NoSQL Databases
MongoDB

Dr. Thomas Zahn

June 2025

NoSQL - Definition

- Umbrella term for many, disparate database technologies, e.g.:
  - Document stores (e.g. MongoDB, CouchBase, Amazon DocumentDB)
  - Graph databases (e.g. Neo4J)
  - Key-Value stores (e.g. Redis, Amazon DynamoDB)
  - Wide column stores (e.g. Apache Cassandra)
- NoSQL is even a bit of a misnomer
  - Some NoSQL DBMS (e.g. CouchBase) actually have a SQL-based query API.
- What is really meant is: **Non-Relational** DBMS
- In practice, arguably, the most-widely used NoSQL DBMS are **Document Stores**

Document Stores / Document Data Model

- Data is not stored in flat rows across two-dimensional tables
- Instead, data is stored in collections of documents
- Typically documents are defined as JSON documents
- Documents often contain embedded documents
  - e.g. a `customer` document might contain an embedded `address` sub-document, etc.
- => Note that the data stored is of course still "relational"!
  - it can still be defined by an ERD
- ***But:*** *we deliberately* ***choose*** *to store it in a* ***denormalized*** form

Document Data Model - Example

```
{
  _id: "648c9913e548ee476696d6ef",
  first: "Jane",
  last: "Doe",
  address: {
    street: "100 High St",
    city: "Cambridge",
    zip: "CB2 2LX",
    country: "UK"
  },
  favorite_colors: ["blue", "red"]
}
{
  _id: "648c9980e548ee476696d6f4",
  first: "Peter",
  last: "Müller",
  address: {
    street: "Hauptstr. 10",
    city: "München",
    zip: "80352",
    country: "Germany"
  },
  favorite_colors: ["blue", "green"]
}
```

```
{
  _id: "649d9513e548ee476696d6ab",
  first: ”John",
  last: ”Smith",
  address: {
    street: ”450 Broadway",
    city: ”New York, NY",
    zip: ”02783",
    country: "USA"
  },
  favorite_colors: ["black", ”yellow"]
}
...
```

Document Data Model – Pros & Cons

Pros

- Read / Write related (i.e. embedded) data in a single database operation
  - No need for joins or maintaining foreign keys => reads & writes very fast
- Flexible schema
  - no fixed table structure => easy to add / remove fields
- More "natural" way of handling data for developers
  - Data often represented as documents inside your application already anyway

