COMP5338 – Advanced Data Models

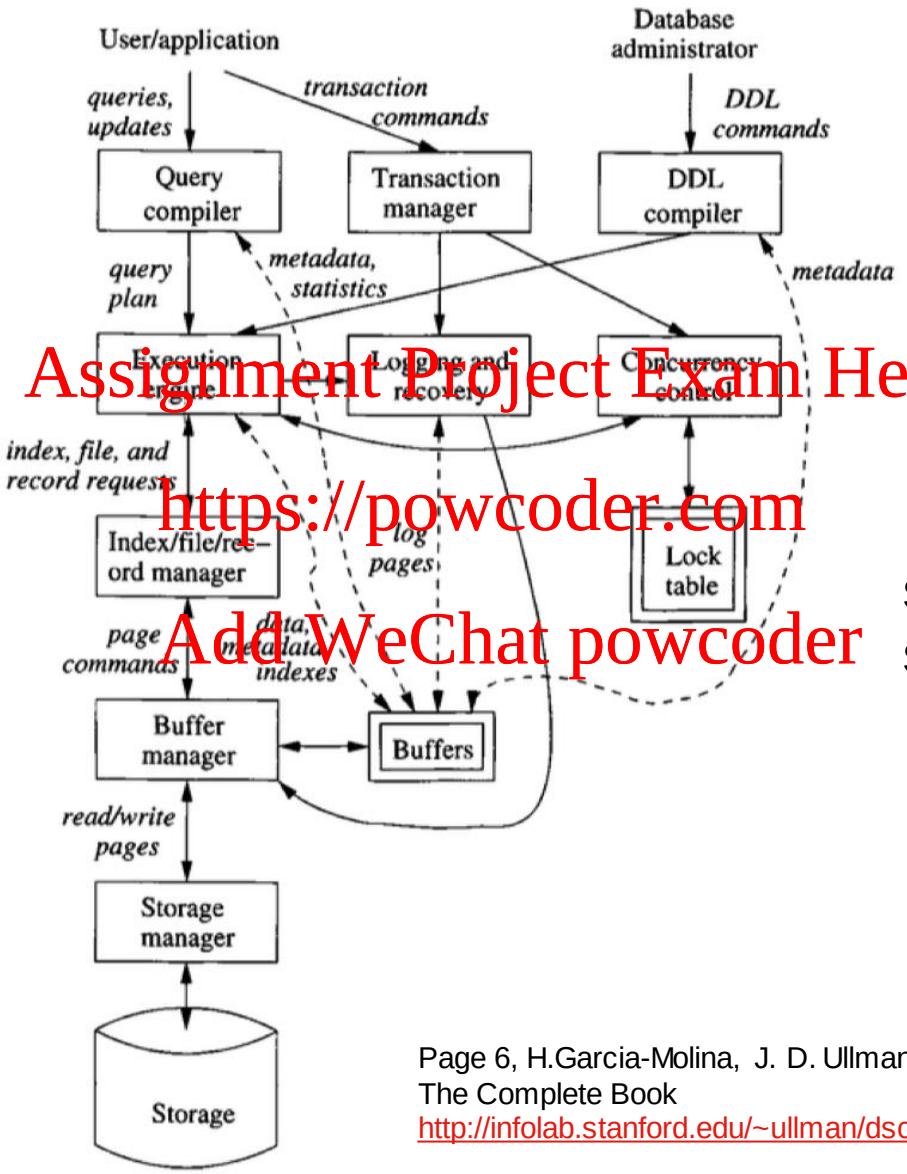
Week 4: MongoDB – Advanced Features

Dr. Ying Zhou
School of Information Technologies

Outline

- **Indexing**
- **Replication**
- **Sharding**

Review: DBMS Components



Disk based storage system

Page 6, H.Garcia-Molina, J. D. Ullman, J. Wildom, Databsae Systems The Complete Book
http://infolab.stanford.edu/~ullman/dscb.html

Storage Engine

- Storage engine is responsible for managing how data is store in memory and disk
- MongoDB supports multiple storage engines
  - WiredTiger is the default one since version 3.2
- Some prominent features of WiredTiger
  - Document level concurrency
  - MultiVersion Concurrency Control (MVCC)
    - Snapshots are provided at the start of operation
    - Snapshots are written to disk (creating checkpoints) at intervals of 60 seconds or 2GB of journal data
  - Journal
    - Write-ahead transaction log
  - Compression

The primitive operations of query

- Read query
  - Load the element of interest from disk to main-memory buffer(s) if it is not already there
  - Read the content to client's address space
- Write query
  - The new value is created in the client's address space
  - It is copied to the appropriate buffers representing the database in the memory
  - The buffer content is flushed to the disk
- Both operations involve data movement between disk and memory and between memory spaces
- Typically disk access is the predominant performance cost in single node settings. Network communication contributes to the cost in cluster setting
- We want to reduce the amount of disk I/Os in read and write queries

Typical Solutions to minimize Disk I/O

- Queries involve reading data from the database
  - Minimize the amount of data need to be moved from disk to memory
  - Use index and data distribution information to decide on a query plan
- Queries involve writing data to the database
  - Minimize the amount of disk I/O in the write path
    - Avoid flushing memory content to disk immediately after each write
    - Push non essential write out the of write path, e.g. do those asynchronously
  - To ensure durability, write ahead log/journal/operation log is always necessary
    - Appending to logs are much faster than updating the actual database file
    - The DB system may acknowledge once the data is updated in memory and appended in the WAL
    - Update to replicas can be done asynchronously, e.g. not in the write path

Indexing

- An index on an attribute/field **A** of a table/collection is a data structure that makes it efficient to find those rows(document) that have a required value for attribute/field **A**.
- An index consists of records (called index entries) each of which has a value for the attribute(s) eg of the form

attr. value	Pointer to data record

- Index files are typically much smaller than the original file


```
db.revisions.stats({scale: 1024});
```

0.005 sec.

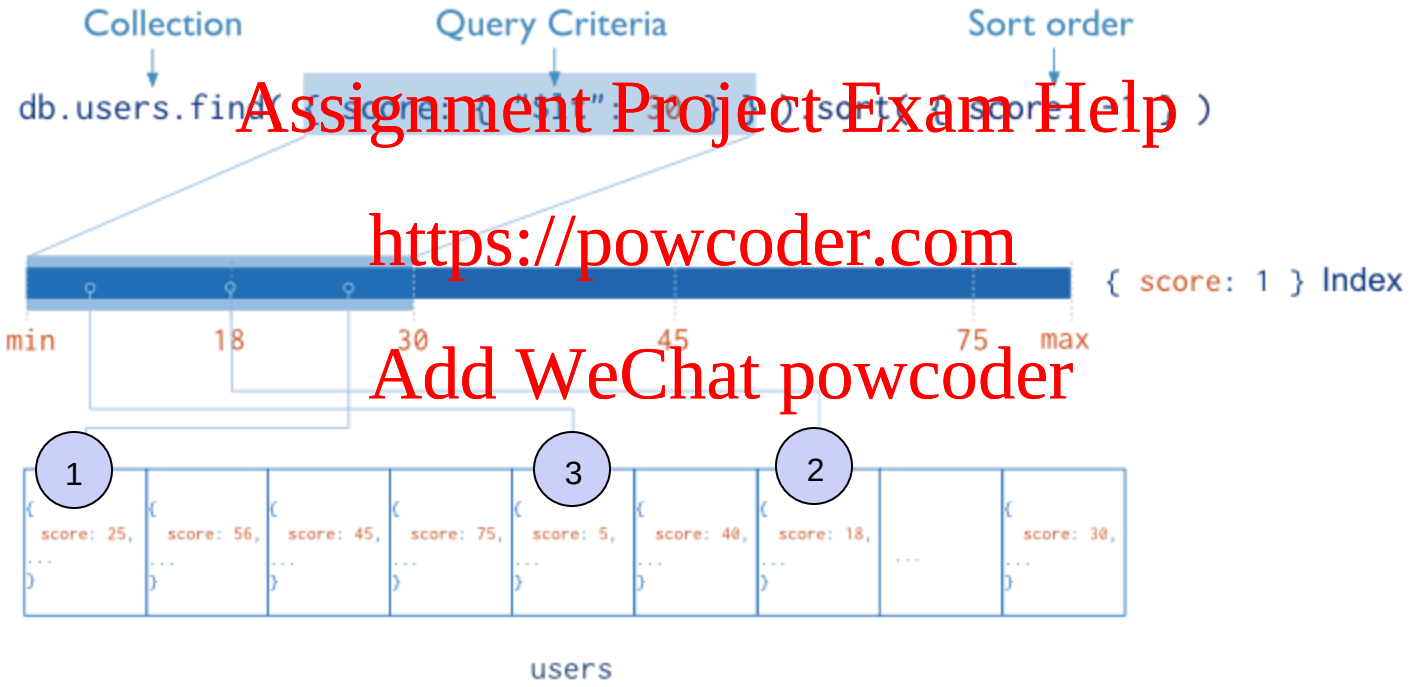
Key	Value	Type
(1)	{ 11 fields }	Object
ns	wikipedia.revisions	String
count	623	Int32
size	188	Int32
avgObjSize	309	Int32
storageSize	204	Int32
capped	false	Boolean
wiredTiger	{ 13 fields }	Object
nindexes	1	Int32
totalIndexSize	28	Int32
indexSizes	{ 1 field }	Object
_id_	28	Int32
ok	1.0	Double

