Document-Oriented & Semi-Structured Data

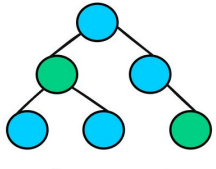
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Document Stores



- **Store data as documents** (JSON, BSON, XML)
  - Documents are self-describing hierarchical structures (aggregates)
  - Documents are typically grouped into collections
  - Natural representation of application objects
- Flexible schema - fields may differ between documents
- Basic operations:
  - Insert/delete a document
  - Retrieve documents by query
  - Update document fields
- **Rich query capabilities**
  - Complex joins are not typical

RavenDB

ArangoDB

CouchDB

mongoDB.

XML vs JSON

```xml
<user id="42">
  <name>Alice</name>
  <email>alice@example.com</email>
  <roles>
    <role>admin</role>
    <role>editor</role>
  </roles>
</user>
```

```json
{
  "id": 42,
  "name": "Alice",
  "email": "alice@example.com",
  "roles": ["admin", "editor"]
}
```

Observations:

- XML uses tags
- JSON uses key–value pairs
- JSON is more compact

XML (Extensible Markup Language)

- Tree-structured document format
- Tag-based representation
- Supports attributes and nested elements
- Designed for document exchange and interoperability
- Strong ecosystem (XSD, XPath, XQuery)

JSON

- Lightweight hierarchical data format
- Based on key–value pairs and arrays
- Designed for data interchange in web applications
- Native to JavaScript and widely used in APIs

Common properties

- hierarchical structure
- semi-structured data
- human-readable text format
- widely supported by programming languages

Structural Differences

Feature	XML	JSON
Data model	Document tree	Object / array structure
Syntax	Tags with opening/closing elements	Key-value pairs
Attributes	Yes	No (use fields instead)
Arrays	Multiple elements	Explicit arrays
Expresiveness	High	Low
Human readability	Moderate	High

- JSON → native for **document databases**
- XML → older **XML databases**

Suitable Use Cases for Document Stores

Event Logging

- Many different applications want to log events
  - Type of data being captured keeps changing

Content Management Systems, Blogging Platforms

- Managing user comments, user registrations, profiles, web-facing documents, …

Web Analytics or Real-Time Analytics

- Parts of the document can be updated
- New metrics can be easily added without schema changes

E-Commerce Applications

- Flexible schema for products and orders
- Evolving data models without expensive data migration

Query-Driven Document Modeling

- Document design should reflect dominant application queries
  - Main decision: embed vs reference
- Optimize for:
  - what is read together
  - what changes frequently
  - what is aggregated often
- Idea: Document schema is not arbitrary, it is shaped by access patterns

Schema-on-Write vs Schema-on-Read

Key idea

- Schema-on-write: structure first → then store data
- Schema-on-read: store data first → interpret structure later

```
CREATE TABLE Orders(
  order_id INT,
  customer VARCHAR(100),
  price DECIMAL
);
```

Data must follow this schema before insertion

Different structures can coexist; the interpretation happens during query processing

```
{ "order_id": 101, "customer": "Alice", "price": 20 }
{ "order_id": 102, "customer": "Bob", "price": "20 EUR" }
{ "order_id": 103, "customer": "Alice", "items": ["book","pen"] }
```

Schema-on-Write vs Schema-on-Read

	Schema-on-Write	Schema-on-Read
Principle	Data must conform to schema before being stored	Schema is applied when data is read
Validation	Strict validation during ingestion	Minimal validation during ingestion
Typical systems	Relational DBMS	Data lakes, most NoSQL systems
Flexibility	Low	High
Query performance	Usually faster (optimized structure)	Can be slower (interpretation at query time)
Data evolution	Schema changes required	Multiple structures can coexist

mongoDB

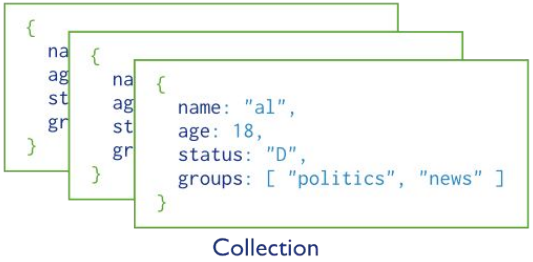
- Initial release: 2009
- Written in C++
  - Open-source
- Cross-platform
- JSON documents
  - Dynamic schemas
- Features:
  - High performance – indices
  - High availability – replication + eventual consistency + automatic failover
  - Automatic scaling – automatic sharding across the cluster
  - MapReduce support

http://www.mongoDB.org/

Terminology

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

Terminology in Oracle and mongoDB



- Each mongoDB instance has multiple databases
- Each database can have multiple collections
- When we store a document, we have to choose database and collection

Documents

- Use JSON
- Stored as BSON
  - Binary representation of JSON
- Have maximum size: 16MB (in BSON)
  - Not to use too much RAM
  - GridFS tool divides larger files into fragments
- Restrictions on field names:
  - `_id` is reserved for use as a primary key
    - Unique in the collection
    - Immutable
    - Any type other than an array
  - The field names cannot start with the `$` character
    - Reserved for operators
  - The field names cannot contain the `.` character
    - Reserved for accessing fields

https://www.mongodb.com/docs/manual/reference/limits/

BSON (Binary JSON)

- Binary-encoded serialization of JSON documents
  - Allows embedding of documents, arrays, JSON simple data types + other types (e.g., date)
- Purpose
  - efficient storage
  - fast traversal of documents

```
{
  "name": "Alice",
  "age": 30
}
```

```
16 00 00 00         total document size
02                  type: string
6e 61 6d 65 00      field name: "name"
06 00 00 00         string length
41 6c 69 63 65 00   value: "Alice"
10                  type: int32
61 67 65 00         field name: "age"
1e 00 00 00         value: 30
00                  end of document
```