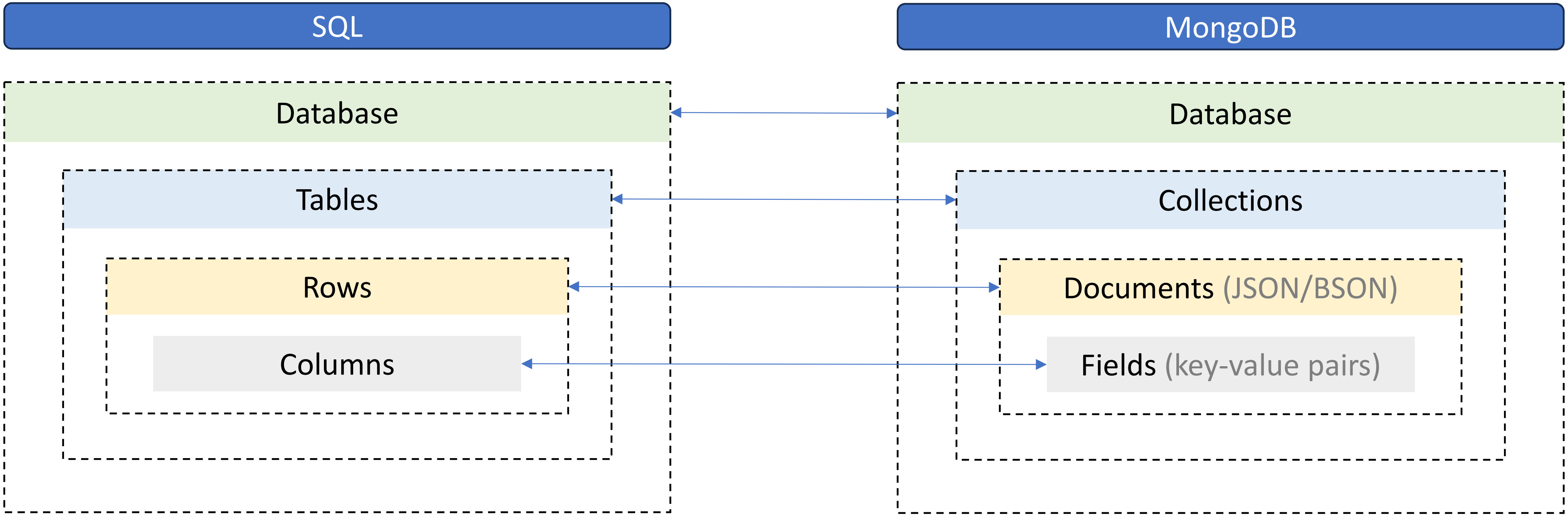
Cons

- Embedding (all) related data can create *very* large documents
  - Think customer with full purchase history
  - Can exceed internal document size limit
- No static schema can lead to messy state over time
  - => schema validation moves into the application layer

MongoDB

- Very popular **document-based** DBMS
- Distributed deployments (replica sets, shards)
- Documents are written and read in JSON
    - MongoDB uses an extended version of JSON to support various data types
- Internally, documents are stored in BSON format
- Documents are stored in collections
- Databases are made up of collections

MongoDB vs SQL Concepts



MongoDB – Creating a Collection

`db.createCollection(`**`name`**`, `*`options doc`*`)`

```
db.createCollection( <name>,
   {
     capped: <boolean>,
     timeseries: {
        timeField: <string
        metaField: <string
        granularity: <stri
        bucketMaxSpanSecond
        bucketRoundingSeconds: <number>
     },
     expireAfterSeconds: <number>,
     clusteredIndex: <document>,
     changeStreamPreAndPostImages: <document>,
     size: <number>,
     max: <number>,
```

Notice how you do not specify a fixed schema!

```
      storageEngine: <document>,
                    <document>,
                   Level: <string>,
                   Action: <string>,
                   nDefaults: <document>,
                   tring>,
                   <pipeline>,
                  : <document>,
      writeConcern: <document>
   }
)
```

MongoDB CRUD – Inserting a document

`db.collection.insertOne(`**`document`**`)`

```
db.customers.insertOne({
   _id: ObjectId("648c9913e548ee476696d6ef"),
   first: "Jane",
   last: "Doe",
   dob: Date("1991-10-30"),
   customer_group: 263,
   address: {
      street: "100 High St",
      city: "Cambridge”,
      ...
   },
   purchases: [
      { item: "Orange juice, 1 liter”,
        qty: 10},
      { item: "Chocolate, 100g",
        qty: 2}
   ]
})
```

Primary key (**every** MDB doc has field **_id**, default type ObjectId)

fields

String

Date

Number (int32)

typed field values

embedded document, aka sub-document

field contains array of sub-documents

MongoDB Data Types

Type	***Notes***	**Type**	***Notes***
Double	*double precision float*	DBPointer	*deprecated*
String		JavaScript	*code*
Object	*embedded document*	Symbol	*deprecated*
Array		JavaScript with scope	*deprecated as of 4.4*
Binary data	*UUIDs, binary blobs*	32-bit integer	*"int"*
Undefined	*deprecated*	Timestamp	
ObjectId	*default type of _id*	64-bit integer	*"long"*
Boolean		Decimal128	
Date	*UNIX epoch*	Min Key	*internal use*
Null		Max Key	*internal use*
Regular Expression			

MongoDB – (Not so) Schemaless

- MongoDB does not have a fixed schema
- You can store any JSON document in your collection
  - => The fields across your documents define an **implicit schema**
- Very easy to add and remove fields
  - Can be handled in-app as your data model evolves / matures
- In practice, you **should not** store completely **unrelated** documents together
  - Makes the application code handling reads very complex
  - Makes it hard to index your data
  - Can make storage optimization hard

MongoDB – Data Modelling

SQL: Normalized Data Model

students

id	first	last
1	Sarah	Miller
2	Paul	Smith

contact_details

_id	student_id	email	phone
<id1>	1	xyz@example.com	0123 456 7890
<id2>	2	abc@example.com	0987 654 3210

grades

_id	student_id	class	grade
<id1>	1	CS 101	A
<id2>	1	CS 102	A
…	…	…	…

MongoDB: Denormalized Data Model

```
{
   _id: 1,
   first: "Sarah",
   last: "Miller",
   contact_details: {
      email: "xyz@example.com",
      phone: "0123 456 7890"
   },
   grades: [
      { class: "CS 101", grade: "A"},
      { class: "CS 102", grade: "B"}
   ]
}
...
```