MongoDB Basic Indexes

- The **_id** index
  - **_id** field is automatically indexed for all collections
  - The **_id** index enforces uniqueness for its keys
- Indexing on other fields
  - Index can be created on any other field or combination of fields
    - `db.<collectionName>.createIndex({<fieldName>:1});`
    - **fieldName** can be a simple field, array field or field of an embedded document (using dot notation)
      - db.blog.createIndex({author:1})
      - db.blog.createIndex({tags:1})
      - db.blog.createIndex({"comments.author":1})
    - the number specifies the direction of the index (1: ascending; -1: descending)
  - Additional properties can be specified for an index
    - **Sparseness**, **uniqueness**, **background**, ..
- Most MongoDB indexes are organized as **B-Tree** structure

http://www.mongodb.org/display/DOCS/Indexes

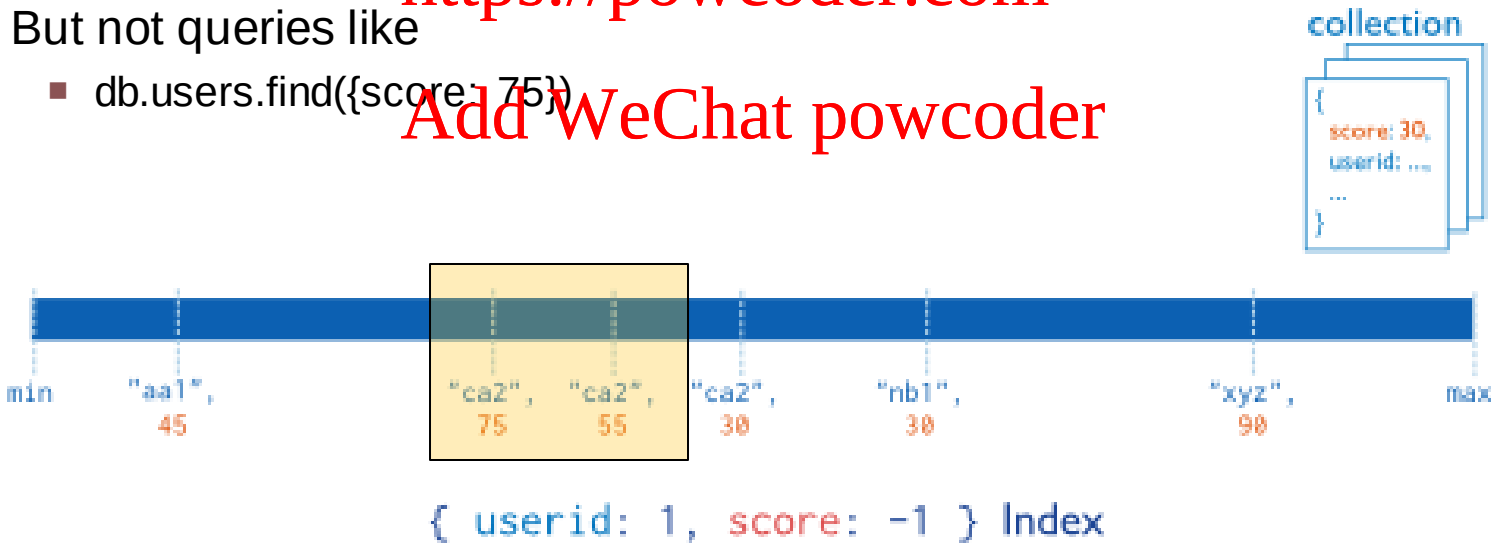
Single field Index



https://docs.mongodb.com/manual/core/index-single/

Compound Index

- Compound Index is a single index structure that holds references to multiple fields within a collection
- The order of field in a compound index is very important
  - The indexes are sorted by the value of the first field, then second, third…
  - It supports queries like
    - db.users.find({userid: "ca2", score: {$gt:30} })
    - db.users.find({userid: "ca2"})
  - But not queries like
    - db.users.find({score: 75})



https://docs.mongodb.com/manual/core/index-compound/

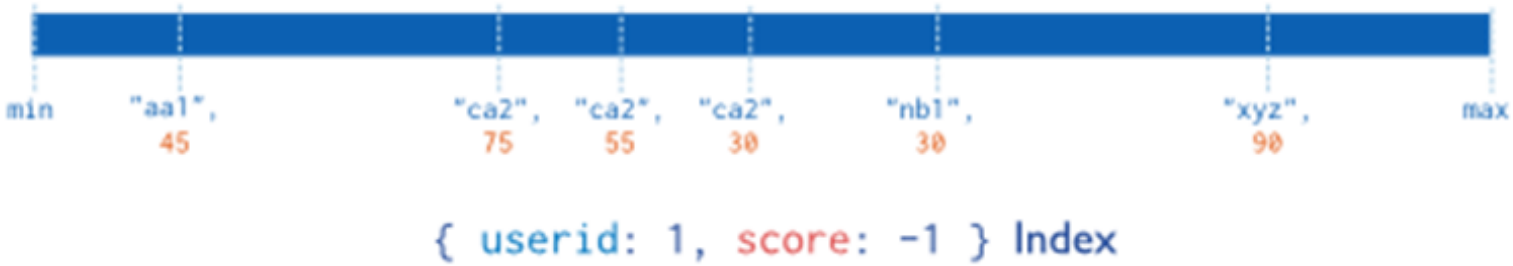
Use Index to Sort (single field)

- Sort operation may obtain the order from index or sort the result in memory
- Index can be traversed in either direction
- Sort with a single field index
  - For single field index, sorting by that field can always use the index regardless of the sort direction
  - E.g. `db.records.createIndex( { a: 1 } )` supports both
    - `db.records.find().sort({a:1})` and
    - `db.records.find().sort({a: -1})`

https://docs.mongodb.com/manual/tutorial/sort-results-with-indexes/

Use Index to Sort (multiple fields)

- Sort on multiple fields
  - Compound index may be used on sorting multiple fields.
  - There are constrains on fields and direction
    - Sort key should have the same order as they appear in the index
    - All field sort have same sort direction, either going forwards or backwards the index
    - E.g. {userid:1, score:-1} and {userid:-1, score:1} can use the index, but not {userid:1, score:1}



Use Index to Sort (multiple fields)

