INDUSTRIAL UNIVERSITY OF HOCHIMINH CITY

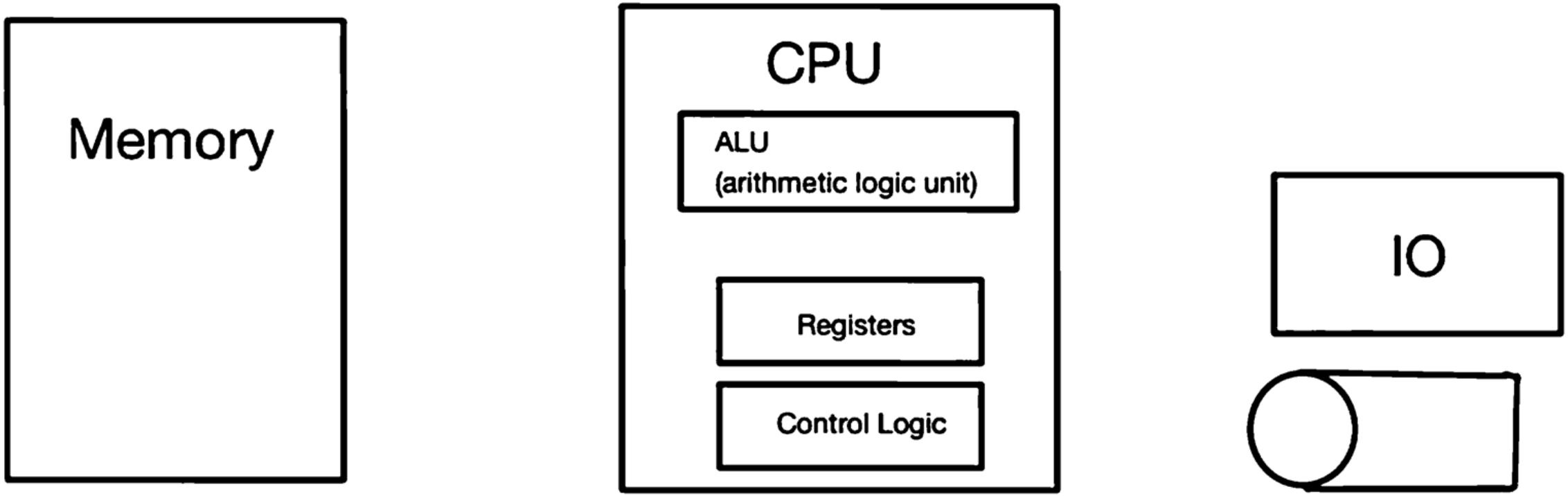
mongoDB

MONGODB PERFORMANCE

Lecturer: Trần Thế Trung

1. Hardware
2. Performance on Clusters
3. Indexes
4. CRUD Optimization

1. Hardware


Memory
CPU
ALU
(arithmetic logic unit)
Registers
Control Logic
IO

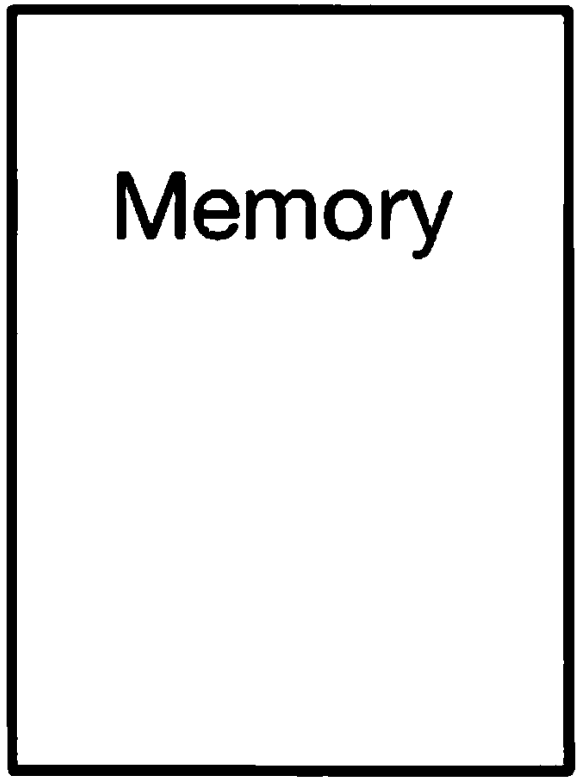
1. Hardware

In MongoDB deployments, as in many other modern databases, memory is a quintessential resource. Over the past few years, the availability of RAM and the fall in its production costs contributed to the development of database architectures.

The fact that RAM or memory is 25 times faster than common SSDs also makes this transition of disk-oriented into RAM-oriented a nice, strong appealing factor for databases to be designed around usage of memory.

As a result of this, MongoDB has storage engines that are either very dependent on RAM, or even completely in memory execution modes for its data management operations.

1. Hardware



- Aggregation
- Index Traversing
- Write Operations
- Query Engine
- Connections

1. Hardware

A significant number of operations rely heavily in RAM.

- Like the aggregation pipeline operations, the index traversing.
- Writes are first performed in RAM allocated pages.
- The query engine requires RAM to retrieve the quarter results.
- And finally, connections are handling memory.

Roughly, one megabyte per established connection. And therefore they require memory space. It is safe to say that the more RAM you have available, the more performance your department of MongoDB will tend to be.

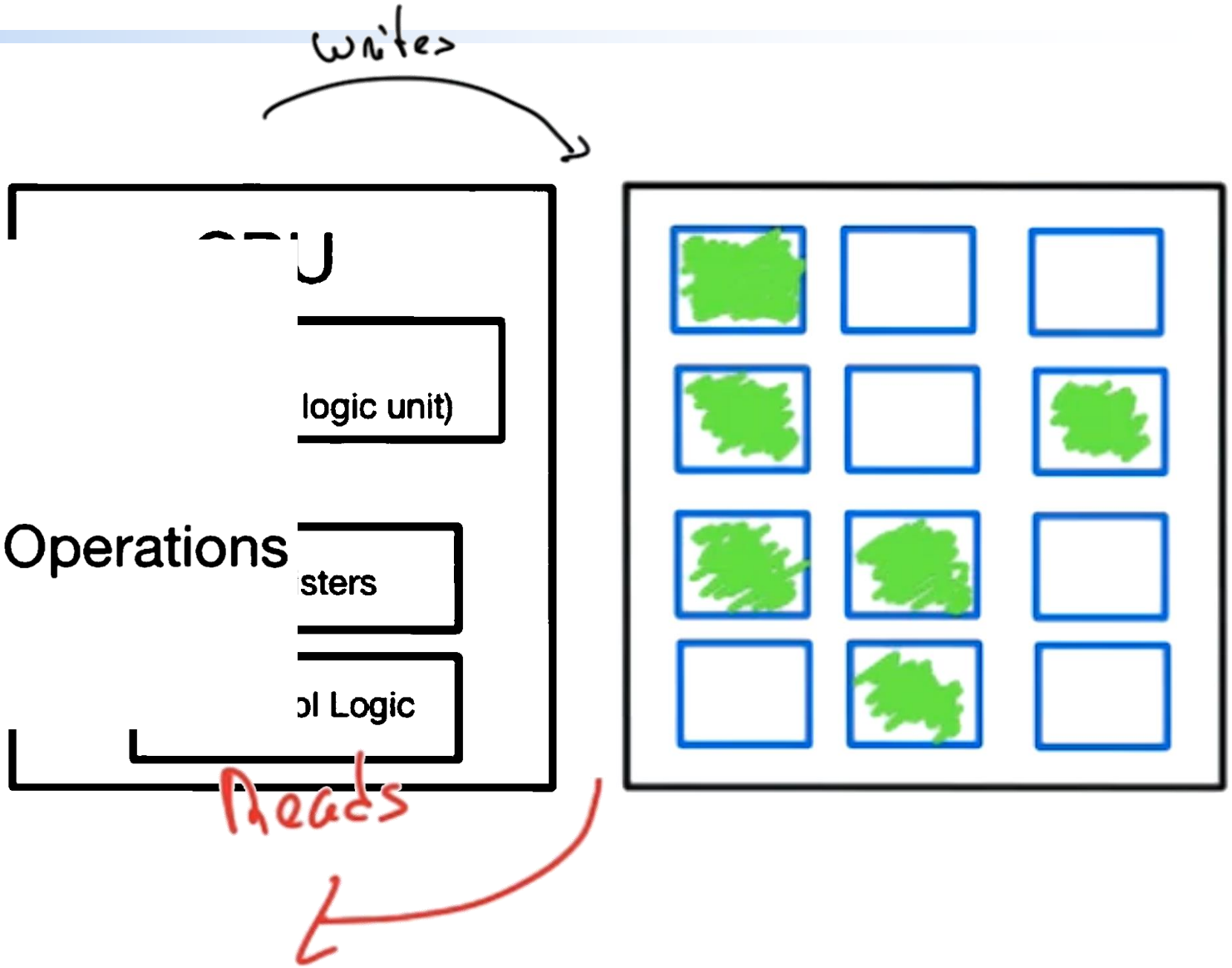
1. Hardware

Page Compression

Data Calculation

Aggregation Framework Operations

Map Reduce



1. Hardware

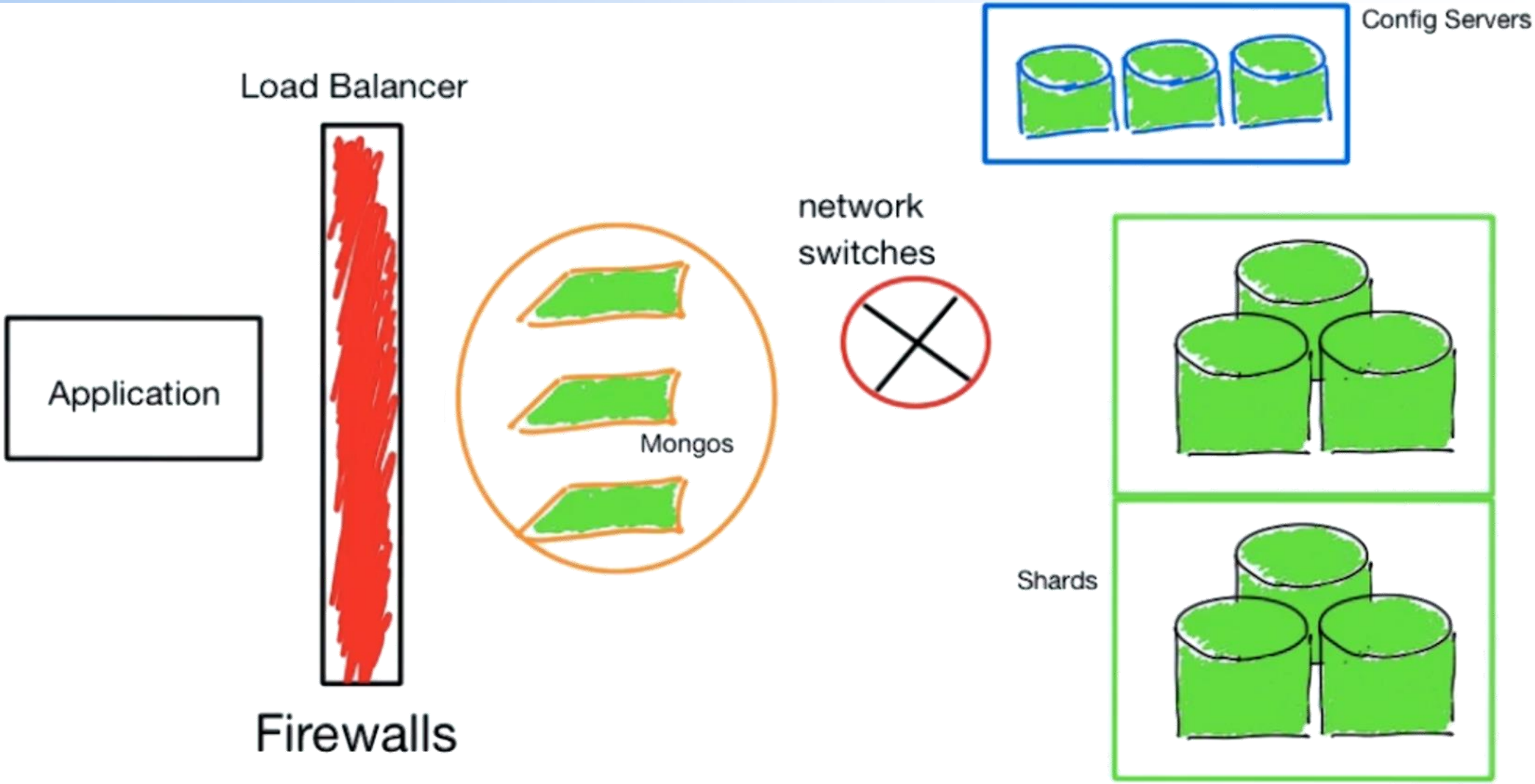
CPU is used by all applications for computational processing. Databases are just another category of applications. MongoDB is no different. But the utilization of this resource is generally more attached with two main factors. Storage engines that we are using, and the concurrency level that your MongoDB instance will be subjected to.

By default, MongoDB will try to use all available CPU cores to respond to incoming requests. Our non-locking concurrency control mechanism, using wired tag or storage engine, rely heavily in the CPU to process these requests.

This means that if we have non-blocking operations, like writing different documents concurrently or responding to an incoming query requests like Reads, MongoDB will perform better the more CPU resources we have available.

Also, there are certain operations, like page compression, data calculation operations, aggregation framework operations, and map reduce, amongst others that will require the availability of CPU cycles.

1. Hardware


Config Servers
Load Balancer
network
switches
Application
Mongos
Shards
Firewalls

1. Hardware

MongoDB deployments also rely on network hardware. Applications will reach the database by establishing connections to the hosts, where MongoDB instance is running.

The faster and the larger the bandwidth is for your network, the better performance you will experience. But this is not the end of the story regarding network utilization with MongoDB.

MongoDB is a distributed database for high availability. But also for horizontal scaling, where the shard and cluster in all its different components, allows you to get a horizontal distribution of your data.

The way that your different hosts that hold the different nodes of your cluster are connected, can affect the overall performance of your system.

1. Hardware

Also, the types of network switches, load balancers, firewalls, and how far apart the cluster nodes are either by being distributed across different data centers or regions.

The type of connections between data centers, especially latency we haven't cracked going faster than the speed of light yet will play a great deal in the performance experienced by your application.

This aligned with the write concern, read concern, and read preference that your application can set while emitting commands or doing requests to the server, needs to be taken into consideration when analyzing the performance of your application.

02. Performance on Clusters

Considerations in Distributed

What is a Distributed System in MongoDB?

- Replica Cluster
- Shard Cluster

02. Performance on Clusters

Considerations in Distributed

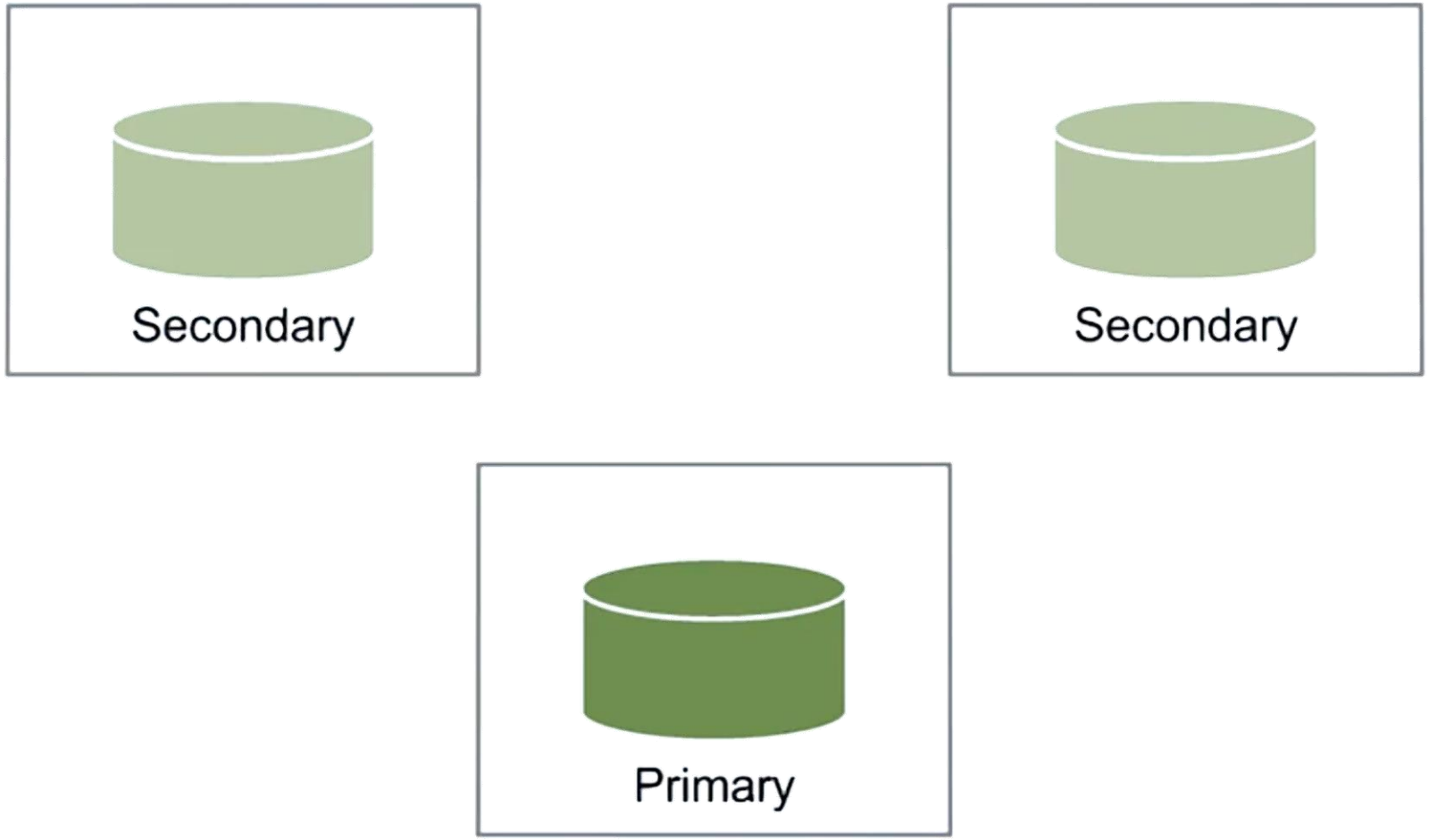
Working with Distributed Systems

- Consider latency;
- Data is spread across different nodes;
- Read implications;
- Write implications.

