

Chapter 20: Advanced MongoDB

20.1 MongoDB Aggregation

20.1.1 What is Aggregation?

- * Aggregation is a powerful feature in MongoDB used to process **data records** and return computed results.
- * It allows you to perform operations like **filtering**, **grouping**, **sorting**, **projecting** and **calculating**.
- * Aggregation is similar to SQL GROUP BY and is used for **data analysis** and reporting.
- * It is performed using an aggregation pipeline.

Key Points:

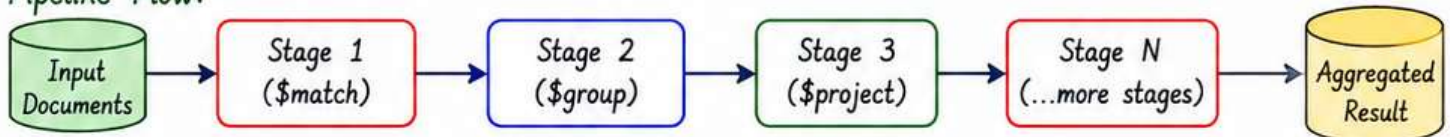
- ✓ Aggregations process data through a series of pipeline stages.
- ✓ Each stage transforms the data and passes it to the next stage.
- ✓ The output of one stage is the input for the next stage.
- ✓ It is very useful for generating reports and analytics.



20.1.2 Aggregation Pipeline

- * Aggregation operations are performed using a pipeline.
- * A pipeline consists of **one or more stages**.
- * Documents pass through the stages in sequence.
- * The output of the last stage is the final result.

Pipeline Flow:



20.1.3 Common Aggregation Stages

Stage	Description
\$match	Filters documents to pass only matching documents to the next stage.
\$group	Groups documents by a specified field and performs calculations.
\$project	Reshapes each document in the stream (include, exclude, add fields).
\$sort	Sorts documents by specified fields.
\$limit	Limits the number of documents passed to the next stage.
\$skip	Skips a specified number of documents.
\$unwind	Deconstructs an array field from the input documents to output a document for each element.
\$lookup	Performs a left outer join to another collection.

Example Pipeline

```
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: { _id: "$customerId",
    total: { $sum: "$amount" } } },
  { $sort: { total: -1 } },
  { $limit: 5 }
])
```



Key Takeaway:



MongoDB Aggregation helps in transforming and analyzing data efficiently. Using the aggregation pipeline, you can filter, group, sort and shape data to get meaningful insights from your collections.

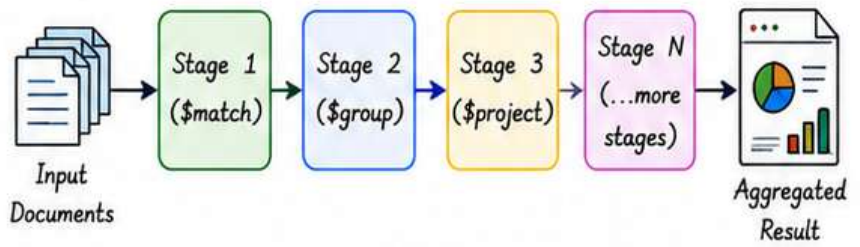


20.2 Aggregation Pipeline

20.2.1 What is Aggregation Pipeline?

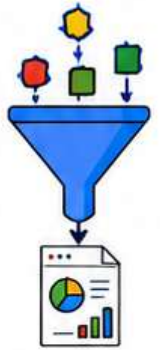
- * Aggregation Pipeline is a framework for data aggregation modeled on the concept of **data processing pipelines**.
- * It allows you to process documents through a **series of stages**.
- * Each stage performs a **specific operation** on the incoming documents.
- * The output of one stage becomes the input to the **next stage**.

Pipeline Concept



Key Points:

- ✓ Aggregation pipeline consists of one or more stages.
- ✓ Documents pass through the stages in sequence.
- ✓ Each stage transforms the documents.
- ✓ The final output is the result of the last stage.
- ✓ It is very flexible and supports complex operations.



20.2.2 Syntax

```
db.collection.aggregate([
  { <stage1>: { <stage1_specification> } },
  { <stage2>: { <stage2_specification> } },
  { ... }
]);
```

Example:

```
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: { _id: "$customerId", total: { $sum: "$amount" } } },
  { $sort: { total: -1 } }
]);
```



20.2.3 Common Pipeline Stages

Stage	Description	Example
\$match	Filters documents to pass only matching documents to the next stage.	<code>{ \$match: { status: "A" } },</code>
\$group	Groups documents by a specified field and performs calculations.	<code>{ \$group: { _id: "\$custId", total: { \$sum: "\$amount" } } }</code>
\$project	Reshapes each document in the stream (include, exclude, add fields).	<code>{ \$project: { name: 1, total: 1, _id: 0 } }</code>
\$sort	Sorts documents by specified fields.	<code>{ \$sort: { total: -1 } }</code>
\$limit	Limits the number of documents passed to the next stage.	<code>{ \$limit: 10 }</code>
\$skip	Skips a specified number of documents.	<code>{ \$skip: 20 }</code>
\$unwind	Deconstructs an array field from the input documents to output a document for each element.	<code>{ \$unwind: "\$items" }</code>
\$lookup	Performs a left outer join to another collection.	<code>{ \$lookup: { from: "customers", localField: "custId", foreignField: "_id", as: "customer" } }</code>

Key Takeaway:



Aggregation Pipeline is a powerful way to process and analyze data in MongoDB. It helps in **filtering**, **grouping**, **sorting**, and **transforming** data to get meaningful results for reporting and analytics.

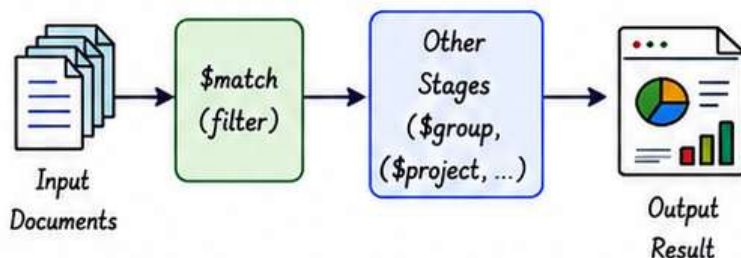


20.3 \$match

20.3.1 What is \$match?

- * **\$match** is the first stage in most aggregation pipelines.
- * It **filters** documents to pass only matching documents to the next stage.
- * It works like the **find()** method query.
- * Placing **\$match** early in the pipeline improves **performance** by reducing the number of documents to process.

Pipeline Illustration:



Key Points:

- ✓ **\$match** filters documents based on a condition.
- ✓ Only matching documents are passed to the next stage.
- ✓ It can use all MongoDB query operators.
- ✓ Using **\$match** as the first stage improves efficiency.
- ✓ It reduces the amount of data processed in the pipeline.



20.3.2 Syntax

```
db.collection.aggregate([
  { $match: { <field1>: <value1>, <field2>: <value2> } },
  { <next_stage>: { ... } },
  ...
]):
```

Example:

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



20.3.3 Common Examples

Use Case	Example Query	Description
Match equal value	<code>{ \$match: { status: "A" } }</code>	Matches documents where status is "A".
Match multiple conditions	<code>{ \$match: { status: "A", amount: { \$gt: 1000 } } }</code>	Matches documents where status is "A" and amount greater than 1000.
Match with OR	<code>{ \$match: { \$or: [{ status: "A" }, { status: "B" }] } }</code>	Matches documents where status is "A" or "B".
Match in array	<code>{ \$match: { tags: "electronics" } }</code>	Matches documents where array field tags contains "electronics".
Match exists	<code>{ \$match: { email: { \$exists: true } } }</code>	Matches documents where email field exists.
Match with regex	<code>{ \$match: { name: { \$regex: "^A" } } }</code>	Matches documents where name starts with "A".