MongoDB – Data Modelling - I

- **Relational DBMS are optimized for storage** (-> normalization)
- Normalization eliminates duplication for data correctness & consistency
- But: Joins are very, very expensive (CPU intensive)
- Joins play a major part in TCO of database infrastructure
- **MongoDB is optimized for compute**
- Embed data that is typically processed together
  - => Very fast reads and writes
  - => Reads and writes become single, atomic operation
  - => This is particularly beneficial for high-velocity queries (i.e. queries run many times per second)

MongoDB – Data Modelling - II

When to denormalize

- Your data is generally used together
- You have "contains" relationships between entities (i.e. One-to-One relationship between entities)
- One-to-Many relationships where the many child entities are always viewed along with their parent entity

When not to denormalize (i.e. choose normalized data model)

- Embedding would result in duplication of data with little read performance improvements to outweigh the implications of duplication.
- Represent more complex many-to-many relationships
- Embedding would result in very large documents (possibly exceeding internal document size limit)

reference: https://www.mongodb.com/docs/manual/core/data-model-design/

MongoDB CRUD – Querying with MQL - I

```
db.collection.find(query)
```

```
> db.customers.find({last: "Miller"})

{ "_id": ObjectId("57d28452ed5d4d54e8686f68"), "first": "Daniel", "last": "Miller", ...}
{ "_id": ObjectId("57d28452ed5d4d54e8686f99"), "first" : "Elizabeth", "last" : "Miller", ...}
{ "_id": ObjectId("57d28452ed5d4d54e8686f51"), "first": "Ryan", "last" : "Miller", ...}
{ "_id": ObjectId("57d28452ed5d4d54e8687069"}, "first": "Eric", "last": "Miller", ...}
{ "_id": ObjectId("57d28452ed5d4d54e8686fd2"), "first": "Pamela", "last": "Miller", ...}
{ "_id": ObjectId("57d28452ed5d4d54e8687256"), "first": "Tammy", "last": "Miller", ...}
>
```

MongoDB CRUD – Querying with MQL - II

```
db.collection.find(query, projection)
```

```
> db.customers.find({last: "Miller", "address.state": "Florida"},
                    {_id: 0, first: 1, last: 1, "address.state": 1})

{ "first" : "Daniel", "last" : "Miller", "address" : { "state" : "Florida" } }
{ "first" : "Ryan", "last" : "Miller", "address" : { "state" : "Florida" } }
>
```

MongoDB CRUD – Querying with MQL - III

```
db.collection.find(query, projection)
```

```
> db.customers.find({ "last": "Miller", $or: [
                        { "address.state": "Florida" },
                        { "address.state": "California" }
                      ] },
                      {_id: 0, first: 1, last: 1, "address.state": 1})

{ "first" : "Daniel", "last" : "Miller", "address" : { "state" : "Florida" } }
{ "first" : "Ryan", "last" : "Miller", "address" : { "state" : "Florida" } }
{ "first" : "Elizabeth", "last" : "Miller", "address" : { "state" : "California" } }

>
```

MongoDB CRUD – Updating a document

```
db.collection.updateOne(query, update)
```

```
> db.fruit_store.find()
{ "_id": "apples", "qty": 5 }
{ "_id": "raspberries", "qty": 3 }
{ "_id": "bananas", "qty": 7 }
{ "_id": "oranges", "qty": { "in stock": 8, "ordered": 12 } }
{ "_id": "avocados", "qty": "fourteen" }

> db.fruit_store.updateOne( { _id: "raspberries" }, { $set: {"qty": 10} } )

> db.fruit_store.find( { qty: { $gt: 4 } } )
{ "_id": "apples", "qty": 5 }
{ "_id": "raspberries", "qty": 10 }
{ "_id": "bananas", "qty": 7 }

>
```