MongoDB stores JSON-like documents, but internally uses BSON

http://bsonspec.org/

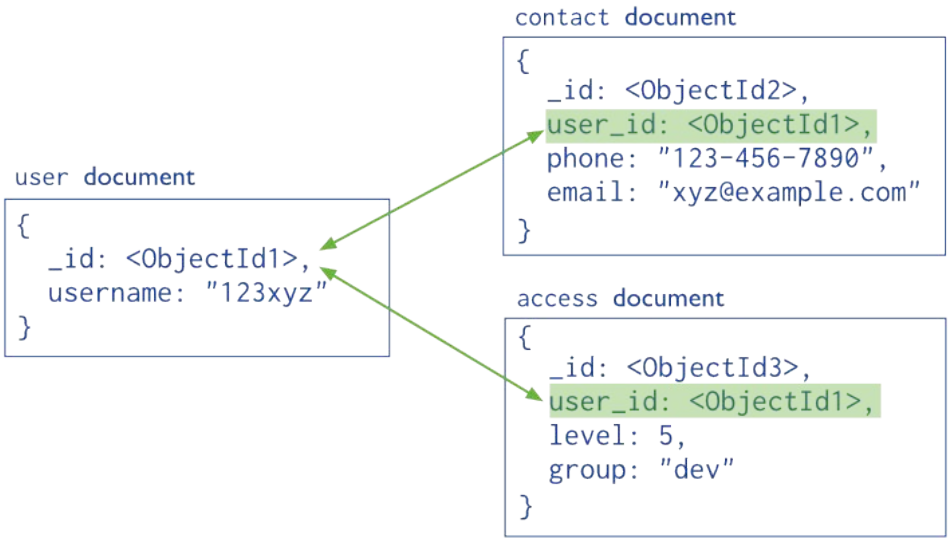
Data Model

mongoDB

- Documents have flexible schema
  - Collections do not enforce structure of data
  - In practice the documents are similar
- Challenge: Balancing
  - the needs of the application
  - the performance characteristics of database engine
  - the data retrieval patterns
- Key design decision: references vs. embedded documents
  - Factors
    - access patterns
    - update frequency
    - data size

References

- Including links / references from one document to another
  - More flexibility than embedding
- Use normalized data models:
  - When embedding would result in duplication of data not outweighted by read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets
- Disadvantages:
  - Can require more roundtrips to the server (follow up queries)



Embedded Data

- Related data in a single document structure
  - Documents can have subdocuments
  - Applications may need to issue less queries
- Denormalized data models
- Allow to manipulate related data in a single database operation
- Provides:
  - Better performance for read operations
  - Ability to retrieve/update related data in a single database operation
- Disadvantages:
  - Documents may significantly grow
  - Only one “view” of the data

Embedding vs Referencing – Trade-off

	Embedding	Referencing
Idea	Related data stored inside one document	Related data stored in separate documents
Read performance	Fast reads (single query)	May require multiple queries or joins
Data redundancy	Duplication of data	Normalized data
Updates	Updating duplicated data may be expensive	Update in one place
Typical usage	Aggregate objects	Shared entities

When to Use Embedding vs Referencing

- Use **Embedding** when
  - data is naturally hierarchical
  - data is usually read together
  - updates are rare
  - document size stays reasonable
- Typical use cases
  - orders with items
  - blog posts with comments
  - user profile with addresses

```
order
 └ items[]
    └ product snapshot
```

- Use **Referencing** when
  - entities are shared across many documents
  - data changes frequently
  - relationships are many-to-many
  - documents would become too large
- Typical use cases
  - customers
  - products
  - large hierarchical structures

```
orders → customer_id
orders → product_id
```

When to Use Embedding vs Referencing

Embedding

- one collection
- customer data are inside the order document

```
db.orders.find(
  {
    "customer.given_name": "James",
    "customer.family_name": "Thompson"
  },
  { _id: 0 }
)
```

Referencing

- customer data are stored separately
- requires an extra step

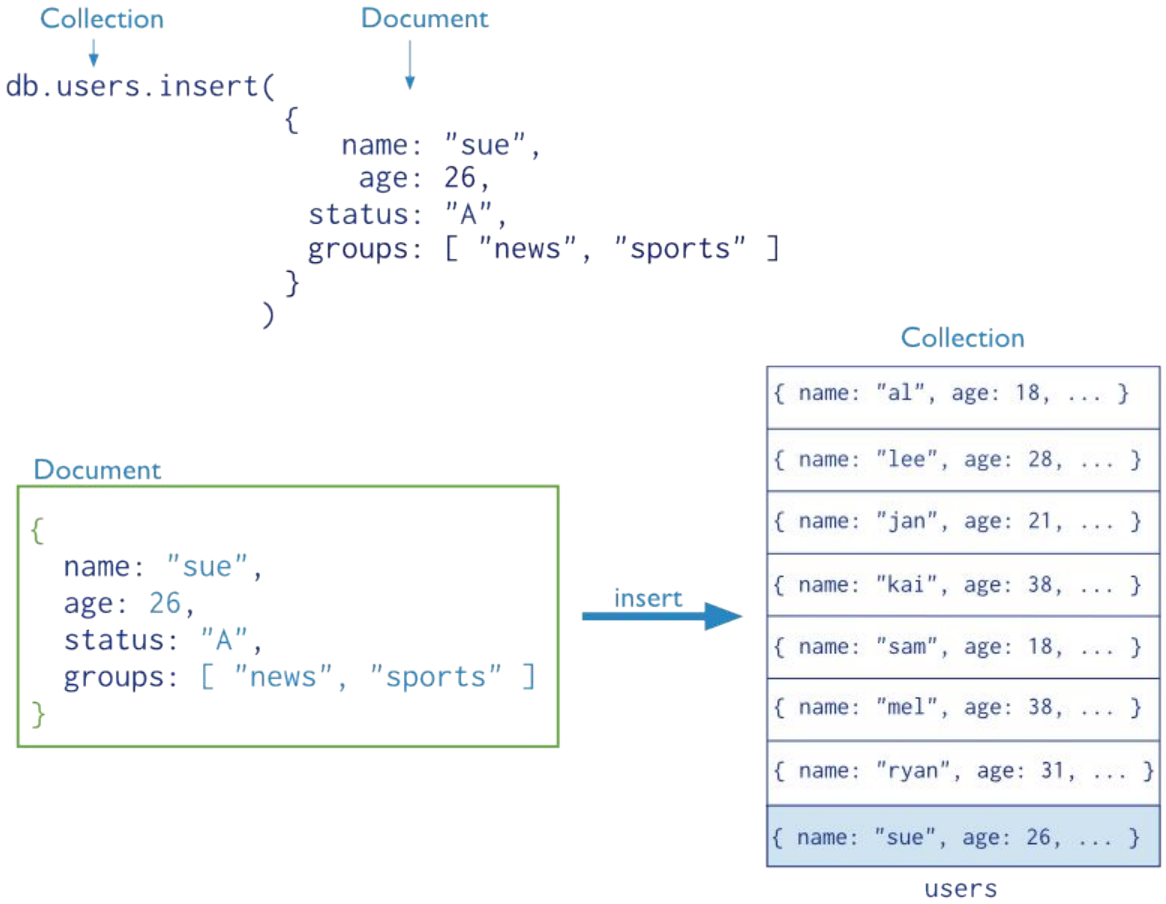
```
db.people.find(
  {
    given_name: "James",
    family_name: "Thompson"
  },
  { _id: 1 }
)

db.orders_flat.find(
  { customer_id: 5595 },
  { _id: 0 }
)
```