Best Practices:

- * Always use **\$match** as the first stage if possible.
- * Filter as early as possible to reduce data processing.
- * Use indexes on fields used in **\$match** for better performance.
- * Avoid complex expressions in **\$match** if simpler ones work.



Key Takeaway:

\$match is used to filter documents in aggregation pipeline. It reduces the number of documents to process and improves performance.

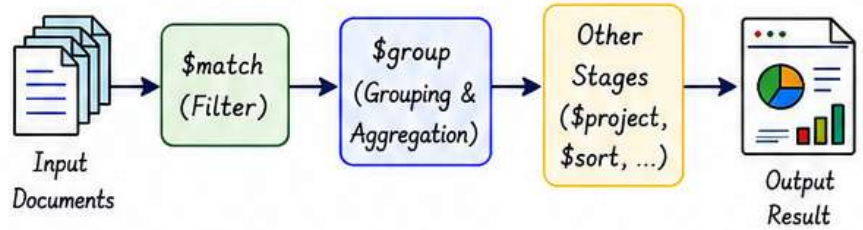


20.4 \$group

20.4.1 What is \$group?

- * **\$group** is an aggregation stage that groups documents by a specified **_id** expression and applies accumulator expressions.
- * It combines multiple documents with the same **_id** into a single document.
- * It is similar to SQL **GROUP BY**.
- * It is used to perform calculations on groups of documents.

Pipeline Illustration:



Key Points:

- ✓ **\$group** groups documents by a specified **_id**.
- ✓ It calculates aggregated values using accumulator operators.
- ✓ Each group produces a single output document.
- ✓ It reduces multiple documents into summarized information.
- ✓ Common accumulators: **\$sum**, **\$avg**, **\$min**, **\$max**, **\$count**, etc.



20.4.2 Syntax

```
db.collection.aggregate([
  { $group: {
    _id: <expression>,
    <field1>: { <accumulator1>: <expression> },
    <field2>: { <accumulator2>: <expression> },
    ...
  }},
  { <next_stage>: { ... } },
  ...
]);
```

Example:

```
db.orders.aggregate([
  { $group: {
    _id: "$customerId",
    totalAmount: { $sum: "$amount" },
    orderCount: { $sum: 1 },
    avgAmount: { $avg: "$amount" }
  }},
  { $sort: { totalAmount: -1 } }
]);
```



20.4.3 Common Accumulators

Accumulator	Description	Example
\$sum	Calculates the sum of numeric values.	{ \$sum : "\$amount" }
\$avg	Calculates the average of numeric values.	{ \$avg : "\$amount" }
\$min	Finds the minimum value.	{ \$min : "\$amount" }
\$max	Finds the maximum value.	{ \$max : "\$amount" }
\$count	Counts the number of documents.	{ \$sum : 1 }
\$first	Returns the first document in each group.	{ \$first : "\$name" }
\$last	Returns the last document in each group.	{ \$last : "\$name" }
\$push	Adds values to an array.	{ \$push : "\$item" }
\$addToSet	Adds unique values to an array.	{ \$addToSet : "\$item" }

Example Use Case:

Find total sales and number of orders placed by each customer.

```
db.orders.aggregate([
  { $group: {
    _id: "$customerId",
    totalSales: { $sum: "$amount" },
    totalOrders: { $sum: 1 }
  } }
]);
```



Key Takeaway:



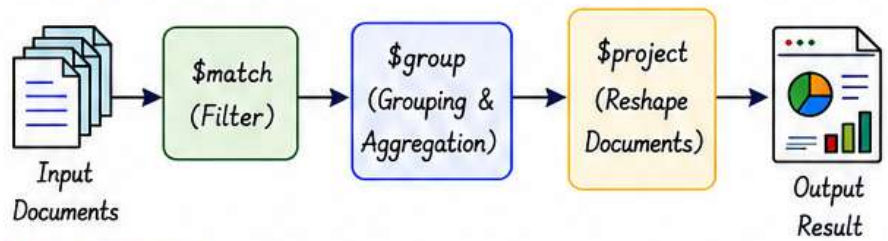
\$group is used to group documents and perform calculations on each group. It helps in generating summary reports and insights from raw data. It is a powerful stage for data analysis and reporting.

20.5 \$project

20.5.1 What is \$project?

- * **\$project** is an aggregation stage used to **reshape** each document in the pipeline.
- * It **includes**, **excludes**, or **creates** fields in the output documents.
- * It allows you to **control the structure** of the data.
- * It is similar to **SELECT** in SQL.

Pipeline Illustration:



Key Points:

- ✓ **\$project** reshapes the documents.
- ✓ It can include specific fields and exclude others.
- ✓ It can create new fields using expressions.
- ✓ Fields can be renamed using \$project.
- ✓ It helps reduce the amount of data passed to the next stage.



20.5.2 Syntax

```
db.collection.aggregate([
  { $project: {
    <field1>: <value1>,
    <field2>: <value2>,
    <fieldN>: <valueN>
  } },
  { <next_stage>: { ... } },
  ...
]);
```

Examples:

```
// Include specific fields
db.users.aggregate([
  { $project: { name: 1, email: 1, _id: 0 } }
]);

// Rename a field and create a new field
db.orders.aggregate([
  { $project: {
    customer: "$custName",
    totalAmount: "$amount",
    tax: { $multiply: [ "$amount", 0.18 ] },
    _id: 0
  } }
]);
```



20.5.3 \$project Operations

Operation	Description	Example	Explanation
Include Fields	Include specific fields in the output.	{ \$project: { name: 1, email: 1, _id: 0 } }	Only name and email are included.
Exclude Fields	Exclude specific fields from the output.	{ \$project: { password: 0, _id: 0 } }	All fields except password and _id.
Rename Fields	Rename a field in the output.	{ \$project: { customerName: "\$name", _id: 0 } }	Field name is renamed to customerName.
Create New Fields	Create new fields using expressions.	{ \$project: { total: "\$amount", tax: { \$multiply: ["\$amount", 0.18] }, } } _id: 0	Creates new field tax based on amount.
Nested Fields	Include nested fields.	{ \$project: { "address.city": 1, name: 1, _id: 0 } }	Includes only address.city and name.

20.5.4 Example Use Case

```
Find customer names and total amount of orders.
db.orders.aggregate([
  { $project: {
    customerName: "$customer.name",
    totalAmount: "$amount",
    _id: 0
  } }
]);
```



Key Takeaway:



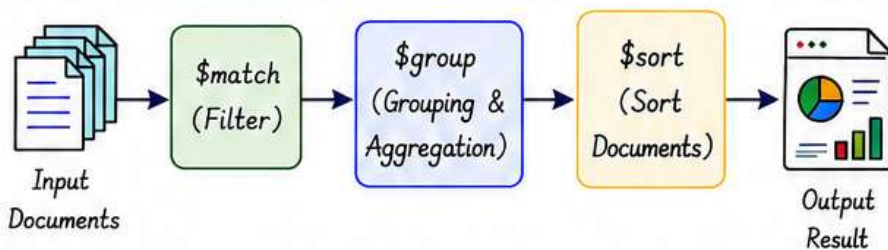
\$project is used to reshape documents by including, excluding, renaming or creating fields. It helps in controlling the output structure and reducing unnecessary data. It is an important stage for preparing data for further processing or reporting.

20.6 \$sort

20.6.1 What is \$sort?

- * **\$sort** is an aggregation stage that **sorts** the documents based on the specified fields.
- * It arranges the documents in either **ascending** or **descending** order.
- * It is similar to **ORDER BY** in SQL.
- * It is used to organize the output documents for better readability or further processing.