MongoDB CRUD – Deleting a document

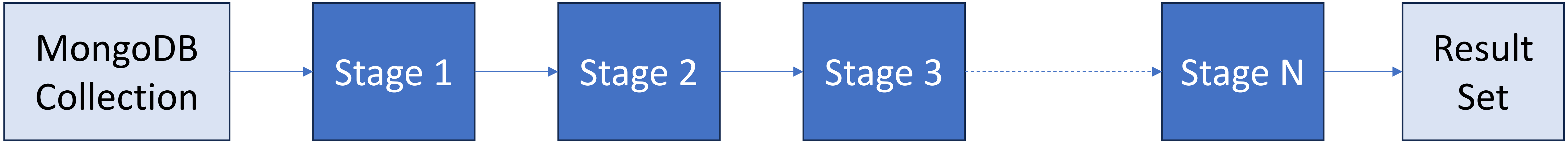
```
db.collection.deleteOne(query)
```

```
> db.fruit_store.find()
{ "_id": "apples", "qty": 5 }
{ "_id": "raspberries", "qty": 10 }
{ "_id": "bananas", "qty": 7 }
{ "_id": "oranges", "qty": { "in stock": 8, "ordered": 12 } }
{ "_id": "avocados", "qty": "fourteen" }

> db.fruit_store.deleteOne( { _id: "raspberries" } )

> db.fruit_store.find( { qty: { $gt: 4 } } )
{ "_id": "apples", "qty": 5 }
{ "_id": "bananas", "qty": 7 }

>
```

MongoDB – Aggregation Framework

- MQL allows us to run key-value match queries
- But what about more complex aggregate queries?
  - E.g. computing sums, average, min or max values across documents
  - Or grouping values from multiple documents together
- For this, MongoDB has another query API
  - => MongoDB **Aggregation Framework**
- An Aggregation query consists of an **Aggregation Pipeline**
- The aggregation pipeline consists of individual **stages**
- The output of a stage is the input of the next stage

MongoDB – Aggregation Pipelines



- The input to a pipeline is always the entire collection
- The output of a stage is the input of the next stage
- The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline
  - Each stage has a stage operator (e.g. $match, $project, $unwind, $lookup, etc)
- An aggregation query is defined as array of stage documents

MongoDB – Aggregation - Example

```
// Stage 1
{
   $match: {
      last: "Miller"
   }
}
```

```
// Stage 2
{
   $project: {
      first: 1, last: 1,
      "address.state": 1,
      _id: 0
   }
}
```

```
// Stage 3
{
   $group: {
      _id: "$address.state",
      count: {$count: {} }
   }
}
```

```
// Stage 4
{
   $sort: {
      count: -1
   }
}
```

MongoDB – Aggregation - Example

```
// Stage 1
{
   $match: {
```

```
// Stage 2
{
   $project: {
```

Stage 1 output:

```
{ _id: ObjectId("57d28452ed5d4d54e8686f9d"), first: "Ernest", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8687022"), first: "Harry", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686f68"), first: "Daniel", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686f99"), first: "Elizabeth", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686f51"), first: "Ryan", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686f55"), first: "Lori", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686fe8"), first: "Eugene", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8687024"), first: "Kimberly", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8687069"), first: "Eric", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686fea"), first: "Dorothy", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686fd2"), first: "Pamela", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e868700f"), first: "Scott", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8686ff8"), first: "Denise", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e86870ac"), first: "Douglas", last: "Miller", ... }
{ _id: ObjectId("57d28452ed5d4d54e8687256"), first: "Tammy", last: "Miller", ... }
```

MongoDB – Aggregation - Example

```
// Stage 1
{
   $match: {
```

```
// Stage 2
{
   $project: {
```

Stage 2 output:

```
{ first: "Ernest", last: "Miller", address: { state: "Ohio" } }
{ first: "Harry", last: "Miller", address: { state: "California" } }
{ first: "Daniel", last: "Miller", address: { state: "Oregon" } }
{ first: "Elizabeth", last: "Miller", address: { state: "District of Columbia" } }
{ first: "Ryan", last: "Miller", address: { state: "Florida" } }
{ first: "Lori", last: "Miller", address: { state: "California" } }
{ first: "Eugene", last: "Miller", address: { state: "Florida" } }
{ first: "Kimberly", last: "Miller", address: { state: "Texas" } }
{ first: "Eric", last: "Miller", address: { state: "California" } }
{ first: "Dorothy", last: "Miller", address: { state: "California" } }
{ first: "Pamela", last: "Miller", address: { state: "North Carolina" } }
{ first: "Scott", last: "Miller", address: { state: "Illinois" } }
{ first: "Denise", last: "Miller", address: { state: "South Carolina" } }
{ first: "Douglas", last: "Miller", address: { state: "Michigan" } }
{ first: "Tammy", last: "Miller", address: { state: "Utah" } }
```

