Mongo DB .2

pm_jat @ daiict

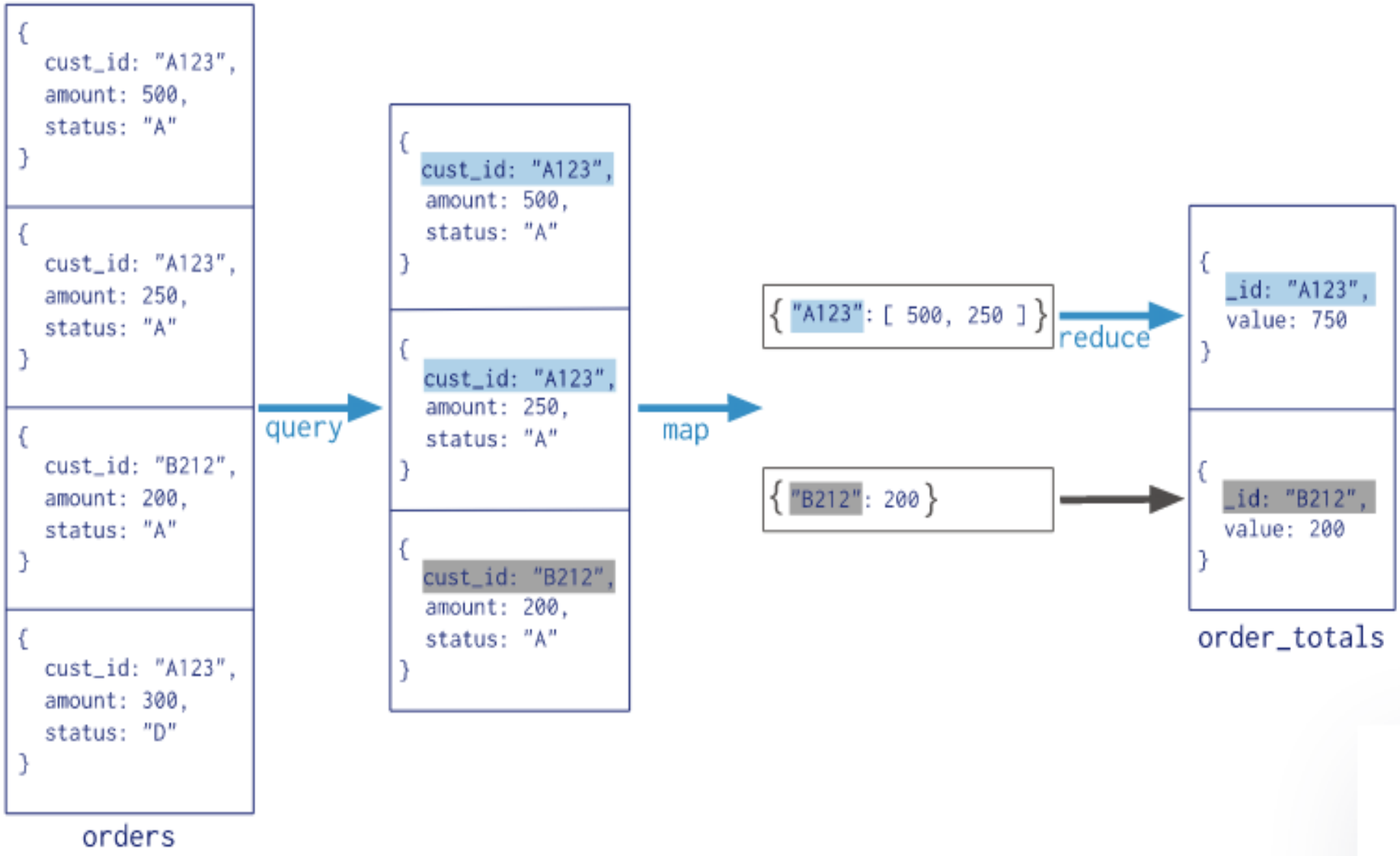
Mongo DB – Map Reduce

- MongoDB allows performing aggregations through map-reduce tasks.

```
db.orders.mapReduce(
        map      ──►  function() { emit( this.cust_id, this.amount ); },
        reduce   ──►  function(key, values) { return Array.sum( values ) },
                      {
        query    ──►    query: { status: "A" },
        output   ──►    out: "order_totals"
                      }
                   )
```

- Query: early filtering/processing
- Map and Reduce functions (in Java script)
- Out: output collection name

Mongo DB – Map Reduce



Aggregation Operations

- Aggregation Operations are performed through "**Aggregation pipeline**"
- Data aggregation framework based data processing pipeline concept.
- Multi-stage pipeline transforms the documents into aggregated results at each stage
- For example (next)