02. Performance on Clusters

Considerations in Distributed

Replica Set



02. Performance on Clusters

Considerations in Distributed

Now, regarding your replica sets, let me reinforce a message around this. Please do use them in production environments. High availability is key to guarantee that your service is not affected by any potential, and probable, system failure. Having a replica set in place is super, super important.

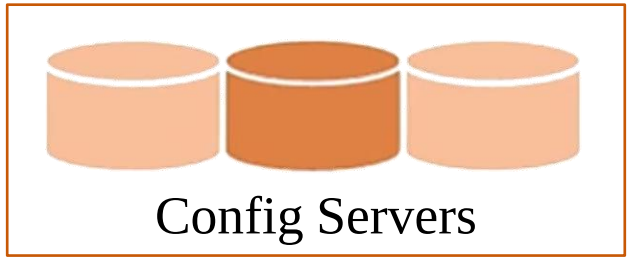
Apart from the main purpose of providing high availability, in case of failure of a node, we will still have availability of a service provided by the remaining nodes, but replica sets can also provide a few other functions, like offloading ventral consistency data to secondaries, privileging your primary for operational workload, or having specific workload with target indexes configuration on secondary nodes.

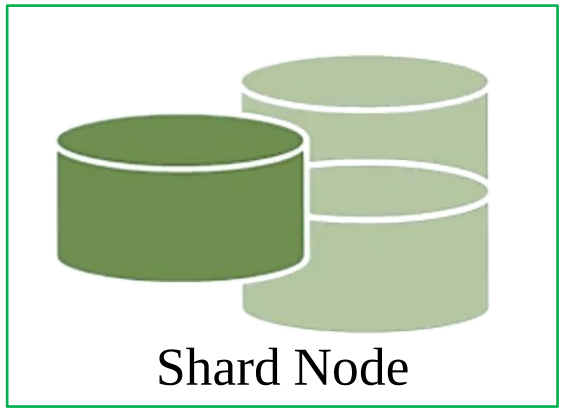
The other side of distributed systems in MongoDB, with a purpose of horizontal scalability, is our Shard Cluster.

02. Performance on Clusters

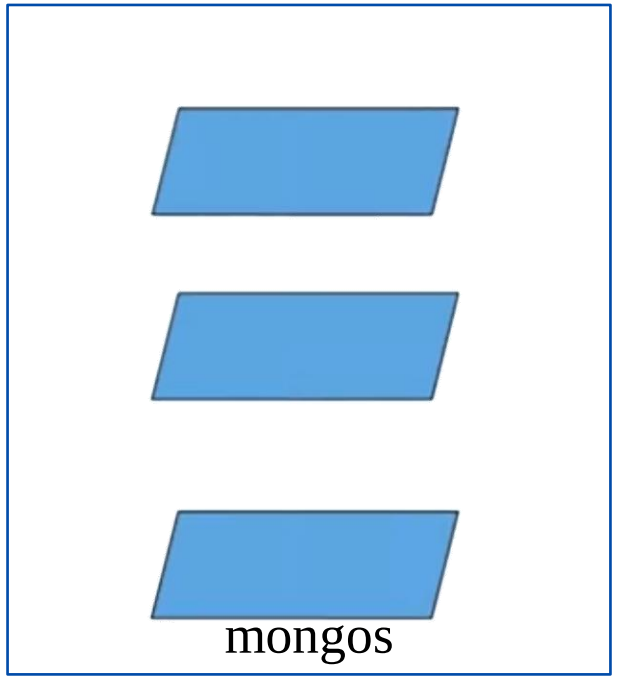
Considerations in Distributed

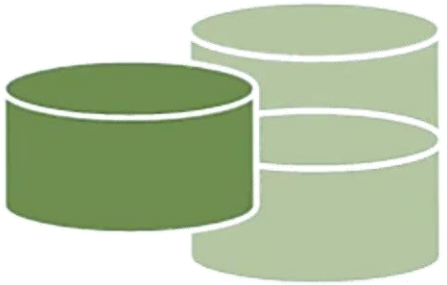
Sharding Cluster





Client Application





02. Performance on Clusters

Considerations in Distributed

In our Shard Cluster, we will have our Mongos, responsible for routing our client application requests to designated nodes.

We're going to have config servers. These nodes are responsible for holding the mapping of our Shard Cluster, where data sits at each point in time, but also the general configuration of our Shard Cluster in its own.

And finally, we have our Shard Nodes. Shard Nodes are responsible for holding the application data. Databases, collections, indexes these will reside in these members, and it will be here that all major workload will be performed.

Shard Nodes are in themselves replica sets.

02. Performance on Clusters

Considerations in Distributed

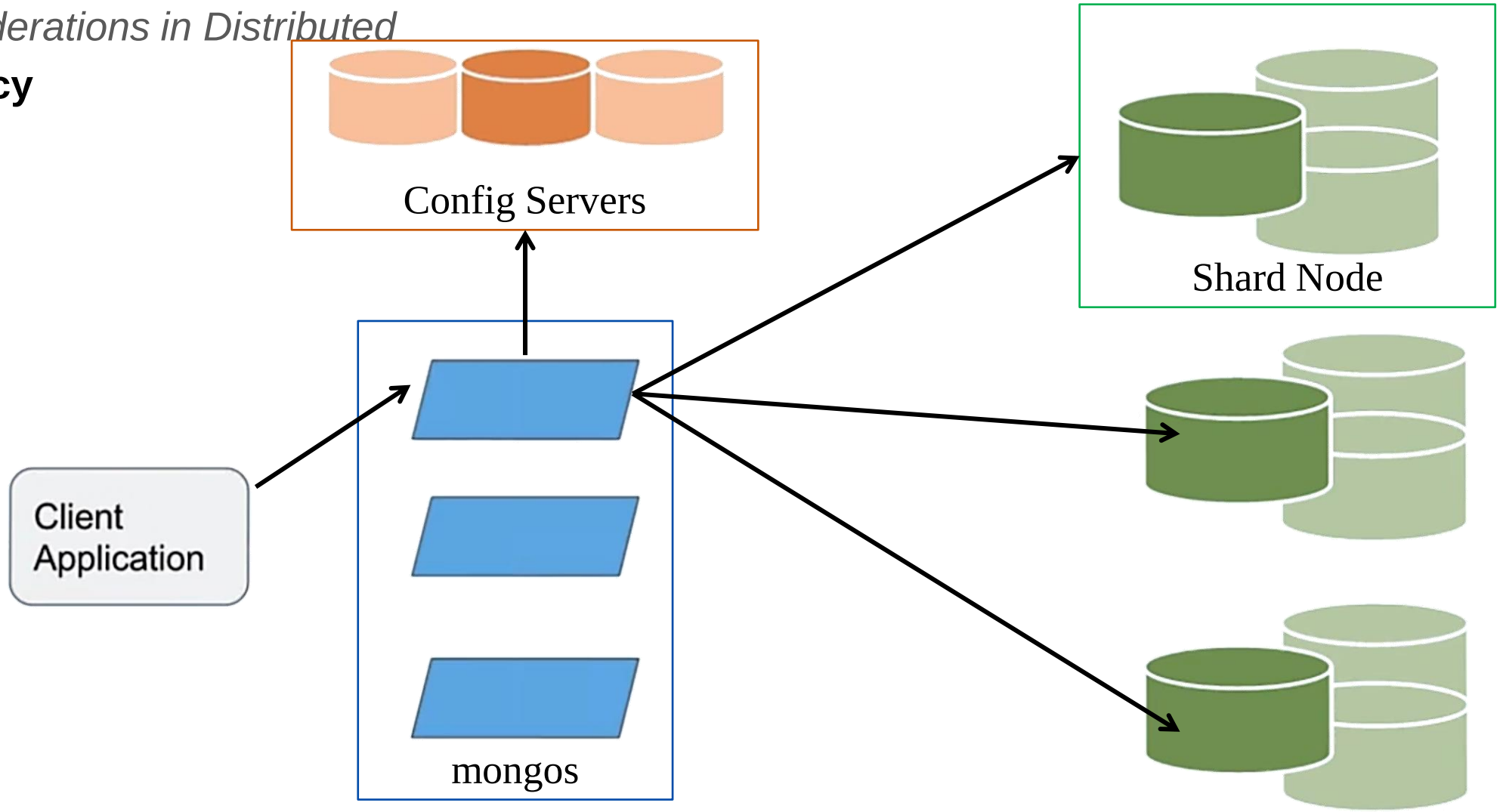
Before Sharding

- Sharding is an horizontal scaling solution;
- Have we reached the limits of our vertical scaling?
- You need to understand how your data grows and your data is accessed;
- Sharding works by defining key based ranges – our shardkey;
- It’s important to get a good shard key

02. Performance on Clusters

Considerations in Distributed

Latency



02. Performance on Clusters

Considerations in Distributed

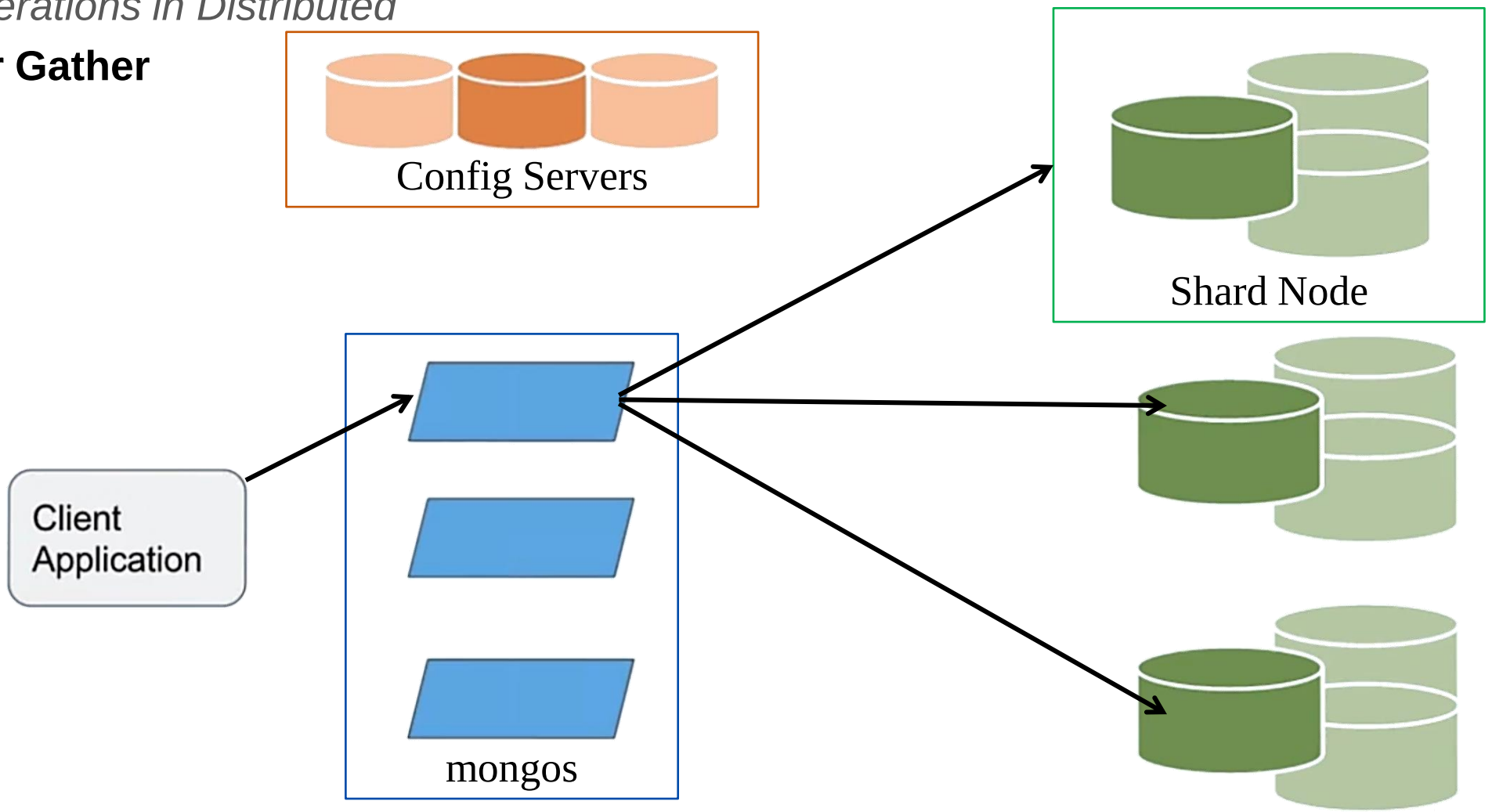
- **Two types of reads:**
  - Scatter Gather
  - Routed Queries

- **When**
  - If we are not using the shard key, we will be performing scattered gathered queries
  - If we are using the shard key, we will be performing scattered Routed Queries