Pipeline Illustration:



Key Points:

- ✓ **\$sort** sorts documents based on one or more fields.
- ✓ You can specify ascending (1) or descending (-1) order.
- ✓ Documents are sorted before passing to the next stage.
- ✓ Sorting large datasets can be resource intensive.
- ✓ Useful for presenting results in a meaningful order.



20.6.2 Syntax

```
db.collection.aggregate([
  { $sort: { <field1>: <order1>,
             <field2>: <order2>, ... } },
  { <next_stage>: { ... } },
  ...
]);
```

Order: 1 for ascending, -1 for descending.

Examples:

```
// Sort by one field (ascending)
db.users.aggregate([
  { $sort: { name: 1 } }
]);

// Sort by one field (descending)
db.users.aggregate([
  { $sort: { age: -1 } }
]);

// Sort by multiple fields
db.orders.aggregate([
  { $sort: { totalAmount: -1, orderDate: 1 } }
]);
```



20.6.3 Sort Orders

Order	Description	Example	Output Order
1	Ascending order (A to Z, 0 to 9, smallest to largest)	{ \$sort: { name: 1 } }	A, B, C ... Z 0, 1, 2 ... 9 Smallest → Largest
-1	Descending order (Z to A, 9 to 0, largest to smallest)	{ \$sort: { name: -1 } }	Z, Y, X ... A 9, 8, 7 ... 0 Largest → Smallest
Multiple Fields	Sort by multiple fields in the specified order	{ \$sort: { total: -1, date: 1 } }	First by total (desc), then by date (asc)

20.6.4 Example Use Case

Find the top 5 most expensive orders.

```
db.orders.aggregate([
  { $sort: { totalAmount: -1 } },
  { $limit: 5 }
]);
```

Result: Returns the 5 orders with highest totalAmount.



Key Takeaway:



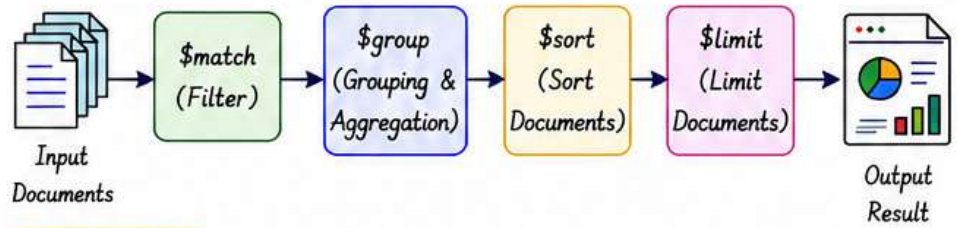
- * **\$sort** is used to sort documents in ascending or descending order.
- * It helps in organizing data for better presentation and analysis.
- * Use **indexes** on sorted fields to improve performance.
- * Place **\$sort** after **\$match** or **\$group** to reduce the amount of data to sort.

20.7 \$limit

20.7.1 What is \$limit?

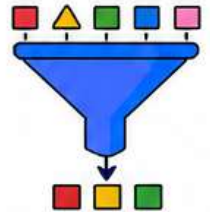
- * **\$limit** is an aggregation stage that **limits** the number of documents passed to the next stage.
- * It specifies the maximum number of documents to output.
- * It is similar to **LIMIT** in SQL.
- * It is used to **restrict** the result set to a specific number of documents.

Pipeline Illustration:



Key Points:

- ✓ **\$limit** restricts the number of documents passed to the next stage.
- ✓ It accepts a positive integer value.
- ✓ Only the specified number of documents are passed forward.
- ✓ It is useful for pagination and limiting large datasets.
- ✓ Using **\$limit** early in the pipeline can improve performance.



20.7.2 Syntax

```
db.collection.aggregate([
  { <previous_stage>: { ... } },
  ...
  { $limit: <number> },
  { <next_stage>: { ... } },
  ...
]);
```

<number> is the maximum number of documents to output.

Examples:

```
// Limit to first 10 documents
db.users.aggregate([
  { $limit: 10 }
]);

// Limit after sorting
db.orders.aggregate([
  { $sort: { totalAmount: -1 } },
  { $limit: 5 }
]);

// Limit after grouping
db.sales.aggregate([
  { $group: { _id: "$custId", total: { $sum: "$amount" } } },
  { $sort: { total: -1 } },
  { $limit: 3 }
]);
```



20.7.3 Common Use Cases

Use Case	Example Query	Description
Get first N documents	<code>{ \$limit: 10 }</code>	Returns the first 10 documents from the pipeline.
Top N results	<code>{ \$sort: { totalAmount: -1 } },</code> <code>{ \$limit: 5 }</code>	Returns top 5 documents with highest totalAmount.
Pagination	<code>{ \$skip: 20 },</code> <code>{ \$limit: 10 }</code>	Skips first 20 documents and returns next 10.
Limit after grouping	<code>{ \$group: { _id: "\$category", total: { \$sum: "\$amount" } } }</code> <code>{ \$limit: 3 }</code>	Returns top 3 groups based on aggregation.
Performance optimization	<code>{ \$match: { status: "A" } },</code> <code>{ \$limit: 100 }</code>	Limits the number of documents early to reduce processing.

20.7.4 Example Use Case

Find the top 5 highest spending customers.

```
db.orders.aggregate([
  { $group: { _id: "$customerId", totalSpent: { $sum: "$amount" } } },
  { $sort: { totalSpent: -1 } },
  { $limit: 5 }
]);
```

Result: Returns the top 5 customers with highest totalSpent.



Key Takeaway:



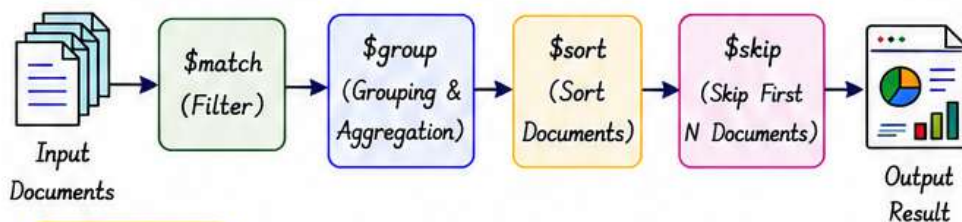
- * **\$limit** is used to restrict the number of documents in the output.
- * It is very useful for top-N queries, pagination and performance optimization.
- * Place **\$limit** after **\$sort** or **\$group** to get meaningful results.
- * Use **\$skip** along with **\$limit** for pagination.

20.8 \$skip

20.8.1 What is \$skip?

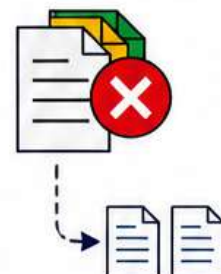
- * **\$skip** is an aggregation stage that skips a specified number of documents and passes the remaining documents to the next stage.
- * It is used to **exclude** the first N documents from the pipeline.
- * It is similar to **OFFSET** in SQL.
- * It is commonly used for pagination along with **\$limit**.

Pipeline Illustration:



Key Points:

- ✓ **\$skip** skips the specified number of documents.
- ✓ It accepts a non-negative integer value.
- ✓ The skipped documents are not passed to the next stage.
- ✓ It is used for pagination and result set navigation.
- ✓ Using **\$skip** alone on large datasets can be slow.
- ✓ It is best used with **\$sort** for consistent results.



20.8.2 Syntax

```
db.collection.aggregate([
  { <previous_stage>: { ... } },
  ...
  { $skip: <number> },
  { <next_stage>: { ... } },
  ...
]);
```

<number> is the number of documents to skip.

