



# Multicore Computing

## Lecture25 – BigData Platforms 2



남 범 석  
bnam@skku.edu



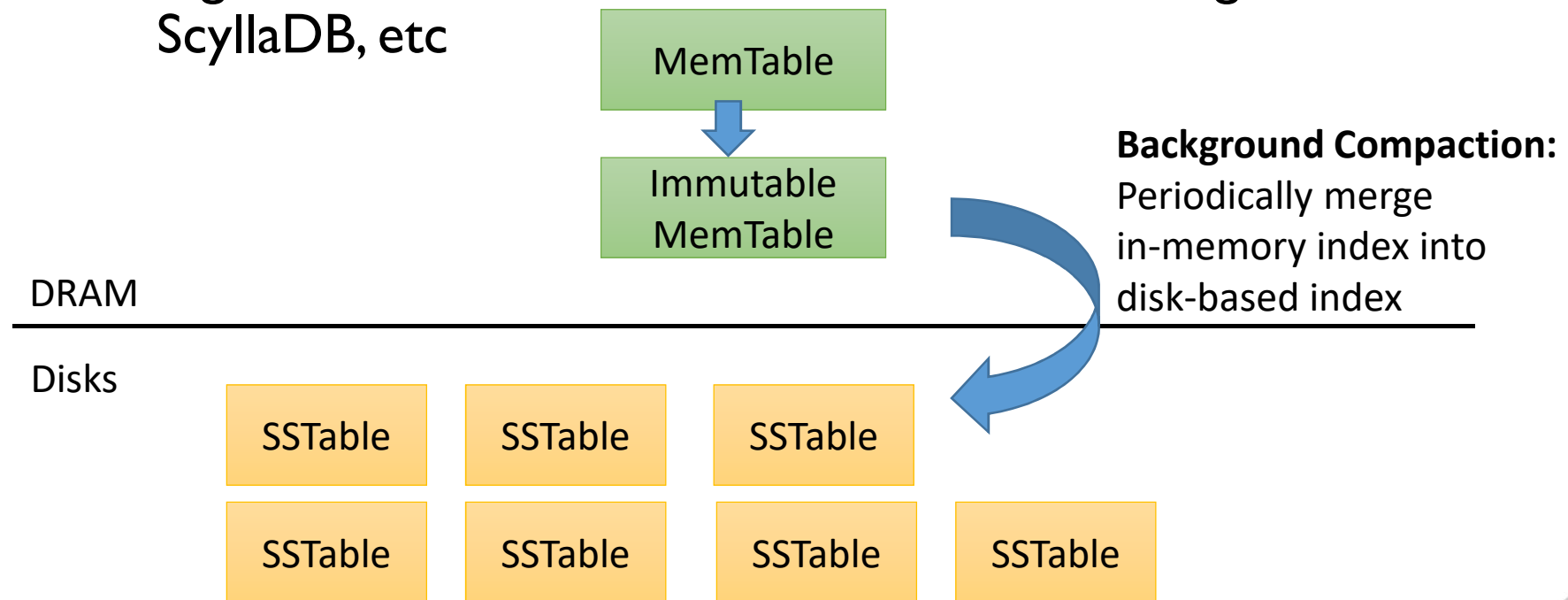


# Log-Structured Merge Tree



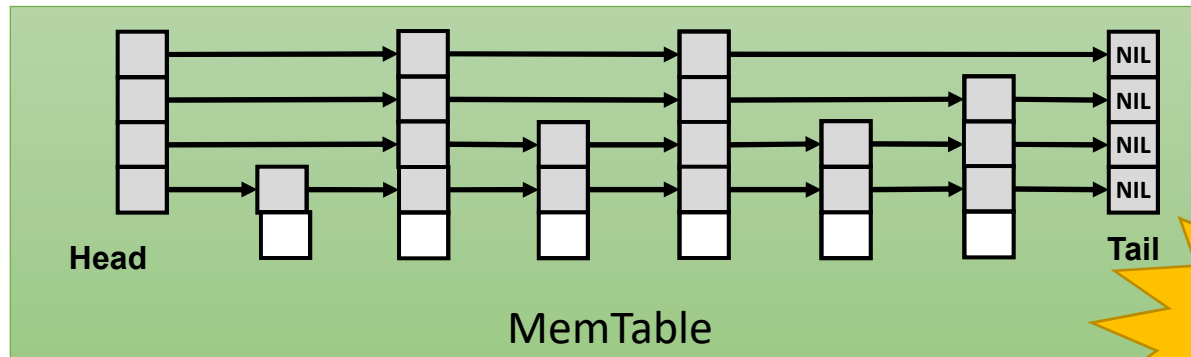
# Background: Log-Structured Merge-Tree

- Designed for Write-intensive workloads
  - Fast insertion
  - Moderate search performance
- Widely used in various key-value stores
  - BigTable, Hbase, Cassandra, RocksDB, WiredTiger, InfluxDB, ScyllaDB, etc