MongoDB – Aggregation - Example

```
Stage 3 output:
{ _id: "Utah", count: 1 }
{ _id: "Oregon", count: 1 }
{ _id: "Texas", count: 1 }
{ _id: "Michigan", count: 1 }
{ _id: "District of Columbia", count: 1 }
{ _id: "California", count: 4 }
{ _id: "South Carolina", count: 1 }
{ _id: "Ohio", count: 1 }
{ _id: "Florida", count: 2 }
{ _id: "North Carolina", count: 1 }
{ _id: "Illinois", count: 1 }
```

```
{
   $group: {
      _id: "$address.state",
      count: {$count: {} }
   }
}
```

```
{
   $sort: {
      count: -1
   }
}
```

MongoDB – Aggregation - Example

Stage 4 output:

```
{ _id: "California", count: 4 }
{ _id: "Florida", count: 2 }
{ _id: "Michigan", count: 1 }
{ _id: "North Carolina", count: 1 }
{ _id: "Illinois", count: 1 }
{ _id: "South Carolina", count: 1 }
{ _id: "Ohio", count: 1 }
{ _id: "District of Columbia", count: 1 }
{ _id: "Utah", count: 1 }
{ _id: "Oregon", count: 1 }
{ _id: "Texas", count: 1 }
```

```
{
   $group: {
      _id: "$address.state",
      count: {$count: {} }
   }
}
```

```
{
   $sort: {
      count: -1
   }
}
```

MongoDB – Aggregation - Example

```
db.getCollection("customers").aggregate(
    [
        // Stage 1
        {
            $match: {
                last: "Miller"
            }
        },
        // Stage 2
        {
            $project: {
                first: 1,
                last: 1,
                "address.state": 1,
                _id: 0
            }
        },
```

```
        // Stage 3
        {
            $group: {
                _id: "$address.state",
                count: {$count: {} }
            }
        },
        // Stage 4
        {
            $sort: {
                count: -1
            }
        }
    ]
);
```

MongoDB – Joins

- MongoDB works naturally with denormalized data
- But, you should not put all unrelated data into one document
- E.g. a company's orders and inventory should be in separate collections
- So, how do you bring data from separate collections together, i.e. how do you join?
- MongoDB has a special aggregation operator for this!
- **$lookup** performs a **left outer join** between collections in the same database

MongoDB – Joins - Example

```
db.orders.insertMany( [
   { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
   { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
   { "_id" : 3  }
] )
```

```
db.inventory.insertMany( [
   { "_id" : 1, "sku" : "almonds", "description": "product 1", "instock" : 120 },
   { "_id" : 2, "sku" : "bread", "description": "product 2", "instock" : 80 },
   { "_id" : 3, "sku" : "cashews", "description": "product 3", "instock" : 60 },
   { "_id" : 4, "sku" : "pecans", "description": "product 4", "instock" : 70 },
   { "_id" : 5, "sku": null, "description": "Incomplete" },
   { "_id" : 6 }
] )
```

example source: https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/

MongoDB – $lookup

```
db.orders.aggregate( [
   {
     $lookup:
       {
         from: "inventory",
         localField: "item",
         foreignField: "sku",
         as: "inventory_docs"
       }
  }
] )
```

example source: https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/

MongoDB – $lookup Result

Resulting Documents:

```
{
   "_id" : 1,
   "item" : "almonds",
   "price" : 12,
   "quantity" : 2,
   "inventory_docs" : [
      { "_id" : 1, "sku" : "                                        120 }
   ]
}
{
   "_id" : 2,
   "item" : "pecans",
   "price" : 20,
   "quantity" : 1,
   "inventory_docs" : [
      { "_id" : 4, "sku" : "                                        0 }
   ]
}
{
   "_id" : 3,
   "inventory_docs" : [
      { "_id" : 5, "sku" : null, "description" : "Incomplete" },
      { "_id" : 6 }
   ]
}
```

SQL Equivalent:

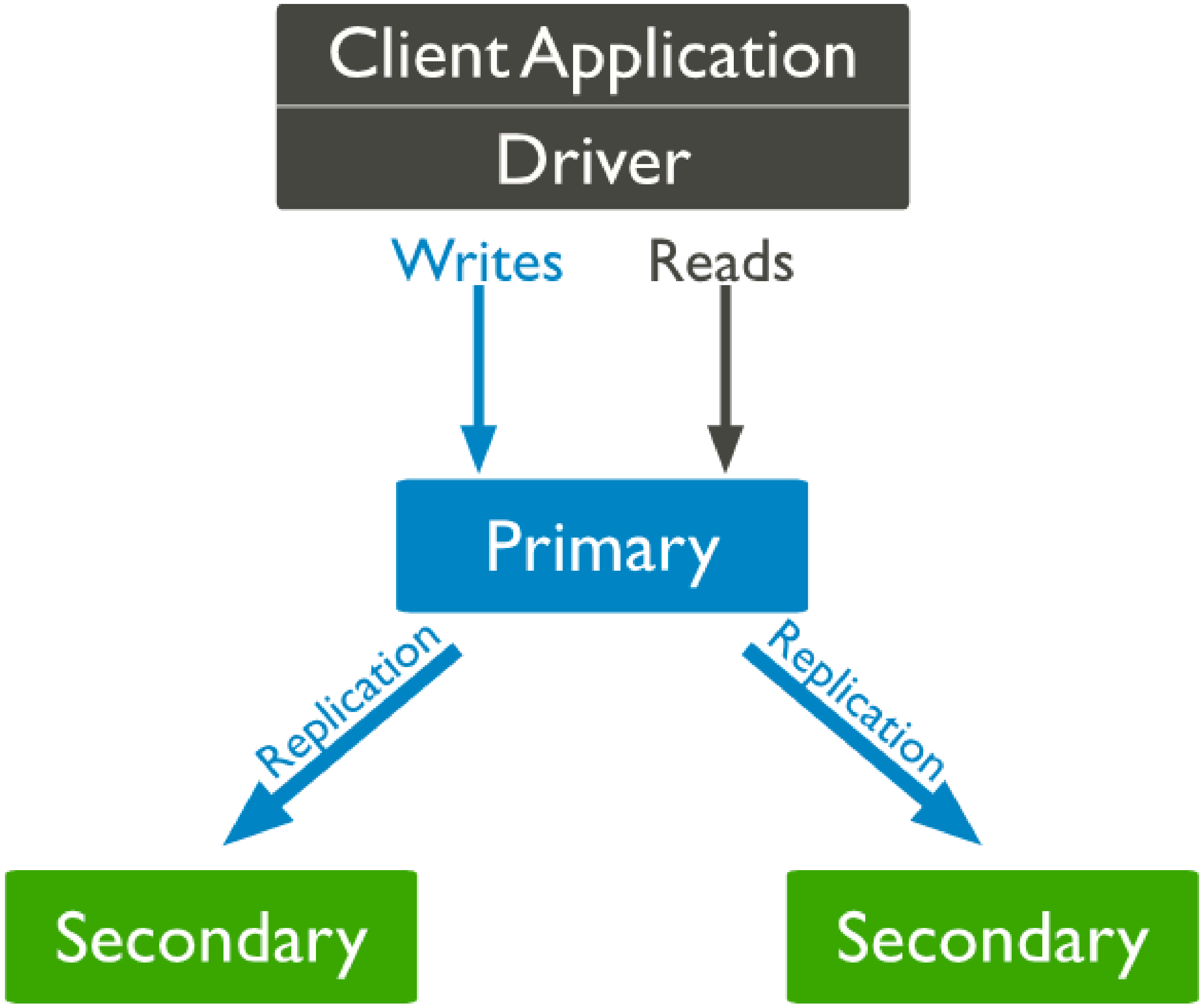
```sql
SELECT *, inventory_docs
FROM orders
WHERE inventory_docs IN (
   SELECT *
   FROM inventory
   WHERE sku = orders.item
);
```

example source: https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/

MongoDB – Deployments

- In its simplest form, just a single server instance (mongod process)
- But, in production, you want redundancy and high availability
- For this, MongoDB supports **Replica Sets**
  - Group of mongod instances that maintain the same data set
  - Contains exactly one **primary** node
  - Can contain multiple **secondary** nodes
  - Optionally, one arbiter node
  - The primary receives all write operations
  - The secondaries then replicate the changes applied to the primary