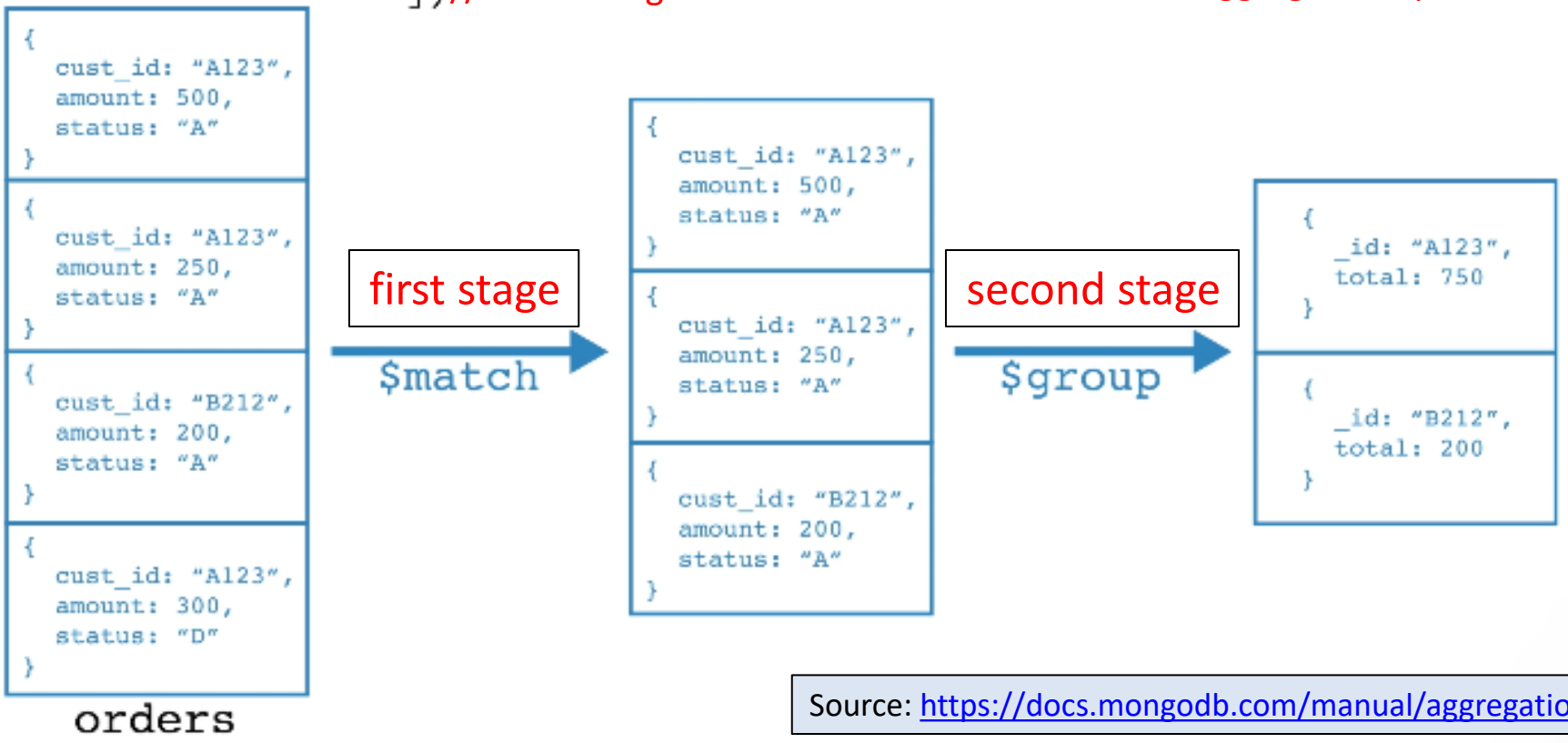
Example: Aggregation Pipeline

Collection

```
db.orders.aggregate( [
  $match stage  → { $match: { status: "A" } },     //first stage
  $group stage  → { $group: { _id: "$cust_id", total: {$sum: "$amount" } } }
                  ])//second stage
```

group by attrib

aggregation op



Source: https://docs.mongodb.com/manual/aggregation/

Aggregation – another example!

- Dataset: **ZIP Code dataset** from MongoDB docs with following fields
- _id: zip code
- city: city name, a city can have more than one zip codes
- state: two letter US state code
- pop: population
- loc: as longitude, latitude pair.