# Background: Log-Structured Merge-Tree

INSERTION

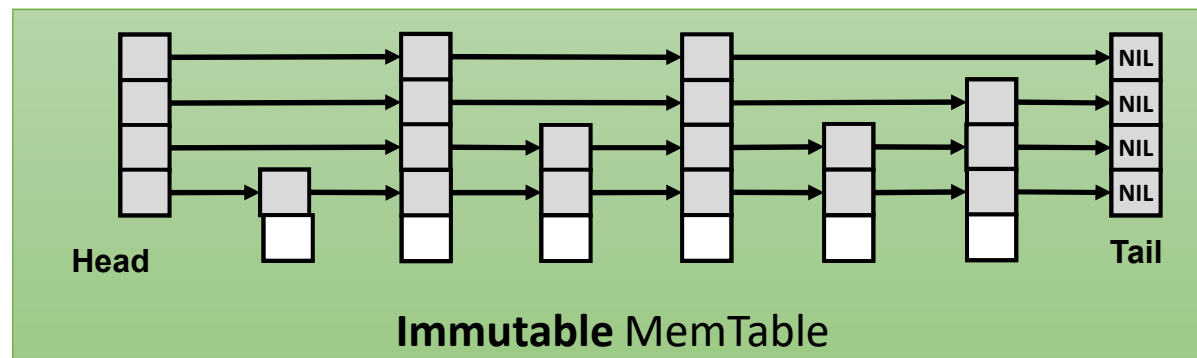


DRAM

Disks



# Background: Log-Structured Merge-Tree



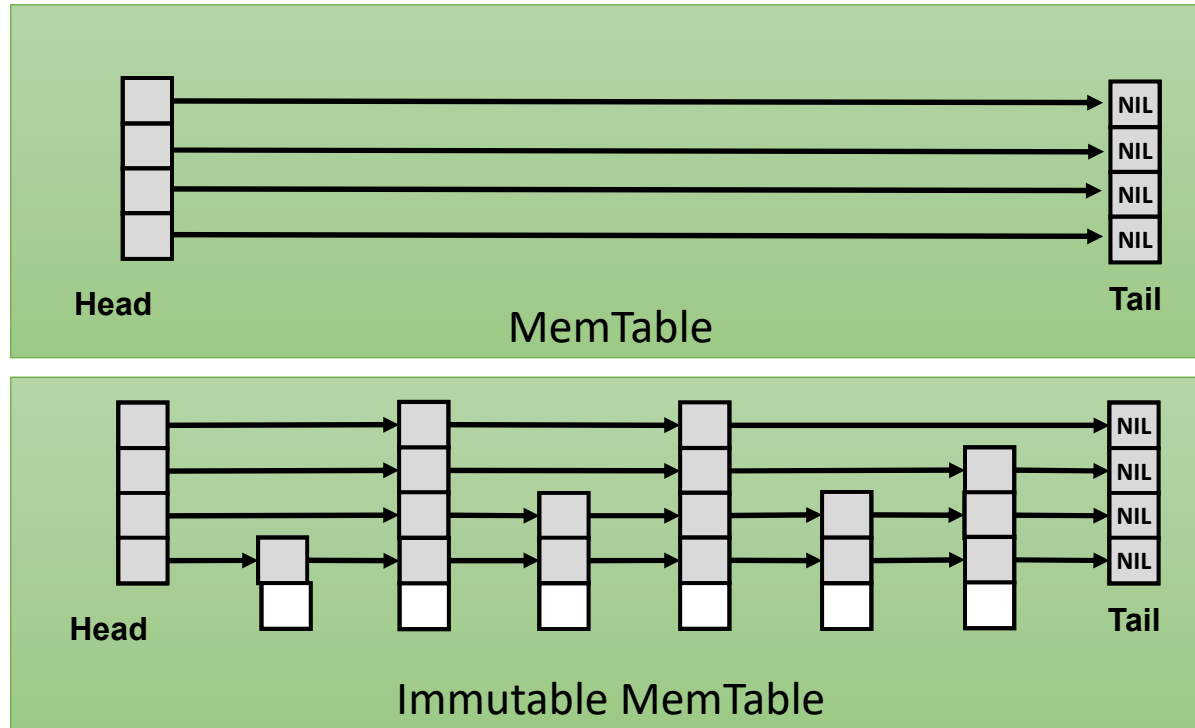
DRAM

Disks

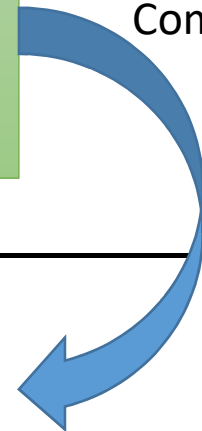


# Background: Log-Structured Merge-Tree

INSERTION



Background  
Compaction



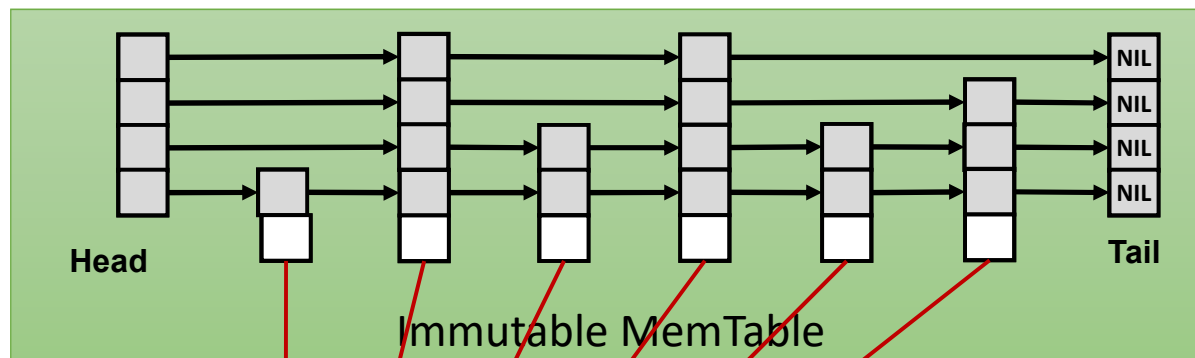
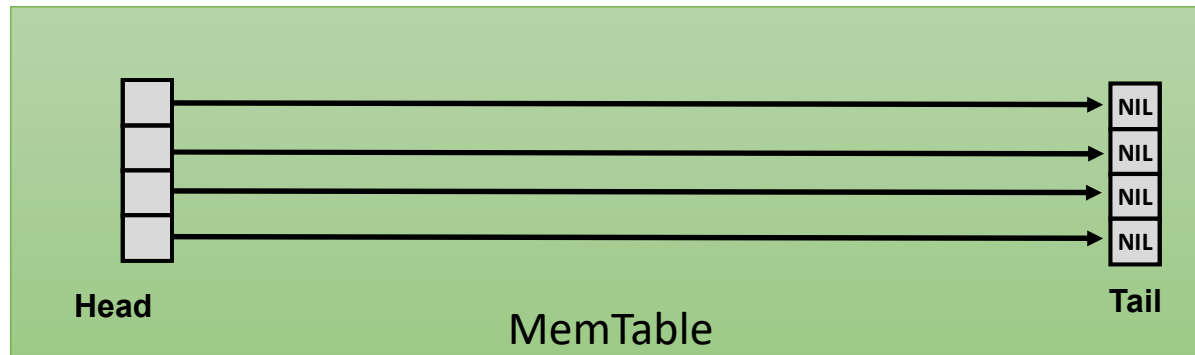
DRAM