Data Modification

- CRUD operations
  - Create - insert new document
  - Read - retrieve documents using queries
  - Update - modify fields in existing documents
  - Delete - remove documents
- Operations always target a collection of documents

MongoDB operations modify whole documents or their fields

```
db.orders.insertOne({...})
db.orders.find({...})
db.orders.updateOne({...})
db.orders.deleteOne({...})
```


Collection
Document
db.users.insert(
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
)
Document
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
insert
Collection
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
users

Example – Insert / Update / Delete

```
db.inventory.insertOne({
    type: "misc",
    item: "card",
    qty: 15
})
```

insertOne() → one network request per document
insertMany() → batch insert (more efficient)

```
db.inventory.insertMany([
{ type: "food", item: "apple", qty: 50, price: 0.5 },
{ type: "food", item: "banana", qty: 80, price: 0.3 },
{ type: "book", item: "notebook", qty: 40, price: 5.0
},
{ type: "misc", item: "pen", qty: 100, price: 1.2 }
])
```

```
db.inventory.updateOne(
    { item: "card" },
    { $set: { qty: 20 } }
)
```

```
db.inventory.deleteOne(
    { item: "card" }
)
```

MongoDB modifies documents, not rows

MongoDB Update Operators

Operator	Purpose	Example
$set	set value of (or add) a field	update price
$inc	increment numeric value	increase quantity
$push	add element to array	add tag
$pull	remove element from array	remove tag

```
{
  item: "notebook",
  qty: 40,
  tags: ["school", "paper"]
}
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $inc: { qty: 10 } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $set: { price: 5.0 } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $pull: { tags: "paper" } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $push: { tags: "office" } }
)
```

Update operators modify fields without replacing the entire document

updateOne vs updateMany vs replaceOne

Operation	Description
updateOne()	updates the first matching document
updateMany()	updates all matching documents
replaceOne()	replaces the entire document

```
{
  item: "notebook",
  qty: 40,
  price: 5
}
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $set: { price: 6 } }
)
```

```
db.inventory.updateMany(
  { type: "food" },
  { $inc: { qty: 10 } }
)
```

```
db.inventory.replaceOne(
  { item: "notebook" },
  { item: "notebook", qty: 50 }
)
```

MongoDB Query Structure

```
{
  item: "apple",
  type: "food",
  qty: 50,
  price: 0.5
}
```

- Basic query syntax:

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

- Parameters
  - `collection` - collection to query
  - `filter` - conditions selecting documents
  - `projection` - fields returned in the result

```
db.inventory.find(
  { type: "food" },
  { item: 1, qty: 1 }
)
```

```
db.inventory.find(
  { type: "food" },
  { item: 1, qty: 1, _id: 0 }
)
```

Querying Nested Documents

- MongoDB allows querying fields inside nested documents using dot notation
- Note:
  - customer.given_name → nested document
  - items.0.product → array

```
{
  _id: 101,
  customer: {
    given_name: "Alice",
    family_name: "Smith"
  },
  total: 120
}
```

```
db.orders.find(
  { "customer.given_name": "Alice" }
)
```

Querying Arrays

```
{
  item: "orange",
  tags: ["fruit", "food", "citrus"]
}
```

```
db.inventory.find(
  { tags: "fruit" }
)
```

```
db.inventory.find(
  { "tags.0": "fruit" }
)
```

```
{
  item: "notebook",
  memos: [
    { by: "shipping", memo: "on time" },
    { by: "billing", memo: "paid" }
  ]
}
```

Query matches if at least one array element satisfies the condition

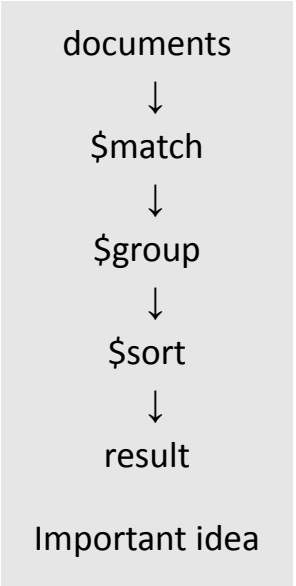
Query	Meaning
{ tags: "fruit" }	array contains value
{ tags: ["fruit","food","citrus"] }	exact array match

```
db.inventory.find(
  { "memos.by": "shipping" }
)
```