02. Performance on Clusters

Considerations in Distributed

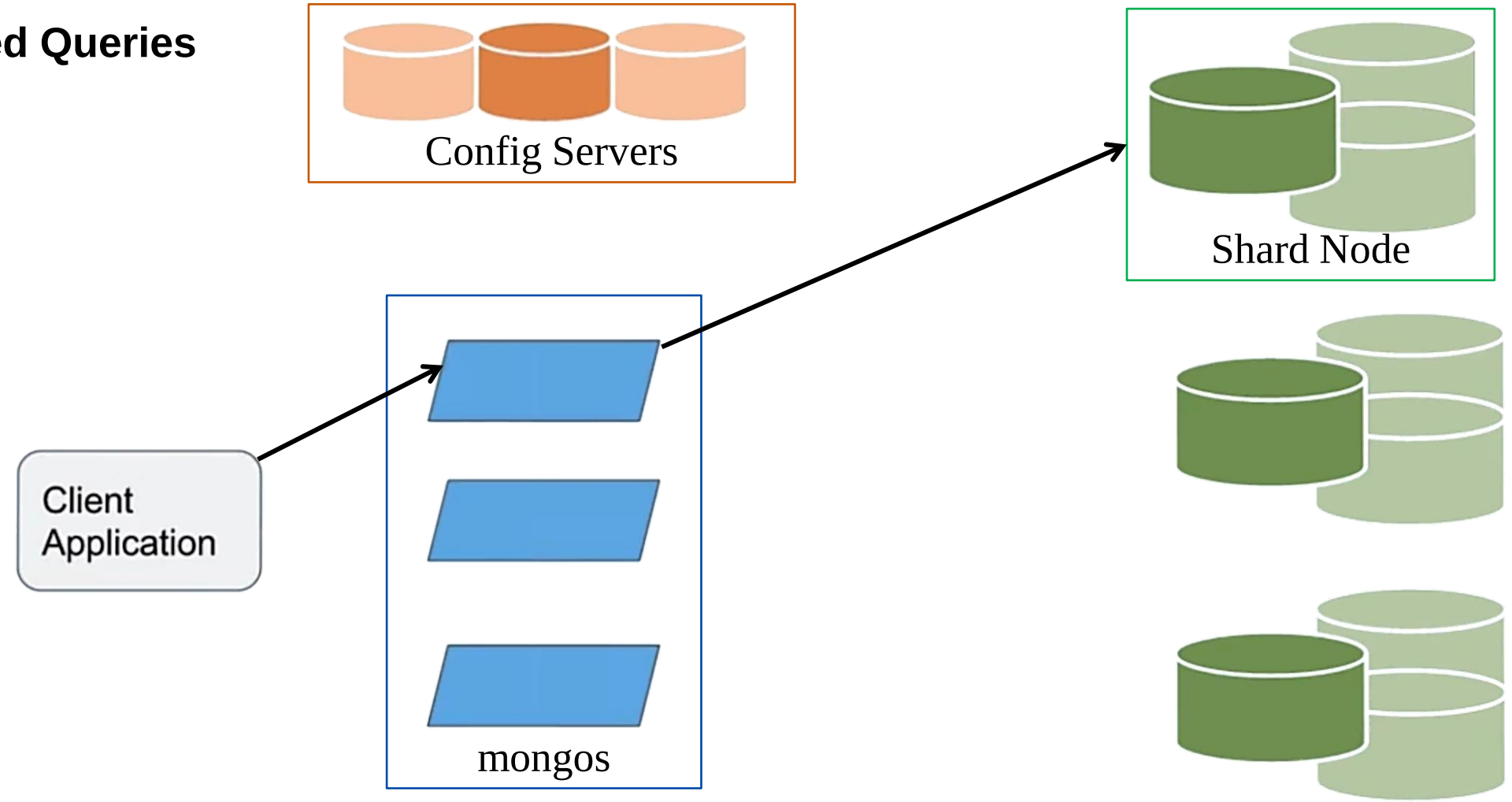
Scatter Gather



02. Performance on Clusters

Considerations in Distributed

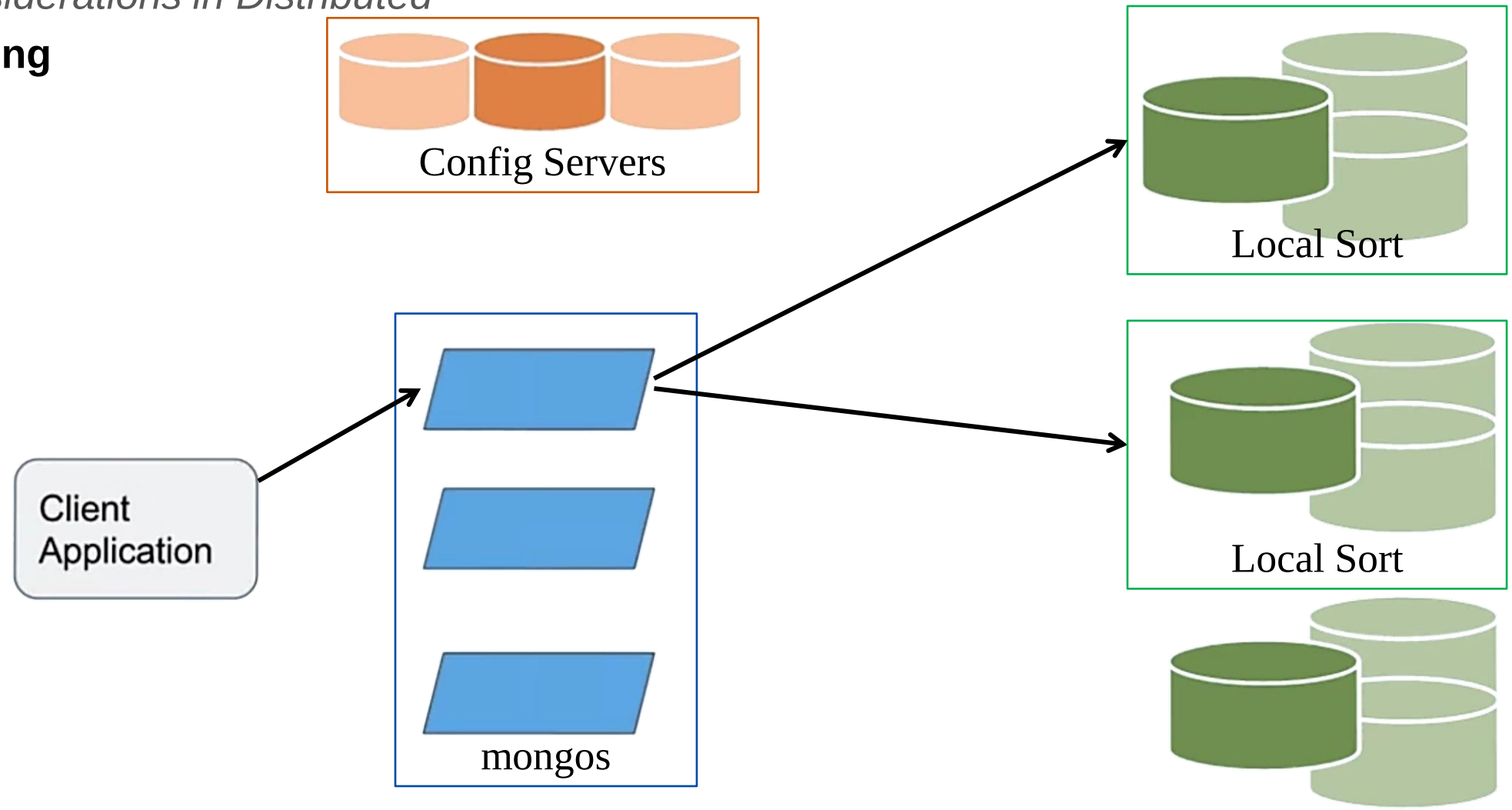
Routed Queries



02. Performance on Clusters

Considerations in Distributed

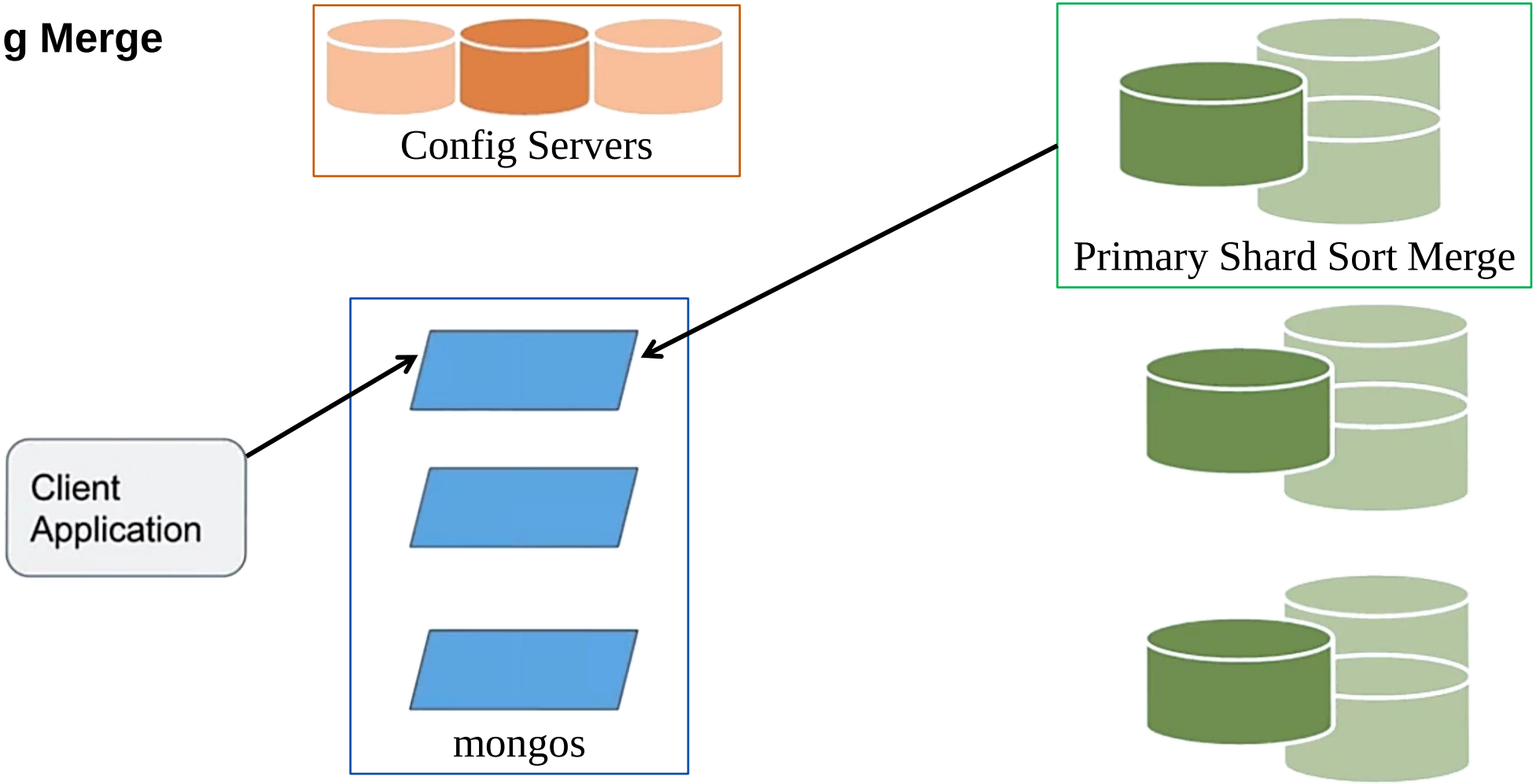
Sorting



02. Performance on Clusters

Considerations in Distributed

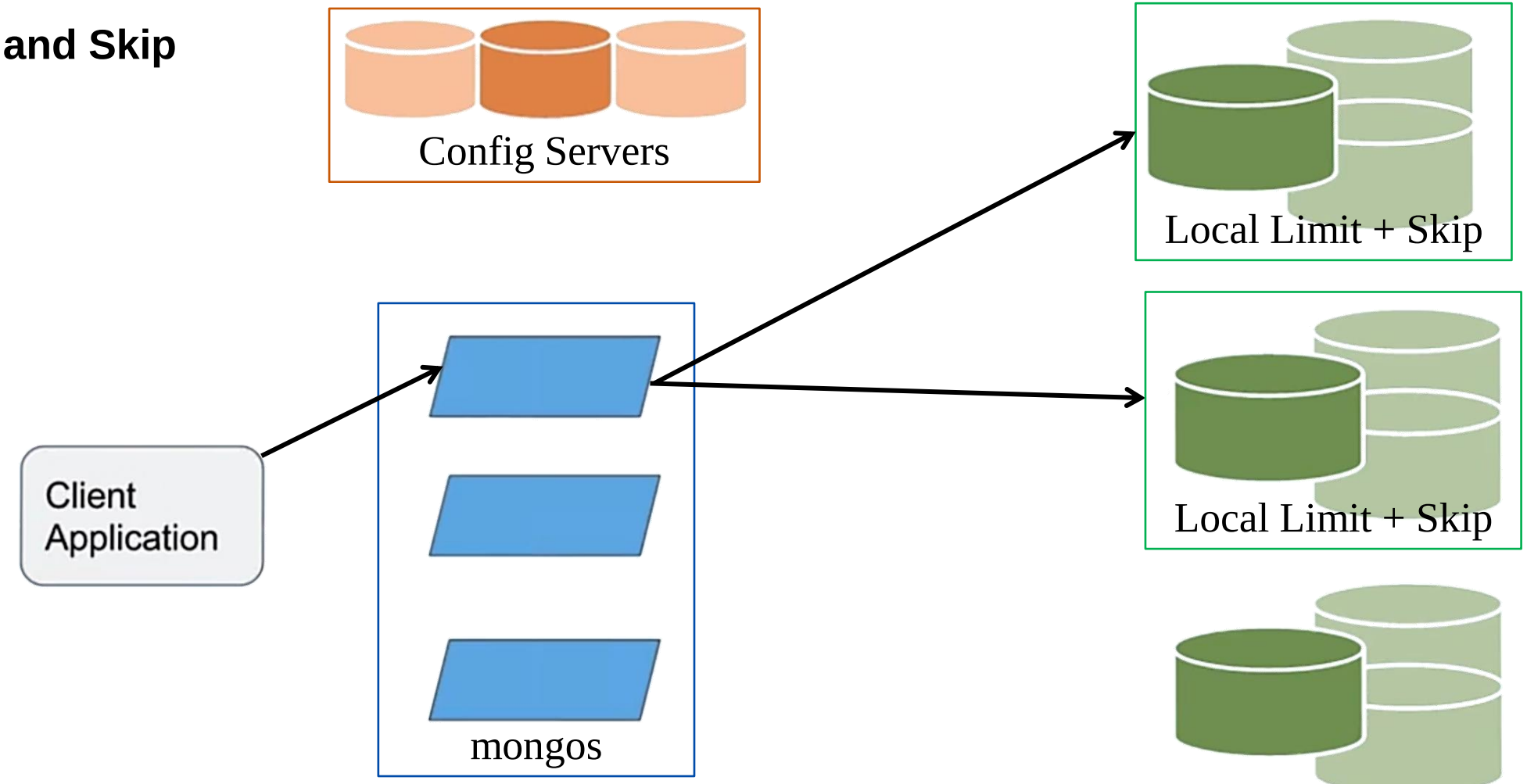
Sorting Merge



02. Performance on Clusters

Considerations in Distributed

Limit and Skip

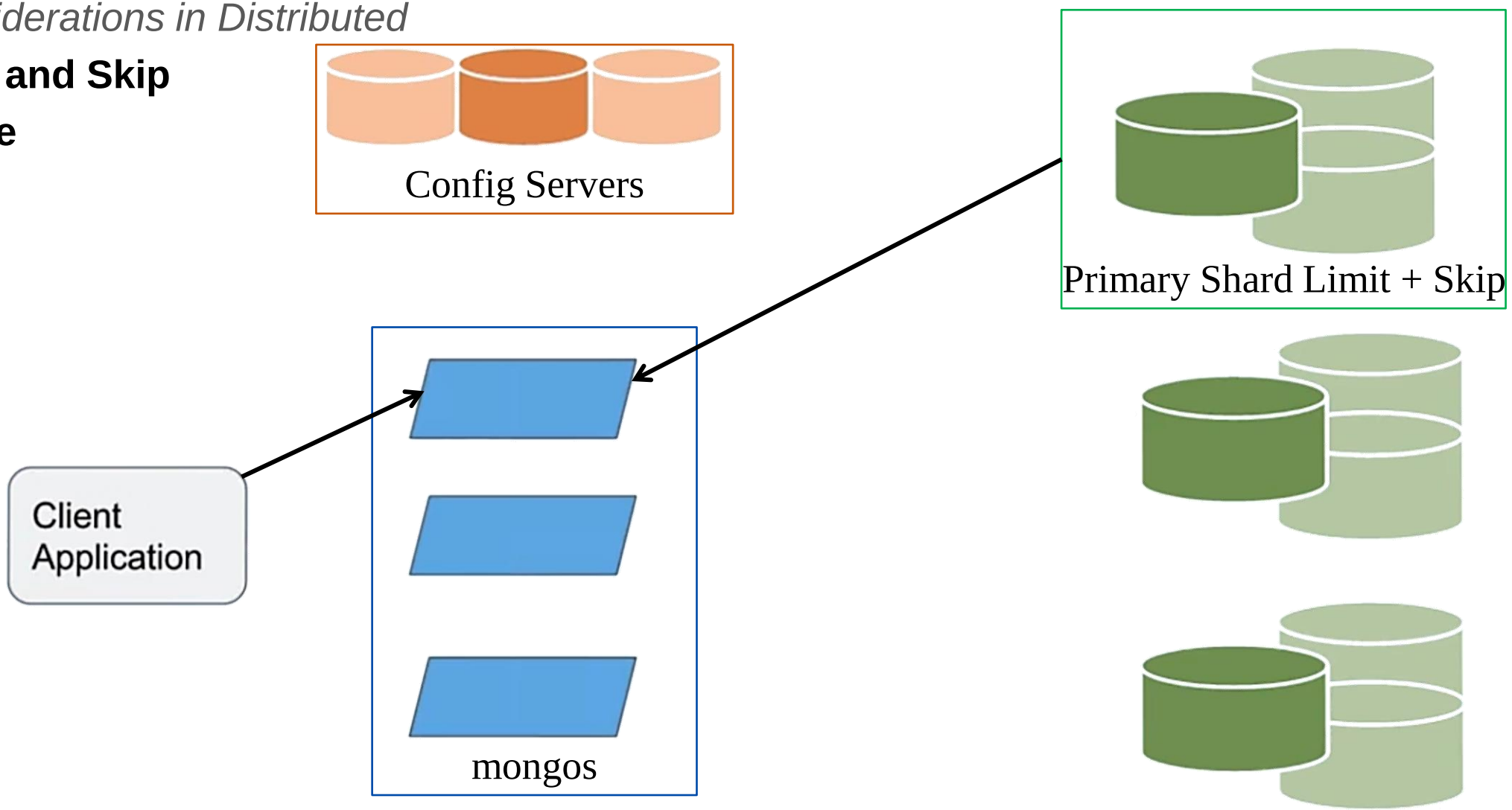


02. Performance on Clusters

Considerations in Distributed

Limit and Skip

Merge



02. Performance on Clusters

Considerations in Distributed

What you've learned

- Considerations before sharding
- Latency
- Scattered gather and routed queries
- Sorting, limit & skip

02. Performance on Clusters

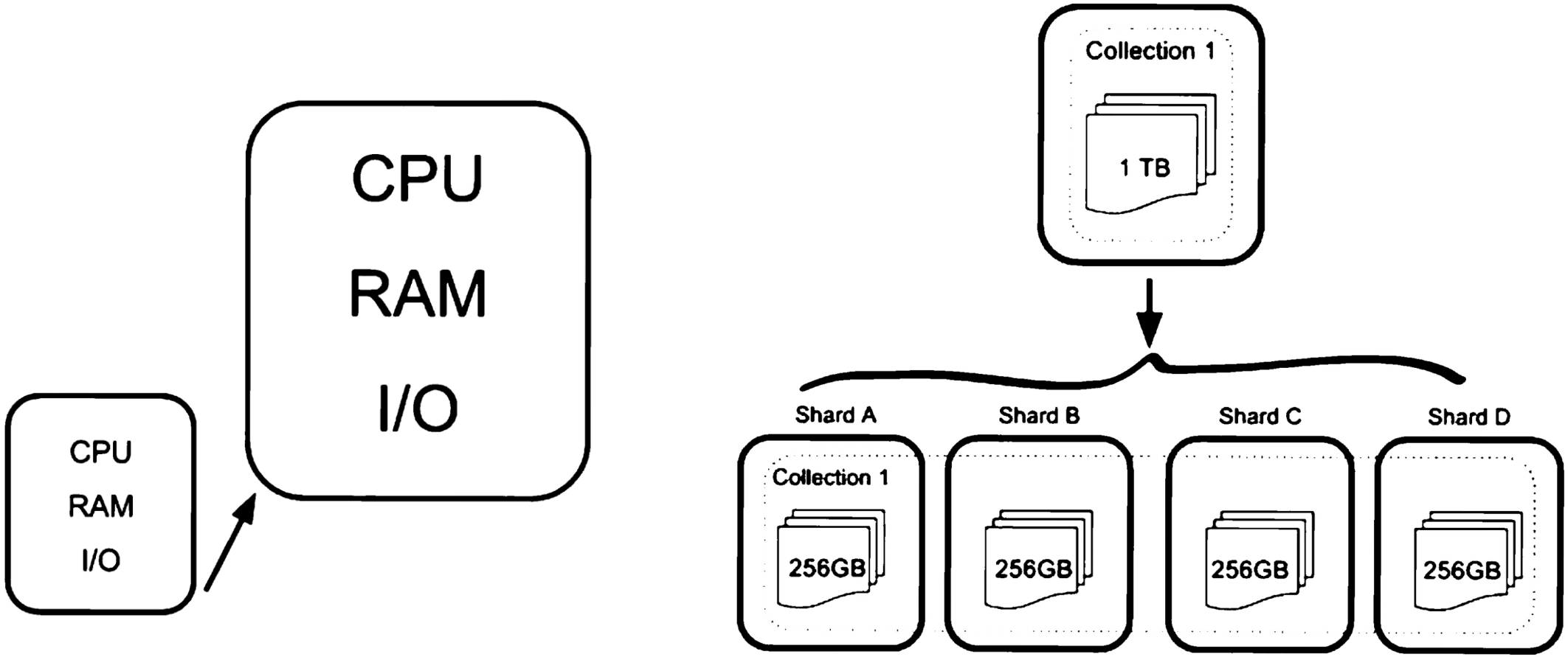
Increasing Write Performance with Sharding

- Vertical vs. horizontal scaling;
- Shard key rules;
- Bulk writes.