Disks



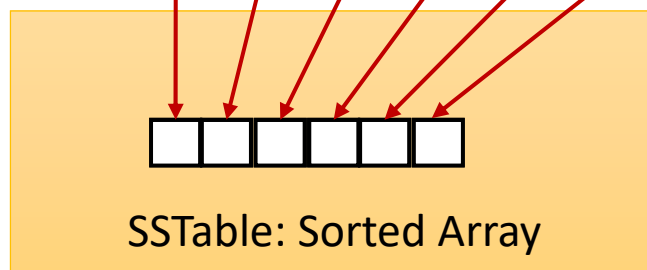
# Background: Log-Structured Merge-Tree

INSERTION



DRAM

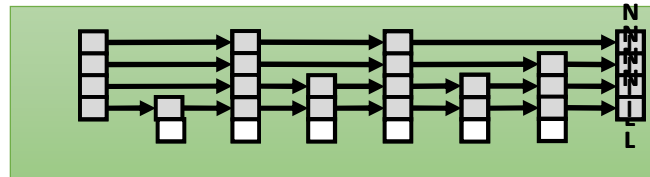
Disks



Background  
Compaction  
(Transform to Array)



# Background: Log-Structured Merge-Tree



MemTable

DRAM

Disks

10, 15, 20, 25  
SSTable

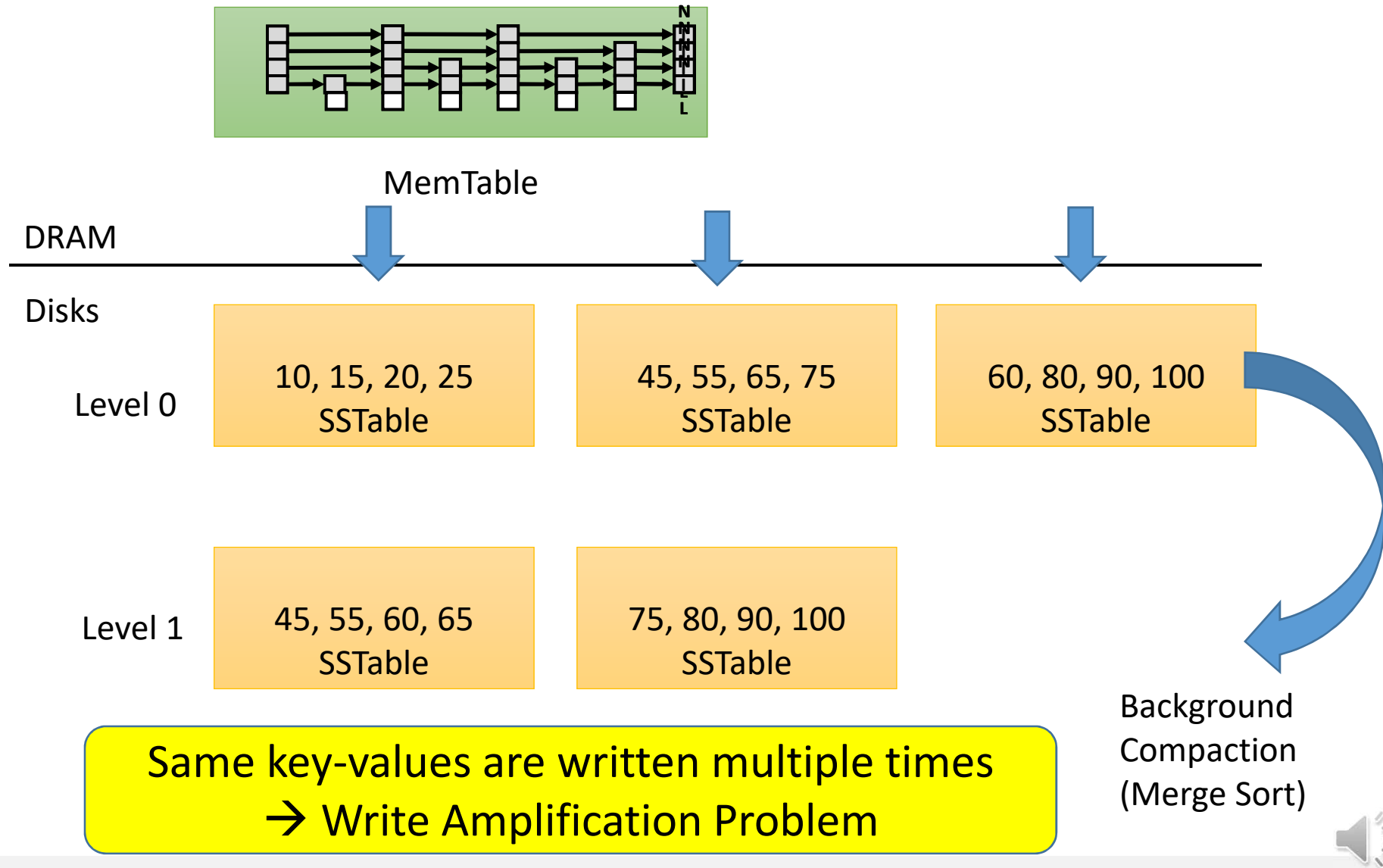
45, 55, 65, 75  
SSTable

60, 80, 90, 100  
SSTable

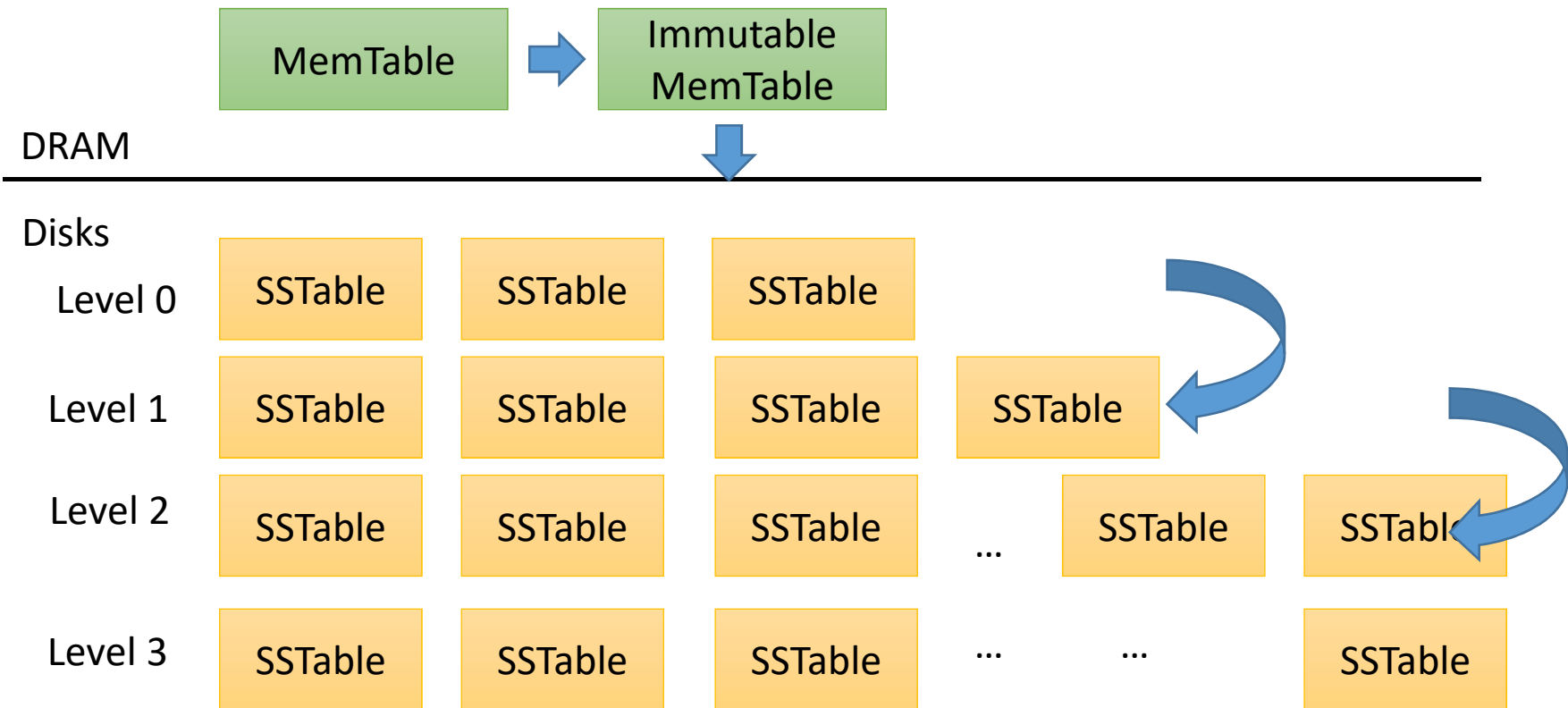




# Background: Log-Structured Merge-Tree



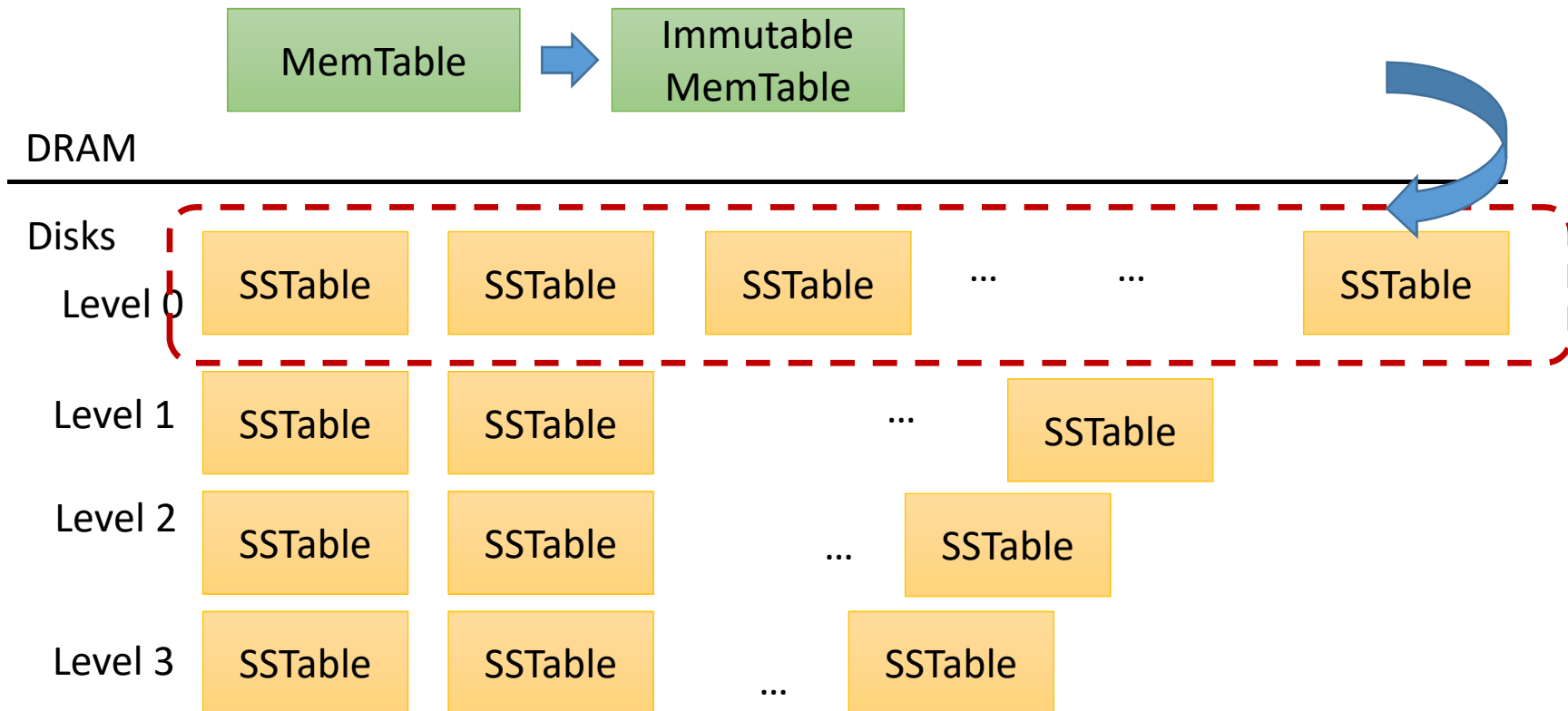
# Background: Log-Structured Merge-Tree



This is what LSM-Tree should be



# Background: Log-Structured Merge-Tree



But, In Reality

- [1] Memory writes are very fast
- [2] Merge sort on disks is slow



# Background: Log-Structured Merge-Tree



No More Writes to MemTable !! → Write Stall Problem

MemTable



Immutable  
MemTable

DRAM

Disks

Level 0

SSTable

SSTable

SSTable

...

...

SSTable

Level 1

SSTable

SSTable

...

SSTable

Level 2

SSTable

SSTable

...

SSTable

Level 3

SSTable

SSTable

...

SSTable

But, In Reality

- [1] Memory writes are very fast