MongoDB Aggregation Pipeline

- Aggregation framework processes documents in multiple stages
  - Pipeline = sequence of transformations
  - Similar to SELECT → WHERE → GROUP BY → ORDER BY
- Each stage transforms the data and passes it to the next stage

```
db.orders.aggregate([
  { $match: {...} },
  { $group: {...} },
  { $sort: {...} }
])
```



Common Aggregation Stages

Stage	Purpose	SQL analogy
$match	filter documents	WHERE
$project	select/rename fields	SELECT
$group	aggregate values	GROUP BY
$sort	order results	ORDER BY
$limit	restrict number of results	LIMIT
$unwind	flatten array	JOIN / explode

Example Aggregation

```
{
  order_id: 1,
  items: [
    { product_id: 10, quantity: 2 },
    { product_id: 11, quantity: 1 }
  ]
}
```

Compute total quantity sold per product

```
db.orders.aggregate([
  { $unwind: "$items" },
  {
    $group: {
      _id: "$items.product_id",
      total_qty: { $sum: "$items.quantity" }
    }
  },
  { $sort: { total_qty: -1 } }
])
```

```
{ "_id": 10, "total_qty": 120 }
{ "_id": 11, "total_qty": 95 }
```

Aggregation pipeline processes documents step by step

find() ≈ SELECT ... WHERE
aggregate() ≈ SELECT ... GROUP BY ...

`find()` **vs** `aggregate()`

	find()	aggregate()
Purpose	retrieve documents	transform and analyze data
Complexity	simple queries	multi-stage processing
Operations	filtering, projection	grouping, sorting, reshaping
Typical use	application queries	analytics and reports

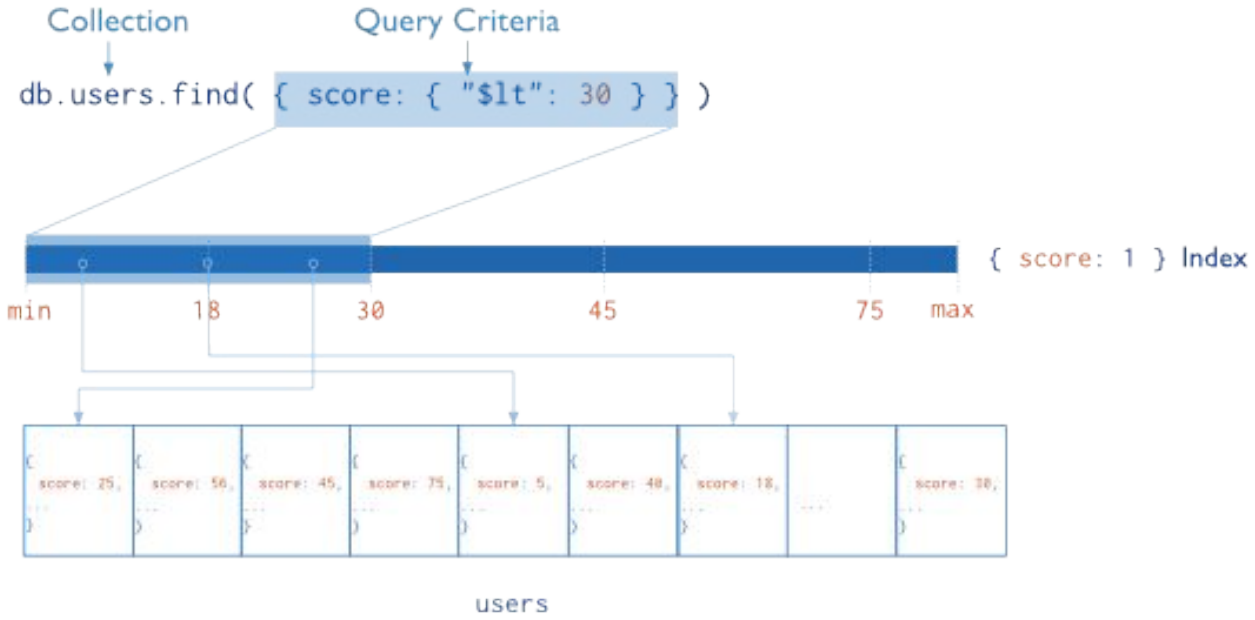
```
db.orders.find({ region: "EU-CZ" })
```

```
db.orders.aggregate([
  { $match: { region: "EU-CZ" } },
  { $group: { _id: "$customer_id", total: { $sum: "$total_price" } } }
])
```