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

Source: https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Aggregate operation #1

States with Populations above 10 Million

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

```
SELECT state, SUM(pop) AS totalPop
FROM zipcodes
GROUP BY state
HAVING totalPop >= (10*1000*1000)
```

Group stage output looks like

```
{
  "_id" : "AK",
  "totalPop" : 550043
}
```

Source: https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Aggregate operation #1

Return Average City Population by State

```
db.zipcodes.aggregate( [
   { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
   { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
] )
```

// grouping attrib: state, city

First group stage output looks like

```
{
  "_id" : {
    "state" : "CO",
    "city" : "EDGEWATER"
  },
  "pop" : 13154
}
```

Second group stage output looks like

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
}
```

Source: https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Aggregate operation #3

Return Largest and Smallest Cities for each State

```
db.zipcodes.aggregate( [
    { $group:
       {
         _id: { state: "$state", city: "$city" },
         pop: { $sum: "$pop" }
       }
    },
    { $sort: { pop: 1 } }, // sort on pop in ascending order
    { $group:
       {
         _id : "$_id.state",
         biggestCity:  { $last: "$_id.city" },
         biggestPop:   { $last: "$pop" },
         smallestCity: { $first: "$_id.city" },
         smallestPop:  { $first: "$pop" }
       }
    },
```

Largest and Smallest Cities by State

- First Stage Output

```
{ $group:
   {
     _id: { state: "$state", city: "$city" },
     pop: { $sum: "$pop" }
   }
},
```

```
{
   "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
   },
   "pop" : 13154
}
```

Source: https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Largest and Smallest Cities by State

- Stage three Output (stage 2 is $sort)

```
{ $sort: { pop: 1 } },
{ $group:
   {
     _id : "$_id.state",
     biggestCity:  { $last: "$_id.city" },
     biggestPop:   { $last: "$pop" },
     smallestCity: { $first: "$_id.city" },
     smallestPop:  { $first: "$pop" }
   }
```

```
{
  "_id" : "WA",
  "biggestCity" : "SEATTLE",
  "biggestPop" : 520096,
  "smallestCity" : "BENGE",
  "smallestPop" : 2
}
```

Source: https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Aggregation Pipeline Stages

- `$count` : returns a count of the number of documents at this stage of the aggregation pipeline.
- `$group` : groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group.
- `$limit`: passes the first n documents unmodified to the pipeline where n is the specified limit.
- `$lookup`: performs a left outer join to another collection in the same database

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Aggregation Pipeline Stages

- `$match`: filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage.
- `$out`: writes the resulting documents of the aggregation pipeline to a collection.
- `$sample:` randomly selects the specified number of documents from its input.
- `$search`: performs a full-text search of the field or fields.
- `$project`: reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Aggregation Pipeline Stages

- `$sort`: Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
- `$sortByCount`: Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
- `$unionWith`: Performs a union of two collections; i.e. combines pipeline results from two collections into a single result set.

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Example: $project

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first: "aaa" },
  copies: 5
}
```

```
db.books.aggregate( [ { $project : { title : 1 , author : 1 } } ] )
```

```
{ "_id" : 1, "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
```

Source: https://docs.mongodb.com/manual/reference/operator/aggregation/project/

Aggregation Accumulators ($group)

==> Equal to “aggregation operation of RA”

- Available for use in the $group stage, accumulators are operators that maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.
- When used as accumulators in the $group stage, these operators take as input a single expression, evaluating the expression once for each input document, and maintain their stage for the group of documents that share the same group key.

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Aggregation Accumulators ($group)

- $sum: returns a sum of numerical values. Ignores non-numeric values.
- $avg : returns an average of numerical values. Ignores non-numeric values.
- $max: returns the highest expression value for each group.
- $min: Returns the lowest expression value for each group.
- $stdDevPop: returns the population standard deviation of the input values.
- $stdDevSamp: returns the sample standard deviation of the input values.

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Aggregation Accumulators ($group)

- $first: returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
- $last: returns a value from the last document for each group.
- $push: returns an array of expression values for each group.
- $accumulator: returns the result of a user-defined accumulator function.

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Accumulators (in $project, and other stages)

- [<==> Aggregation without group by]
- $avg.
- $max
- $min
- $stdDevPop
- $stdDevSamp
- $sum

Source: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Single Purpose Aggregation Operations

- MongoDB provides special (single purpose) aggregation operations:

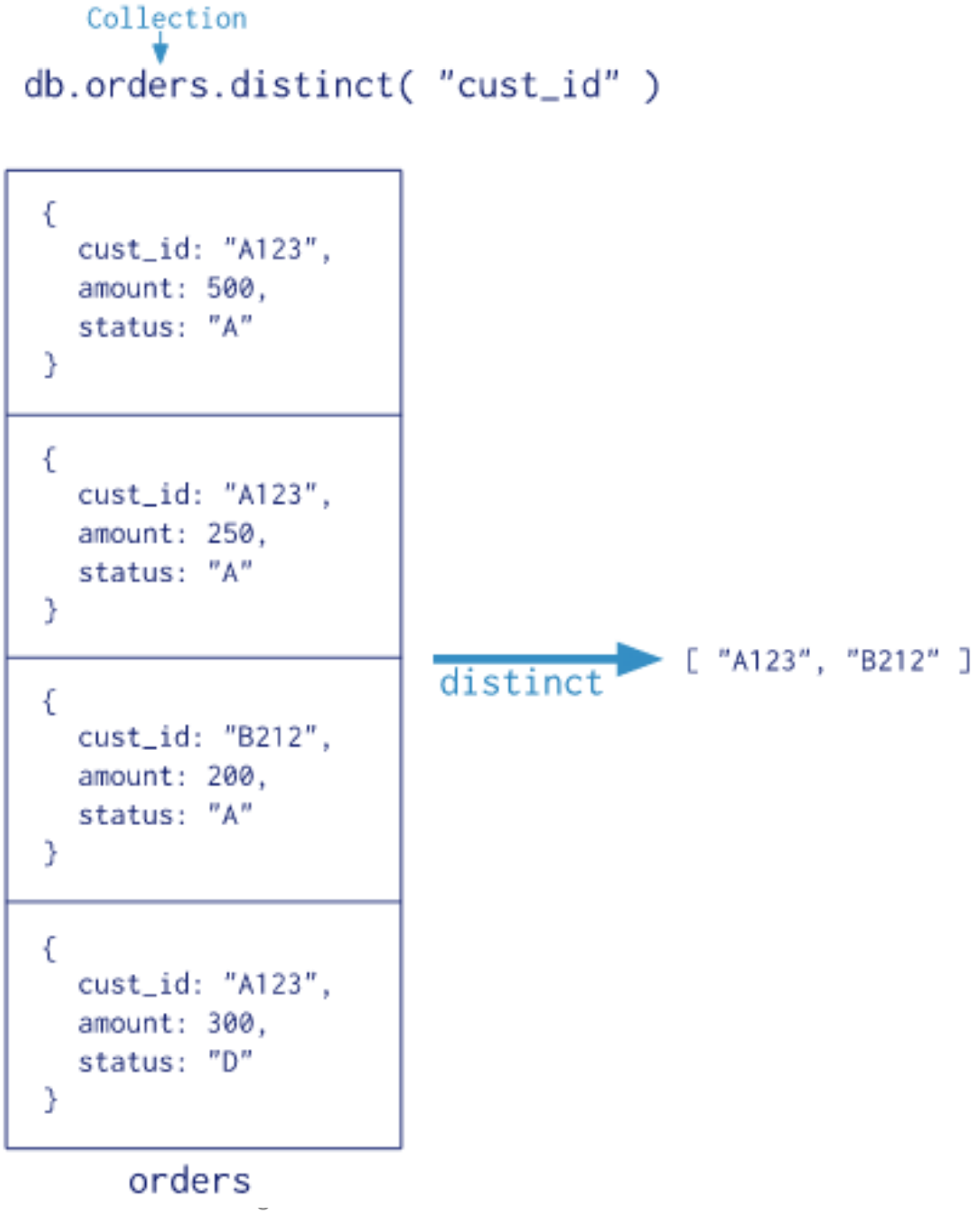
  db.collection.estimatedDocumentCount(),
  does not actually count but queries metadata

  db.collection.count(), and

  db.collection.distinct()

- The basic motivation of making them separate method is to make them more efficient in terms of query execution.

Example: distinct



LEFT JOIN operation: example

- Consider following collections “orders” and “inventory”. Orders documents have references to inventory collection!

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

```
db.inventory.insert([
    { "_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 }
```

LEFT JOIN : example

- $lookup is used in aggregation pipeline for joining. It is LEFT JOIN
- Aggregate applied on “orders”
- Performs lookup in “inventory”
- Result: inventory is embedded inside orders.
- Have a look at Result Snapshot on next slide.

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

LEFT JOIN : example

- Result: joined document from right collection are embedded in left relation (new collection is produced)

```
    "_id" : 1,
    "item" : "almonds",
    "price" : 12,
    "quantity" : 2,
    "inventory_docs" : [
        { "_id" : 1, "sku" : "almonds", "description" : "product 1", "instock" : 120 }
    ]
}
{
    "_id" : 2,
    "item" : "pecans",
    "price" : 20,
    "quantity" : 1,
    "inventory_docs" : [
        { "_id" : 4, "sku" : "pecans", "description" : "product 4", "instock" : 70 }
    ]
```

Aggregation Pipelining

- The “aggregation pipeline” can operate on a sharded collection.
- The aggregation pipeline can use indexes to improve its performance during some of its stages.
- In addition, the aggregation pipeline has an internal “optimization phase”.