02. Performance on Clusters

Increasing Write Performance with Sharding

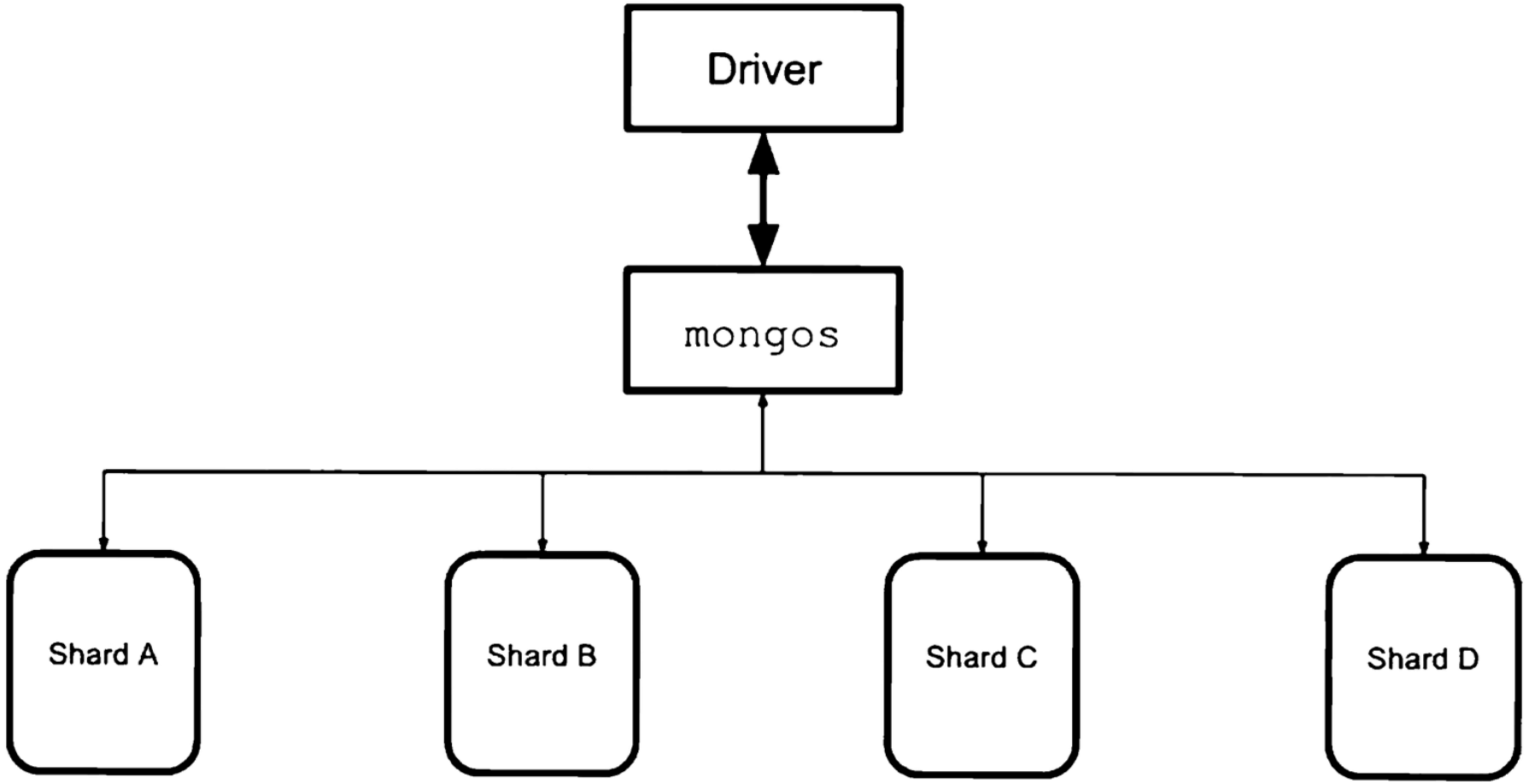
Vertical vs. horizontal scaling



02. Performance on Clusters

Increasing Write Performance with Sharding

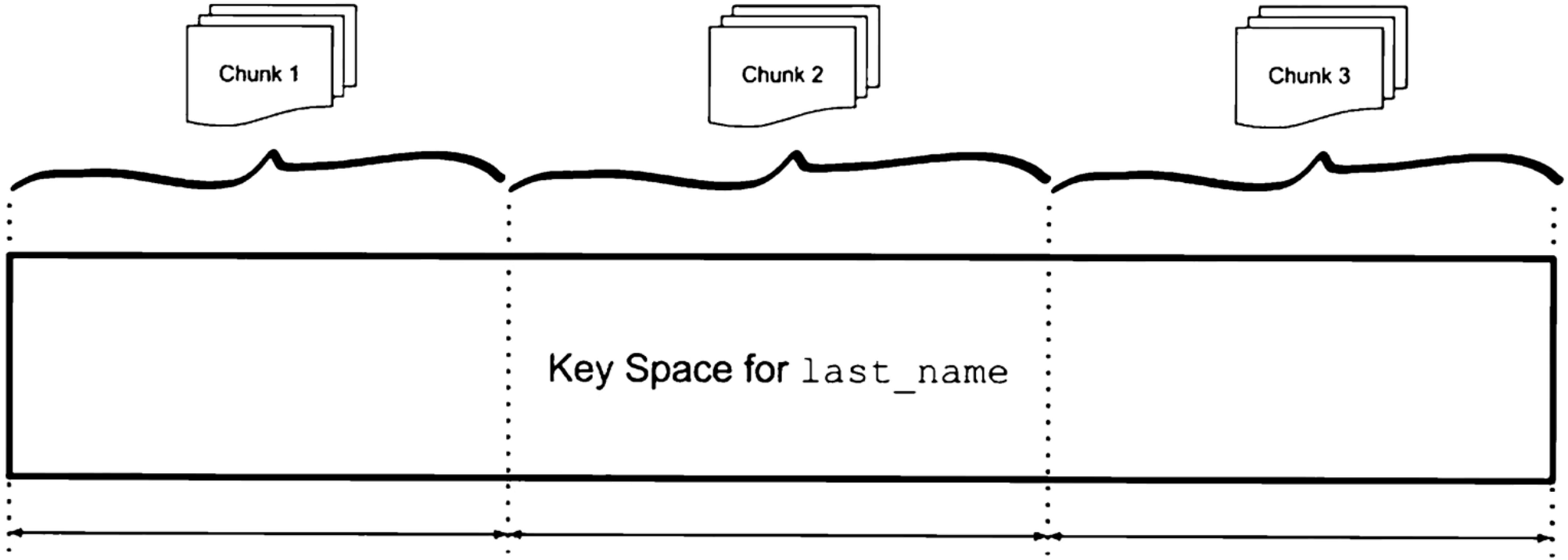
sh.shardCollection('m201.people', { last_name : 1 })



02. Performance on Clusters

Increasing Write Performance with Sharding

sh.shardCollection(‘m201.people’, { last_name : 1 })



last_name: “Abbott” last_name: “Fisher” last_name: “Rodgers” last_name: “Zuniga”

02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors

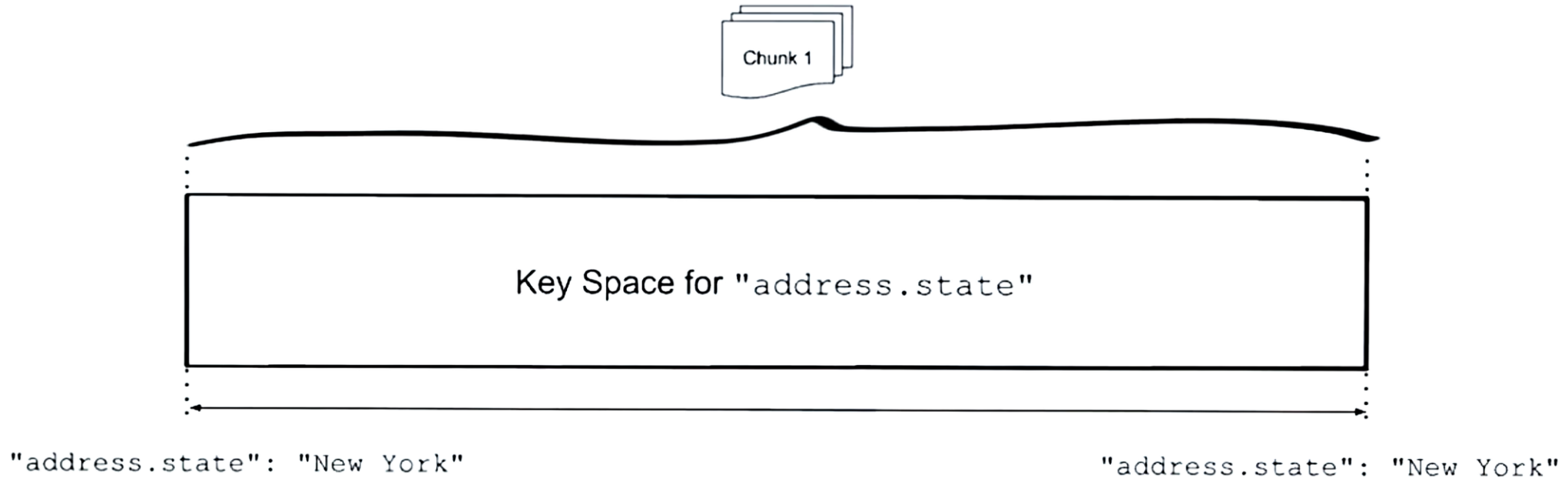
- Cardinality;
- Frequency;
- Rate of change.

02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors - Cardinality

```
sh.shardCollection('m201.people', { "address.state": 1 })
```

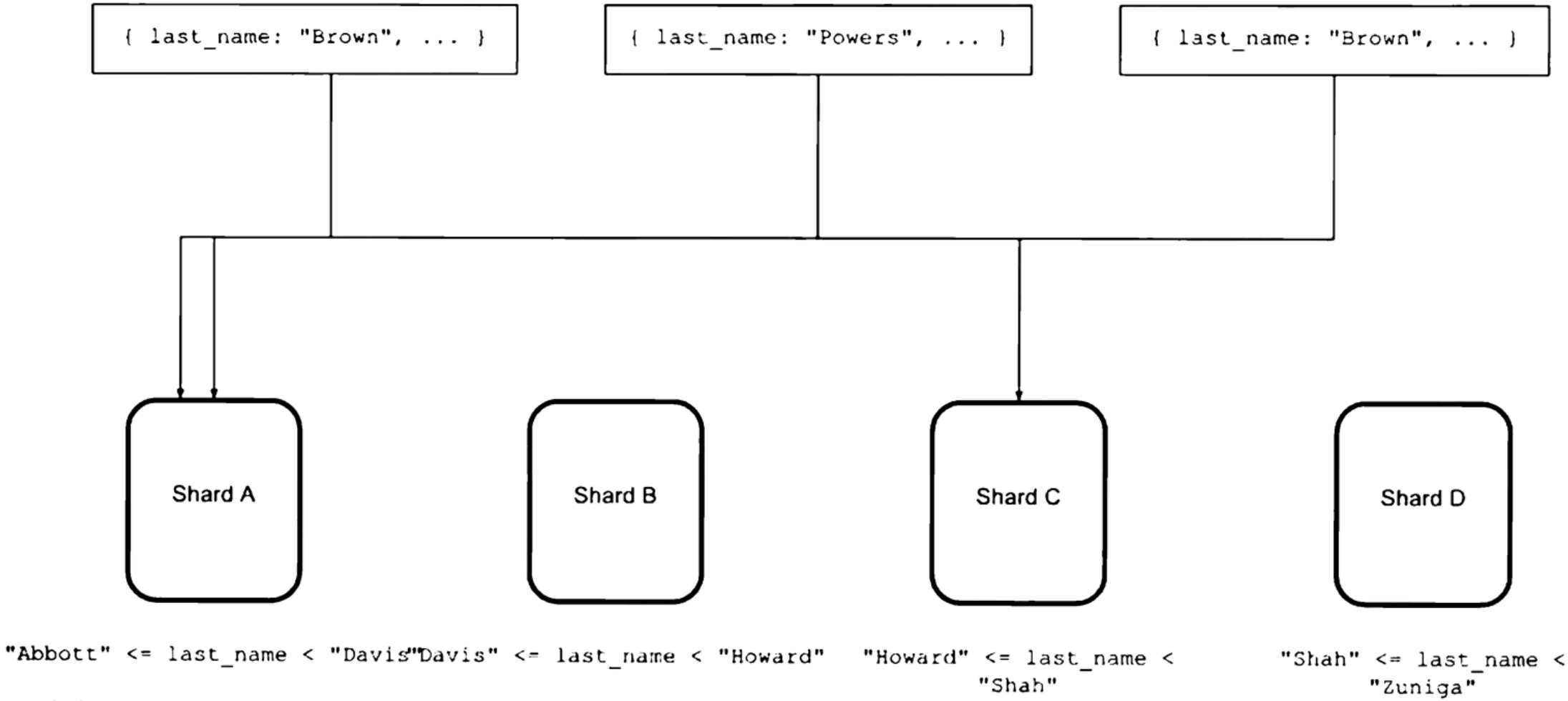


sh.shardCollection('m201.people', { "address.sate" : 1, last_name: 1 })