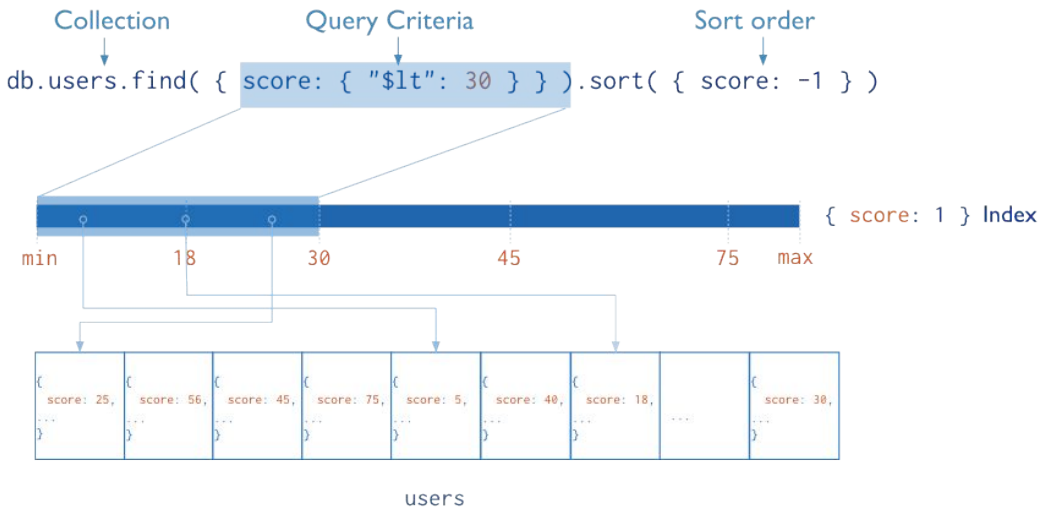
Indices

- Without indices:
  - MongoDB scans all documents (operation called COLLSCAN)
- With index
  - MongoDB searches using index structure (operation called IXSCAN)
- Indices store a portion of the collection's data set in an easy to traverse form
  - Stores the value of a specific field or a set of fields ordered by the value of the field
  - B-tree like structures
- Defined at collection level
- Purpose:
  - To speed up common queries
  - To optimize the performance of other operations in specific situations

Indices – Example

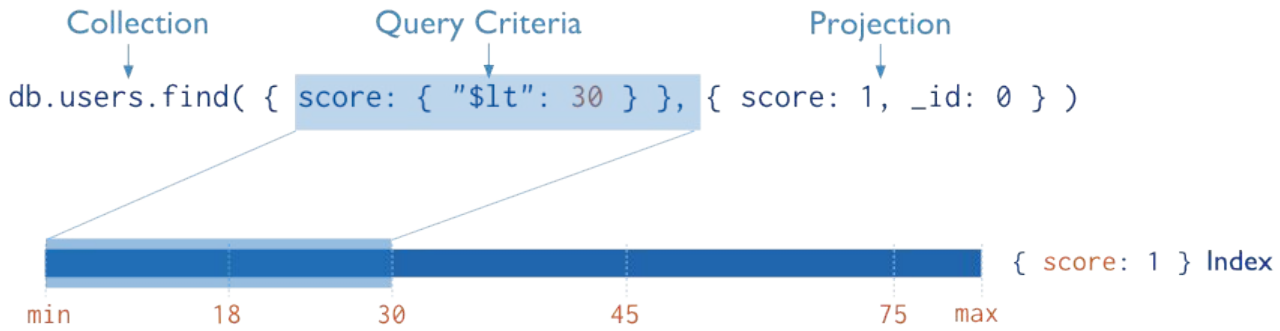


Indices – Sorted Results



- The index stores `score` values in ascending order
- mongoDB can traverse the index in either ascending or descending order to return sorted results (without sorting)

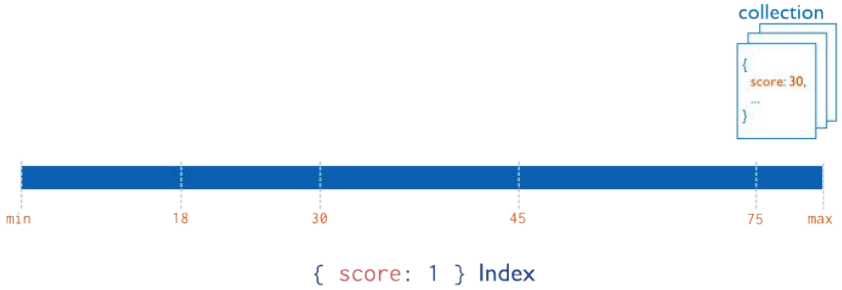
Indices – Covered Results



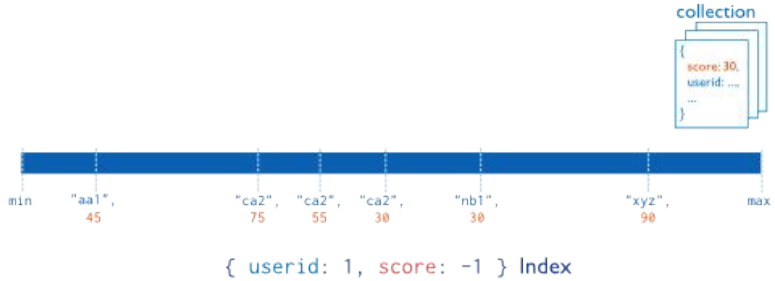
- mongoDB does not need to inspect data outside of the index to fulfil the query

Index Types

- **Default `_id`**
  - Exists by default
    - If applications do not specify `_id`, it is created automatically
  - Unique by default
- **Single Field**
  - User-defined indices on a single field of a document
- **Compound**
  - User-defined indices on multiple fields
- **Multikey index**
  - To index the content stored in arrays
  - Creates separate index entry for every element of the array



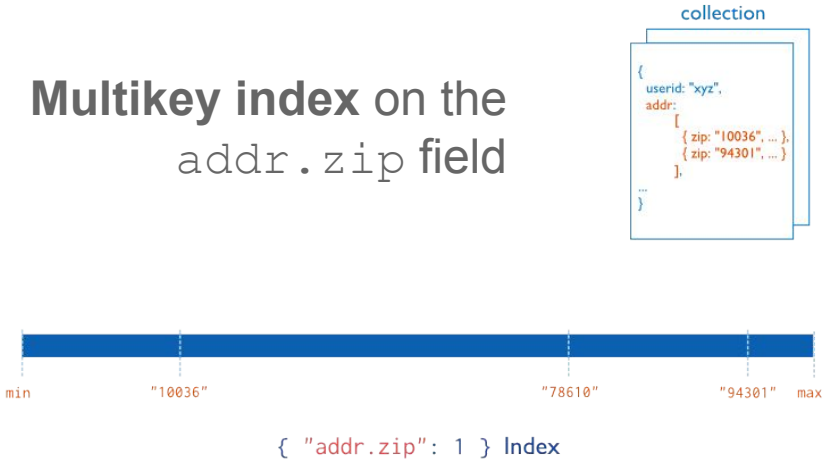
Single field index on the `score` field (ascending).