- [2] Merge sort on disks is slow





# MongoDB



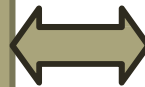
# RDB ACID to NoSQL BASE

**A**tomicity

**C**onsistency

**I**solation

**D**urability



**B**asically

**A**vailable (CP)

**S**oft-state

(State of system may change over time)

**E**ventually  
consistent

(Asynchronous propagation)





## Key Value Storage Systems

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are **partitioned** across multiple machines and
- Queries are routed by the system to appropriate machine
- Records are also **replicated** across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are **consistent**



# What is MongoDB?

- Developed by 10gen
  - Founded in 2007
- A document-oriented, NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - In practice, this means you can store hashes with any keys and values that you choose
      - Keys are stored as strings
      - Document Identifiers (`_id`) will be created for each document, field name reserved by system
    - Application tracks the schema and mapping
    - Uses BSON format
      - Based on JSON – B stands for Binary







# Functionality of MongoDB

- Dynamic schema
  - No DDL
- Document-based database
- Secondary indexes
- Query language via an API
- Atomic writes and fully-consistent reads
  - If system configured that way
- Master-slave replication with automated failover (replica sets)
- Built-in horizontal scaling via automated range-based partitioning of data (sharding)
- No joins nor transactions





## Why use MongoDB?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
  - No ERD diagram
- Not well suited for heavy and complex transactions systems



# Data Model

- Stores data in form of BSON (Binary JSON) *documents*

{

name: "travis",

salary: 30000,

designation: "Computer Scientist",

teams: [ "front-end", "database" ]

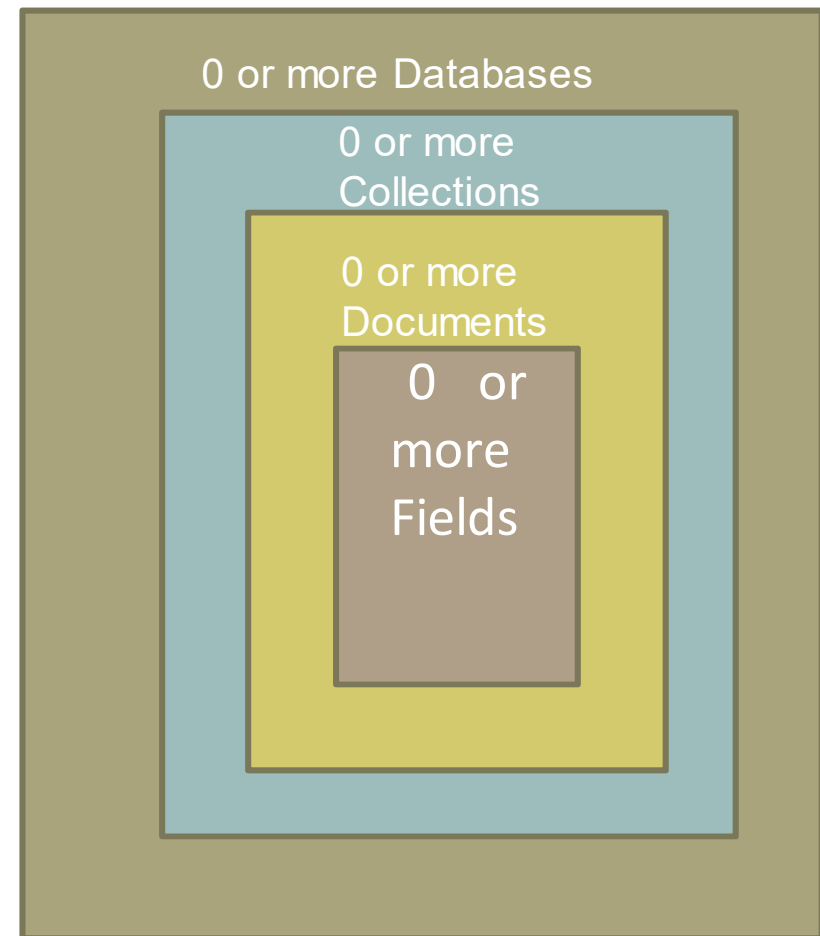
}

- Group of related *documents* with a shared common index is a *collection*



# MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more 'databases'
- A database may have zero or more 'collections'.
- A collection may have zero or more 'documents'.
- A document may have one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.