02. Performance on Clusters

Increasing Write Performance with Sharding

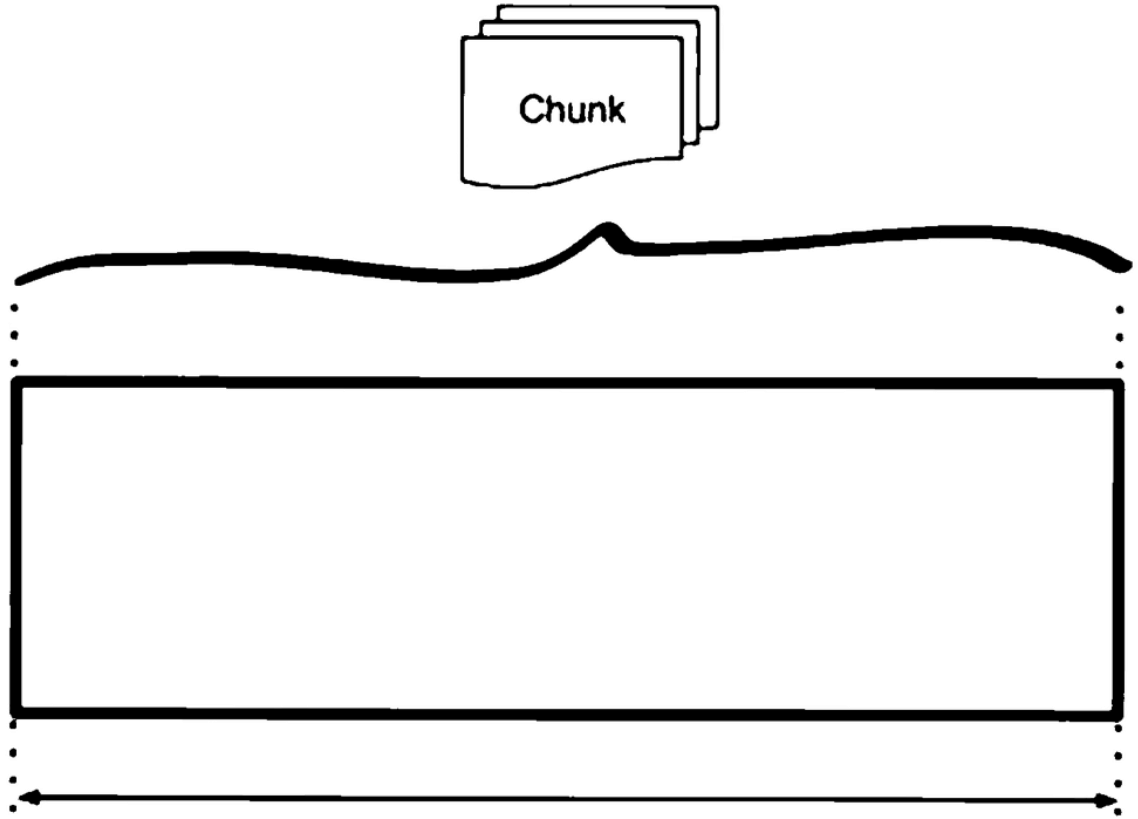
Shard Key Factors - Frequency



02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors - Frequency



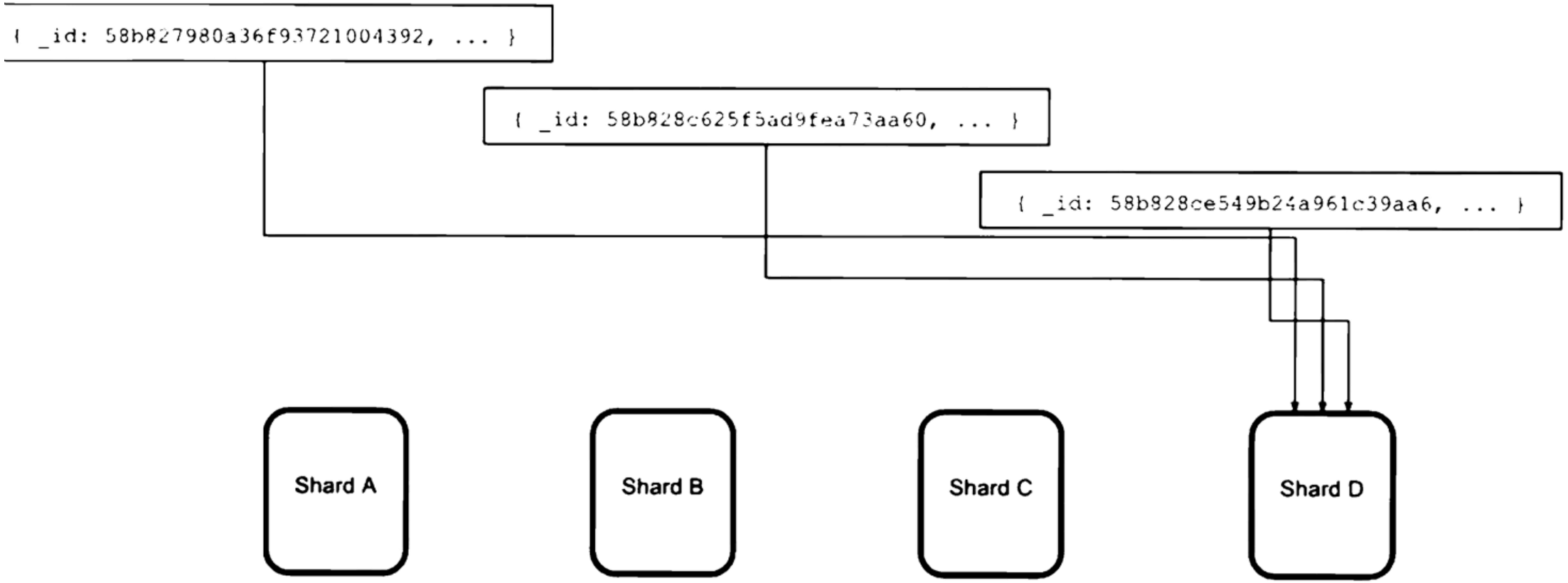
"last_name": "Brown"

"last_name": "Brown"

02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors – Rate of Change



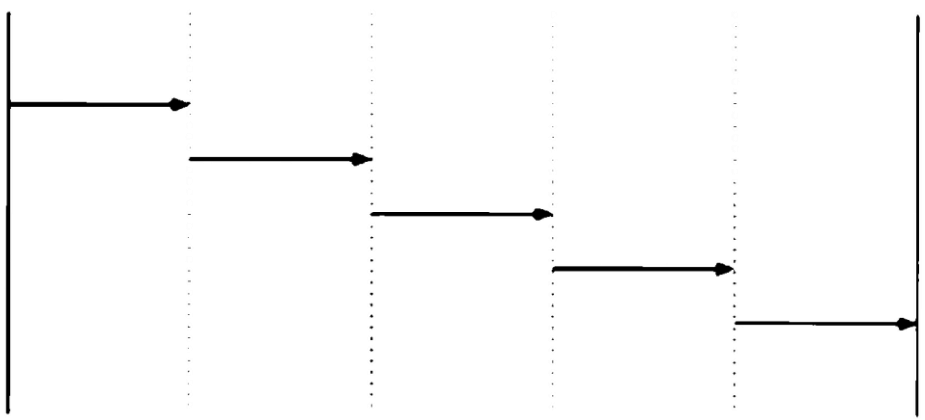
sh.shardCollection('m201.people', { "address.sate" : 1, last_name: 1 })

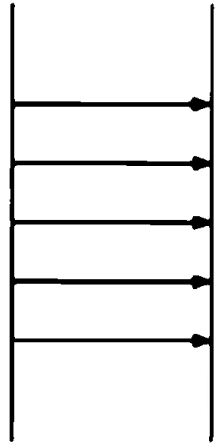
02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors – Bulk Writes

```
db.collection.bulkWrite(
   [ <operation 1>, <operation 2>, ... ],
   { ordered : true }
)
```

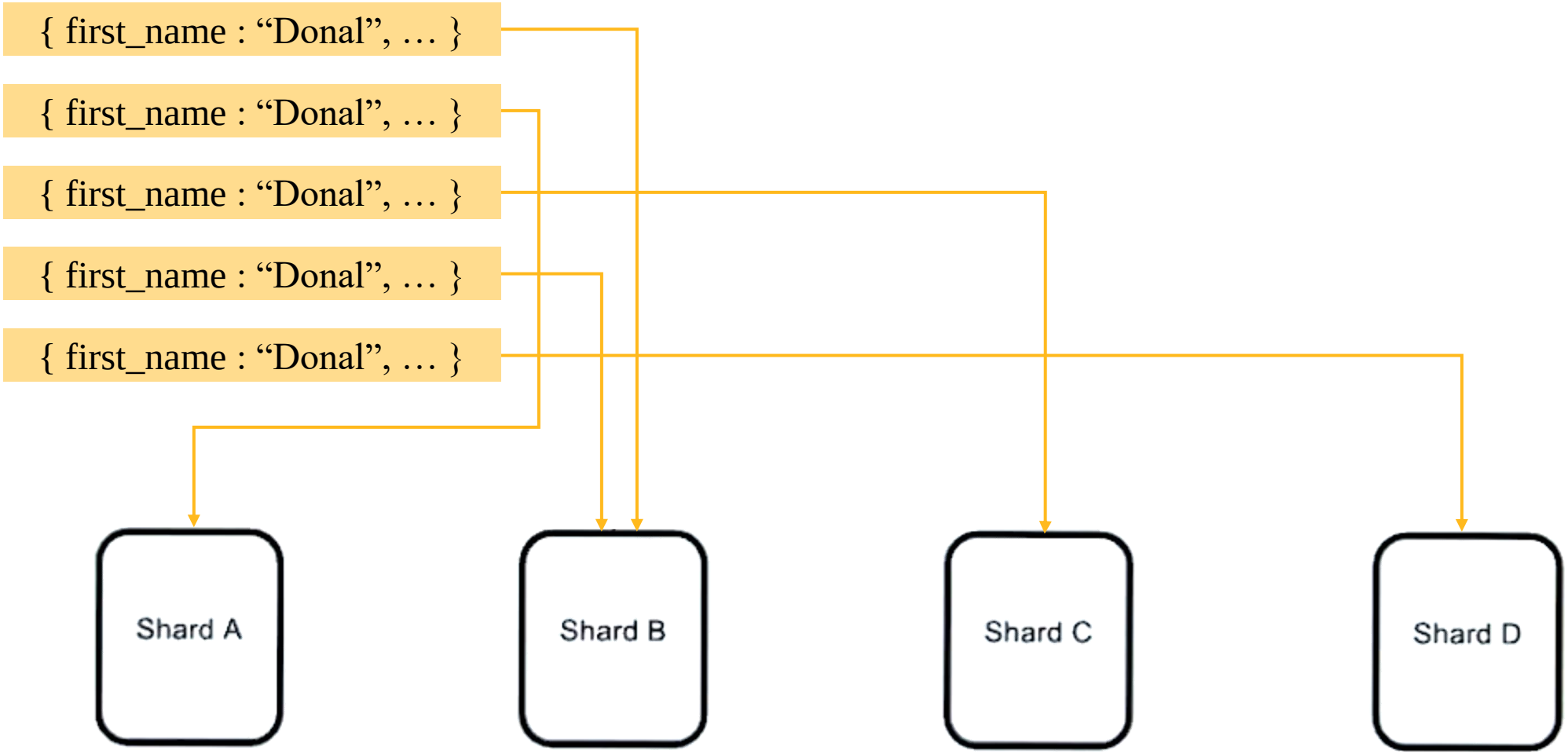




02. Performance on Clusters

Increasing Write Performance with Sharding

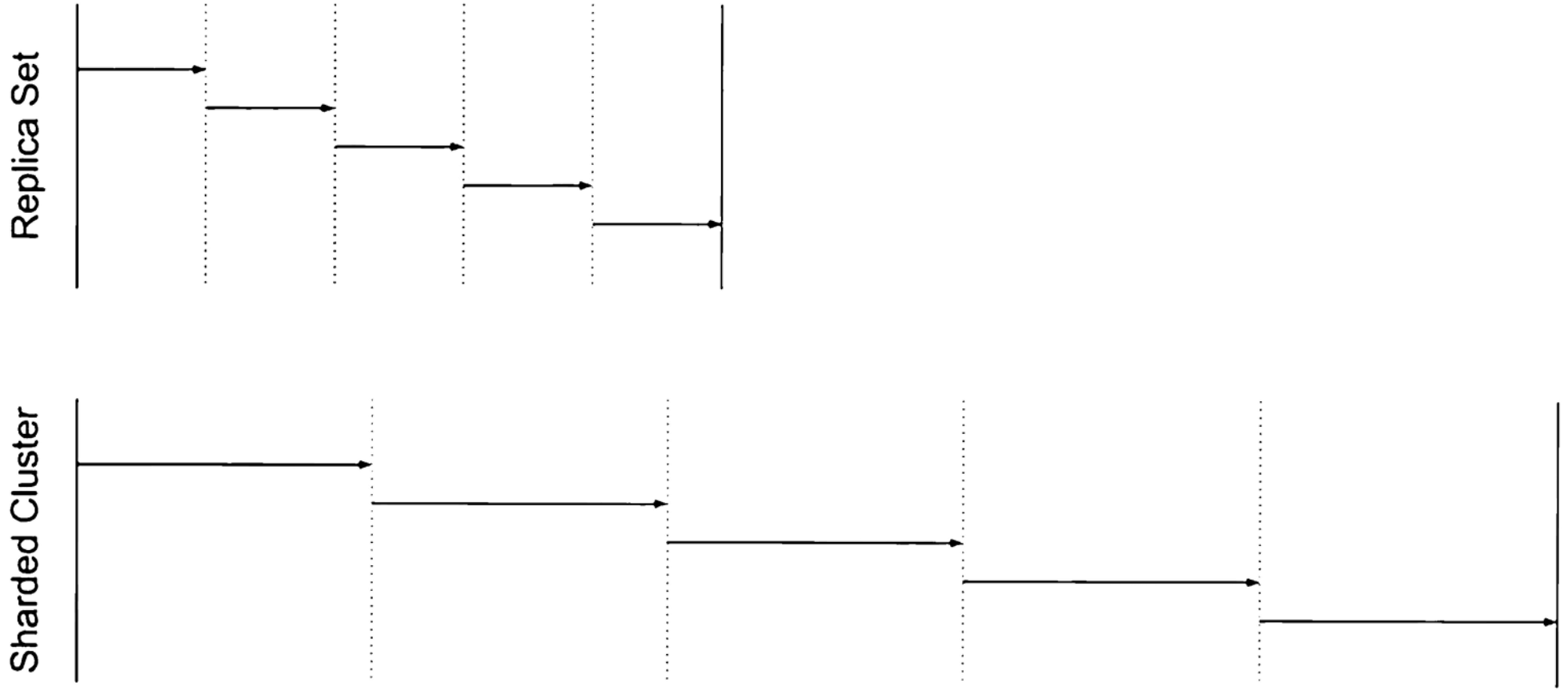
Shard Key Factors – Bulk Writes



02. Performance on Clusters

Increasing Write Performance with Sharding

Shard Key Factors – Bulk Writes



02. Performance on Clusters

Increasing Write Performance with Sharding

What you've learned

- Vertical vs. horizontal scaling;
- Shard key rules;
- Bulk writes.

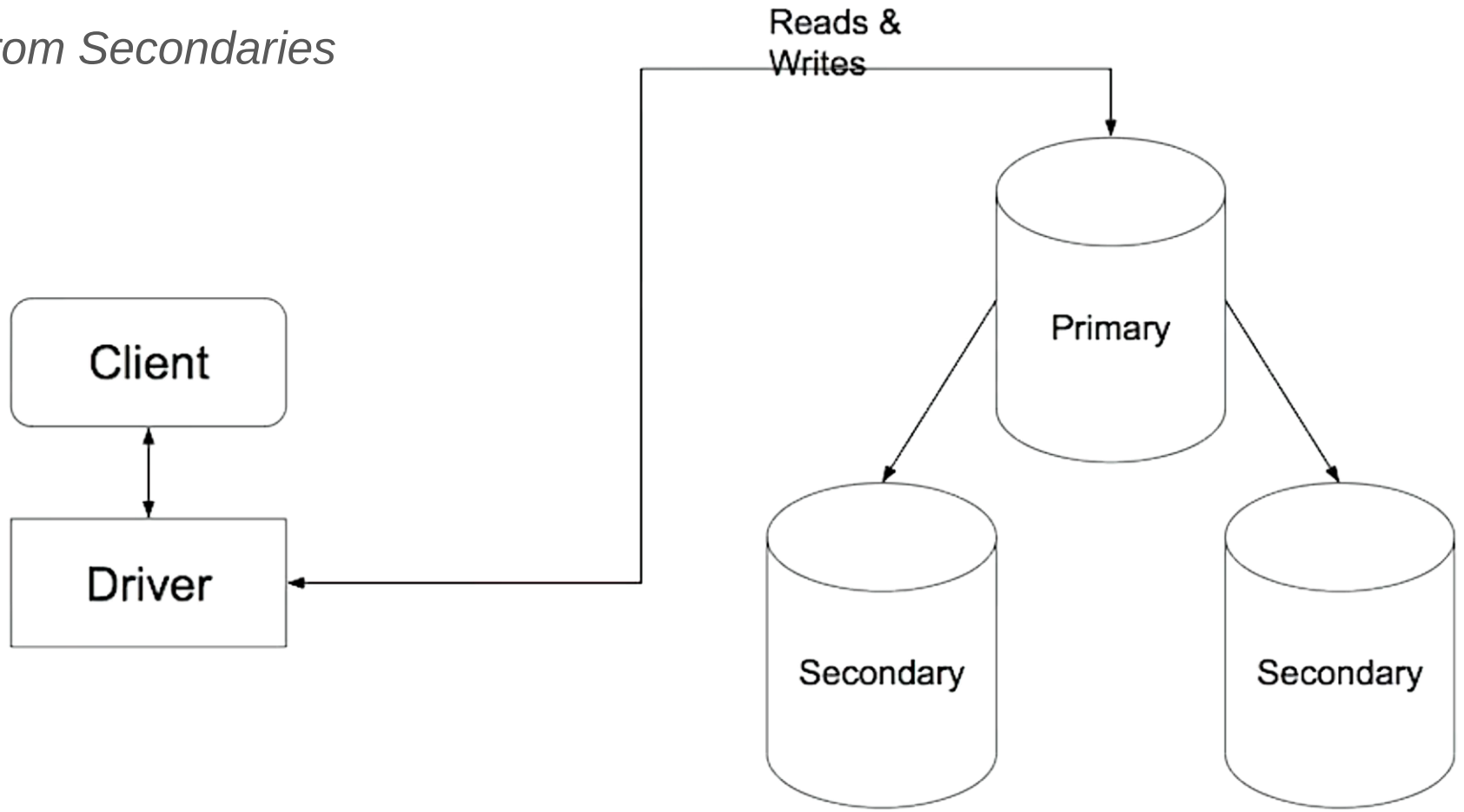
02. Performance on Clusters

Reading from Secondaries

*db.people.find().readPref("**primary**")*
*db.people.find().readPref("**primaryPreferred**")*
*db.people.find().readPref("**secondary**")*
*db.people.find().readPref("**secondaryPreferred**")*
*db.people.find().readPref("**nearest**")*