Compound index on the `userid` field (ascending) and the `score` field (descending).

- sorts first by `userid` and then, within each `userid` value, sort by `score`

Multikey index on the `addr.zip` field



Index Types

- **Geospatial Field**
  - 2d indexes = use planar geometry when returning results
    - For data representing points on a two-dimensional plane
  - 2sphere indexes = use spherical (Earth-like) geometry to return results
    - For data representing longitude, latitude
- **Text Indexes**
  - Searching for string content in a collection
- **Hash Indexes**
  - Indexes the hash of the value of a field
  - Only support equality matches (not range queries)

Indexes

```
db.people.createIndex( { "phone-number": 1 } )
```

- Creates a single-field index on the `phone-number` field of the `people` collection

```
db.products.createIndex( { item: 1, category: 1, price: 1 } )
```

- Creates a compound index on the `item`, `category`, and `price` fields

```
db.accounts.createIndex( { "tax-id": 1 }, { unique: true } )
```

- Creates a unique index
  - Prevents applications from inserting documents that have duplicate values for the inserted fields

```
db.collection.createIndex( { _id: "hashed" } )
```

- Creates a hashed index on `_id`

Indexes + `explain()`

- Without index
  - Likely COLLSCAN
  - Many examined documents
  - Possible extra sort stage
- After index
  - IXSCAN
  - Fewer examined documents
  - Sort can be index-supported

```
db.orders.aggregate([
  { $match: { region: "EU-CZ" } },
  { $sort: { order_date: -1 } },
  { $limit: 20 }
]).explain("executionStats")

db.orders.createIndex({
  region: 1,
  order_date: -1 })
```

MongoEngine

- Object Document Mapper for Python
  - Maps collections and embedded documents to classes
- Improves readability of application code
- Same MongoDB logic underneath
  - ODM does not remove modeling trade-offs

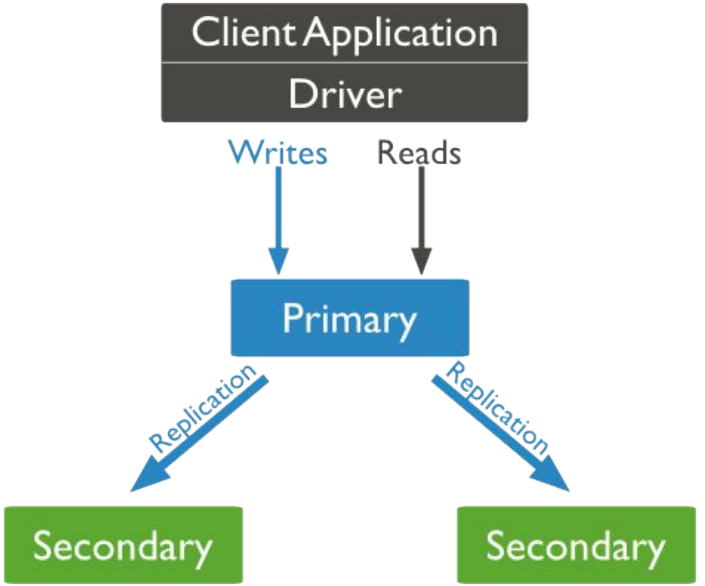
```
class Customer(EmbeddedDocument):
    id = IntField(required=True)
    given_name = StringField()
    family_name = StringField()

class Order(Document):
    meta = {"collection": "orders"}
    region = StringField()
    order_date = DateTimeField()
    customer = EmbeddedDocumentField(Customer)

Order.objects(region="EU-CZ")
     .order_by("-order_date") [:20]
```

```
db.orders.find(
  { region: "EU-CZ" }
)
.sort({ order_date: -1 })
.limit(20)
```

More on Internals of 

Replication

- Master/slave replication
- Replica set = group of instances that host the same data set
  - **primary** (master) – receives all write operations
  - **secondaries** (slaves) – apply operations from the primary so that they have the same data set



Replication Steps

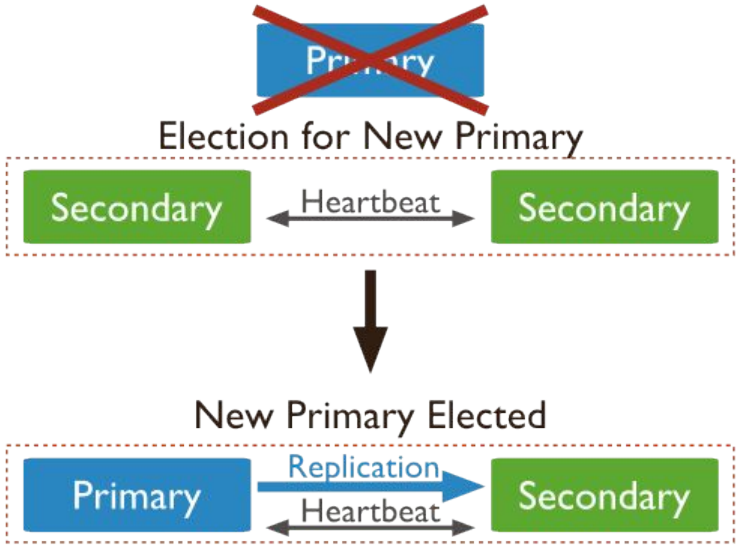
- **Write:**
  1. mongoDB applies write operations on the primary
  2. mongoDB records the operations to the primary's oplog
  3. Secondary members replicate oplog + apply the operations to their data sets
- **Read:** All members of the replica set can accept read operations
  - By default, an application directs its read operations to the primary member
    - Guaranties the latest version of a document
    - Decreases read throughput
  - Read preference mode can be set