# RDB Concepts to NO SQL

| RDBMS       |   | MongoDB           |
|-------------|---|-------------------|
| Database    | ⇒ | Database          |
| Table, View | ⇒ | Collection        |
| Row         | ⇒ | Document (BSON)   |
| Column      | ⇒ | Field             |
| Index       | ⇒ | Index             |
| Join        | ⇒ | Embedded Document |
| Foreign Key | ⇒ | Reference         |
| Partition   | ⇒ | Shard             |

Collection is not strict about what it Stores

Schema-less

Hierarchy is evident in the design

Embedded Document ?





# MongoDB Processes and configuration

- Mongod – Database instance
- Mongos - Sharding processes
  - Analogous to a database router
  - Processes all requests
  - Decides how many and which *mongods* should receive the query
  - *Mongos* collates the results, and sends it back to the client.
- Mongo – an interactive shell ( a client)
  - Fully functional JavaScript environment for use with a MongoDB
- You can have one *mongos* for the whole system no matter how many *mongods* you have
- OR you can have one local *mongos* for every client if you wanted to minimize network latency.



# Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection could have different data
  - Addresses NULL data fields

```
{name: "will",  
  eyes: "blue",  
  birthplace: "NY",  
  aliases: ["bill", "la ciacco"],  
  loc: [32.7, 63.4],  
  boss: "ben"}
```

```
{name: "jeff",  
  eyes: "blue",  
  loc: [40.7, 73.4],  
  boss: "ben"}
```

```
{name: "brendan",  
  aliases: ["el diablo"]}
```

```
{name: "ben",  
  hat: "yes"}
```

```
{name: "matt",  
  pizza: "DiGiorno",  
  height: 72,  
  loc: [44.6, 71.3]}
```

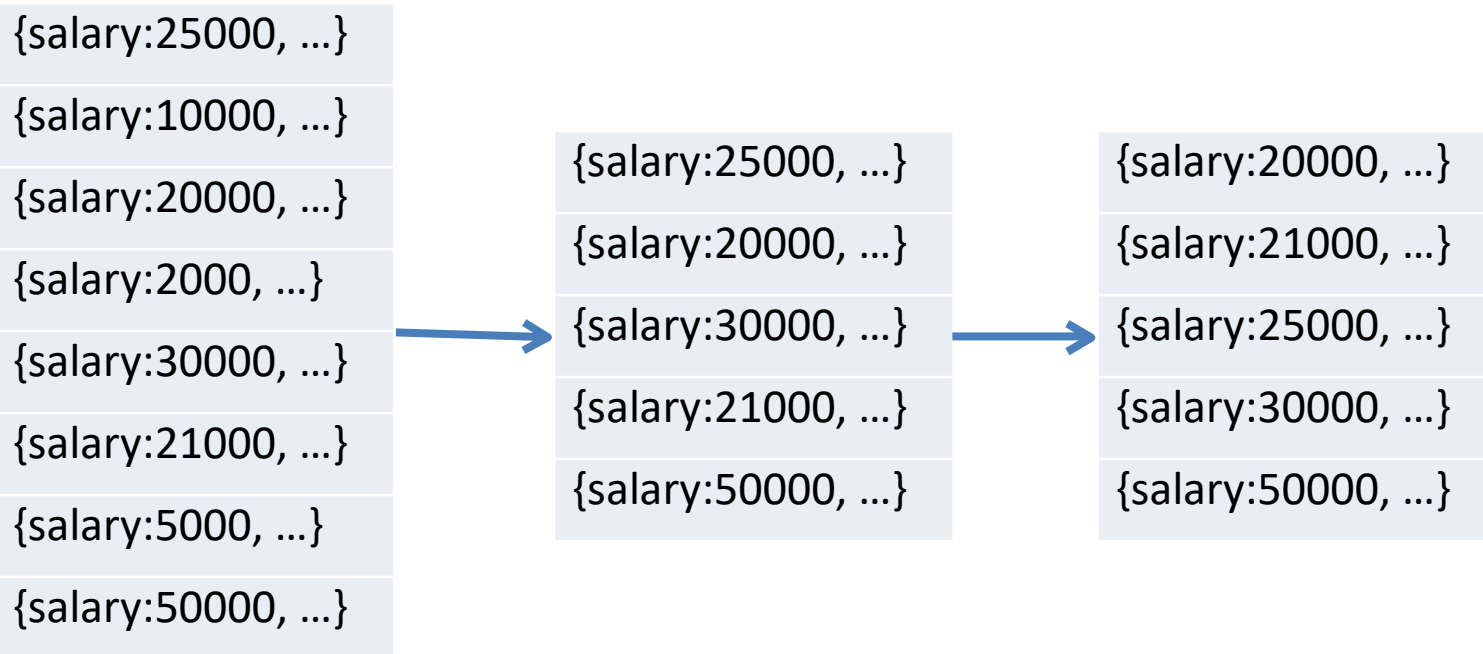


# Query

- Query all employee names with salary greater than 18000 sorted in ascending order

```
db.users.find({salary:{$gt:18000}, {name:1}}).sort({salary:1})
```

Collection                      Condition                      Projection                      Modifier





# Query Operators

| Name        | Description   |
|-------------|---|
| \$eq        | Matches value that are equal to a specified value                   |
| \$gt, \$gte | Matches values that are greater than (or equal to a specified value |
| \$lt, \$lte | Matches values less than or ( equal to ) a specified value          |
| \$ne        | Matches values that are not equal to a specified value              |
| \$in        | Matches any of the values specified in an array                     |
| \$nin       | Matches none of the values specified in an array                    |
| \$or        | Joins query clauses with a logical OR returns all                   |
| \$and       | Join query clauses with a logical AND                               |
| \$not       | Inverts the effect of a query expression                            |
| \$nor       | Join query clauses with a logical NOR                               |
| \$exists    | Matches documents that have a specified field                       |





## Insert

- Insert a row entry for new employee Sally

```
db.users.insert({  
    name: "sally",  
    salary: 15000,  
    designation: "MTS",  
    teams: [ "cluster-management" ]  
})
```