02. Performance on Clusters

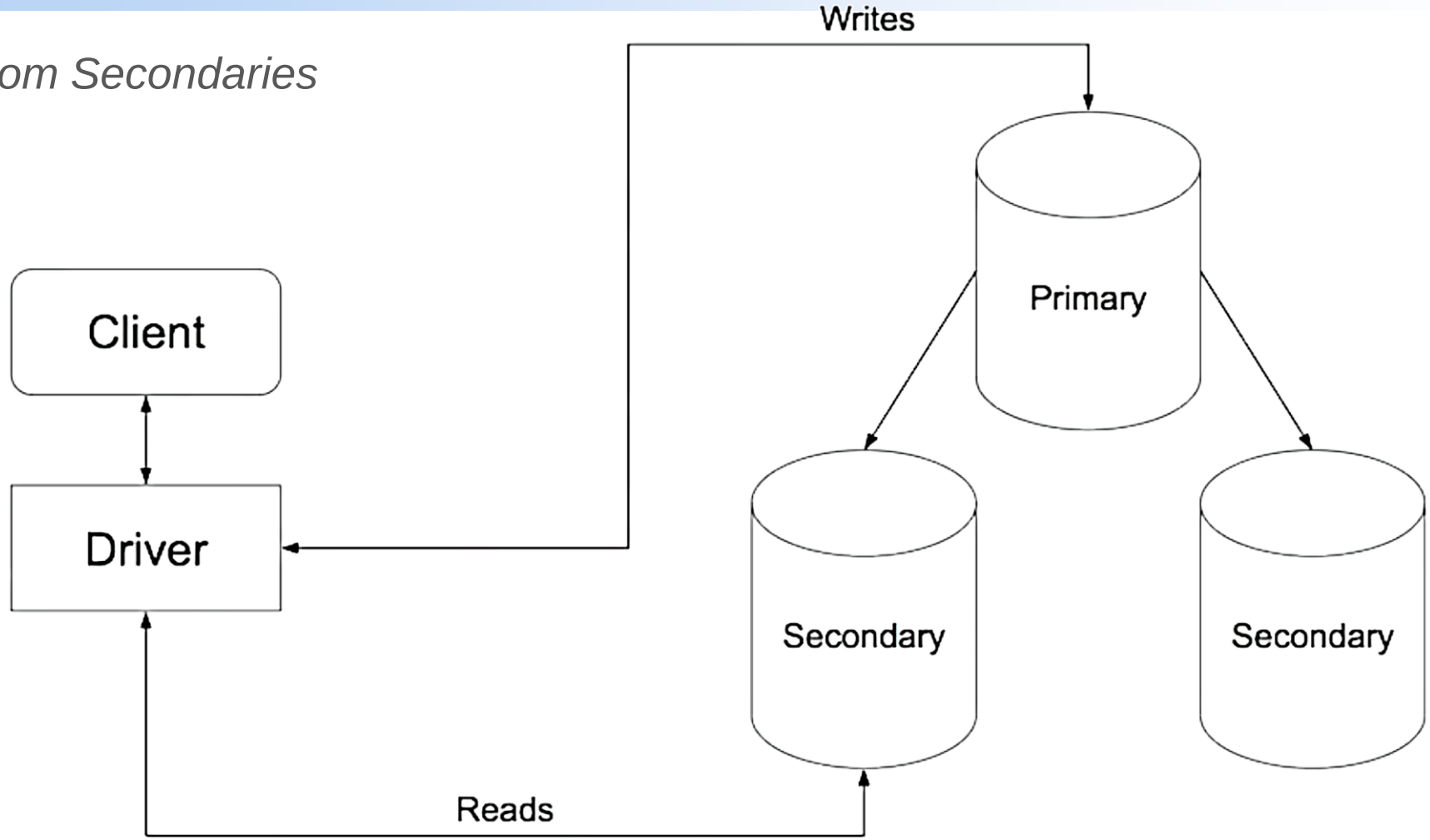
Reading from Secondaries



*db.people.find().readPref("**primary**")*

02. Performance on Clusters

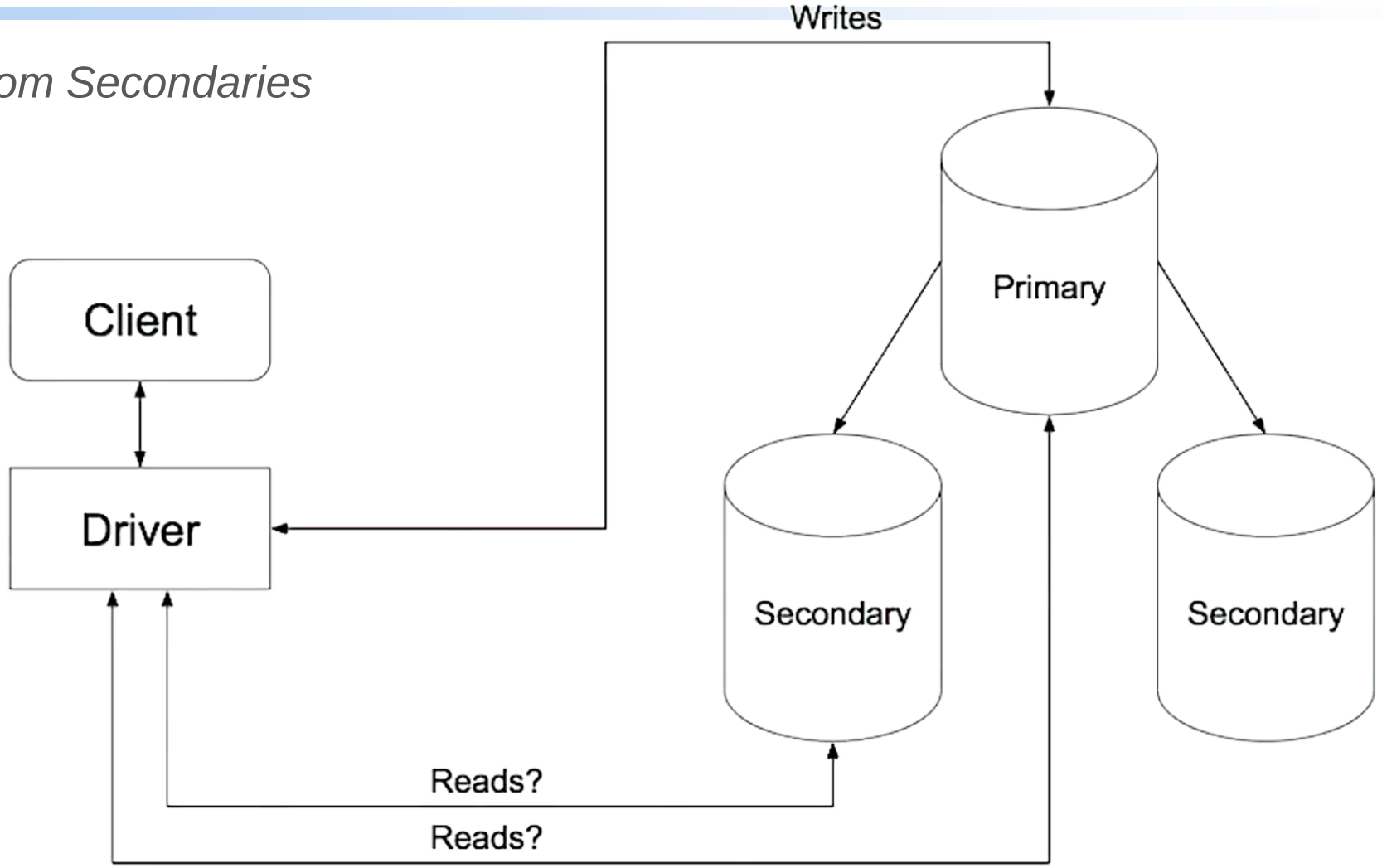
Reading from Secondaries



*db.people.find().readPref("**secondary**")*

02. Performance on Clusters

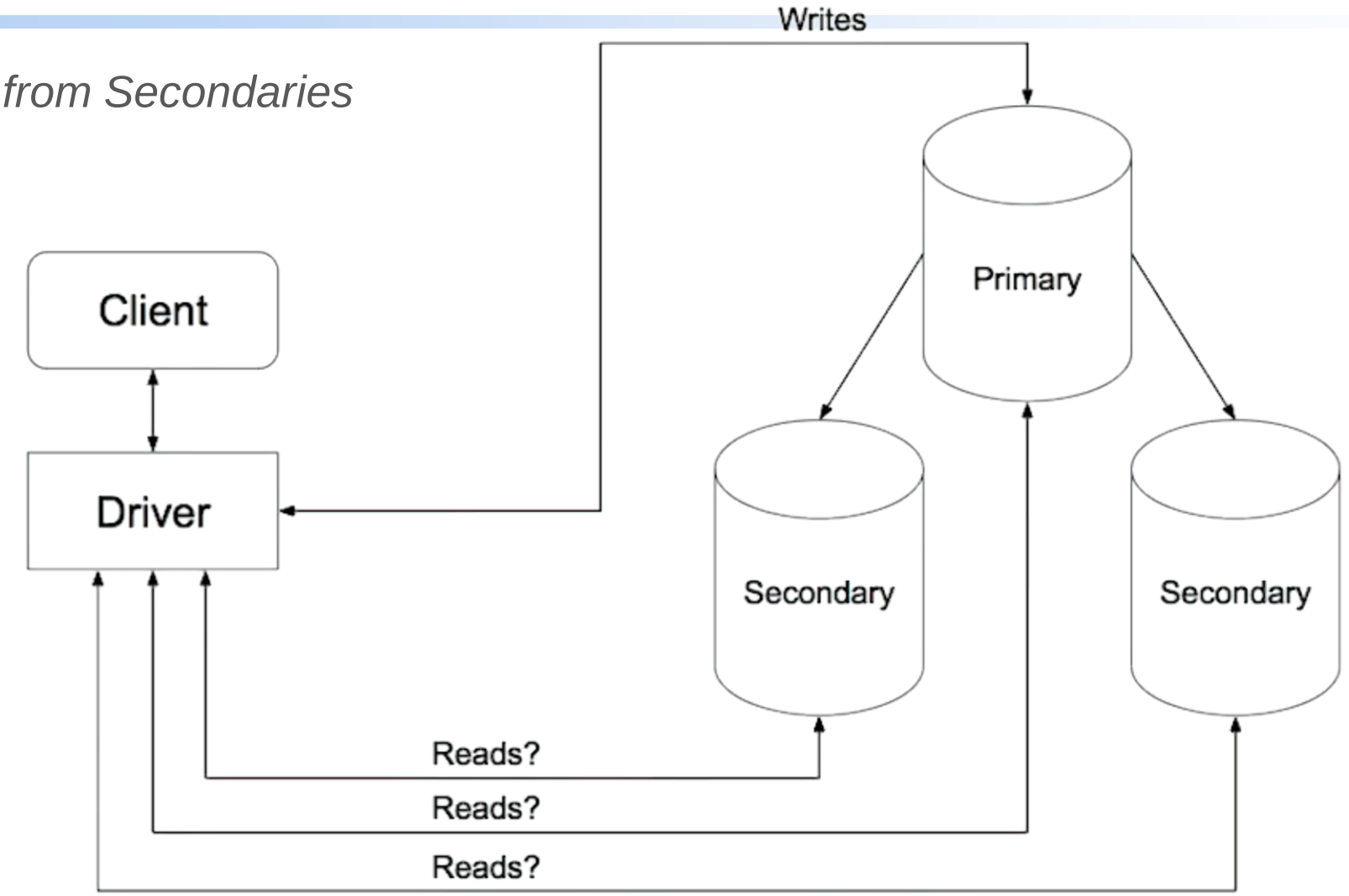
Reading from Secondaries



*db.people.find().readPref("**secondaryPreffered**")*

02. Performance on Clusters

Reading from Secondaries



*db.people.find().readPref("**nearest**")*

02. Performance on Clusters

Reading from Secondaries

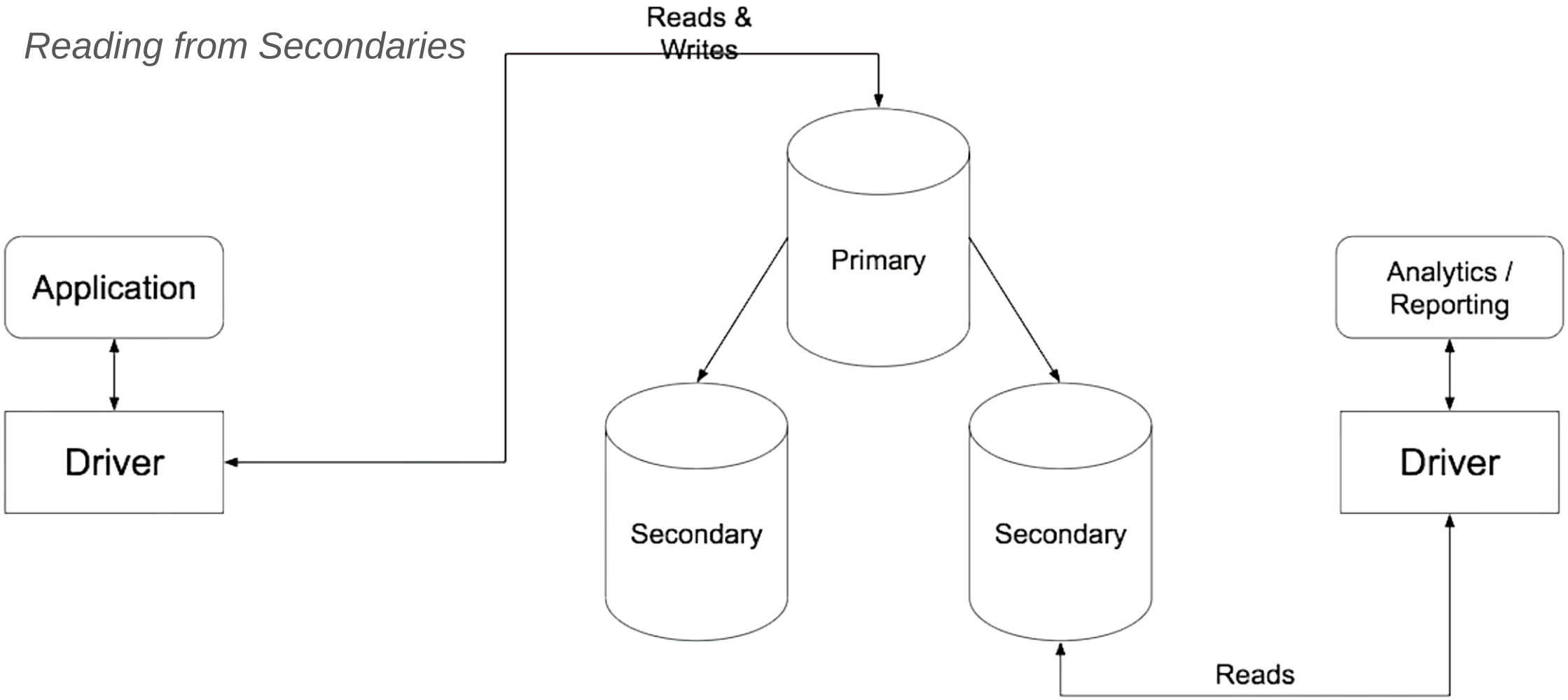
When Reading from a Secondary is a Good Idea