Indexes in Mongo DB

- Advantages: "**Searching becomes very efficient**"

Indexes and Query execution

- Hope you know indexes are used in Relational Databases?
- How indexes will affect execution of following query?
  1. SELECT * FROM PRODUCTS WHERE CAT=5
  2. SELECT * FROM PRODUCTS WHERE CAT=5 and PRICE >=2000 and PRICE < 5000
     - Composite Index (CAT, PRICE)
  3. SELECT * FROM CUSTOMERS WHERE STATE="ABCD"
  4. SELECT * FROM CUSTOMERS WHERE ZIP="122345"
- Availability of indexes on attributes in predicates makes query execution efficient?

Indexes and Query execution

- For instance: SELECT * FROM PRODUCTS WHERE CAT=5
- For getting PRODUCTS of given CAT, wee have two options
  - Sequentially scan all products, and select the one that have CAT=5. BUT this is brute force and not efficient.
  - Other option is, TABLE PRODUCTS have index on CAT. Using Index we can determine address of PRODUCTS for given CAT. This is efficient in terms of execution of query.
- Interestingly DBMS uses indexes transparently, and we do not have to tell if index is to be used.
- In case of second query here, indexes can be composite on CAT and PRICE, or can be individually. Composite would be better?

Indexes and Query execution

- Suppose we have query “SELECT * FROM PRODUCTS WHERE SELLER = 123 AND CAT=3”
- Which index will be most efficient:
  - Composite (SELLER, CAT)
  - Composite (CAT, SELLER)
  - Individual Indexes on SELLER and CAT

Indexes and Query execution

- SELECT * FROM PRODUCTS WHERE PRICE >=2000 and PRICE < 5000
- Which index will be most efficient:
  - Composite (PRICE, CAT): YES
  - Composite (CAT, PRICE): ORDERED by CAT, PRICE : NO

Good and Downside of Indexes

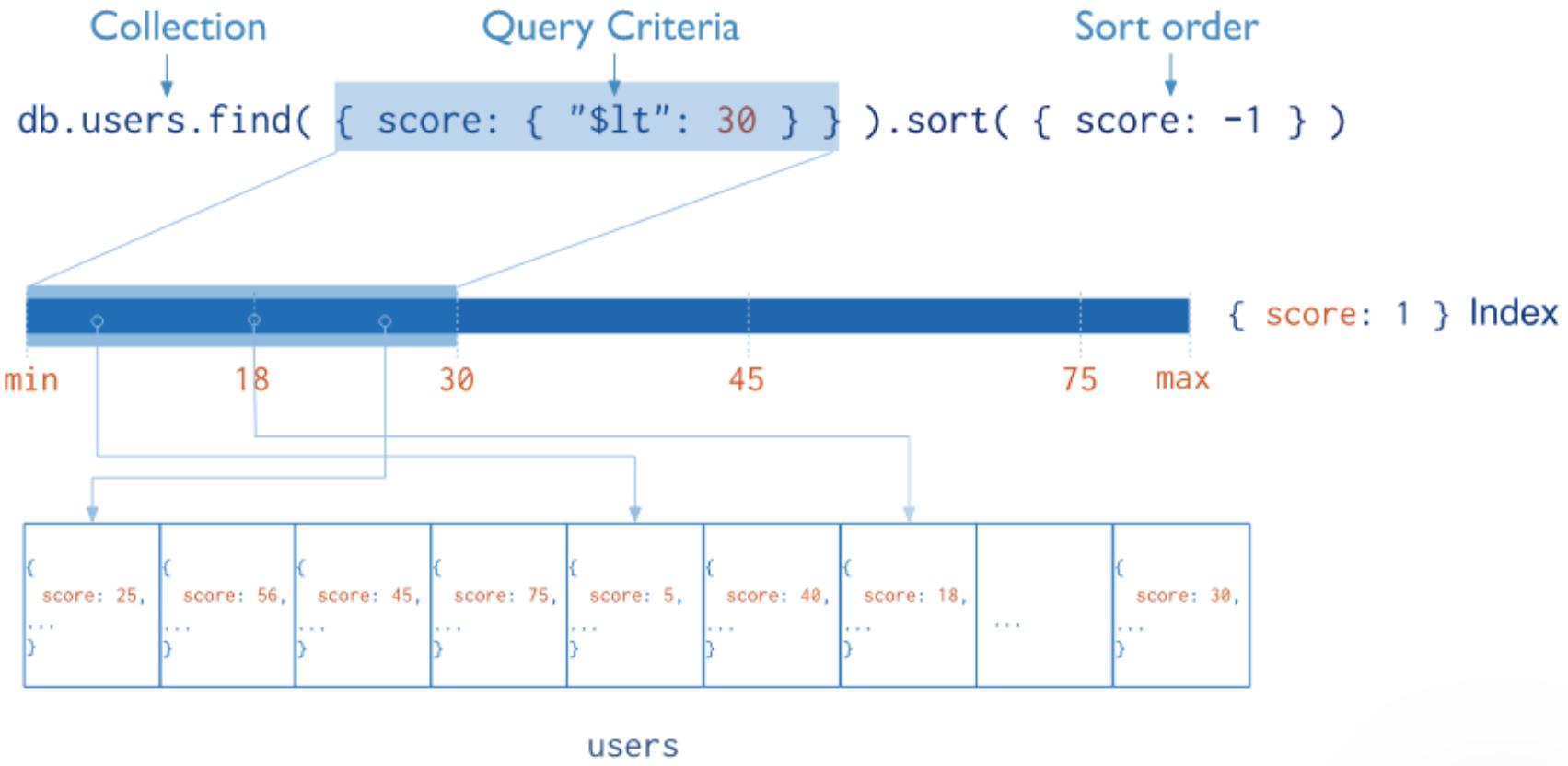
- Good:
  - Helps in improving searches
- Downside:
  - Since DBMS require updating indexes also when data items are getting added and updated.
  - This affects the performance of update operations and hence update operations become slower.
- Therefore, we can not have index on every field and we can not afford having no index at all?