Replication – Read Preference Mode

Read Preference Mode	Description
`primary`	operations read from the current replica set primary
`primaryPreferred`	operations read from the primary, but if unavailable, operations read from secondary members
`secondary`	operations read from the secondary members
`secondaryPreferred`	operations read from secondary members, but if none is available, operations read from the primary
`nearest`	operations read from the nearest member (= shortest ping time) of the replica set, irrespective of the member's type

Replica Set Elections

- Replica set can have at most one primary
- If the current primary becomes unavailable, an election determines a new primary
- Note:
  - Elections need some time
    - Approx. 1 minute
  - No primary ⇒ no writes



Replica Set Elections – Influencing Factors

- **Heartbeat** (ping)
  - Every 2s sent to each other
  - No response for 10s ⇒ node is inaccessible
- **Priority comparisons**
  - Higher priority = preferred to be voted
  - Members with priority = 0
    - Cannot become primary (not eligible)
    - Cannot trigger election, but can vote
  - The current primary has the highest priority and is within 10s of the latest oplog entry ⇒ OK
  - A higher-priority member catches up to within 10s of the latest oplog entry of the current primary ⇒ elections
    - The higher-priority node has a chance to become primary
- **Connections**
  - A node cannot become primary unless it can connect to a majority of the members

Replica Set Elections – Mechanism

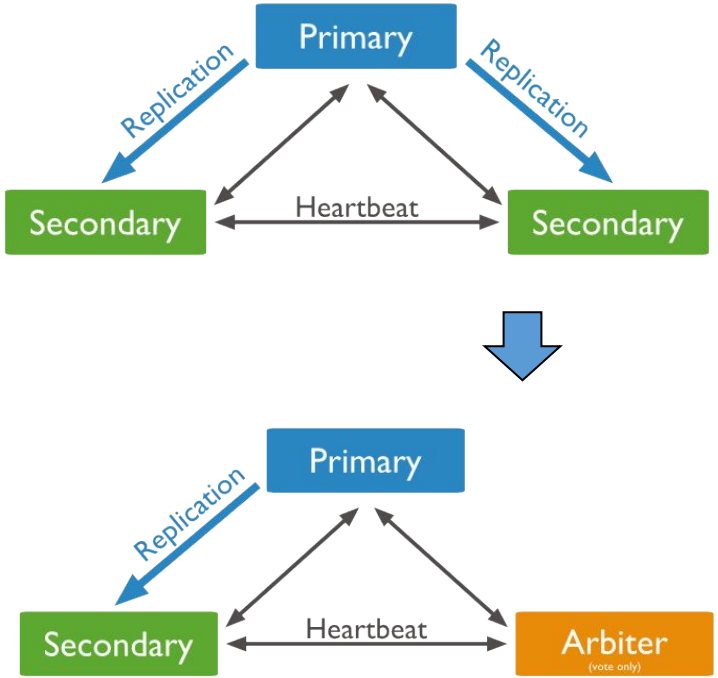
- Replica sets hold an election any time there is no primary:
  - Initiation of a new replica set
  - A secondary loses contact with a primary
  - A primary steps down
- A primary will step down:
  - After receiving the `replSetStepDown` command
    - Forces a primary to become a secondary
  - If one of the current secondaries is eligible for election and has a higher priority
  - If it cannot contact a majority of the members of the replica set

Replica Set Elections – Mechanism

- The replica set elects an eligible member with the highest priority value as primary
  - By default, all members have a priority of 1
    - Can be adjusted
- The first member to receive the majority of votes becomes primary
  - By default, all members have 1 vote
    - Can be disabled = non-voting members
      - Hold copies of data
      - Can become primary
    - Not recommended to set more than 1 (better use priority)
- All members of a replica set can veto an election, e.g.,
  - If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set
  - If the member seeking an election has a lower priority than another member in the set that is also eligible for election
  - …

Replication – Arbiters

- Arbiter
  - A special node
  - Does not maintain a data set
    - Does not require dedicated hardware
  - Cannot be a primary
  - Exists to vote in elections
    - For replicas with even number of members

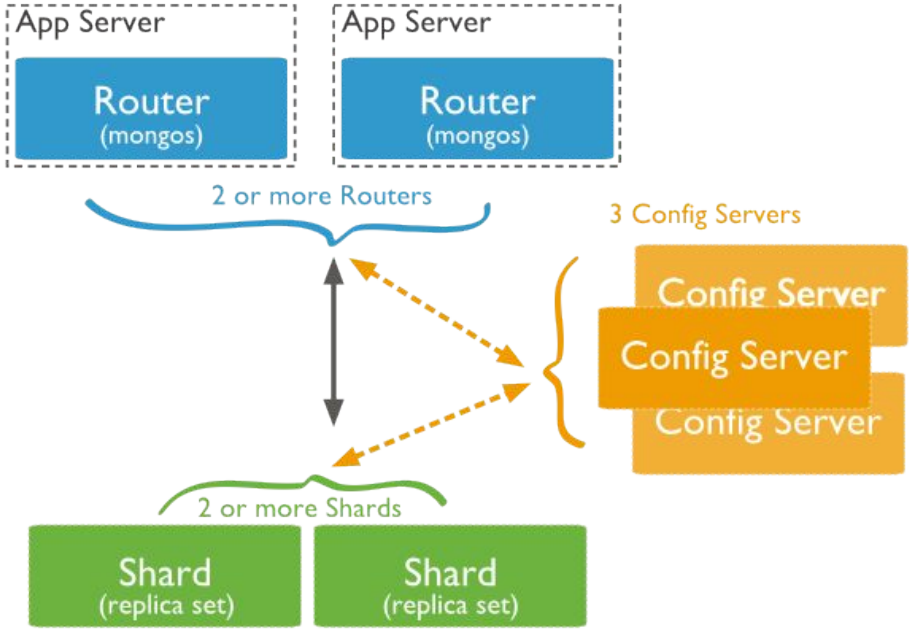


Replication – Secondaries

- A secondary can be configured as:
  - **Priority 0** – to prevent it from becoming a primary in an election
    - e.g., a standby
  - **Hidden** – to prevent applications from reading from it
    - Just replicates the data for special usage
    - Can vote in elections
  - **Delayed** – to keep a running “historical” snapshot
    - For recovery from errors like unintentionally deleted databases