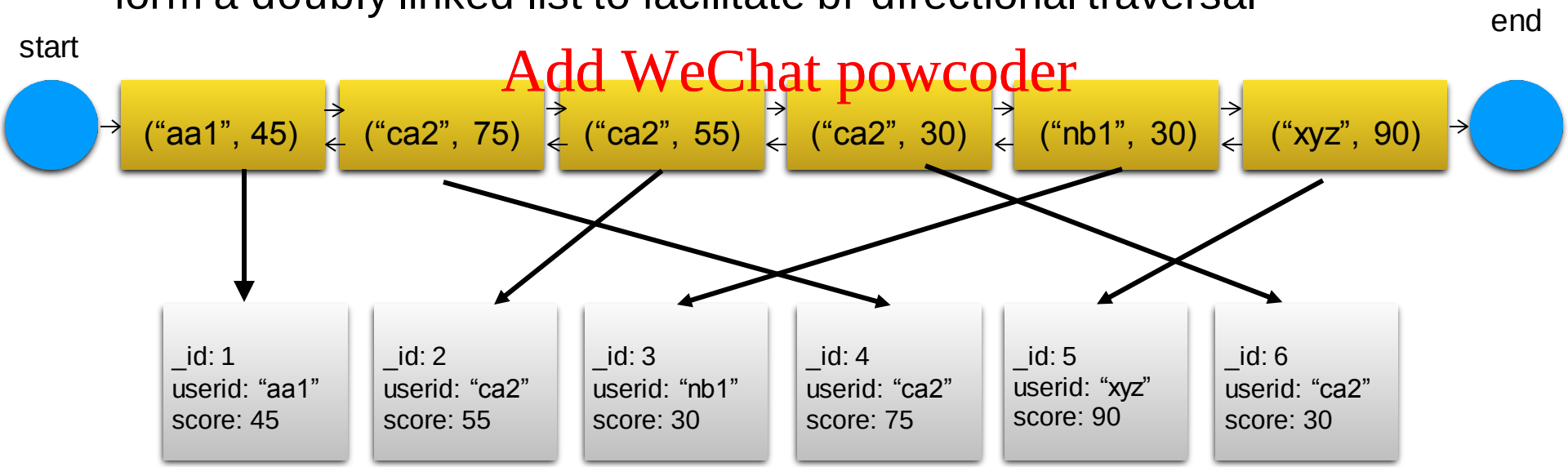
- Sort and Index Prefix
  - If the sort keys correspond to the index keys or an index *prefix*, MongoDB can use the index to sort the query results.
    - E.g. db.data.createIndex({ a:1, b: 1, c: 1, d: 1 })
    - Supported query:
    - db.data.find().sort({ a: -1 })
    - db.data.find().sort({ a: 1, b: 1 })
    - db.data.find({ a: { $gt: 4 } }).sort({ a: 1, b: 1 })
- Sort and Non-prefix Subset of an Index
  - An index can support sort operations on a non-prefix subset of the index key pattern if the query include **equality** conditions on all the prefix keys that precede the sort keys.
    - e.g supported query: db.data.find({ a: 5 }).sort({ b: 1, c: 1 })
    - db.data.find({ a: 5, b: { $lt: 3} }).sort({ b: 1 })

Running Example

- Suppose we have a **users** collection with the following 6 documents stored in the order of `_id` values

Index Entries

- Now we create a compound index on **userid** and **score** fields :
  `db.users.createIndex(userid:1, score:-1)`
- With the current data, the index has six entries because we have 6 unique values for (**`userid, score`**) in the collection
  - New entry will be added each time we insert a document with a (**`userid,score`**) different to the ones already there
- Our index entry structure would look like this: the index entries usually form a doubly linked list to facilitate bi-directional traversal

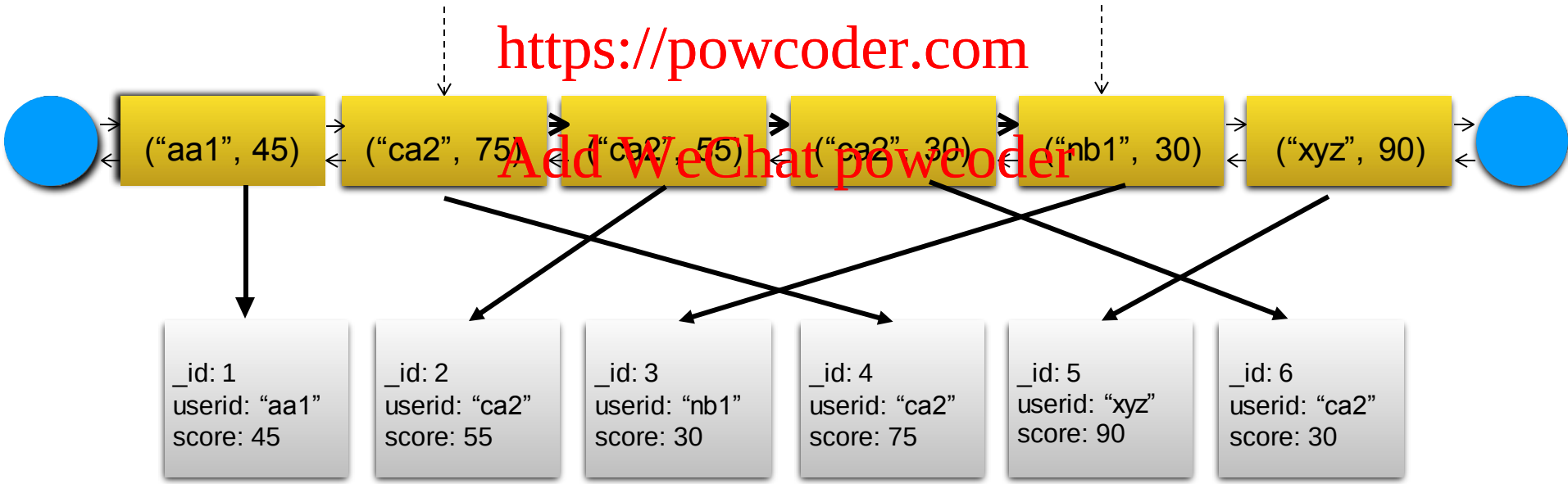


Using index to find documents

- For queries that are able to use index, the first step is to find the boundary entries on the list based on given query condition
- for instance, if we want to look for userid greater than “b” but less than “s”
  - `db.users.find({userid:{$gt: “b”, $lt:“s”}})`
- This query is able to use the compound index and the two bounds are:(“ca1”, 75) and (“mb1”,30) inclusive at both ends

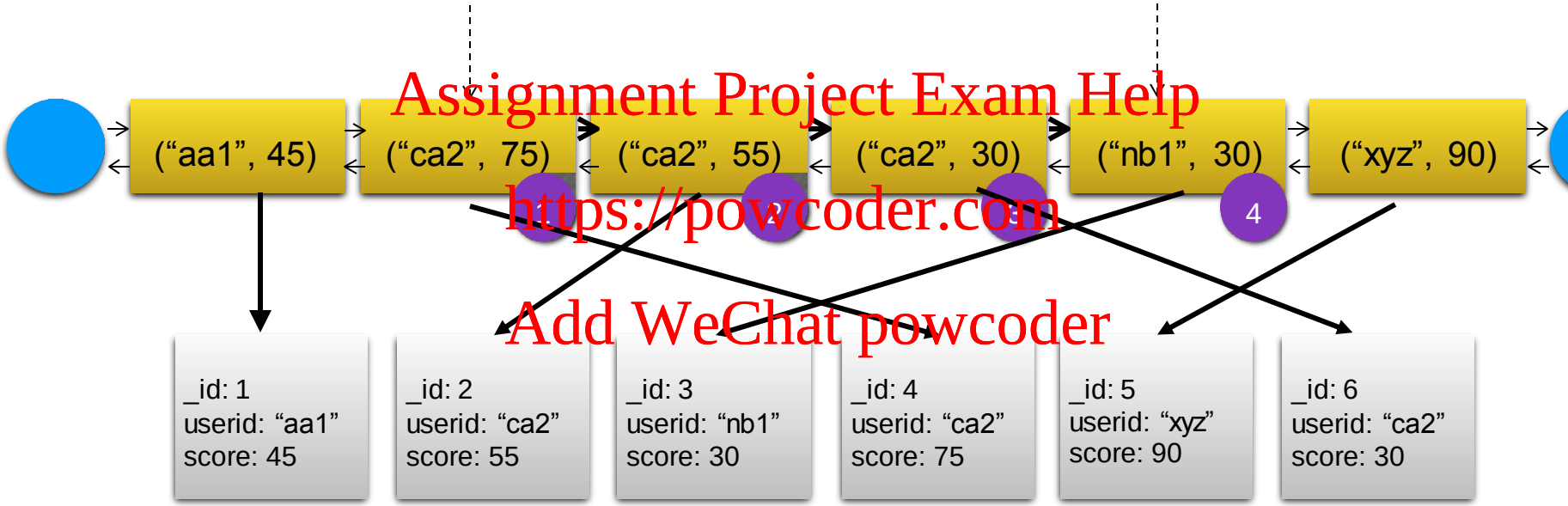
Using index to find documents

- The four documents with **`_id`** equals: 4, 2, 6 and 3 are the result of the above query