Examples:

```
// Skip first 10 documents
db.users.aggregate([
  { $skip: 10 }
]);

// Skip first 20 documents and then limit to next 5
db.orders.aggregate([
  { $sort: { totalAmount: -1 } },
  { $skip: 20 },
  { $limit: 5 }
]);

// Skip after grouping
db.sales.aggregate([
  { $group: { _id: "$category", total: { $sum: "$amount" } } },
  { $sort: { total: -1 } },
  { $skip: 5 }
]);
```



20.8.3 Common Use Cases

Use Case	Example Query	Description
Pagination (Skip & Limit)	<pre>{ \$skip: (page-1) * pageSize }, { \$limit: pageSize }</pre>	Skips documents for previous pages and returns the next set of results.
Skip Top N Results	<pre>{ \$sort: { totalAmount: -1 } }, { \$skip: 10 }</pre>	Skips top 10 highest documents and returns the rest.
Skip After Filtering	<pre>{ \$match: { status: "A" } }, { \$skip: 15 }</pre>	Skips first 15 documents after filtering the data.
Skip After Grouping	<pre>{ \$group: { _id: "\$city", total: { \$sum: "\$amount" } } }, { \$sort: { total: -1 } }, { \$skip: 5 }</pre>	Skips first 5 grouped results and returns the remaining.
Large Dataset Navigation	<pre>{ \$sort: { createdAt: -1 } }, { \$skip: 1000 }, { \$limit: 20 }</pre>	Navigates to a specific position in a large dataset.

20.8.4 Example Use Case

Find the customers after skipping the first 15 default sorted by name (A to Z).

```
db.customers.aggregate([
  { $sort: { name: 1 } },
  { $skip: 15 },
  { $limit: 10 }
]);
```

Result: Returns the next 10 customers after skipping the first 15 names.



Key Takeaway:

- * **\$skip** is used to skip a specified number of documents.
- * It is useful for pagination and navigating result sets.
- * Always use **\$sort** with **\$skip** for consistent and predictable results.
- * Avoid using **\$skip** on large datasets without proper indexing as it can impact performance.
- * Combine **\$skip** with **\$limit** for efficient paging.

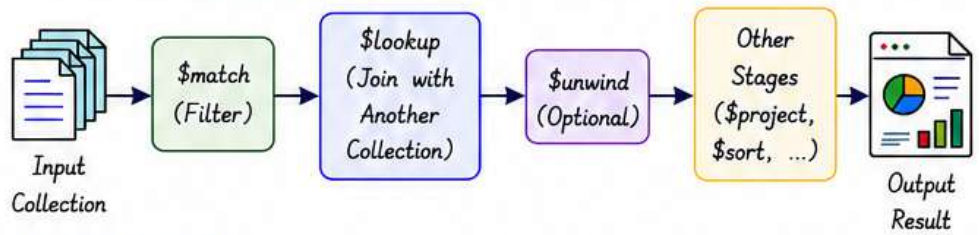


20.9 \$lookup

20.9.1 What is \$lookup?

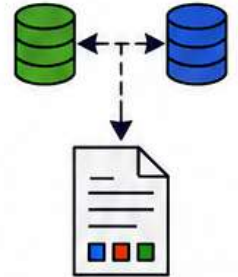
- * **\$lookup** is an aggregation stage used to perform a left outer join between two collections.
- * It combines documents from the input collection with matching documents from a joined collection.
- * It adds an array field to each document that contains the matching documents from the joined collection.
- * It is similar to **JOIN** in SQL.
- * It is very useful for combining related data from multiple collections.

Pipeline Illustration:



Key Points:

- ✓ **\$lookup** performs a left outer join between collections.
- ✓ It requires the local field and foreign field to match.
- ✓ The result is added as an array field in the output document.
- ✓ If no match is found, an empty array is returned.
- ✓ It helps in combining data from related collections.
- ✓ Use **\$unwind** after **\$lookup** to deconstruct the array (if needed).



20.9.2 Syntax

```
db.collection.aggregate([
  { $lookup: {
    from: "<joined_collection>",
    localField: "<local_field>",
    foreignField: "<foreign_field>",
    as: "<output_array_field>"
  } },
  { <next_stage>: { ... } },
  ...
]);
```

from : The collection to join with.
localField : Field from the input collection.
foreignField : Field from the joined collection.
as : Name of the new array field to store matched documents.

20.9.3 Example

Example:

Suppose we have two collections:

orders (input)

_id	customerId	amount
1	C101	250
2	C102	300
3	C103	150

customers (joined)

_id	name	city
C101	Alice	Hyderabad
C102	Bob	Bangalore
C104	Charlie	Chennai

Query: Get orders along with customer details.

```
db.orders.aggregate([
  { $lookup: {
    from: "customers",
    localField: "customerId",
    foreignField: "_id",
    as: "customerDetails"
  } }
]);
```



20.9.4 How \$lookup Works?

For each document in the input collection, **\$lookup** finds matching documents in the joined collection and adds them as an array.

orders (Input Collection)

_id	customerId	amount
1	C101	250
2	C102	300
3	C103	150

\$lookup
(from: customers,
localField: customerId,
foreignField: _id,
as: customerDetails)

Output (After \$lookup)

_id	customerId	amount	customerDetails (Array)
1	C101	250	[{ _id: "C101", name: "Alice", city: "Hyderabad" }]
2	C102	300	[{ _id: "C102", name: "Bob", city: "Bangalore" }]
3	C103	150	[] (no match)

20.9.5 Example Use Case

Get all orders along with customer details and total amount greater than 200.

```
db.orders.aggregate([
  { $match: { amount: { $gt: 200 } } },
  { $lookup: {
    from: "customers",
    localField: "customerId",
    foreignField: "_id",
    as: "customer"
  } },
  { $unwind: "$customer" },
  { $project: { _id: 0, customer.name: 1,
    customer.city: 1, amount: 1 } }
]);
```



Key Takeaway:

- * **\$lookup** is used to join collections in MongoDB.
- * It performs a left outer join.
- * Use **\$unwind** after **\$lookup** if you want to work with individual matched documents.
- * It helps to build powerful reports and dashboards by combining related data.
- * Always index the fields used in **\$lookup** for better performance.

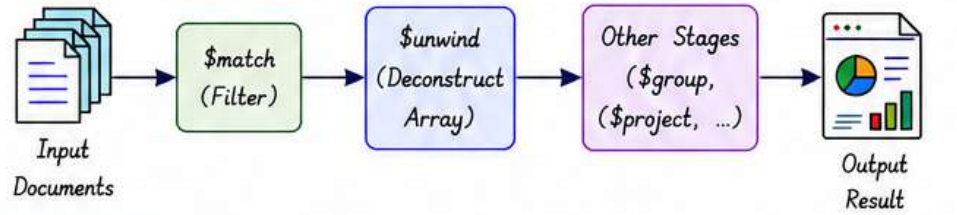


20.10 \$unwind

20.10.1 What is \$unwind?

- * **\$unwind** is an aggregation stage used to deconstruct an array field from the input documents and output a separate document for each element of the array.
- * It transforms an array field into multiple documents.
- * It is the **opposite** of **\$group**.
- * It is very useful when we need to work with individual array elements.
- * It is similar to **UNNEST** in SQL.

Pipeline Illustration:



Key Points:

- ✓ **\$unwind** deconstructs an array field.
- ✓ It creates a new document for each element in the array.
- ✓ The original array field becomes a single value in the output.
- ✓ If the field is not an array, it treats the field as a single element.
- ✓ Use **preserveNullAndEmptyArrays** option to keep documents with null, missing or empty arrays.