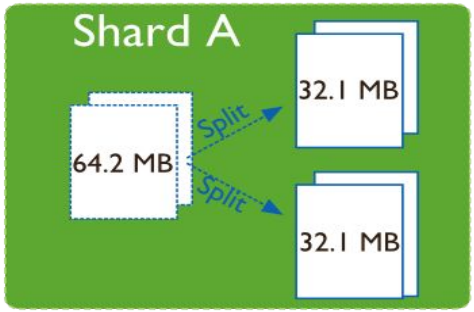
Sharding

- Supported through sharded clusters
  - **Shards** – store the data
    - Each shard is a replica set
  - **Query routers** – interface with client applications
    - Direct operations to the appropriate shard(s) + return the result to the user
    - More than one ⇒ to divide the client request load
  - **Config servers** – store the cluster's metadata
    - Mapping of the cluster's data set to the shards
    - Recommended number: 3



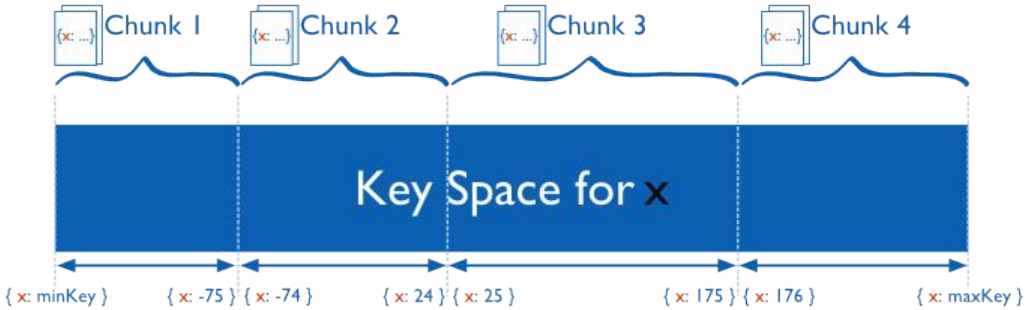
Data Partitioning

- Partitions a collection's data by the shard key
  - Indexed (possibly compound) field that exists in every document in the collection
    - Immutable
  - Divided into chunks distributed across shards
    - **Range-based partitioning**
    - **Hash-based partitioning**
  - When a chunk grows beyond the chunk size, it is split
    - Small chunks ⇒ more even distribution at the expense of more frequent migrations
    - Large chunks ⇒ fewer migrations



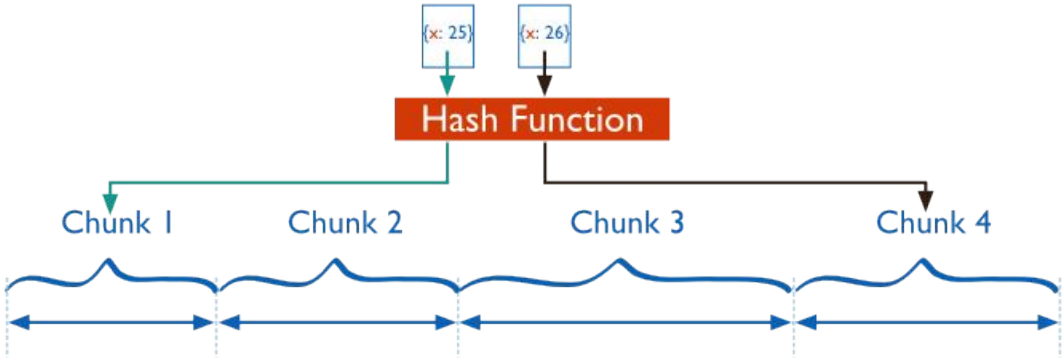
Range-Based Partitioning

- Each value of the shard key falls at some point on line from negative infinity to positive infinity
- The line is partitioned into non-overlapping chunks
- Documents with “close” shard key values are likely to be in the same chunk
  - More efficient range queries
  - Can result in an uneven distribution of data



Hash-Based Partitioning

- Computes a hash of a field's value
  - Hashes form chunks
- Ensures a more random distribution of a collection in the cluster
  - Documents with "close" shard key values are unlikely to be a part of the same chunk
  - A range query may need to target most/all shards



Transactions

- Write operations are atomic at the level of a single document
  - Including nested documents (sufficient for many cases, but not all)
- When a single write operation modifies multiple documents, it is not atomic
  - Other operations may interleave
- Isolation of a single write operation that affects multiple documents
  - No client sees the changes until the operation completes or errors out
  - `db.foo.update( { field1 : 1 , $isolated : 1 }, { $inc : { field2 : 1 } } , { multi: true } )`

mongoDB Enterprise

- Commercial edition of mongoDB
- Includes:
  - **Advanced Security** – Kerberos authentication
  - **Management Service** – a suite of tools for managing mongoDB deployments
    - Monitoring, backup capabilities, helping users optimize clusters, …
  - **Enterprise Software Integration** – SNMP support to integrate mongoDB with other tools
  - **Certified OS Support** – has been tested and certified on Red Hat/CentOS, Ubuntu, SuSE and Amazon Linux
  - …

https://www.mongoDB.com/products/mongoDB-enterprise

References

- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Tiny mongoDB Browser Shell: https://mongoplayground.net/
- mongoDB Manual: http://docs.mongoDB.org/manual/