## Update

- All employees with salary greater than 18000 get a designation of Executive

|                        |   |
|------------------------|---|
| <i>Update Criteria</i> | <code>db.users.update(</code>                   |
| <i>Update Action</i>   | <code>{salary:{\$gt:18000}},</code>             |
| <i>Update Option</i>   | <code>{\$set: {designation: "Manager"}},</code> |
|                        | <code>{multi: true}</code>                      |
|                        | <code>)</code>                                  |

- Multi option allows multiple document update





## Delete

- Remove all employees who earn less than 10000

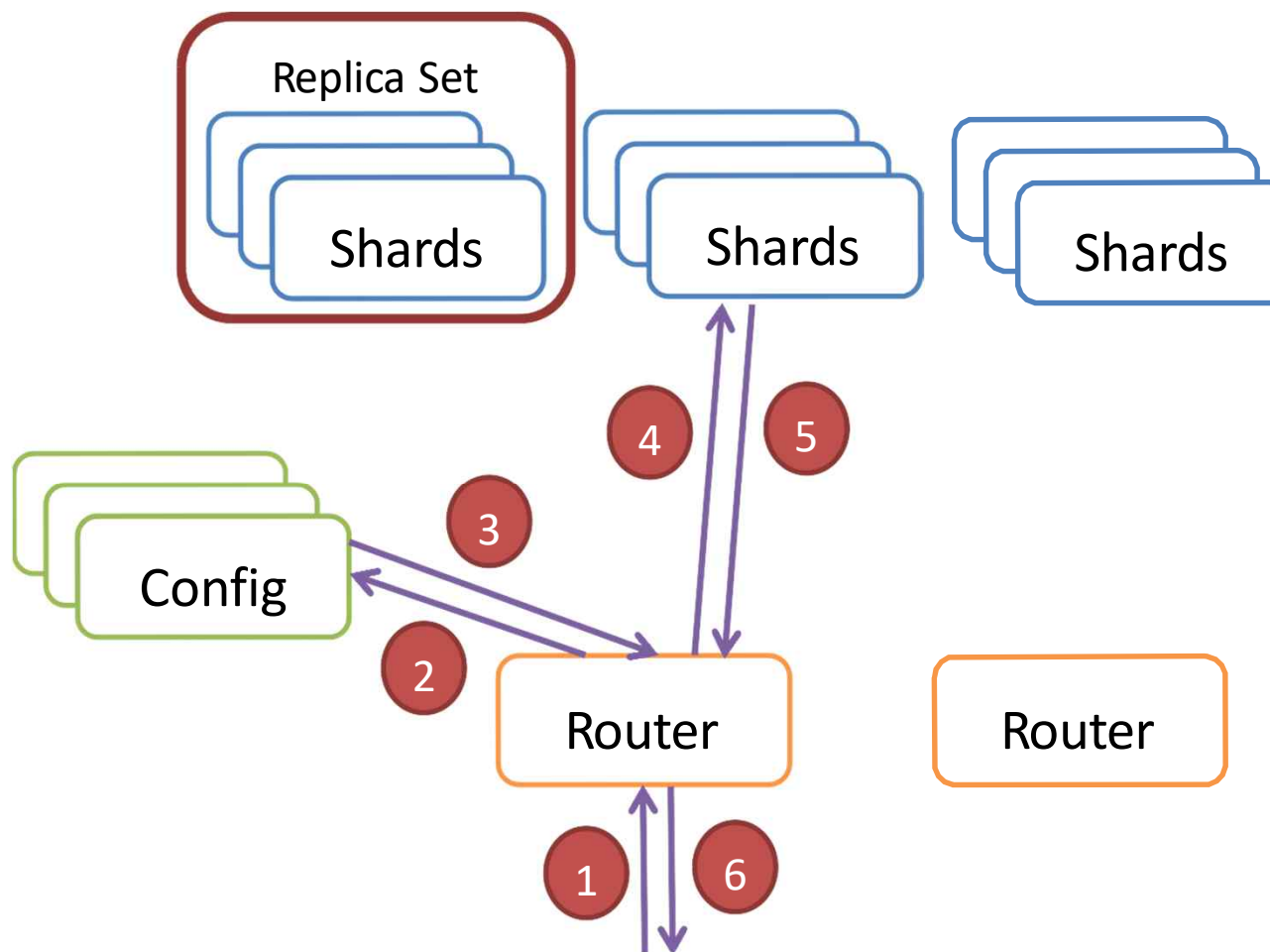
*Remove Criteria*

```
db.users.remove(  
  {salary:{$lt:10000}},  
)
```

- Can accept a flag to limit the number of document removal



# Typical MongoDB Deployment



- **Shards:** mongod servers store the data
- Multiple shard servers form a *replica set*
- Replica set maintain same replica of data
- **Routers:** mongos interfaces with clients and routers operations to appropriate shards
- **Config:** Stores collection level metadata.





## Read Preference

- Determine where to route read operation
- Default is primary. Possible options are secondary, primary-preferred, etc.
- Helps reduce latency, improve throughput
- Reads from secondary may fetch stale data





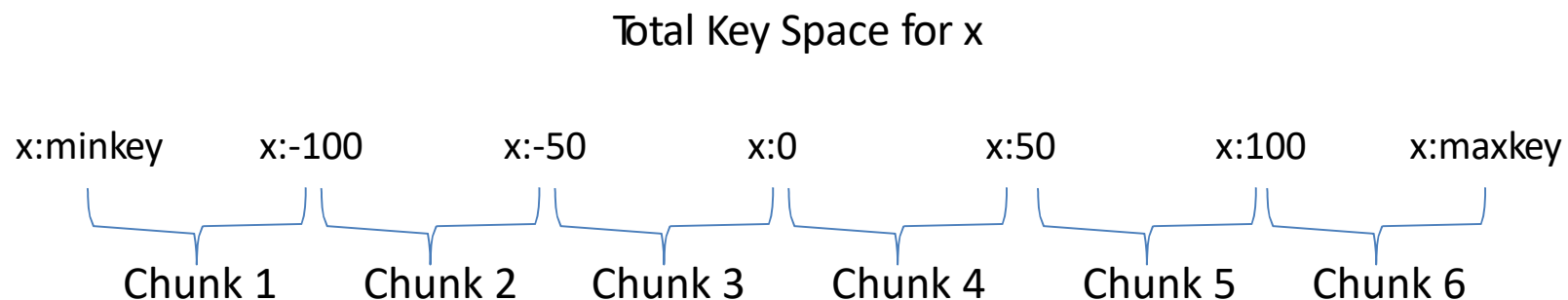
## Write Concern

- Determines the guarantee that MongoDB provides on the success of a write operation
- Default is *acknowledged*. Others are unacknowledged, replica-acknowledged, etc
- For the default case, primary replicas acknowledge the success of a write operation
- Weaker write concern implies faster write time