Analytics queries

Local reads

02. Performance on Clusters

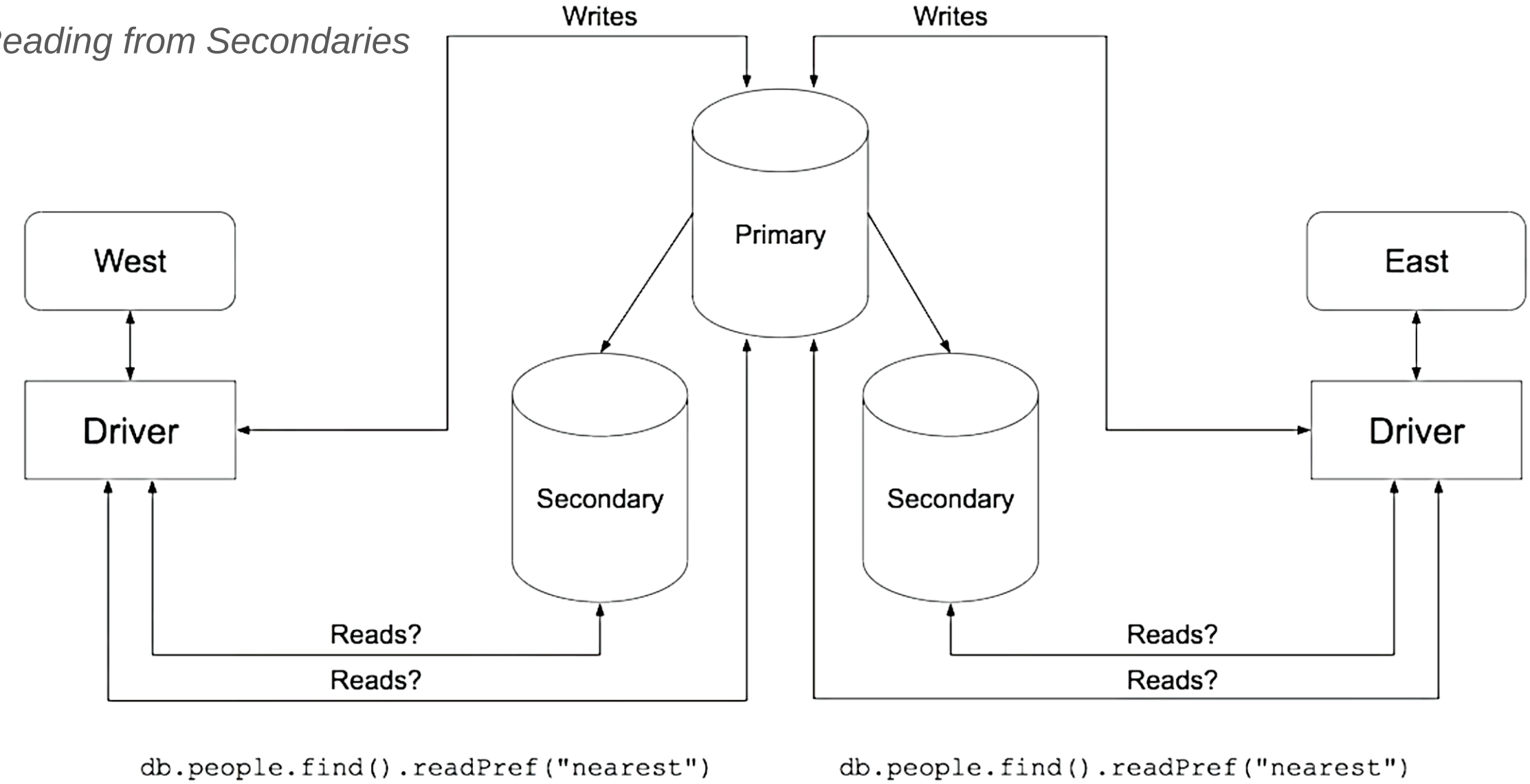
Reading from Secondaries



*db.people.find().readPref("**secondary**")*

02. Performance on Clusters

Reading from Secondaries



02. Performance on Clusters

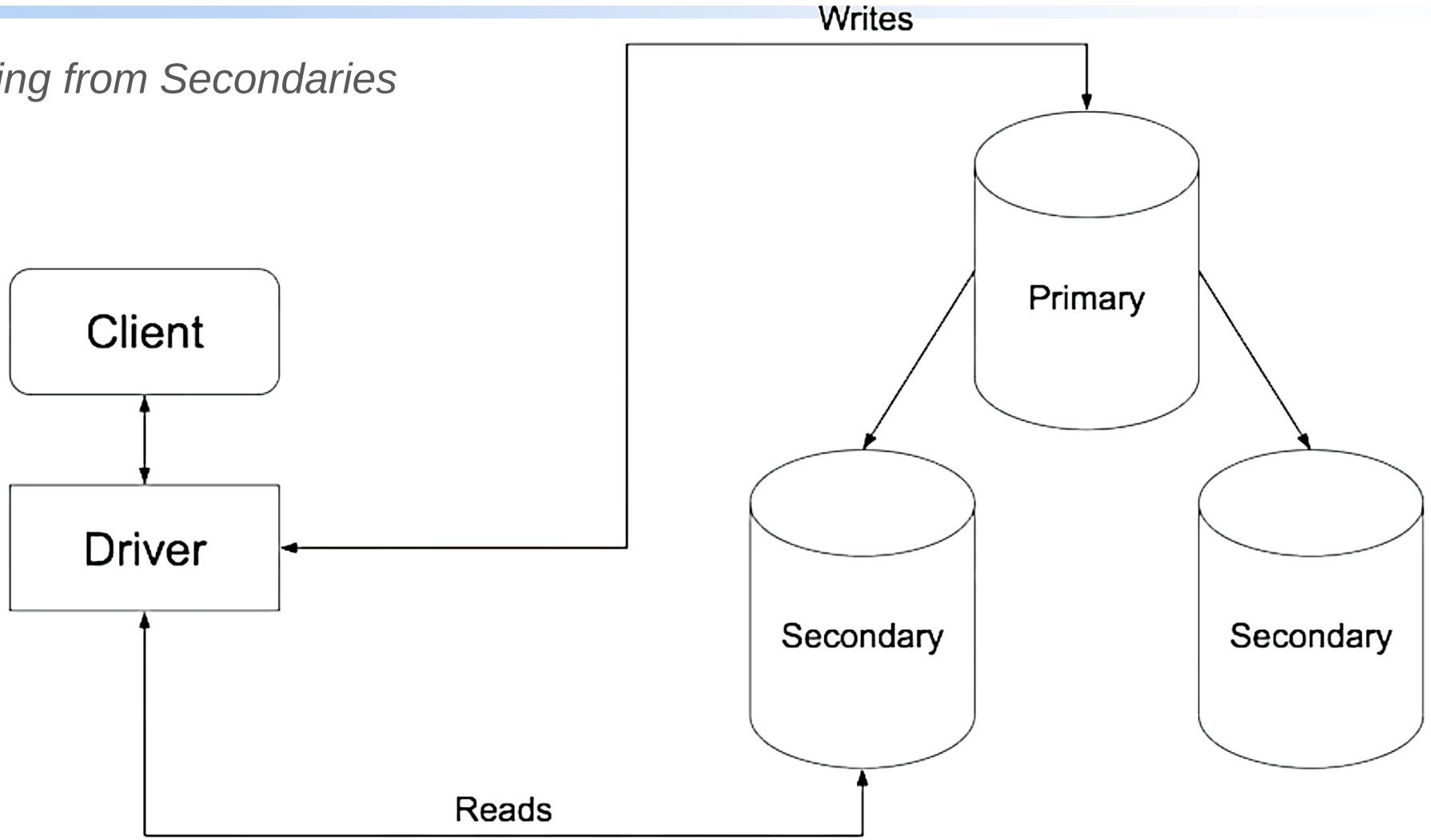
Reading from Secondaries

When Reading from a Secondary is a Bad Idea

Providing extra capacity for reads

02. Performance on Clusters

Reading from Secondaries



*db.people.find().readPref("**secondary**")*

02. Performance on Clusters

Reading from Secondaries

What you've learned

- Read preferences associated with performance;
- When it's a good idea
  - Analytics queries;
  - Local reads.
- When it's a bad idea

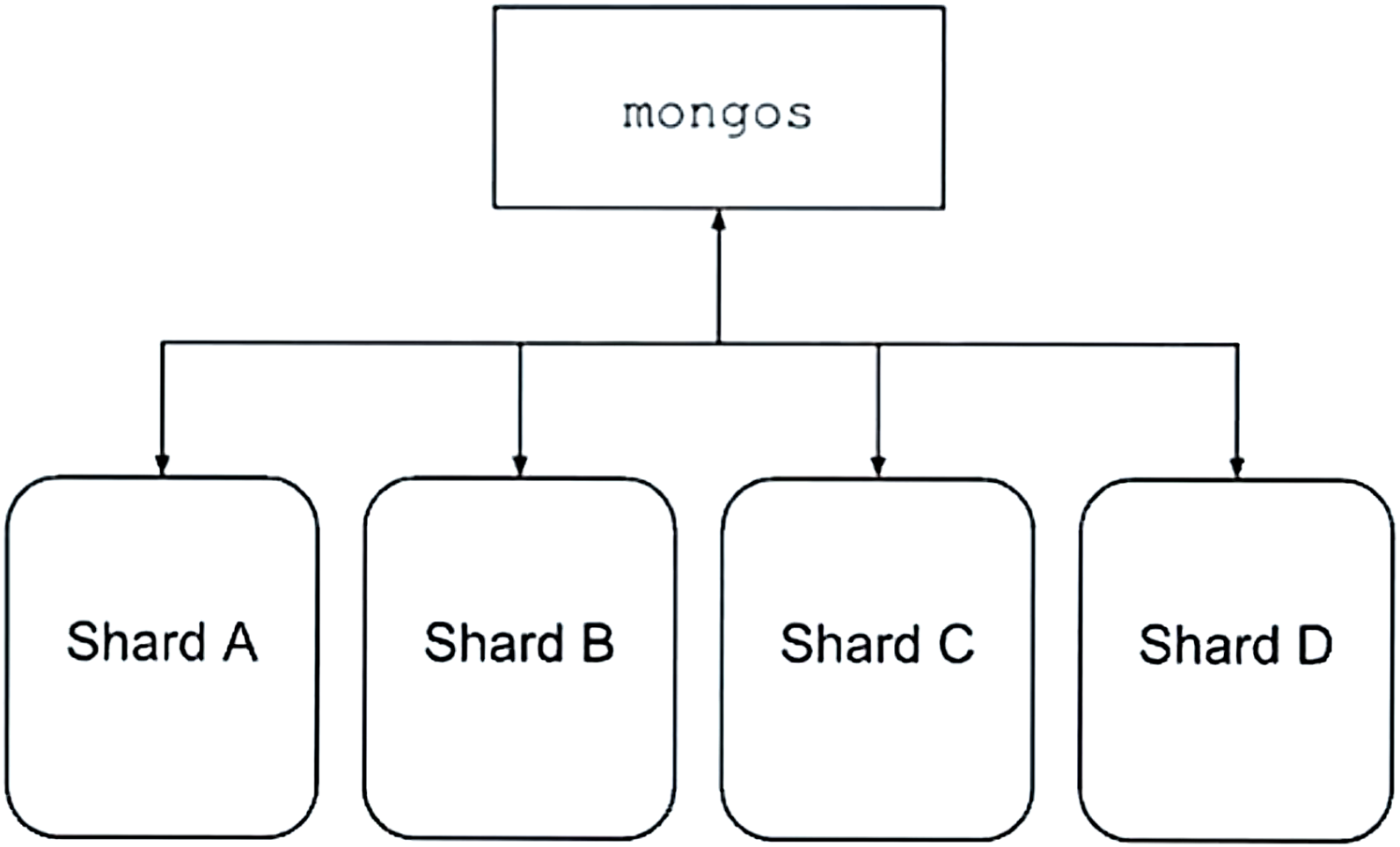
02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

- How it works;
- Where operations are completed;
- Optimizations.

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

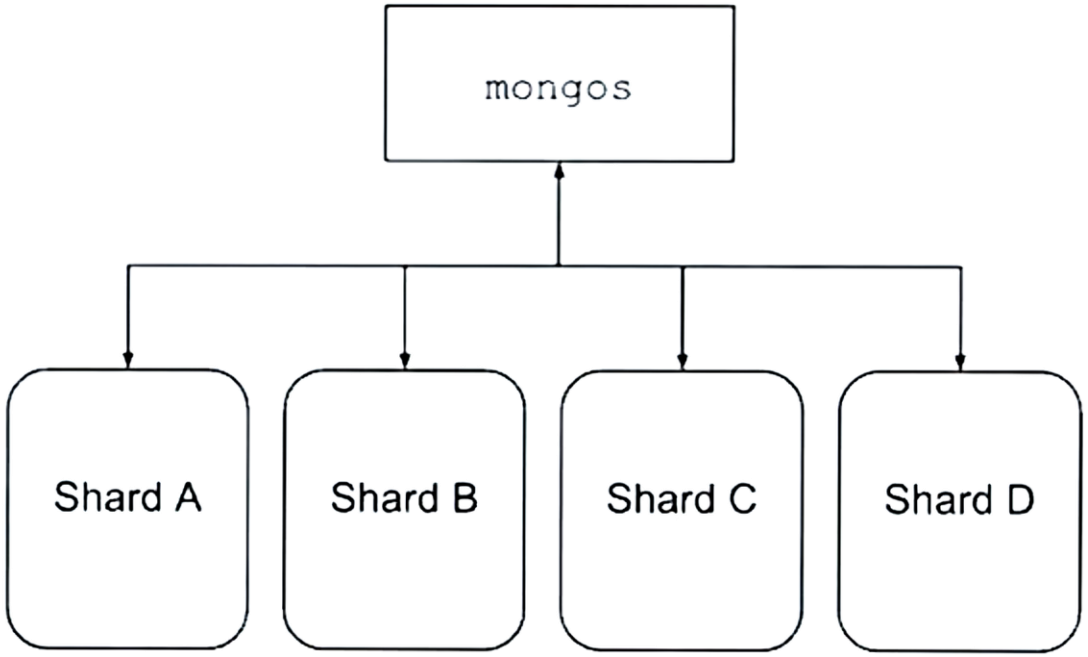


02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

```
sh.shardCollection('m201.restaurants, { "address.state": 1 })
```

```
db.restaurants.aggregate([
  {
    $match: { 'address.state': 'NY' }
  },
  {
    $group: {
      _id: '$address.state',
      avgStars: { $avg: '$stars' }
    }
  }
])
```

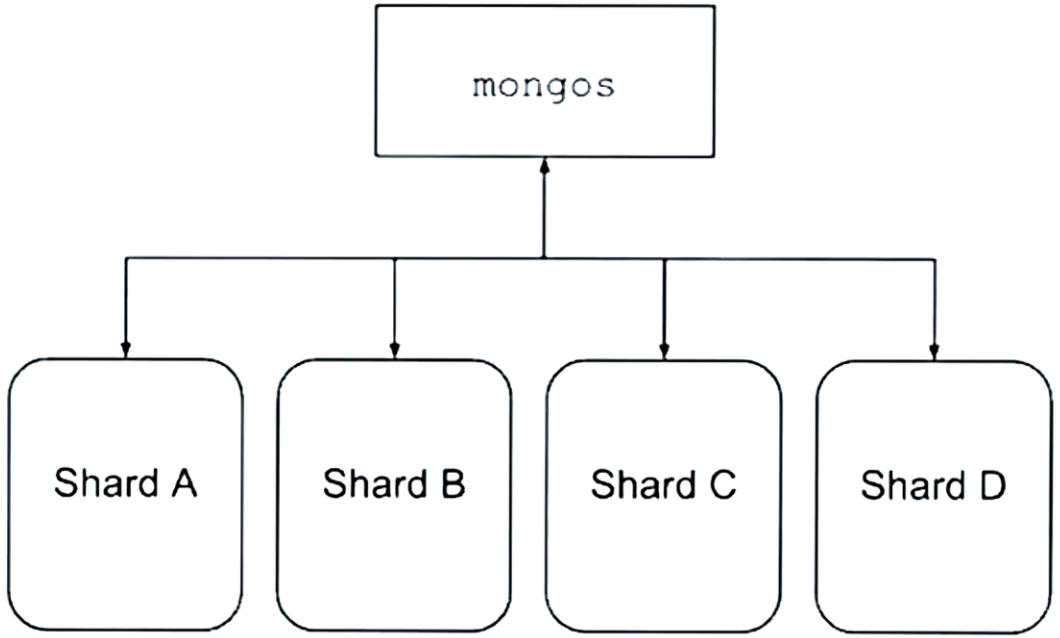


02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

```
sh.shardCollection('m201.restaurants, { "address.state": 1 })
```

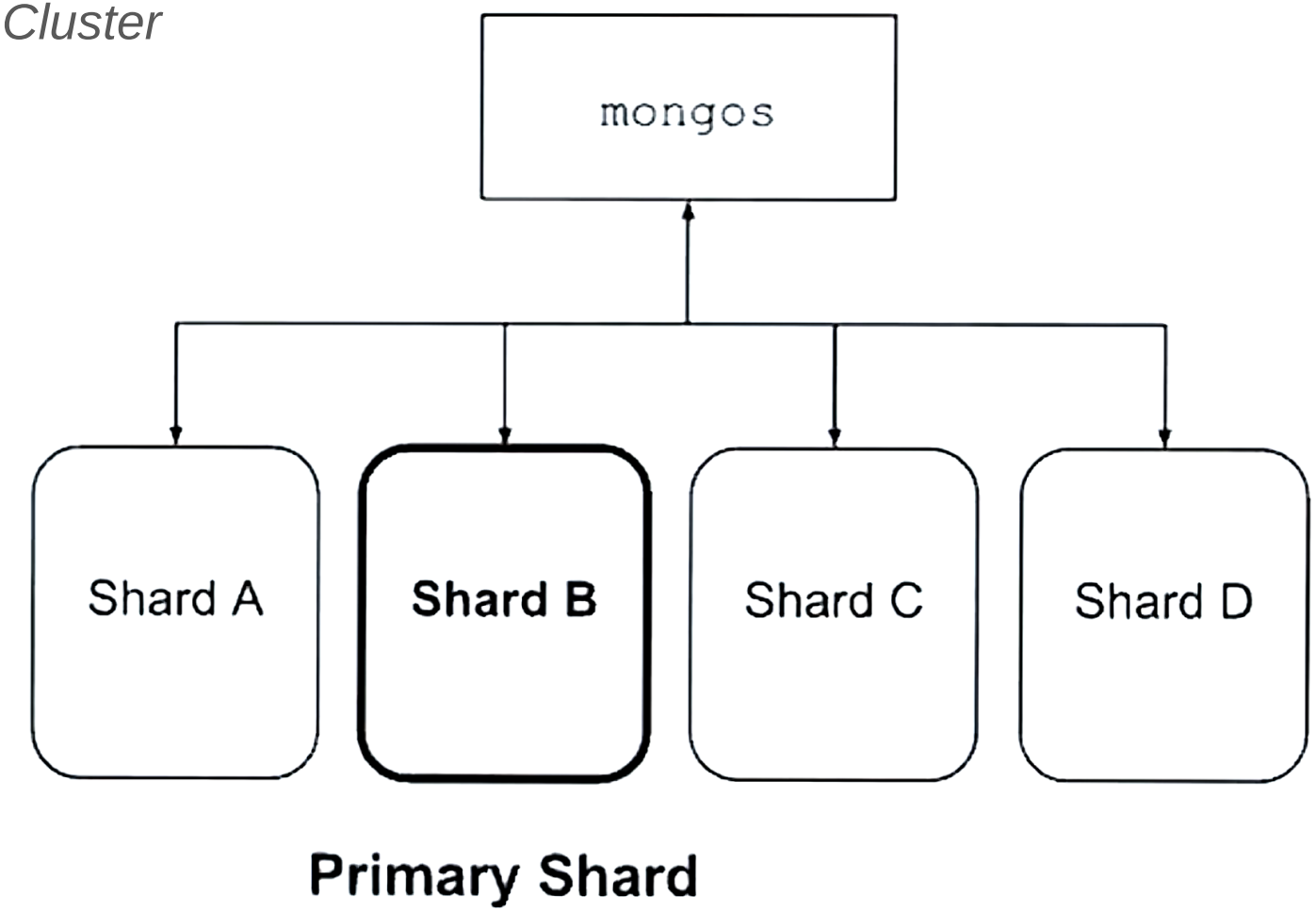
```
db.restaurants.aggregate([
  {
    $group: {
      _id: '$address.state',
      avgStars: { $avg: '$stars' }
    }
  }
])
```



02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

- $out;
- $facet;
- $lookup;
- $graphLookup.