Mongo DB Indexes

- Mongo provides very comprehensive support for Indexes
- Mongo DB indexes are B+ tree based
- Mongo DB automatically makes use of indexes wherever indexes are available!
- Most content put here is from: https://docs.mongodb.com/manual/indexes/
- There is also a very naïve videos' available on MongoDB University: https://university.mongodb.com/mercury/M201/2021_March_16/overview

Mongo Indexes

- How availability of index on score (ascending) on users will be used in answering following query?



Source: https://docs.mongodb.com/manual/indexes/

Default _id Index

- MongoDB creates a unique index on the _id field during the creation of a collection.
- The _id index prevents clients from inserting two documents with the same value for the _id field.
- You cannot drop this index on the _id field.

Source: https://docs.mongodb.com/manual/indexes/

Index Properties

Unique Indexes

- A unique index causes MongoDB to reject duplicate values for the indexed field.

Partial Indexes

- Partial indexes only index the documents in a collection that meet a specified filter expression.
- By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.
- Partial indexes offer a superset of the functionality of sparse indexes and should be preferred over sparse indexes.

Source: https://docs.mongodb.com/manual/indexes/

Index Properties

Sparse Indexes

- The sparse property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that do not have the indexed field.
- You can combine the sparse index option with the unique index option to prevent inserting documents that have duplicate values for the indexed field(s) and skip indexing documents that lack the indexed field(s).

TTL Indexes

- TTL (Time To Live) Indexes lives only for specified time.

Source: https://docs.mongodb.com/manual/indexes/

Creating Index

- Method creatIndex
- Parameters: key and other index specifications, and options

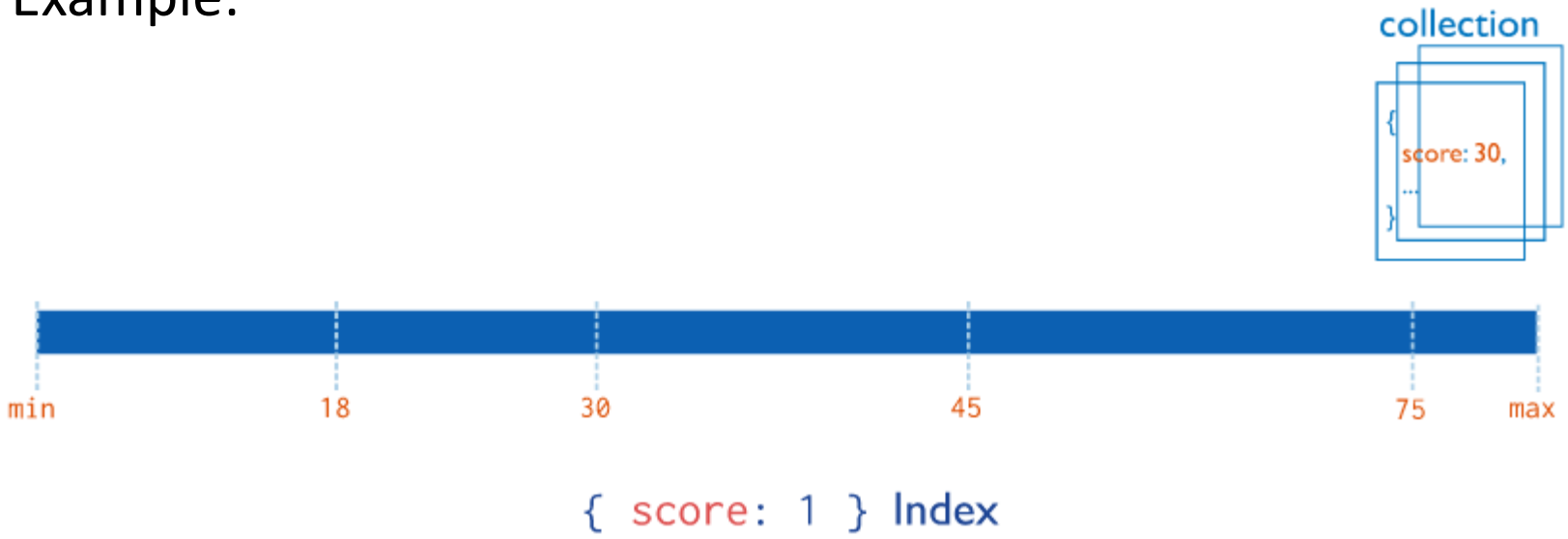
```
db.collection.createIndex( <key and index type specification>, <options> )
```