Using Index to sort

- If our queries include a sorting criteria
  - `db.users.find({userid:{$gt: “b”, $lt:“s”}}).sort({userid:1, score:-1})`
- as before, the db engine will start from the **lower** bound, following the **forward links** to the **upper** bound and return all documents pointed by the entries
- They are :
  - {_id:4, userid:“ca2”, score: 75} {_id:2,userid:“ca2”, score: 55} {_id:6, userid:“ca2”, score 30} {_id:3,userid:“nb1”, score: 30}
  - The results satisfy the condition and are in correct order

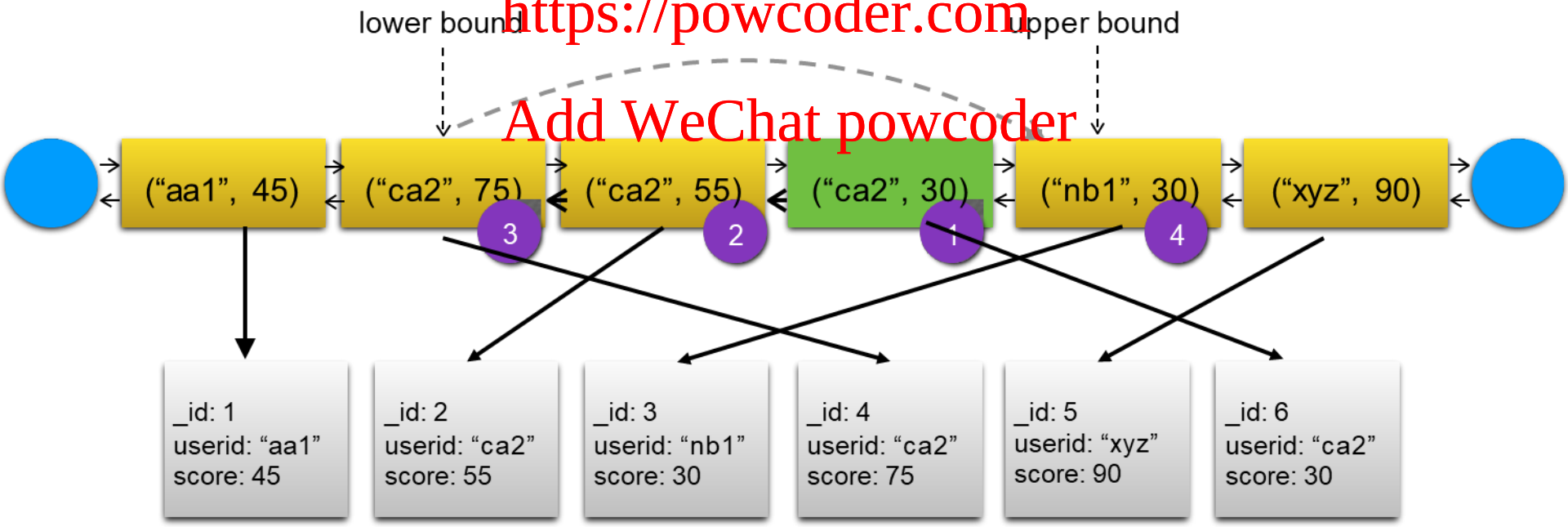


Sorting that cannot use index

- If our query includes yet another sorting criteria
  - db.users.find({userid:{$gt: “b”, $lt:“s”}}).sort({**userid:1, score:1**})
- We can still use the index to find the bounds and the four documents satisfying the query condition, but we are not able to follow a single forward or backward link to get the correct order of the documents

Sorting that cannot use index

- If we want to use the index entry list to obtain the correct, we would start from a mysterious position ("ca2",30), follow the backward links to ("ca2",75), and make a magic jump to the entry ("nb1", 30).
  - complexity involved:
    - how do we find the start point in between lower and upper bound?
    - how do we decide when and where to jump in another direction?
  - The complexity of such algorithm makes it less optimal than a memory sort of the actual documents.

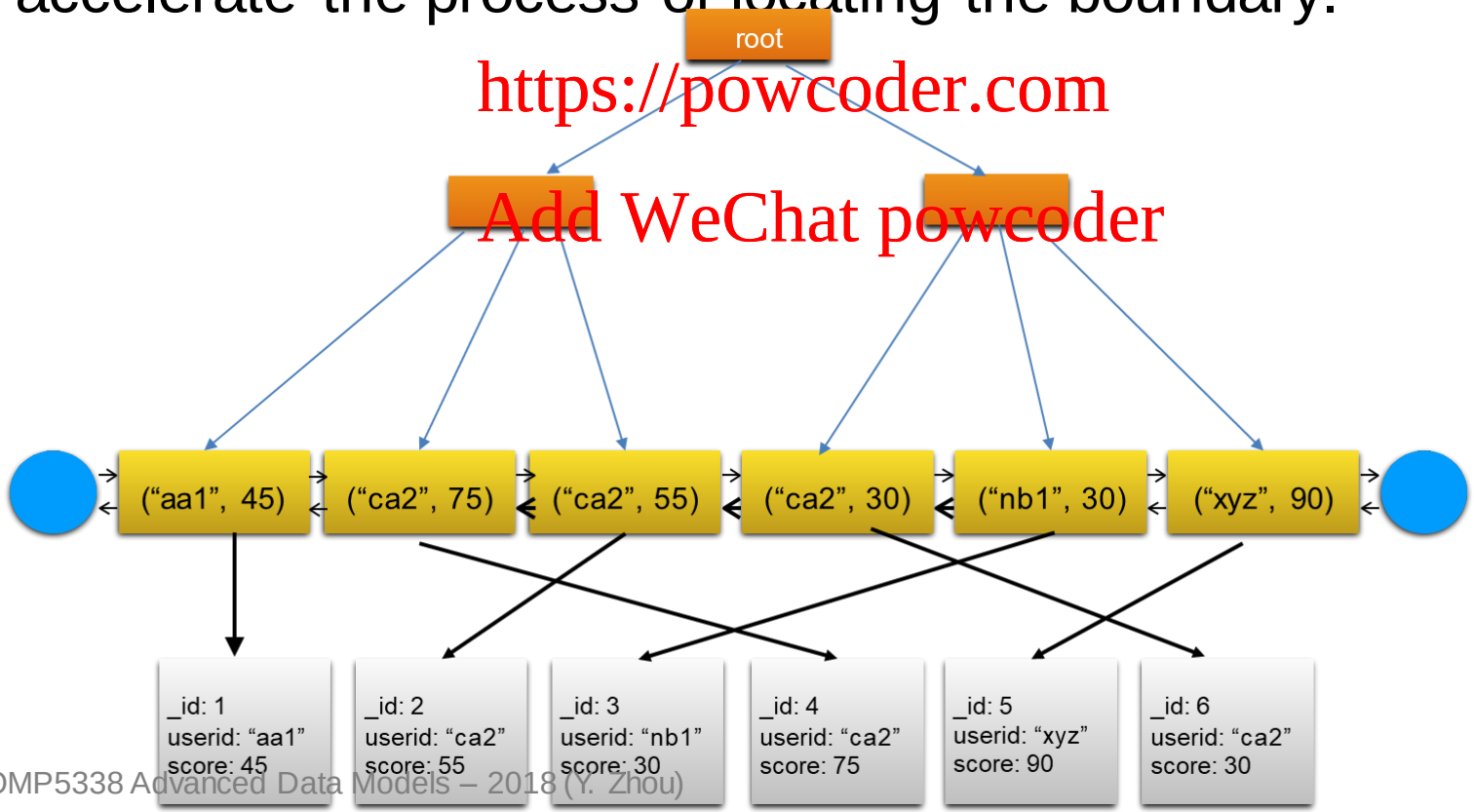


General rules

- If you are able to traverse the list between the upper and lower bounds as determined by your query condition in one direction to obtain the correct order as specified in the sort condition, the index will be used to sort the result
- Otherwise you may still use index to obtain the results but have to sort them in memory