02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

Aggregation Optimizations

```
db.restaurants.aggregate([
  {
    $sort: { stars : -1 }
  },
  {
    $match: { cuisine: 'Sushi' }
  }
])
```

```
db.restaurants.aggregate([
  {
    $match: { cuisine: 'Sushi' }
  },
  {
    $sort: { stars : -1 }
  }
])
```

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

Aggregation Optimizations

```
db.restaurants.aggregate([
   {
     $skip: 10
   },
   {
     $limit: 5
   }
])
```

```
db.restaurants.aggregate([
   {
     $limit: 15
   },
   {
     $skip: 10
   }
])
```

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

Aggregation Optimizations

```
db.restaurants.aggregate([
   {
     $limit: 10
   },
   {
     $limit: 5
   }
])
```

```
db.restaurants.aggregate([
   {
     $limit: 15
   }
])
```

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

Aggregation Optimizations

```
db.restaurants.aggregate([
  {
    $match: { cuisine: 'Sushi' }
  },
  {
    $match: { stars: 5.0 }
  }
])
```

```
db.restaurants.aggregate([
  {
    $match: {
      cuisine: 'Sushi',
      stars: 5.0
    }
  }
])
```

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

What you've learned

- How it works;
- Where operations are completed;
- Optimizations;

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

02. Performance on Clusters

Aggregation Pipeline on a Sharded Cluster

Indexes

In the database world, index plays a vital role in a performance, that not an exception with MongoDB

Index Type	Index Properties
• Single Field Indexes	• TTL Indexes
• Compound Indexes	• Unique Indexes
• Multikey Indexes	• Partial Indexes
• Text Indexes	• Case Insensitive Indexes
• Wildcard Indexes	• Hidden Indexes
• 2dsphere Indexes	• Sparse Indexes
• 2d Indexes	
• geoHaystack Indexes	
• Hashed Indexes	

Indexing strategies

- Use the ESR (Equality, Sort, Range)
- Create Indexes to Support Your Queries
- Use Indexes to Sort Query
- Ensure Indexes Fit in RAM
- Create Queries that Ensure Selectivity

Follow ESR Rule in Compound Indexes

Equality

- "Equality" refers to an exact match on a single value.
- Example:

```
db.cars.find( { model: "Cordoba" } )

db.cars.find( { model: { $eq: "Cordoba" } } )
```

- Place fields that require exact matches first in your index.

- An index may have multiple keys for queries with exact matches. The index keys for equality matches can appear in any order. However, to satisfy an equality match with the index, all of the index keys for exact matches must come before any other index fields.

- Exact matches should be selective. To reduce the number of index keys scanned, ensure equality tests eliminate at least 90% of possible document matches.

Follow ESR Rule in Compound Indexes

Sort

- "Sort" determines the order for results. Sort follows equality matches because the equality matches reduce the number of documents that need to be sorted.

- An index can support sort operations when the query fields are a subset of the index keys. Sort operations on a subset of the index keys are only supported if the query includes equality conditions for all of the prefix keys that precede the sort keys.

- Example: queries the cars collection, the output is sorted by model

```
db.cars.find( { manufacturer: "GM" } ).sort( { model: 1 } )
```

- To improve query performance, create an index on the manufacturer and model fields:

```
db.cars.createIndex( { manufacturer: 1, model: 1 } )
```

Follow ESR Rule in Compound Indexes

Sort - Blocking sort

- A blocking sort indicates that MongoDB must consume and process all input documents to the sort before returning results. Blocking sorts do not block concurrent operations on the collection or database.

- If MongoDB cannot use an index or indexes to obtain the sort order, MongoDB must perform a blocking sort operation on the data.

- MongoDB to use temporary files on disk to store data exceeding the 100 megabyte system memory limit while processing a blocking sort operation.

- Sort operations that use an index often have better performance than blocking sorts.

Follow ESR Rule in Compound Indexes

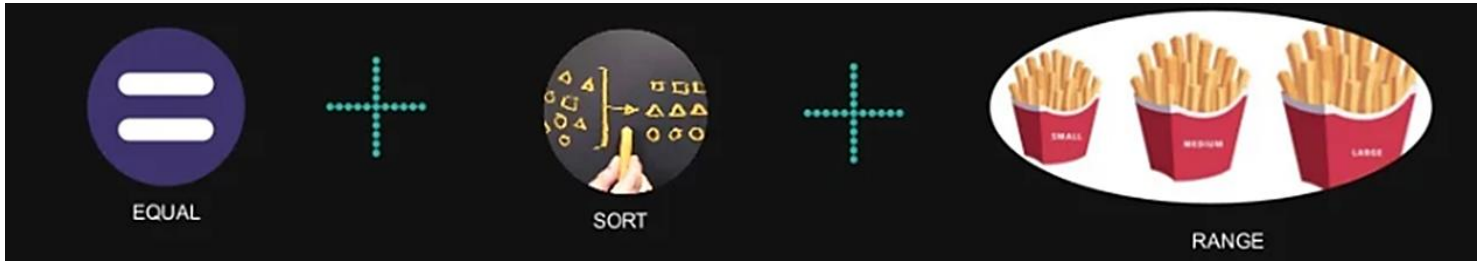
Range

- "Range" filters scan fields. The scan doesn't require an exact match, which means range filters are loosely bound to index keys. To improve query efficiency, make the range bounds as tight as possible and use equality matches to limit the number of documents that must be scanned.
- Range filters resemble the following:

```
db.cars.find( { price: { $gte: 15000} } )
db.cars.find( { age: { $lt: 10 } } )
db.cars.find( { priorAccidents: { $ne: null } } )
```

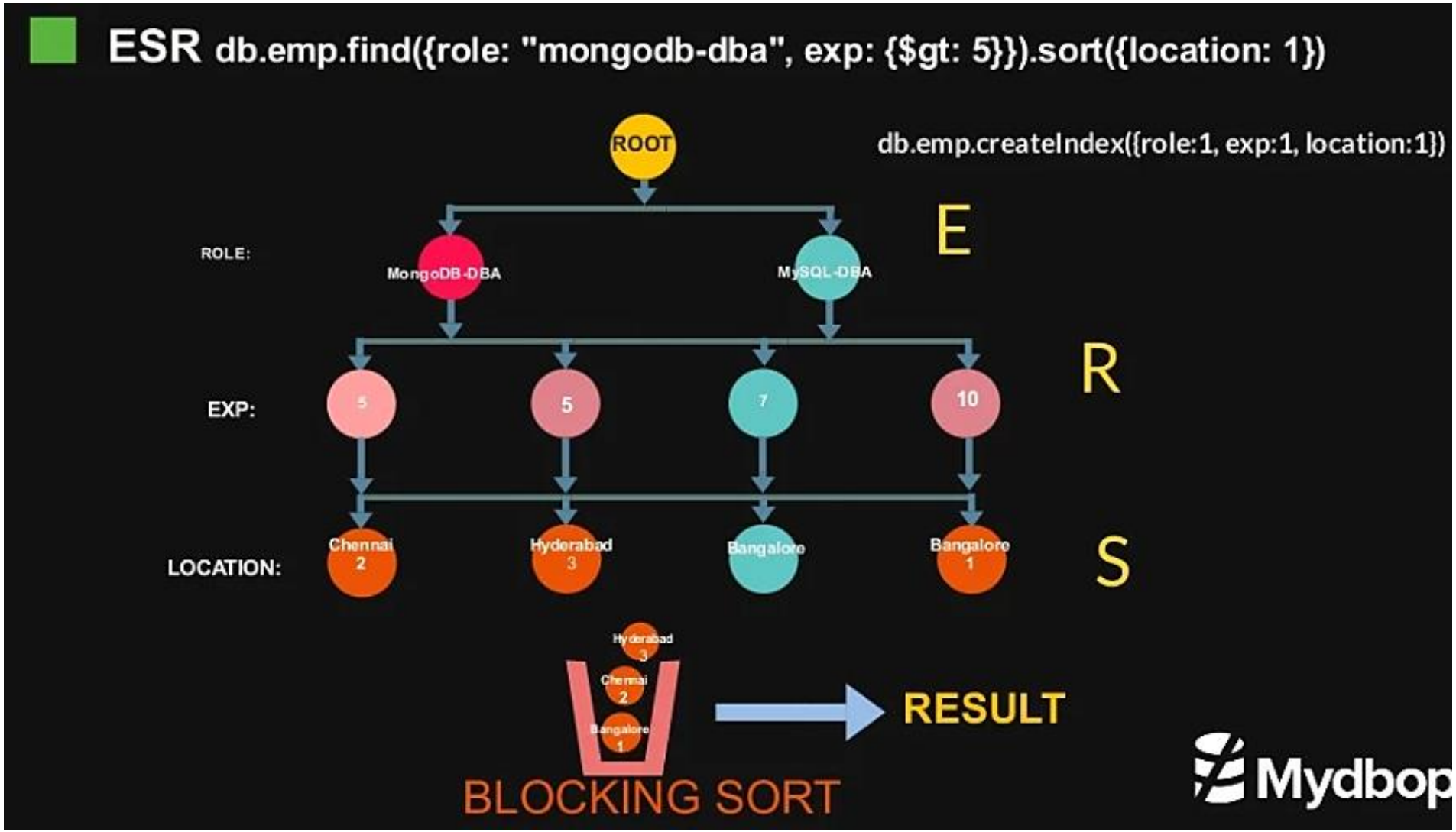
Follow ESR Rule in Compound Indexes

- For compound indexes, this rule of thumb is helpful in deciding the order of fields in the index:
  - First, add those fields against which **Equality** queries are run.
  - The next fields to be indexed should reflect the **Sort** order of the query.
  - The last fields represent the **Range** of data to be accessed.

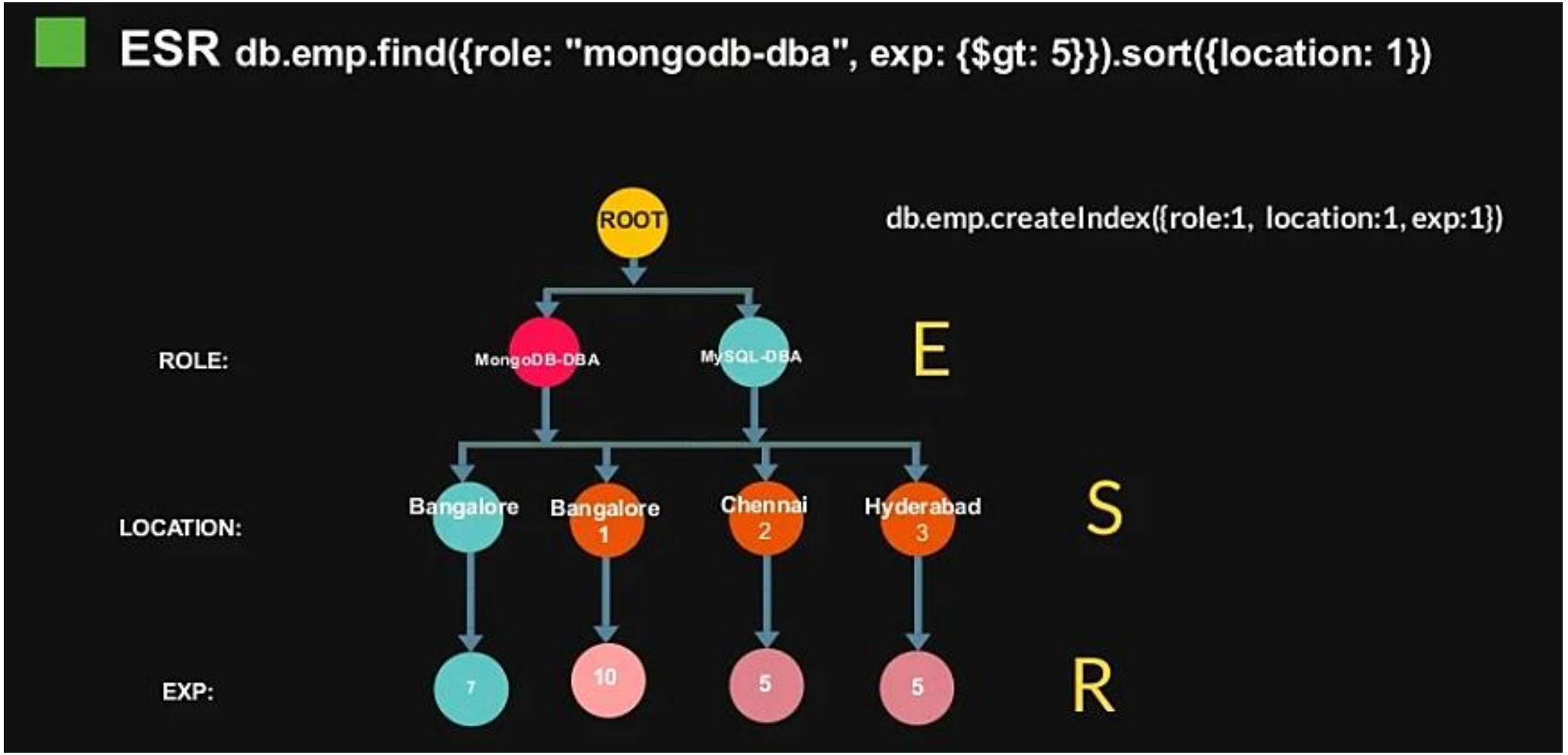


- If we put equality key first, we will limit the of data we looking
- Avoid blocking/in-memory sorting

Follow ESR Rule in Compound Indexes



Follow ESR Rule in Compound Indexes



B-Tree & Prefix Compression: Query Performance & Disk usage

- In B-Tree indexes, Low Cardinality value actually harm performance
- In Low Cardinality value preference to use Partial Index

B-Tree & Prefix Compression: Query Performance & Disk usage

Cardinality

- Cardinality is defined to be number of unique elements present in a set. The lower the cardinality, the more duplicated elements.

- So if a set has 5 elements made of Boolean values, then the cardinality of the set is going to be two. So, all sets made of Booleans will have a max cardinality of two and a min cardinality of one.

B-Tree & Prefix Compression: Query Performance & Disk usage

How cardinality impacts indexing

- If a Boolean field is indexed, there is not much the index will improve in terms of performance.
  - Have just Booleans with a 50/50 split. The index will allow you to skip 50% of the documents, but still be a sequential scan of the rest 50%.
  - If there is a 80/20 split between true and false, then the index is pretty much useless when querying over the true part because you will have to do a sequential scan of 80% of the documents (But the queries looking for false will benefit from the index).
  - This applies to any field with a low cardinality, if a field has an enum of five values and thousands of documents in each category, A similar effect can be observed.

- Indexes must be built carefully in conditions like these. One more side effect of having an index on such fields is that it impacts writes as well.

B-Tree & Prefix Compression: Query Performance & Disk usage

Partial Index

- 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.

- For example, the following operation creates a compound index that indexes only the documents with a rating field greater than 5.

```
db.restaurants.createIndex(
      { cuisine: 1, name: 1 },
      { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

Use Covered Queries When Possible

Covered query

- A covered query is a query that can be satisfied entirely using an index and does not have to examine any documents. An index covers a query when all of the following apply:
  - all the fields in the query are part of an index, and
  - all the fields returned in the results are in the same index.
  - no fields in the query are equal to null (i.e. {"field" : null} or {"field" : {$eq : null}}).

- For example, a collection inventory has the following index on the type and item fields:

```
db.inventory.createIndex( { type: 1, item: 1 } )
```

- This index will cover the following operation which queries on the type and item fields and returns only the item field:

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

Use Covered Queries When Possible

Covered query

- For the specified index to cover the query, the projection document must explicitly *specify _id: 0* to exclude the _*id* field from the result since the index does not include the _*id* field.

- For example, consider a collection userdata with documents of the following form:

```
{ _id: 1, user: { login: "tester" } }
```

- The collection has the following index:

```
{ "user.login": 1 }
```

- The { "user.login": 1 } index will cover the query below:

```
db.userdata.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
```

Key Consideration

- Index create in foreground will do collection level locking
- Index creation in the background helps to overcome the locking bottleneck but decrease the efficiency of index traversal
- Recommend the developer to write the covered query. The kind of query will be entirely satisfied with an index. So zero documents need to be inspected to satisfy the query, and this makes the query run lot faster. All the projection keys need to be indexed
- Use Index to sort the result and avoid blocking sort
- Remove Duplicate and unused index, it also improve the disk throughput and memory optimization