- MongoDB indexes use a B-tree data structure.
- Here is an example: Index for collection products.
  Search Keys: item: ascending, quantity: descending

```
db.products.createIndex(
   { item: 1, quantity: -1 } ,
   { name: "query for inventory" }
)
```

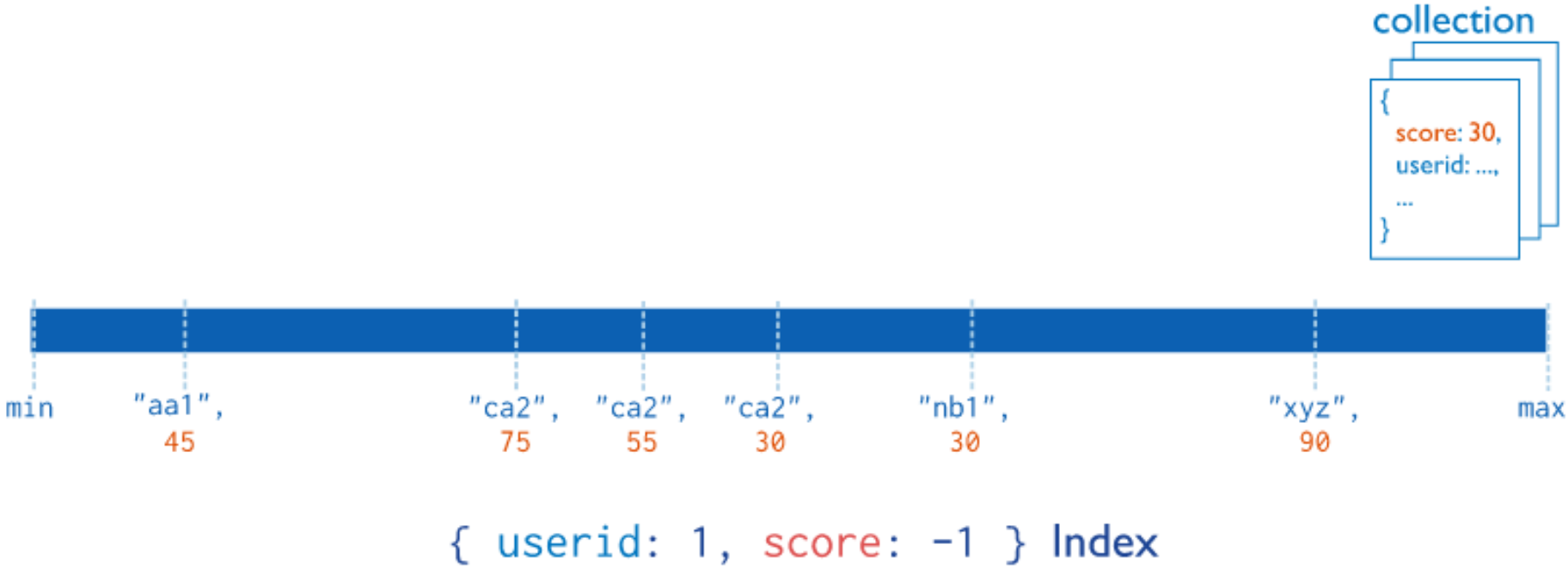
Index Types

- MongoDB provides a number of index types to support specific types of data and queries.
- Single Field: On a single field: ascending/descending
- Example:



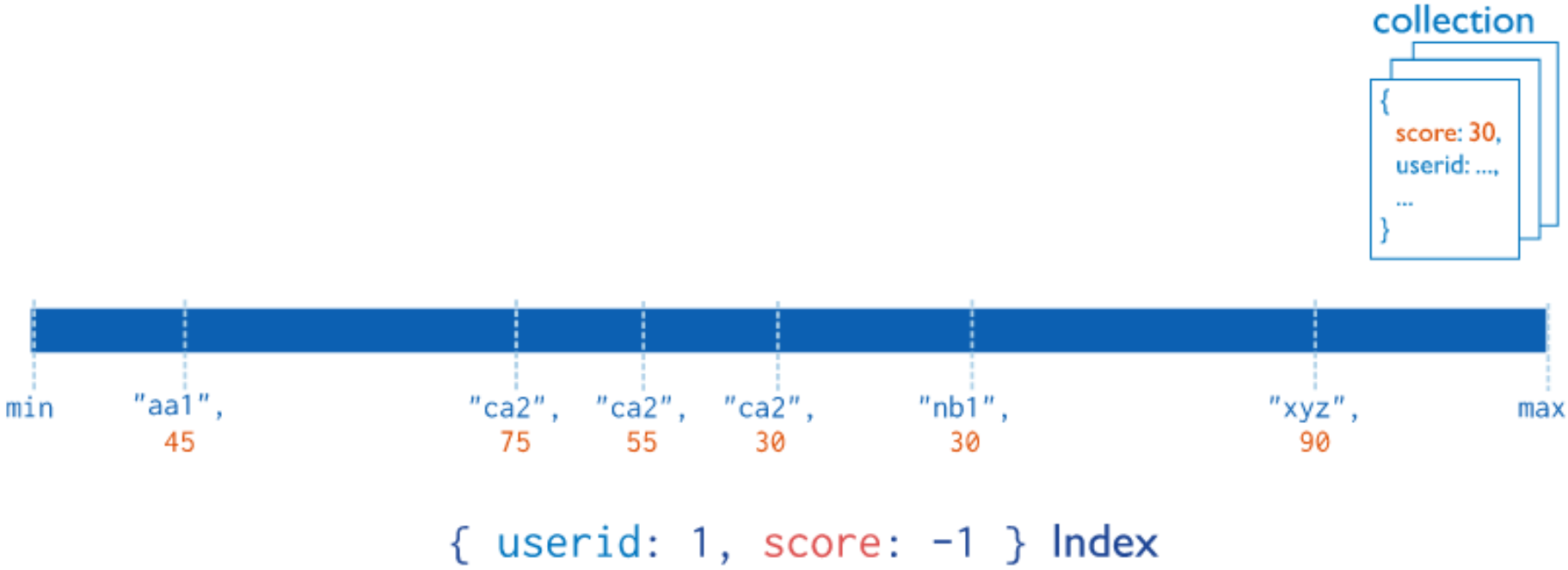
Index Types: Compound Index

- Indexes on multiple fields.



Index Types: Compound Index

- Which query will take advantage of the index?
  - Query: find * where userid=101 and score > 50
  - Query: find users with score > 50



Index types: Multikey Index

- MongoDB uses multikey indexes to index the content stored in arrays.
- Searching based on array element becomes efficient!

Other index types

- **Geospatial Index**: To support efficient queries of geospatial coordinate data
- **Text Indexes**: to support full text search. Index/Searching is done based root words, etc.
- **Hashed Indexes**: To support hash based sharding, MongoDB provides a hashed index type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries.

Full Text Search | Text Search

- Full Text Searching (or just text search) provides the capability to search in natural-language documents
- The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.
- Notions of query and similarity are very flexible and depend on the specific application.
- The simplest search considers query as a set of words and similarity as the frequency of query words in the document (popular metric is cosine similarity based on Term Frequency and TFIDF)

Full Text Search in Mongo DB

- MongoDB allows integrating powerful full text search engine like Lucene with it.
- “MongoDB Atlas”, the cloud based server provides full text search through Lucene. May look at

Getting started with MongoDB Atlas Full-Text Search
https://www.mongodb.com/blog/post/getting-started-with-mongodb-atlas-fulltext-search

Query Planner

- For a query, the MongoDB query optimizer chooses and caches the most efficient query plan given the available indexes.
- The evaluation of the most efficient query plan is based on the number of “work units” (works) performed by the query execution plan when the query planner evaluates candidate plans.
- The associated plan cache entry is used for subsequent queries with the same “query shape”.
- To view query plan for a query, we can use `db.collection.explain()` or the `cursor.explain()`. Here we see what plan is winner and its details.

db.collection.explain()

- Allows you finding:
  - Is your query using the index you expect?
  - Is your query using an index to provide a sort?
  - Is your query using an index to provide the projection?
  - How selective is your index?
  - Which part of your plan is the most expensive?

References

[1] Chapter 9, No-SQL Distilled

[2] https://docs.mongodb.com/manual/