Before \$unwind

```
{
  "_id": 1,
  "items": [
    "A",
    "B",
    "C"
  ]
}
```

After \$unwind

```
{
  "_id": 1,
  "items": "A"
}
{
  "_id": 1,
  "items": "B"
}
{
  "_id": 1,
  "items": "C"
}
```

20.10.2 Syntax

```
db.collection.aggregate([
  { $unwind: <path> },
  { <next_stage>: { ... } },
  ...
])
```

<path> : The field path to an array.

20.10.3 Examples

Example 1: Unwind simple array

Input (students)

_id	name	subjects
1	Alice	["Maths", "CS", "DBMS"]
2	Bob	["OS", "CN"]

After \$unwind: "\$subjects"

_id	name	subjects
1	Alice	"Maths"
1	Alice	"CS"
1	Alice	"DBMS"
2	Bob	"OS"
2	Bob	"CN"

Example 2: With preserveNullAndEmptyArrays

Input (products)

_id	name	tags
1	Laptop	["Electronics", "Computer"]
2	Chair	[]
3	Table	null
4	Pen	["Stationery"]

After \$unwind: "\$tags"

(preserveNullAndEmptyArrays: true)

_id	name	tags
1	Laptop	"Electronics"
1	Laptop	"Computer"
2	Chair	null
3	Table	null
4	Pen	"Stationery"



20.10.4 How \$unwind Works?

Input Document

```
{
  "_id": 1,
  "items": [
    "X",
    "Y",
    "Z"
  ]
}
```

Output Documents

```
{
  "_id": 1,
  "items": "X"
}
{
  "_id": 1,
  "items": "Y"
}
{
  "_id": 1,
  "items": "Z"
}
```

- * Each element of the array becomes a separate document.
- * The array field becomes the individual element in each output document.

Useful Options with \$unwind

Option	Description
preserveNullAndEmptyArrays: true	Includes documents where the array is null, missing or empty.
preserveNullAndEmptyArrays: false (default)	Excludes such documents.
includeArrayIndex: "index"	Adds a new field with the index of the array element.

20.10.5 Example Use Case

Find total sales per product from orders where each order has multiple products.

```
db.orders.aggregate([
  { $unwind: "$products" },
  { $group: {
    _id: "$products.name",
    totalSales: { $sum: "$products.price" }
  } },
  { $sort: { totalSales: -1 } }
])
```



Key Takeaway:



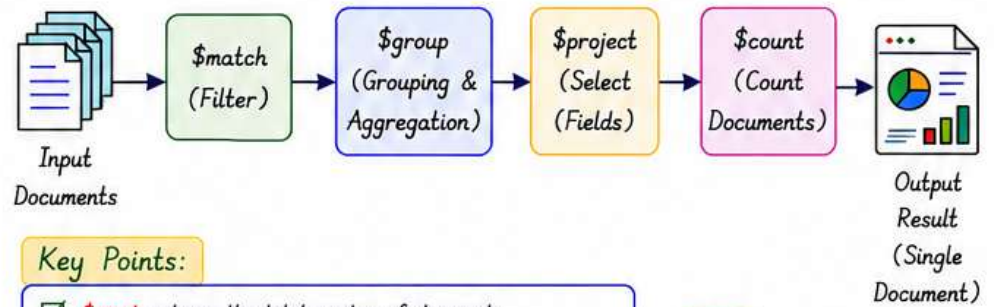
- * **\$unwind** is used to deconstruct array fields.
- * It outputs a separate document for each array element.
- * It is the opposite of **\$group**.
- * Use **preserveNullAndEmptyArrays** to keep empty, null or missing arrays.
- * Often used with **\$group**, **\$lookup** and **\$project** in real-world pipelines.

20.11 \$count

20.11.1 What is \$count?

- * **\$count** is an aggregation stage that returns the total number of documents that pass through the previous stage of the pipeline.
- * It outputs a single document containing the count.
- * It is similar to **COUNT(*)** in SQL.
- * It is usually the last stage in the pipeline.
- * It is useful to get the total number of documents after filtering, grouping or other operations.

Pipeline Illustration:



Key Points:

- ✓ **\$count** returns the total number of documents.
- ✓ It must be the last stage in the aggregation pipeline.
- ✓ It returns a single document with the specified field name.
- ✓ The field name must be a string (e.g., "totalCount").
- ✓ It is useful after **\$match**, **\$group** or other filtering stages.
- ✓ It helps in reporting, analytics and pagination.



```
Result
{
  "totalCount": 125
}
```

20.11.2 Syntax

```
db.collection.aggregate([
  { <previous_stage>: { ... } },
  ...
  { $count: "<field_name>" }
]);
```

<field_name> : The name of the field that will store the count.

20.11.3 Examples

Example 1: Count all documents in a collection

```
db.users.aggregate([
  { $count: "totalUsers" }
]);
```

Result:

totalUsers
8



Example 2: Count after filtering

```
db.users.aggregate([
  { $match: { status: "Active" } },
  { $count: "activeUsers" }
]);
```

Result:

activeUsers
5

Example 3: Count after grouping

```
db.orders.aggregate([
  { $group: { _id: "$status", total: { $sum: 1 } } },
  { $count: "statusCount" }
]);
```

Result:

statusCount
3



20.11.4 Common Use Cases

Use Case	Example Query	Description
Total documents in a collection	<pre>{ \$count: "totalCount" }</pre>	Counts all documents in the collection.
Count after filtering	<pre>{ \$match: { status: "A" } }, { \$count: "activeCount" }</pre>	Counts documents that match a condition.
Count after grouping	<pre>{ \$group: { _id: "\$city" } }, { \$count: "cityCount" }</pre>	Counts the number of groups.
Pagination (Total Records)	<pre>{ \$match: { age: { \$gte: 18 } } }, { \$count: "totalRecords" }</pre>	Finds total records for pagination.
Reports & Analytics	<pre>{ \$group: { _id: "\$category" } }, { \$count: "categoryCount" }</pre>	Counts categories, statuses, or other groups.

20.11.5 Example Use Case

Find the total number of orders where totalAmount > 500.

```
db.orders.aggregate([
  { $match: { totalAmount: { $gt: 500 } } },
  { $count: "highValueOrders" }
]);
```

Result:

highValueOrders
12



Key Takeaway:



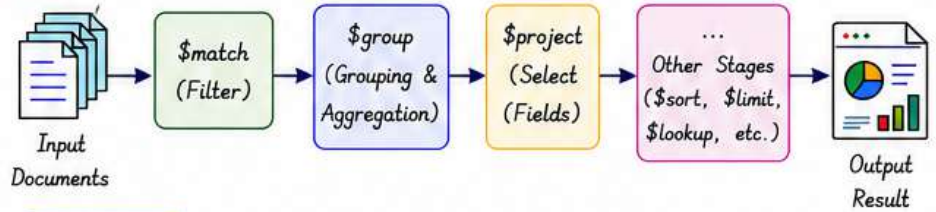
- * **\$count** returns the total number of documents as a single document.
- * It must be the last stage in the pipeline.
- * Use it after **\$match**, **\$group** or other stages to get meaningful counts.
- * Provide a field name (e.g., "totalCount").
- * Very useful for reporting, dashboards, analytics and pagination.

20.12 Aggregation with Mongoose

20.12.1 What is Aggregation in Mongoose?

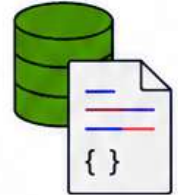
Pipeline Illustration:

- * Mongoose provides the `aggregate()` function to perform aggregation operations on MongoDB collections.
- * It is a wrapper around the MongoDB Aggregation Framework.
- * It allows us to process data through a pipeline of stages and return the results.
- * It is very useful for reporting, analytics, filtering, grouping, joining and more.
- * It returns a promise which resolves to an array.