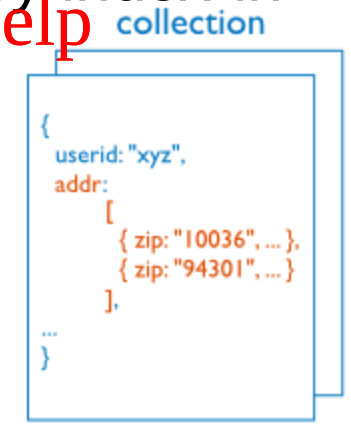
BTree motivation

- Finding the boundaries could be time consuming if we only have the list structure and can only start from one of the two ends
- B-Tree structure is built on top of the index values to accelerate the process of locating the boundary.



Multi key index

- Index can be created on array field, the key set include each element in the array. It behaves the same as single index field otherwise
- There are restrictions on including multi key index in compound index



{ "addr.zip": 1 } Index

Text Indexes

- Text indexes support efficient text search of string content in documents of a collection
- To create a text index
  - `db.<collectionName>.createIndex({<fieldName>:"text"});`
  - text index tokenizes and stems the terms in the indexed fields for the index entries.
- To perform text query
  - `db.find($text:{$search:<search string>})`
    - No field name is specified
- Restrictions:
  - A collection can have at most one text index, but it can include text from multiple fields
  - Different field can have different weights in the index, results can be sorted using text score based on weights
  - Sort operations cannot obtain sort order from a text index

Other Indexes

- Geospatial Index
  - MongoDB can store and query spatial data in a flat or spherical surface
    - 2d indexes and 2dsphere indexes
- Hash indexes
  - Index the hash value of a field
  - Only support equality match, but not range query
  - Mainly used in hash based sharding

Indexing properties

- Similar to index in RDBMS, extra properties can be specified for index
- We can enforce the *uniqueness* of a field by create a unique indexes
  - `db.members.createIndex( { "user_id": 1 }, { unique: true } )`
- We can reduce the index storage by specifying index as *sparse*
  - Only documents with the indexed field will have entries in the index
  - By default, non-sparse index contain entries for all documents. Documents without the indexed field will be considered as having **`null`** value.
- MongoDB also supports TTL indexes and partial index

Indexing strategy

- Indexing cost
  - Storage, memory, write latency
- Performance consideration
  - In general, MongoDB only uses one index to fulfil specific queries
    - $or query on different fields may use different indexes
    - MongoDB may use intersection of multiple indexes
  - When index fits in memory, you get the most performance gain
- Build index if the performance gain can justify the cost
  - Understand the query
  - Understand the index behaviour

Performance Monitoring Tools

- Profiler
  - Collects execution information about queries running on a database
  - IT can be used to identify various underperforming queries
    - Slowest queries
    - Queries not using any index
    - Queries running slower than some threshold
    - Custom tagged queries, e.g. by commenting
    - And more
- Explain method
  - Collect detailed information about a particular query
    - How the query is performed
    - What execution plans are evaluated
    - Detailed execution statistics, e.g. how many index entries or documents have been examined

https://studio3t.com/knowledge-base/articles/mongodb-query-performance/

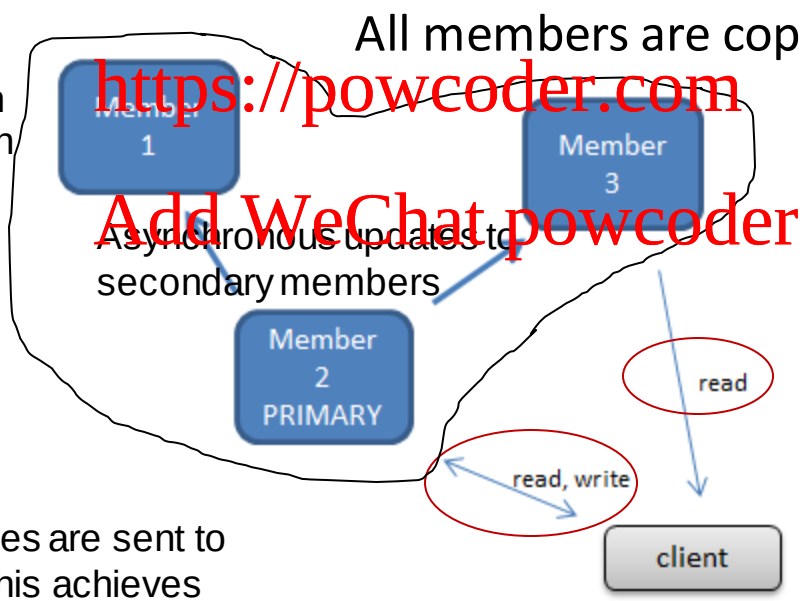
Outline

- Indexing
- **Replication**
- Sharding

Replication

- MongoDB uses replication to achieve durability, availability and/or read scalability.
- A basic master/slave replication component in MongoDB is called a **`replica set`**

MongoDB applies database operations on the *primary* and then records the operations on the primary's **`oplog`** (operation log). The *secondary* members then replicate this log and apply the operations to themselves in an asynchronous process



User may indicate that it is safe to read from secondary (slave) member; ***strong consistency*** cannot be guaranteed; achieves eventual consistency

By default all reads/writes are sent to primary member only; this achieves ***strong consistency***

http://www.mongodb.org/display/DOCS/Replica+Sets+-+Basics

Replica Set

- Data Integrity
  - Single Master (primary)
  - Write happens only on Master
  - Read from secondary (slave) member may return previous value
  - Read may return *uncommitted* value
- Primary Election
  - May be triggered at different time
    - Newly formed replica set
    - Primary is down
    - ...
  - Replica set members send heartbeats (pings) to each other every 2 seconds.
  - The first member to receive votes from a **majority** of members in a set becomes the next primary until the next election
    - Replica set needs to have odd number of members
    - Arbiter is a member of the replica set that does not hold data but are able to vote during primary election

