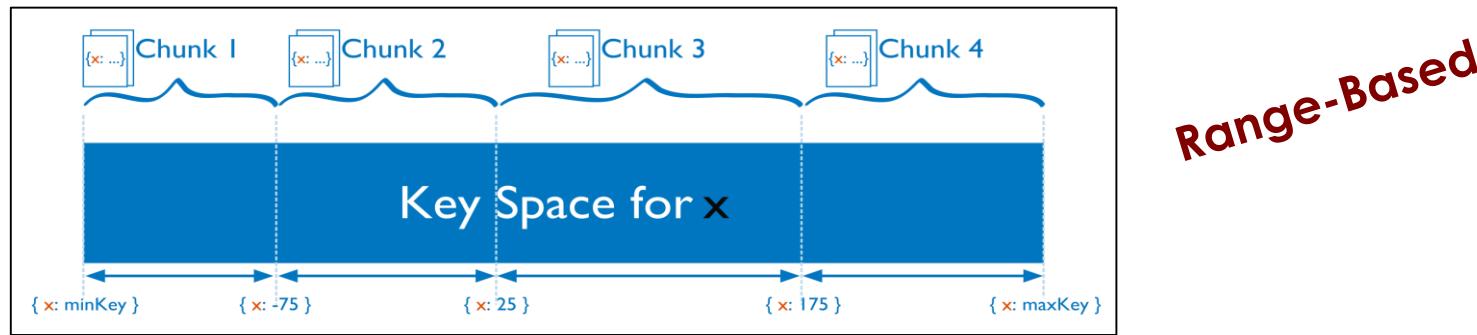


Round-robin partitioning



Shard Key

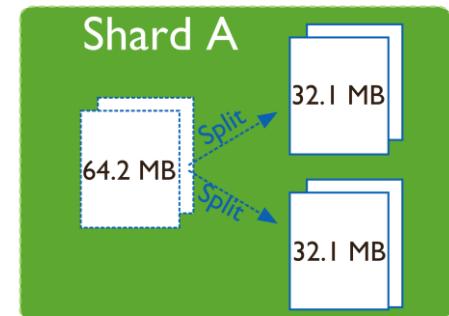
- A collection is sharded based on a **key** into chunks
- **Key:** must be present in each document (and indexed)



Keeping Balanced Shards

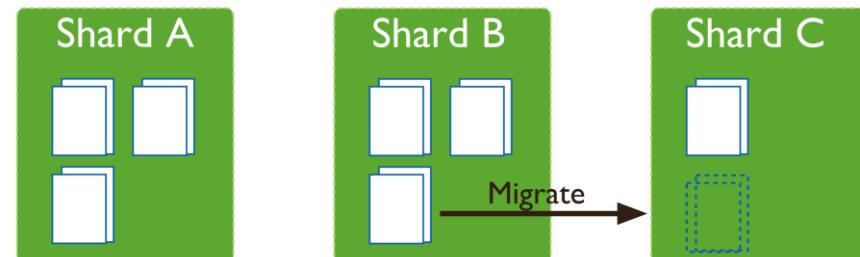
- **Splitter**

- Splits a big chunk into two
- Triggered by inserts/updates



- **Balancer**

- Migrates chunks from one shard to another (least occupied)



Resources

- MongoDB University

<https://university.mongodb.com/>

- Queries Cheat Sheet

http://s3.amazonaws.com/info-mongodb-com/mongodb_qrc_queries.pdf

- Aggregation Queries

<https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>