Key Points:

- ✓ Use Model.`aggregate()` to run aggregation.
- ✓ The pipeline is an array of stages.
- ✓ Each stage is an object with a single operator.
- ✓ The last stage returns the final output.
- ✓ It is ideal for complex queries and data analysis.



20.12.2 Basic Syntax

```
Model.aggregate([
  { <stage1> },
  { <stage2> },
  ...
]).then(result => {
  console.log(result);
}).catch(err => {
  console.error(err);
});
```

Model : Mongoose model (collection)
aggregate() : Starts the aggregation pipeline
pipeline : Array of stages
result : Output documents (array)
err : Error object (if any)

20.12.3 Example

Example: Get total sales per category

Assume a collection: sales

Fields: `_id`, `product`, `category`, `amount`, `date`

Requirement: Find total sales amount per category.

```
const mongoose = require('mongoose');
const Sale = require('./models/sale'); // Sale model

async function getSalesByCategory() {
  try {
    const result = await Sale.aggregate([
      { $group: {
        _id: "$category",
        totalSales: { $sum: "$amount" },
        count: { $sum: 1 }
      } },
      { $sort: { totalSales: -1 } }
    ]);
    console.log("Result:", result);
  } catch (err) {
    console.error("Error:", err);
  }
}

getSalesByCategory();
```

Sample Output

<code>_id</code> (category)	<code>totalSales</code>	<code>count</code>
Electronics	12500	45
Clothing	8200	30
Books	4100	18
Accessories	2500	12



20.12.4 Common Aggregation Stages in Mongoose

Stage	Description	Example Use
<code>\$match</code>	Filters documents.	{ <code>\$match</code> : { <code>status</code> : "Active" } }
<code>\$group</code>	Groups documents and performs aggregations.	{ <code>\$group</code> : { <code>_id</code> : "\$category", <code>total</code> : { <code>\$sum</code> : "\$amount" } } }
<code>\$project</code>	Includes, excludes or reshapes fields.	{ <code>\$project</code> : { <code>_id</code> : 0, <code>name</code> : 1, <code>total</code> : 1 } }
<code>\$sort</code>	Sorts documents.	{ <code>\$sort</code> : { <code>total</code> : -1 } }
<code>\$limit</code>	Limits the number of documents.	{ <code>\$limit</code> : 10 }
<code>\$skip</code>	Skips a number of documents.	{ <code>\$skip</code> : 20 }
<code>\$lookup</code>	Performs a left outer join with another collection.	{ <code>\$lookup</code> : { <code>from</code> : "users", <code>localField</code> : "userId", <code>foreignField</code> : "_id", <code>as</code> : "userInfo" } }
<code>\$unwind</code>	Deconstructs an array field.	{ <code>\$unwind</code> : "\$items" }
<code>\$count</code>	Counts the number of documents.	{ <code>\$count</code> : "totalCount" }

20.12.5 Another Example (Using \$lookup)

Get all orders along with user details.

```
Order.aggregate([
  { $lookup: {
    from: "users",
    localField: "userId",
    foreignField: "_id",
    as: "userInfo"
  } },
  { $unwind: "$userInfo" },
  { $project: { _id: 1, total: 1, "userInfo.name": 1, "userInfo.email": 1 } }
]).then(res => console.log(res))
.catch(err => console.error(err));
```



Key Takeaway:



- * `aggregate()` in Mongoose gives powerful data processing capabilities.
- * Use pipeline stages to filter, group, join, sort and transform data.
- * It is very useful for reports, dashboards and analytics.
- * Always handle errors using `try...catch` with `async/await`.

20.13 Advanced Indexing

20.13.1 What is Advanced Indexing?

- * Advanced indexing in MongoDB allows us to create special types of indexes for better performance and more powerful query capabilities.
- * It goes beyond simple single-field indexes.
- * It includes compound indexes, unique indexes, sparse indexes, partial indexes, TTL indexes, text indexes, geospatial indexes, hashed indexes and wildcard indexes.

Key Points:

- ✓ Indexes improve read performance.
- ✓ More indexes may slow down write operations.
- ✓ Choose the right index based on query patterns.
- ✓ Use `explain()` to analyze index usage.
- ✓ Do not over-index; it consumes storage.
- ✓ Monitor index usage and remove unused indexes.

20.13.2 Syntax

```
// Create an index
collection.createIndex({ field1: 1, field2: -1 }, options);

// Drop an index
collection.dropIndex("index_name");

// List indexes
collection.getIndexes();

// Explain query (to see index usage)
collection.find(query).explain("executionStats");
```

1 -> Ascending order

-1 -> Descending order

options -> Additional settings specific to the index

20.13.4 Advanced Index Types at a Glance

Index Type	Syntax / Example	Use Case	Notes
Compound Index	<code>db.collection.createIndex({ a: 1, b: -1 })</code>	Queries filtering/sorting on multiple fields	Supports equality, range and sort operations efficiently
Unique Index	<code>db.collection.createIndex({ email: 1 }, { unique: true })</code>	Enforce uniqueness of a field	Duplicate values are not allowed
Sparse Index	<code>db.collection.createIndex({ field: 1 }, { sparse: true })</code>	Index documents that have the field only	Documents missing the field are not indexed
Partial Index	<code>db.collection.createIndex({ status: 1 }, { partialFilterExpression: { status: "A" } })</code>	Index subset of documents matching a condition	Smaller index size, better performance for filtered queries.
TTL Index	<code>db.collection.createIndex({ createdAt: 1 }, { expireAfterSeconds: 86400 })</code>	Auto delete old documents (e.g., logs, sessions)	Time To Live in seconds
Text Index	<code>db.collection.createIndex({ name: "text", description: "text" })</code>	Full text search	Use <code>\$text</code> in queries
Geospatial Index	<code>db.collection.createIndex({ location: "2dsphere" })</code>	Location based queries (nearest, within, etc.)	Requires GeoJSON data
Hashed Index	<code>db.collection.createIndex({ userId: "hashed" })</code>	Sharding and even data distribution	Cannot support range queries
Wildcard Index	<code>db.collection.createIndex({ "\$**": 1 })</code>	Index all fields (flexible schemas)	Increases storage; use carefully

20.13.5 Explain (Index Usage)

```
db.orders.find({ customerId: "C101" })
    .explain("executionStats");
```

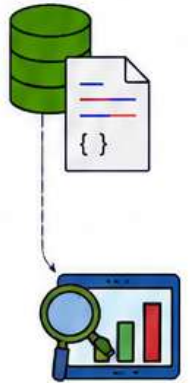
Output (summary):

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "IXSCAN",
      "indexName": "customerId_1"
    },
    "executionStats": {
      "totalKeysExamined": 10,
      "totalDocsExamined": 10,
      "executionTimeMillis": 2
    }
  }
}
```



Index Types Overview:

Compound Index	Index on multiple fields.
Unique Index	Ensures all values are unique.
Sparse Index	Index only documents that contain the field.
Partial Index	Index documents that match a filter expression.
TTL Index	Automatically removes documents after a specified time.
Text Index	Supports full-text search.
Geospatial Index	Supports location-based queries.
Hashed Index	Indexes the hash of a field value.
Wildcard Index	Indexes all fields in a document (or subdocument).



20.13.3 Examples

1. Compound Index

```
db.orders.createIndex({
  customerId: 1, date: -1
});
// Index on customerId (asc) and
// date (desc)
```

2. Unique Index

```
db.users.createIndex({
  email: 1
}, { unique: true });
// Ensures email is unique
```

3. Sparse Index

```
db.products.createIndex({
  discount: 1
}, { sparse: true });
// Only documents with 'discount'
// field are indexed
```

4. Partial Index