http://docs.mongodb.org/manual/core/replication-internals/

Replica Set – cont'd

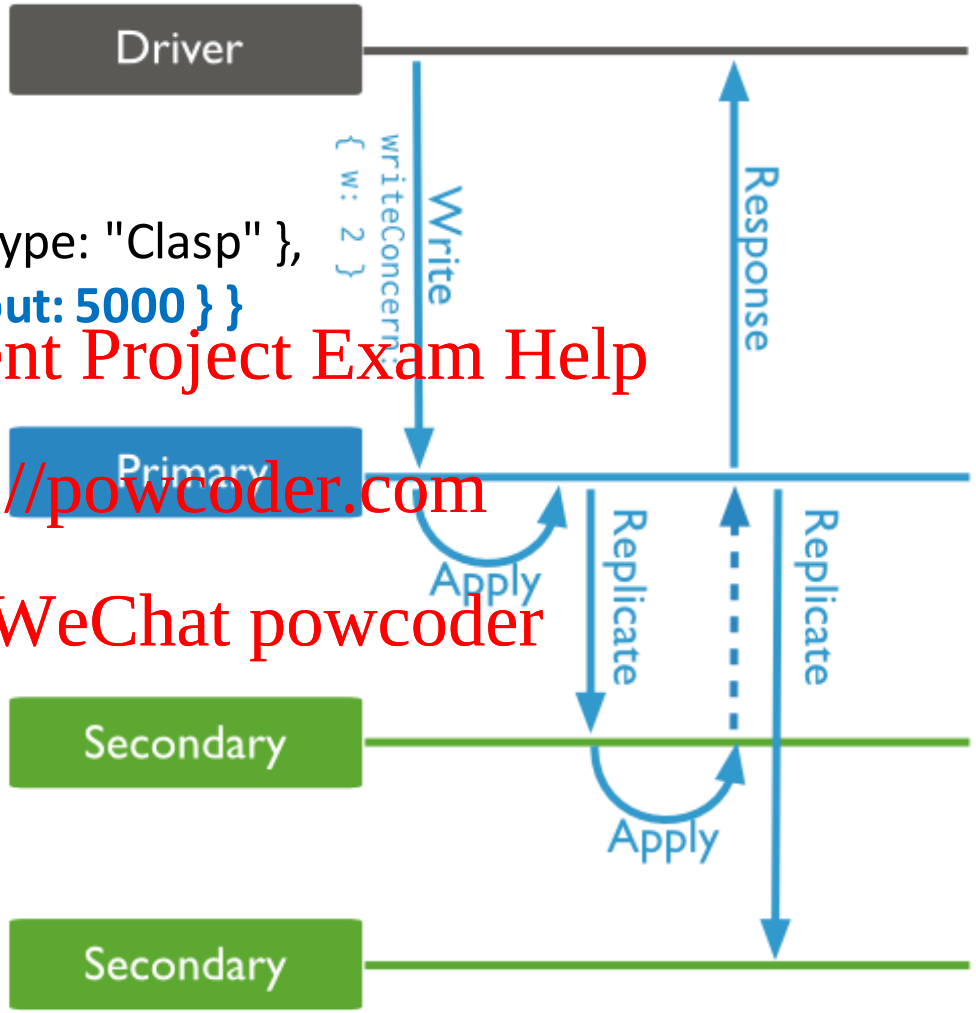
- Network Partition
  - Members in a replica set may belong to different racks or different data centers to maximize durability and availability
  - During primary election if network partition happens and neither side of the partition has a majority on its own, the set will not elect a new primary and the set will become read only

Replica Set Read/Write Options

- By *default*, read operations are answered by the primary member and always return the latest value being written
- By *default*, replication to the secondary member happens asynchronously
- Client can specify "Read Preference" to read from different members
  - Primary(*default*), secondary, nearest, etc
- To maintain consistency requirements, client can specify different levels of "Write Concern"
  - By *default*, write is considered successful when it is written on the primary member
  - This can be changed to include write operations on secondary members.