# Partition

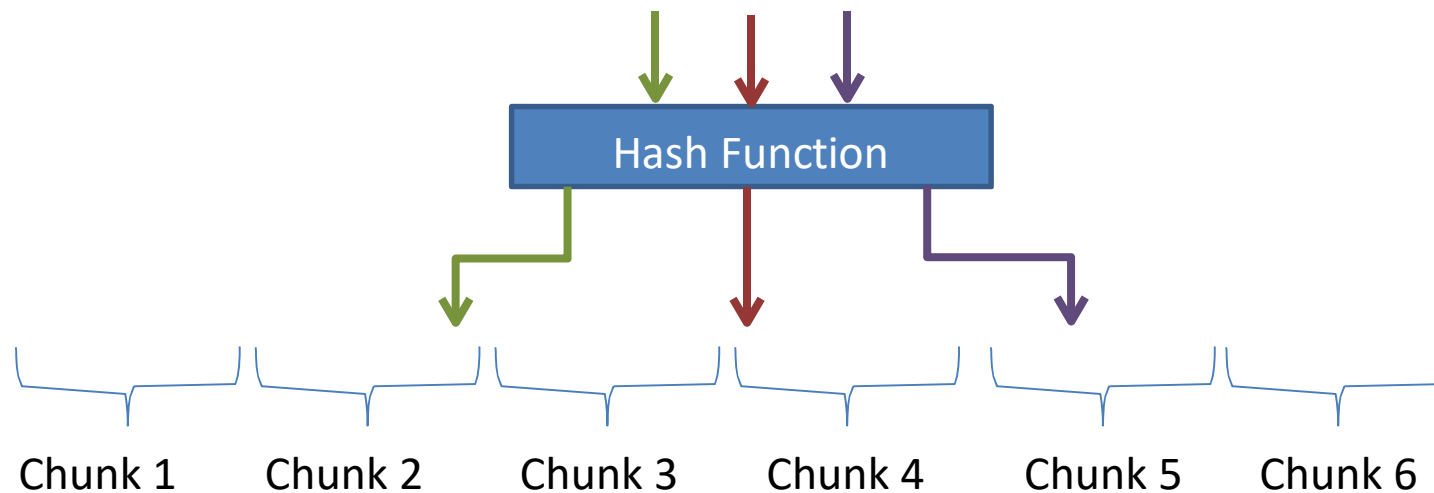
- Shard Key: Single or compound field in schema used for data partitioning
- Partitions are called *chunks*. Two strategies:
  - [1] **Range based**: Shard Key Values are partitioned into ranges





## Partition

- [2] Hash based: Hash of shard key values are partitioned into ranges



- Range Queries are efficient for the first strategy
- Hash Scheme leads to better data balancing



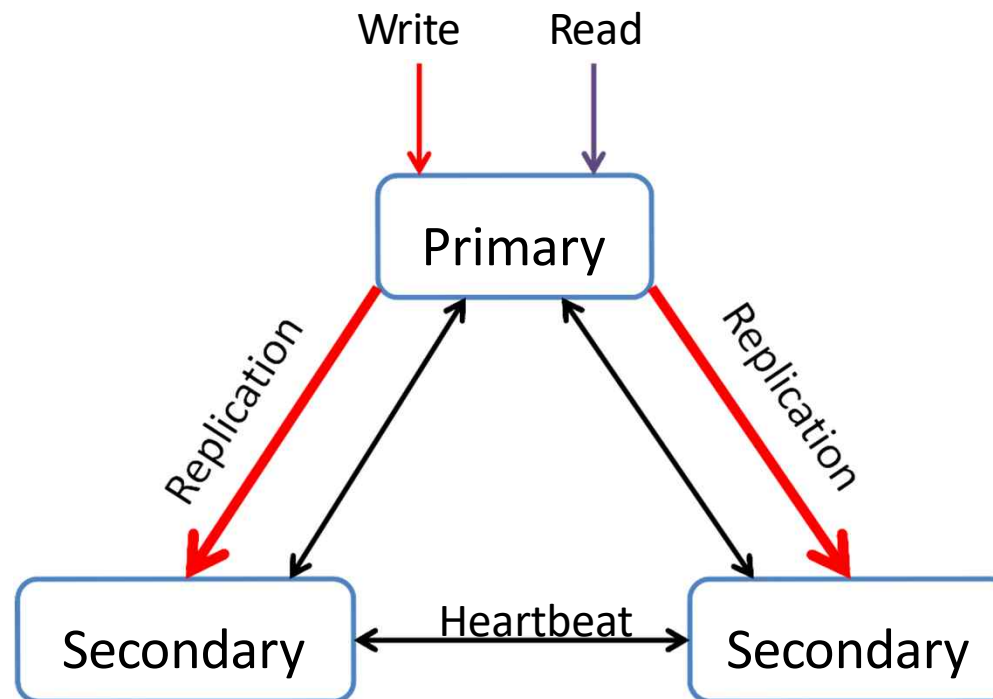


## Balancing

- Splitting: Background process which splits when a chunk grows beyond a threshold
- Balancing: Migrates chunks among shards if there is an uneven distribution



# Replication





# Consistency

- Strongly Consistent: Read Preference is Master
- Eventually Consistent: Read Preference is Slave
- CAP Theorem: Under partition, MongoDB becomes write unavailable thereby ensuring consistency