```
db.orders.createIndex({
  status: 1
}, { partialFilterExpression: { status: "active" } });
// Index only active orders
```

5. TTL Index

```
db.sessions.createIndex({
  lastAccessed: 1
}, { expireAfterSeconds: 3600 });
// Documents expire 1 hour after lastAccessed
```

6. Text Index

```
db.articles.createIndex({
  title: "text", content: "text"
});
// Enables text search on title and content
```

20.13.6 Best Practices

- ✓ Index fields used in queries, sorting, and joins.
- ✓ Create compound indexes that match your query patterns.
- ✓ Avoid too many indexes.
- ✓ Monitor index usage with `db.collection.getIndexes()` and `explain()`.
- ✓ Remove unused indexes regularly.
- ✓ Use TTL indexes for automatic data cleanup.

Key Takeaway:



- * Advanced indexing helps in building fast, scalable and efficient queries.
- * Choose the right index type based on your data and query needs.
- * Always analyze performance before and after indexing.
- * Good indexing strategy = Better performance + Lower costs.

20.14 Compound Indexes

20.14.1 What are Compound Indexes?

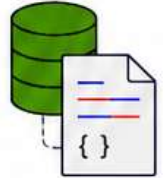
- * A compound index indexes multiple fields in a specific order.
- * It improves the performance of queries that filter, sort or group by multiple fields.
- * The order of fields in the index is very important.
- * MongoDB can use a compound index for queries that match the index prefix (from the left).
- * It reduces the need for in-memory sorting and collection scans.

How Compound Index Works?



Key Points:

- ✓ Fields are indexed in the order specified.
- ✓ MongoDB can use the index if the query filters on the leading field(s).
Example: Index on { a: 1, b: 1, c: 1 }
 - Can support queries on { a: value }
 - Can support queries on { a: value, b: value }
 - Can support queries on { a: value, b: value, c: value }
 - Not efficient for queries starting with { b: value } only
- ✓ Useful for sort() and groupBy (aggregation) on multiple fields.
- ✓ Choose the field order based on query patterns and selectivity.



20.14.2 Syntax

```
// Create a compound index
collection.createIndex({ field1: 1, field2: -1, ... })

// With options
collection.createIndex(
  { field1: 1, field2: -1 },
  { unique: true, name: "idx_field1_field2" }
)

// Drop the index
collection.dropIndex("idx_field1_field2")

// List indexes
collection.getIndexes()

// Explain query (to see index usage)
collection.find(query).explain("executionStats")
```

- 1 → Ascending order
- 1 → Descending order

20.14.3 Examples

1. Create Compound Index

```
db.orders.createIndex({
  customerId: 1,
  orderDate: -1
});
```

// Index on customerId (asc) and orderDate (desc)

2. Unique Compound Index

```
db.users.createIndex({
  email: 1,
  phone: 1
}, { unique: true });
```

// Ensures combination of email and phone is unique

3. Query using Compound Index

```
db.orders.find({
  customerId: "C101",
  orderDate: { $gte: ISODate("2024-01-01") }
}).sort({ orderDate: -1 });
```

// Uses index { customerId: 1, orderDate: -1 }
// for filtering and sorting

4. Partial Usage (Prefix)

```
db.orders.find({ customerId: "C101" });
// Even though query has only customerId,
// MongoDB can still use the index
// { customerId: 1, orderDate: -1 }
```

20.14.4 Compound Index Behavior

Index	Query Filter	Index Used?	Why?
{ a: 1, b: 1, c: 1 }	{ a: 10 }	Yes	Uses the leftmost field (a).
{ a: 1, b: 1, c: 1 }	{ a: 10, b: 20 }	Yes	Uses prefix (a, b).
{ a: 1, b: 1, c: 1 }	{ a: 10, b: 20, c: 30 }	Yes	Uses full index (a, b, c).
{ a: 1, b: 1, c: 1 }	{ b: 20 }	No	Does not start with leading field (a).
{ a: 1, b: -1 }	{ a: 10 } sort({ b: -1 })	Yes	Supports sorting on b because order is -1.
{ a: 1, b: 1 }	{ a: 10 } sort({ b: -1 })	No	Sort order on b does not match index order.

20.14.5 Real Time Use Case

E-commerce: Find recent orders of a customer.

Index:
db.orders.createIndex({ customerId: 1, orderDate: -1 })

Query:
db.orders.find({ customerId: "C101" })
.sort({ orderDate: -1 })
.limit(10);

Why it is fast?

- ✓ Filters by customerId (leading field).
- ✓ Sorts by orderDate using the index.
- ✓ Returns top 10 recent orders quickly.



20.14.6 Explain Output (Example)

```
db.orders.find({ customerId: "C101" })
.explain("executionStats")
```

Output (summary):

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "IXSCAN",
      "keyPattern": { "customerId": 1, "orderDate": -1 },
      "indexName": "customerId_1_orderDate_-1"
    }
  },
  "executionStats": {
    "totalDocsExamined": 10,
    "totalKeysExamined": 10,
    "executionTimeMillis": 2
  }
}
```



Key Takeaway:

- * Compound indexes are built on multiple fields in a specific order.
- * They speed up queries that involve multiple fields, sorting and grouping.
- * Always design index field order based on your most common query patterns.
- * Use **explain()** to verify index usage.
- * Avoid over-indexing to save storage and write performance.



20.14.7 Best Practices for Compound Indexes

- ✓ Place the most selective fields first.
- ✓ Put equality fields before range or sort fields.
- ✓ Match the query and sort order in the index.
- ✓ Avoid including too many fields in one index.
- ✓ Monitor index usage and remove unused indexes.
- ✓ Test with **explain()** and real workload.
- ✓ Use partial indexes if indexing only a subset is needed.
- ✓ Combine with projection and limit for best performance.

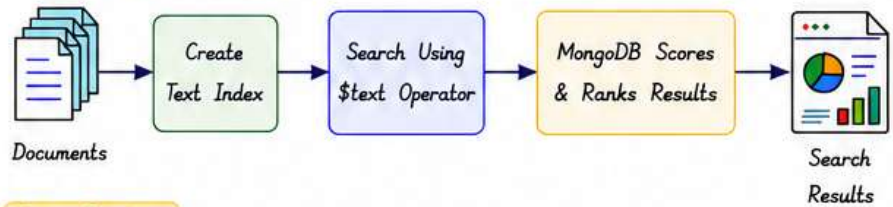


20.15 Text Search

20.15.1 What is Text Search?

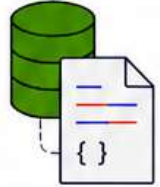
- Text search in MongoDB allows us to search for a string of words in one or more text **indexed fields**.
- It supports language stemming, stop words, case-insensitive search and relevance scoring.
- Text search is useful for searching articles, blogs, products, courses, etc.
- MongoDB text search uses a special **text index type**.

Text Search Process:



Key Points:

- ✓ A text index can be created on one or more string fields.
- ✓ Only one text index is allowed per collection.
- ✓ Text index ignores certain common words called "stop words".
- ✓ It returns matching documents along with a relevance score.
- ✓ Use **\$text** operator in the query to perform text search.
- ✓ Results are sorted by score (highest first) by default.