Verify Write to Replica Set

```
db.products.insert(
  { item: "envelopes", qty : 100, type: "Clasp" },
  { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

Timeout mechanism is used to prevent blocking indefinitely

http://docs.mongodb.org/manual/core/replica-set-write-concern/

Read Preference

- Read preference describes how MongoDB clients ***route*** read operations to the members of a replica set.

Mode	Description
Primary	Default one. All operations read from the primary node
PrimaryPreferred	in most situations, operations read from the primary but if it is unavailable, operations read from secondary members.
Secondary	All operations read from the secondary members of the replica set.
SecondaryPreferred	In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary.
nearest	Operations read from member of the replica set with the least network latency, irrespective of the member's type.

Read Isolation (Read Concern)

- How read operation is carried out **inside** MongoDB with replica set to control the consistency and availability
- There are many levels
- New release may introduce new level(s) to satisfy growing consistency requirement
- To understand what sort of consistency you will get, all three properties need to be looked at
  - Write Concern
  - Read Preference
  - Read Concern

Read Concern Levels

- *local*: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members (i.e. may be rolled back)
  - Default for read against primary, or reads against secondaries if the reads are associated with causally consistent sessions
- *available*: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members
  - Default for read against secondaries if the reads are **not** associated with causally consistent sessions
- *majority*: The query returns the data that has been acknowledged by a majority of the replica set members. The documents returned by the read operation are durable, even in the event of failure.
- linearizable
- Snapshot

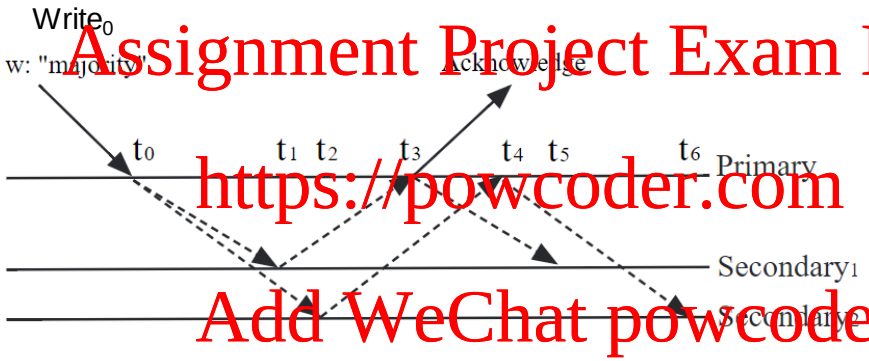
Read uncommitted behaviour may happen with local and available level

Default Behaviour

- Write concern:
  - Write is considered successful when it is written on the primary member
  - replication to the secondary members happen asynchronously
  - There is no rollback once the write is applied successfully in the primary
- Read Preference:
  - primary: All read operations are sent to the primary
- Read Concern
  - Local: returns data from the instance (in this case, the primary) with no guarantee that the data has been written to a majority of the replica set members
- What we get with default setting
  - The strongest consistency level: strong consistency at single document level
- What are trade offs
  - Availability and latency
  - All write/read happens at primary, secondaries have little use in terms of live traffic

Customized Behaviour: Write: majority

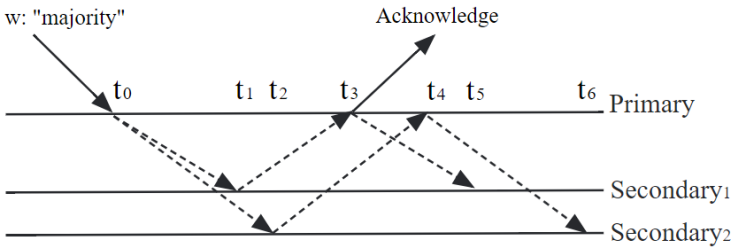
- Write concern: “majority”
  - Requests acknowledgement that write operations have propagated to the majority of voting nodes, including the primary



- All writes prior to $Write_0$ have been successfully replicated to all members.
- $Write_{prev}$ is the previous write before $Write_0$.
- No other writes have occured after $Write_0$.

https://docs.mongodb.com/manual/reference/read-concern-local/#readconcern.%22local%22

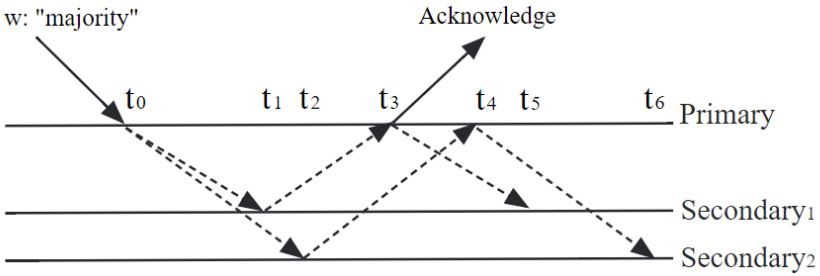
Write: majority example case



Time	Event	Most Recent Write	Most Recent w: "majority" write
t_0	Primary applies $Write_0$	Primary: **$Write_0$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$	Primary: **$Write_{prev}$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_1	$Secondary_1$ applies $write_0$	Primary: **$Write_0$** $Secondary_1$: $Write_0$ $Secondary_2$: $Write_{prev}$	Primary: **$Write_{prev}$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_2	$Secondary_2$ applies $write_0$	Primary: **$Write_0$** $Secondary_1$: $Write_0$ $Secondary_2$: $Write_0$	Primary: **$Write_{prev}$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_3	Primary is aware of successful replication to $Secondary_1$ and sends acknowledgement to client	Primary: **$Write_0$** $Secondary_1$: $Write_0$ $Secondary_2$: $Write_0$	Primary: **$Write_0$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_4	Primary is aware of successful replication to $Secondary_2$	Primary: **$Write_0$** $Secondary_1$: $Write_0$ $Secondary_2$: $Write_0$	Primary: **$Write_0$** $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_5	$Secondary_1$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write	Primary: **$Write_0$** $Secondary_1$: **$Write_0$** $Secondary_2$: $Write_0$	Primary: **$Write_0$** $Secondary_1$: **$Write_0$** $Secondary_2$: $Write_{prev}$