MongoDB – Replica Sets

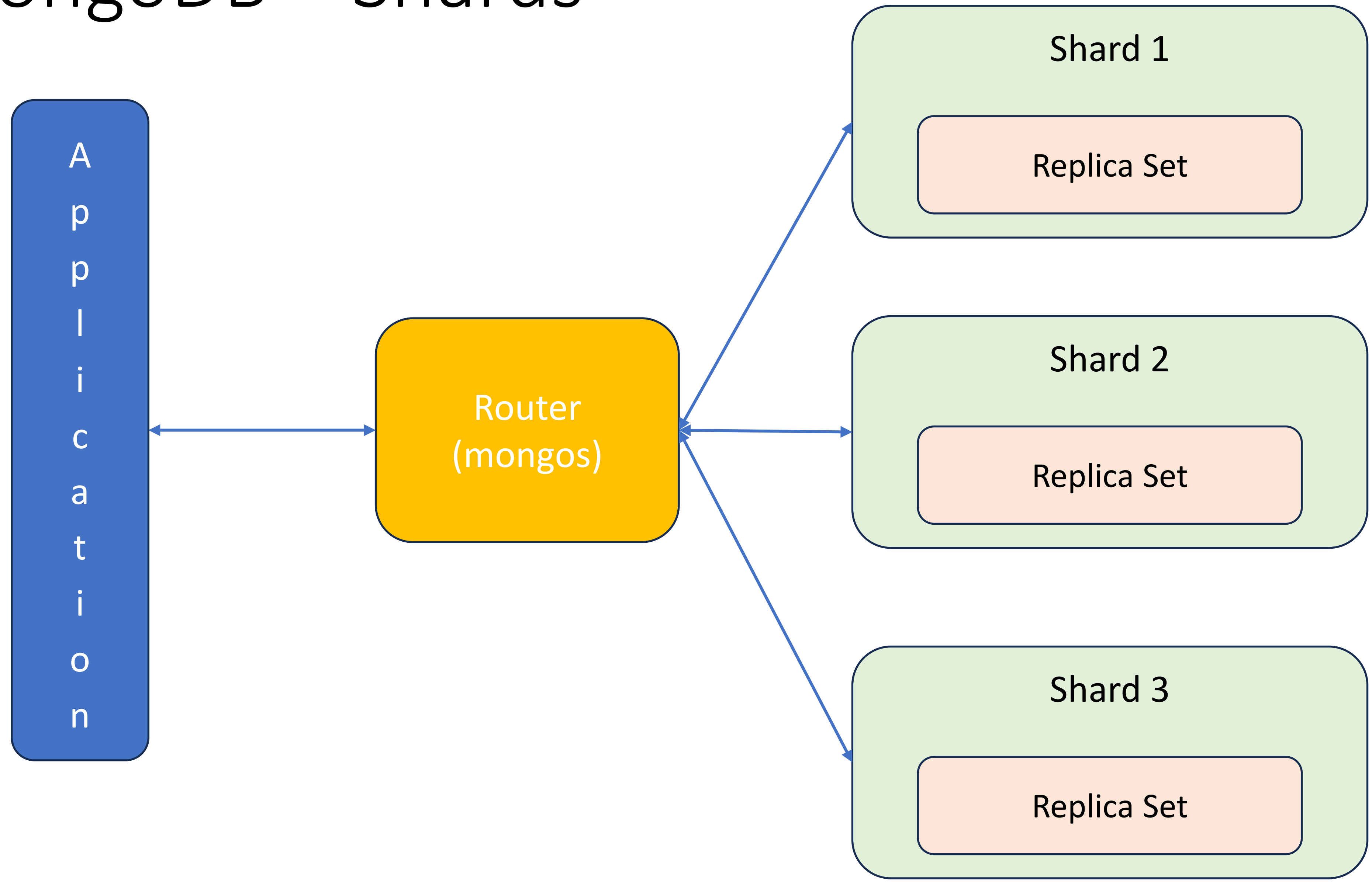


source: https://www.mongodb.com/docs/manual/replication/

MongoDB – Sharding

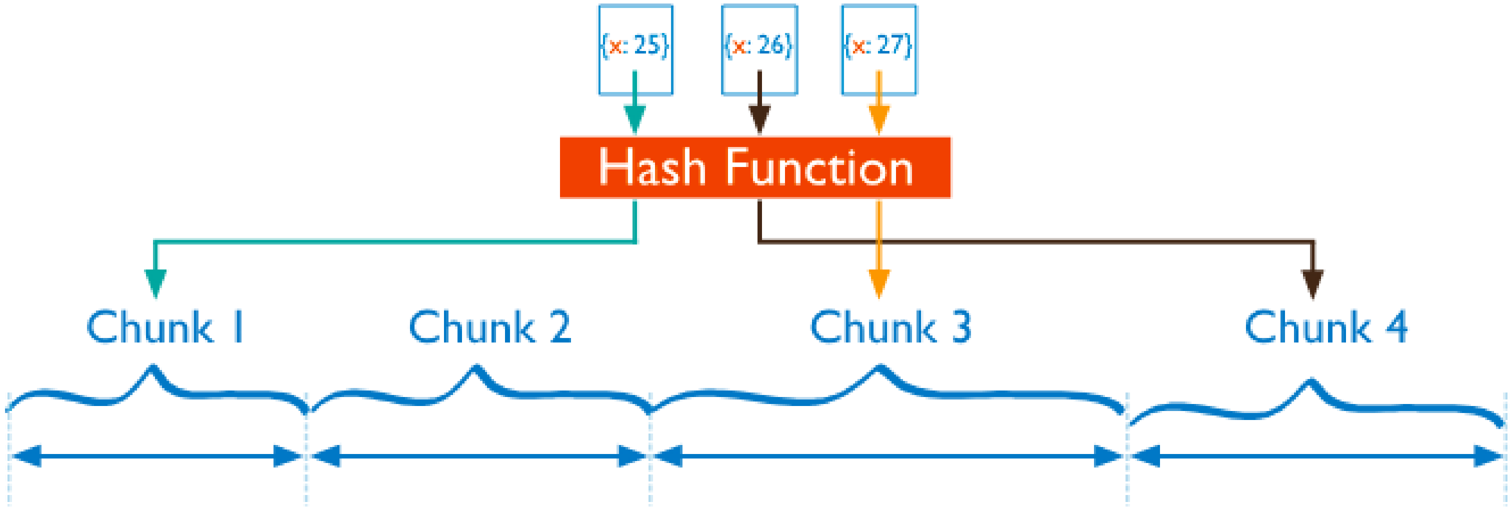
- MongoDB uses sharding to scale for very large data sets and/or high throughput operations
- Idea: split data set across multiple **shards**
- An individual shard then typically consists of a replica set storing a subset of the total data set
- The DBA selects a **shard key** (one or multiple fields)
- A mongos router then distributes documents based on their shard key values
- This is referred to as **horizontal scaling**

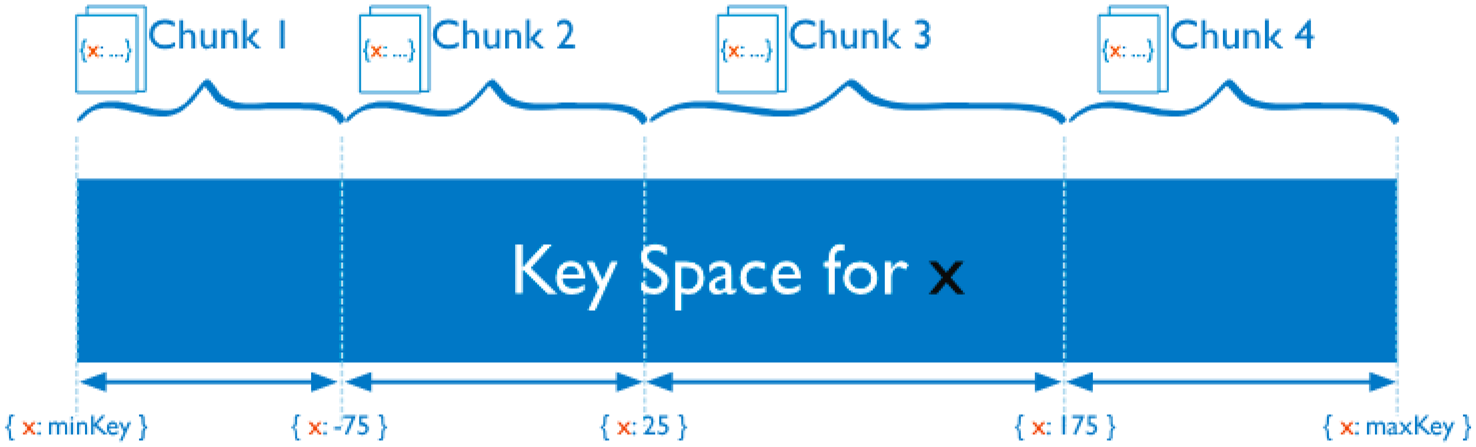
MongoDB – Shards



MongoDB – Sharding Strategies

Hashed Sharding





Ranged Sharding

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

Summary

- NoSQL umbrella term for all sorts of non-relational DBMS technologies
  - (note that the data stored may still be "relational")
- Document stores choose a denormalized data model
- That makes reads and writes of related data very fast
  - => NoSQL is a great fit for (most) OLTP workloads
- Denormalized data model also feels natural to develop against
- MongoDB is a popular distributed document store DBMS
- MongoDB also supports (left-outer) joins

- Contact: t.zahn@yahoo.de