20.15.2 Syntax

```
// Create text index on one or more fields
db.collection.createIndex({
  field1: "text",
  field2: "text",
  ...
})

// Search using $text
db.collection.find({
  $text: { $search: "search words" }
})

// Project relevance score
db.collection.find(
  { $text: { $search: "search words" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

20.15.3 Examples

1. Create Text Index

```
db.articles.createIndex({
  title: "text",
  content: "text",
  tags: "text"
})
// Creates text index on title, content and tags
```

2. Basic Text Search

```
db.articles.find({
  $text: { $search: "mongodb indexing" }
})
// Returns documents containing
// "mongodb" AND/OR "indexing"
```

3. Get Relevance Score

```
db.articles.find(
  { $text: { $search: "machine learning" } },
  { title: 1, score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
// Returns matching documents with
// relevance score (higher score = more relevant)
```

4. Search with Additional Filter

```
db.articles.find({
  $text: { $search: "database" },
  status: "published"
})
// Text search with other query conditions
```

20.15.4 Example Collection

Sample Documents (articles)

_id	title	content	tags	status
1	Getting Started with MongoDB	MongoDB is a NoSQL database. It stores data in documents...	["mongodb", "database", "nosql"]	published
2	Advanced Indexing in MongoDB	Indexes improve query performance. Text index helps in search...	["mongodb", "indexing"]	published
3	Introduction to Machine Learning	Machine learning enables computers to learn from data...	["machine", "learning", "ai"]	published
4	Full Text Search in MongoDB	Text search allows searching words in text fields efficiently...	["mongodb", "search", "text"]	draft
5	Data Modeling Best Practices	Good schema design is important for scalable applications...	["mongodb", "schema", "design"]	published

20.15.5 Real Time Use Case

Search Blog Articles

Goal: Search blog articles using keywords.

Steps:

- ✓ Create text index on title and content.
- ✓ Search using keywords entered by user.
- ✓ Show results sorted by relevance.
- ✓ Display only published articles.



20.15.6 Output Example

Query:

```
db.articles.find(
  { $text: { $search: "mongodb search" }, status: "published" },
  { title: 1, score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

Output:

_id	title	score
4	Full Text Search in MongoDB	12.87
1	Getting Started with MongoDB	8.35
2	Advanced Indexing in MongoDB	6.10



Key Takeaway:

- Text search enables powerful keyword search on string fields.
- Create text index using field: "text".
- Use **\$text** with **\$search** to search.
- Results are ranked by relevance score.
- Only one text index per collection.
- Useful for blogs, articles, products, courses, documentation, etc.



20.15.7 Best Practices

- ✓ Create text index only on the fields that are needed.
- ✓ Keep content fields meaningful and well-structured.
- ✓ Avoid storing huge unnecessary text in documents.
- ✓ Combine text search with other filters for better results.
- ✓ Use projection to include textScore for ranking.
- ✓ Test queries and analyze performance.
- ✓ Use pagination for large result sets.
- ✓ Update index if document structure changes.

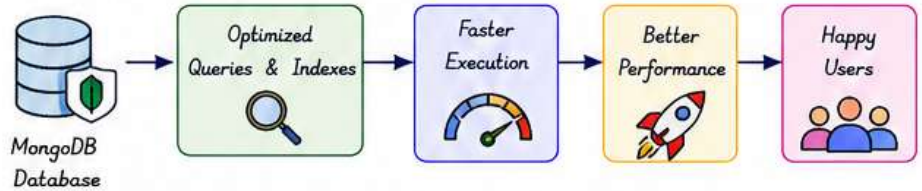


20.16 MongoDB Performance Optimization

20.16.1 What is Performance Optimization?

- * Performance optimization means making our MongoDB queries and operations faster and more efficient.
- * It helps in reducing response time, lowering server load and improving the overall application performance.
- * Good optimization leads to better scalability and user experience.

How Performance Optimization Helps?



20.16.2 Why Optimize?

Problem (Without Optimization)	Solution (Optimization)	Benefit
Slow queries (high response time)	Proper indexing, efficient queries	Faster response time
High server load	Optimize schema, use projection, limit result set	Lower server load
High memory usage	Avoid large documents, use pagination and projection	Reduced memory usage
Poor scalability	Sharding, indexing, caching	Better scalability
Poor user experience	All optimizations combined	Happy users 😊

Key Points:

- ✓ Optimize schema design.
- ✓ Use indexes wisely.
- ✓ Write efficient queries.
- ✓ Avoid large documents and unbounded arrays.
- ✓ Use projection to return only required fields.
- ✓ Monitor performance using explain().
- ✓ Use aggregation pipeline efficiently.
- ✓ Regularly monitor, analyze and tune database.

20.16.3 Syntax (explain())

```
db.collection.find({ <filter> })
    .explain("executionStats");

// Returns statistics about how the query
// was executed and helps in performance
// analysis.
```



20.16.4 Example (Performance Analysis)

1. Without Index

```
db.orders.find({ status: "shipped" })
    .explain("executionStats");
```

// Sample Output

```
"executionStats" : {
  "totalDocsExamined" : 150000,
  "executionTimeMillis" : 120,
  "nReturned" : 250
}
```

// More documents examined and
// higher execution time.

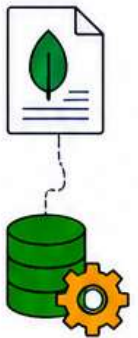
2. After Creating Index

```
db.orders.createIndex({ status: 1 });
db.orders.find({ status: "shipped" })
    .explain("executionStats");
```

// Sample Output

```
"executionStats" : {
  "totalDocsExamined" : 250,
  "executionTimeMillis" : 3,
  "nReturned" : 250
}
```

// Less documents examined and
// lower execution time.



20.16.5 Best Practices for Performance Optimization

Optimization Area	Best Practice	Description	Example
Indexing	Create indexes on commonly queried fields.	Indexes improve query performance.	<code>db.users.createIndex({ email: 1 });</code>
Query Optimization	Use specific filters and avoid unnecessary regex.	Specific queries use indexes better.	<code>db.users.find({ status: "active" })</code>
Projection	Return only required fields using projection.	Reduces data transfer and memory usage.	<code>db.users.find({}, { name: 1, email: 1, _id: 0 })</code>
Limit & Skip	Use <code>limit()</code> and <code>skip()</code> for pagination.	Avoids loading large result sets.	<code>db.products.find().skip(100).limit(20)</code>
Schema Design	Avoid large documents and unbounded arrays.	Keeps documents small and efficient.	Embed related data, avoid huge arrays.
Aggregation Pipeline	Use <code>\$match</code> early and <code>\$project</code> to reduce data.	Improves aggregation performance.	<code>[{ \$match: { status: "A" } }, { \$project: { name: 1 } }]</code>
Monitoring	Regularly monitor using <code>explain()</code> , profile and slow query logs.	Helps identify slow queries.	<code>db.setProfilingLevel(1, { slowms: 100 });</code>
Sharding	Use sharding for large datasets.	Distributes data and improves scalability.	Enable sharding on collection.
Caching	Use application level caching if applicable.	Reduces repeated database hits.	Use Redis / in-memory cache.

20.16.6 Common Mistakes to Avoid

- ✗ Not using indexes for frequently queried fields.
- ✗ Using too many indexes.
- ✗ Returning unnecessary fields.
- ✗ Fetching large result sets without pagination.
- ✗ Using regex queries without indexes.
- ✗ Ignoring `explain()` and performance metrics.
- ✗ Having huge documents and arrays.

20.16.7 Quick Checklist

- ✓ Are important fields indexed?
- ✓ Are queries using indexes (check `explain()`)?
- ✓ Are you returning only required fields?
- ✓ Are large result sets paginated?
- ✓ Are aggregation pipelines optimized?
- ✓ Are slow queries monitored?
- ✓ Is schema designed for performance?



Key Takeaway:

- * Performance optimization makes MongoDB applications fast and efficient.
- * Use indexes, efficient queries and proper schema design.
- * Analyze queries using `explain()`.
- * Always monitor and optimize continuously.
- * A well optimized database = better performance and happy users! 🚀💡