t_6	$Secondary_2$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write	Primary: **$Write_0$** $Secondary_1$: **$Write_0$** $Secondary_2$: **$Write_0$**	Primary: **$Write_0$** $Secondary_1$: **$Write_0$** $Secondary_2$: **$Write_0$**

Read Concern: *local* example

Read Preference: *Primary*, *PrimaryPreferred*, *SecondaryPreferred*, *Nearest*



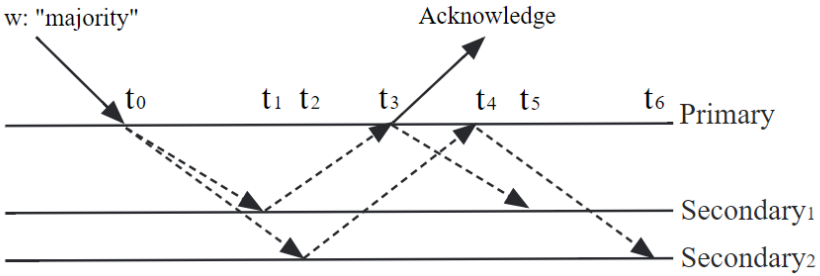
Read uncommitted before t3

Read Target	Time T	State of Data
Primary	After t_0	Data reflects $Write_0$.
$Secondary_1$	Before t_1	Data reflects $Write_{prev}$
$Secondary_1$	After t_1	Data reflects $Write_0$
$Secondary_2$	Before t_2	Data reflects $Write_{prev}$
$Secondary_2$	After t_2	Data reflects $Write_0$

Read Concern: available has similar behaviour

Read Concern: *majority* example

Read Preference:
Primary,
PrimaryPreferred,
SecondaryPreferred,
Nearest



Primary has the most recent update $Write_0$ since t_1, but before t_3 it knows that majority of the replica has the previous value $Write_{prev}$

Read Target	Time T	State of Data
Primary	Before t_3	Data reflects $Write_{prev}$
Primary	After t_3	Data reflects $Write_0$
$Secondary_1$	Before t_5	Data reflects $Write_{prev}$
$Secondary_1$	After t_5	Data reflects $Write_0$
$Secondary_2$	Before or at t_6	Data reflects $Write_{prev}$
$Secondary_2$	After t_6	Data reflects $Write_0$

t_2	$Secondary_2$ applies $write_0$	Primary: $Write_0$ $Secondary_1$: $Write_0$ $Secondary_2$: $Write_0$	Primary: $Write_{prev}$ $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$
t_3	Primary is aware of successful replication to $Secondary_1$ and sends acknowledgement to client	Primary: $Write_0$ $Secondary_1$: $Write_0$ $Secondary_2$: $Write_0$	Primary: $Write_0$ $Secondary_1$: $Write_{prev}$ $Secondary_2$: $Write_{prev}$

Consequence

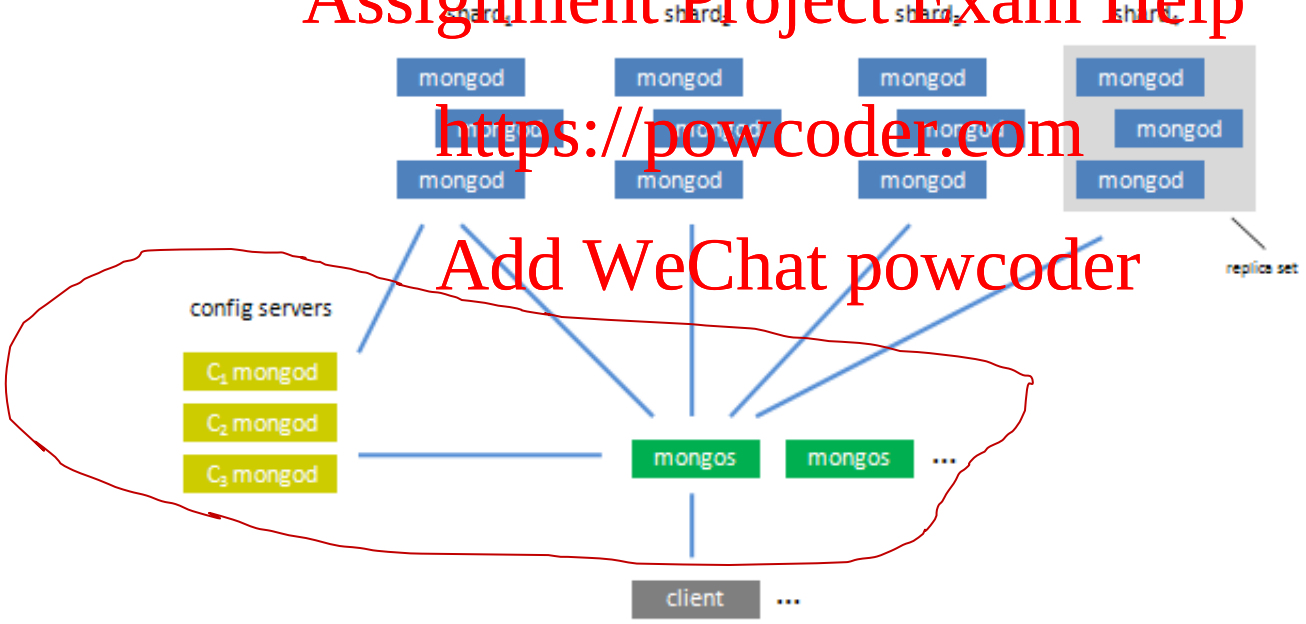
- When write concern is set to *majority*,
  - Read concern "*local*" can return the latest value as soon as it is applied locally, it has the danger of read uncommitted, e.g. return a value that should not exist if rolled back
  - Read concern "*majority*" will return old value some time after the write happens even if the target is set to primary node; it does not return uncommitted value regardless of the targe node.
- Customized setting will have better scalability by allowing read to happen at the secondary node
  - There are various trade offs depending on the actual setting

Outline

- Indexing
- Replication
- **Sharding**

Sharding

- MongoDB uses sharding mechanism to **scale out**
- The main database engine **`mongod`** is not distributed
- Sharding is achieved by running an extra coordinator service **`mongos`** together with a **config server** set on top of **`mongod`**

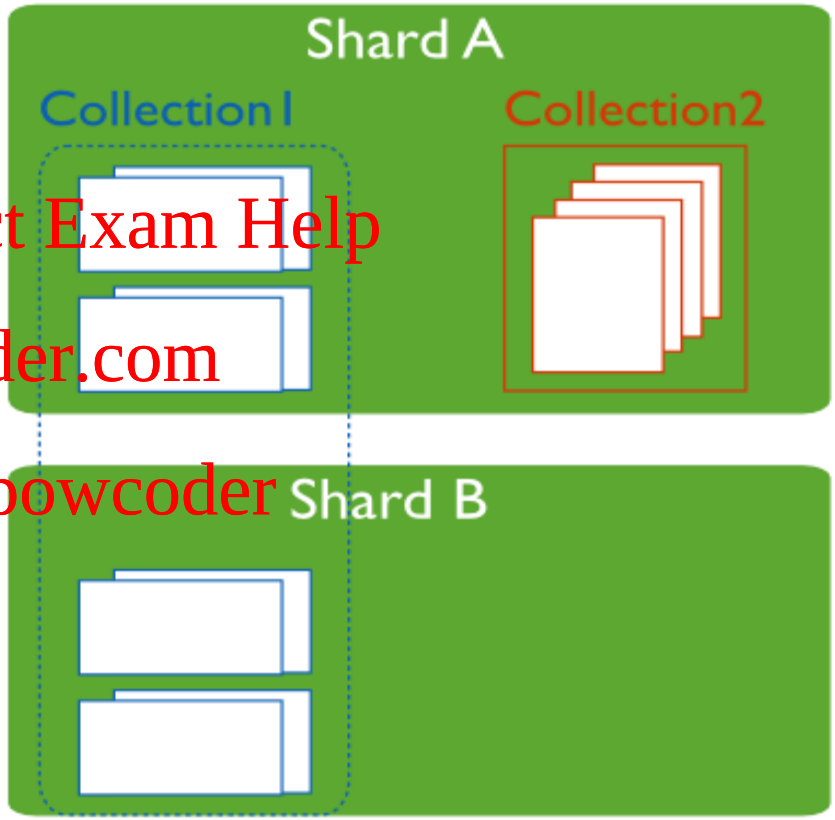


http://docs.mongodb.org/manual/core/sharding/

http://www.mongodb.org/display/DOCS/Sharding+Introduction

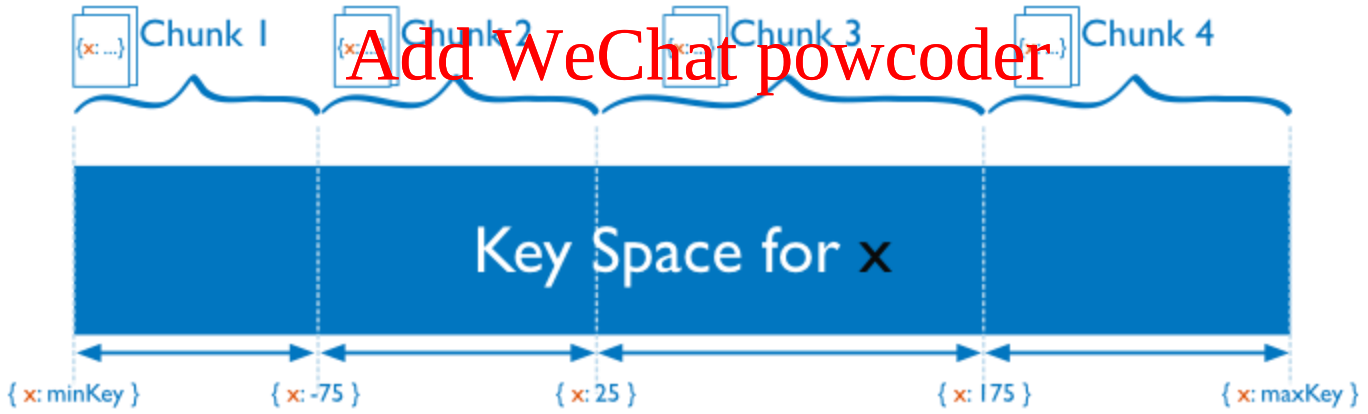
Shard

- Each shard is a standalone mongod server or a replica set (with one primary and a few secondary members)
- Each shard stores a portion of large collection partitioned by a **shard key**
- Primary Shard
  - Every database has a primary shard that holds all unsharded collections for a database



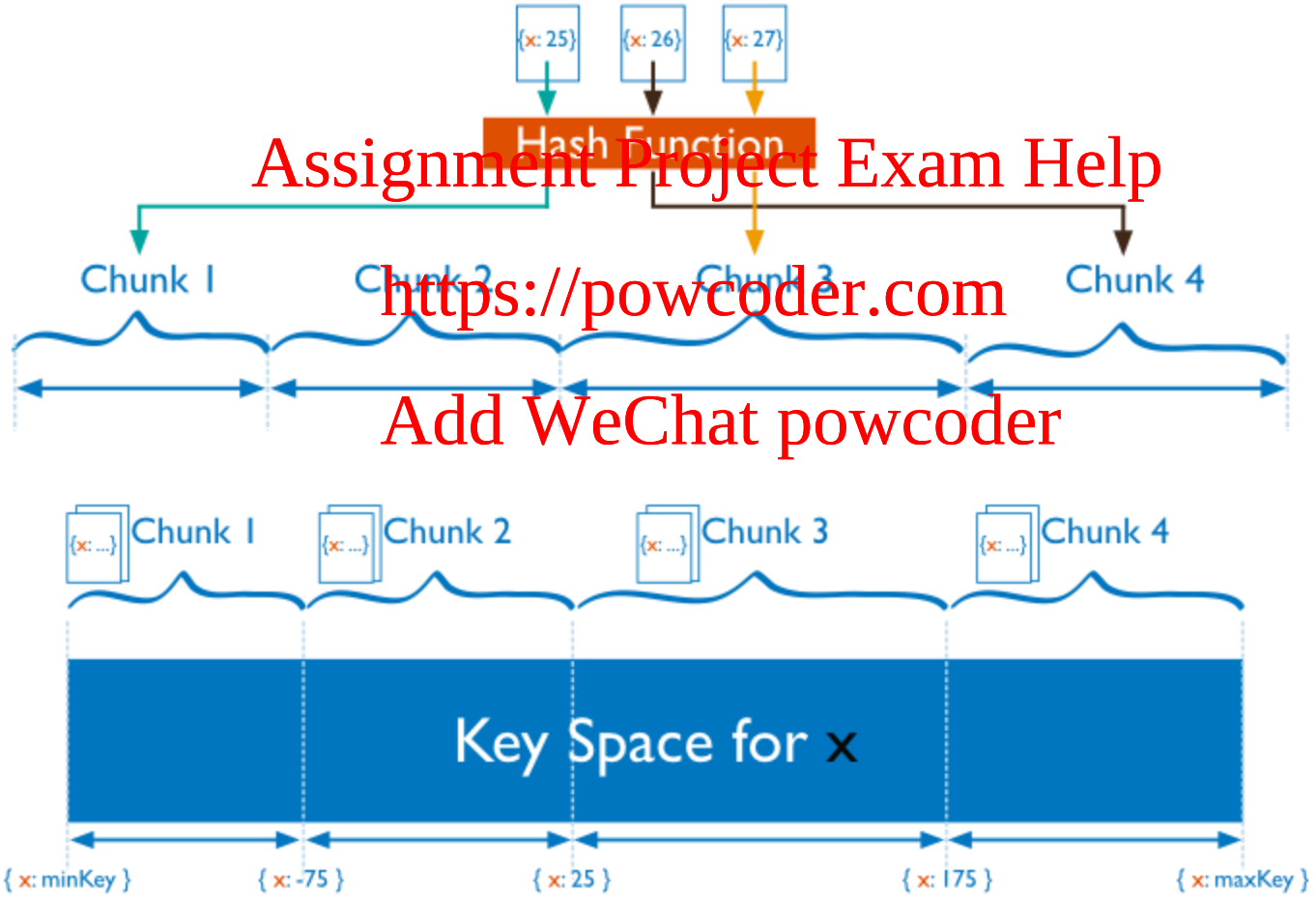
Shard Keys

- The shard key determines the distribution of the collection's documents among the cluster's shards.
- Data stored in each shard are organized as fixed sized **chunks** (usually 64MB)
  - Chunk is the basic data distribution units (we move around chunks between shards)



Sharding strategy

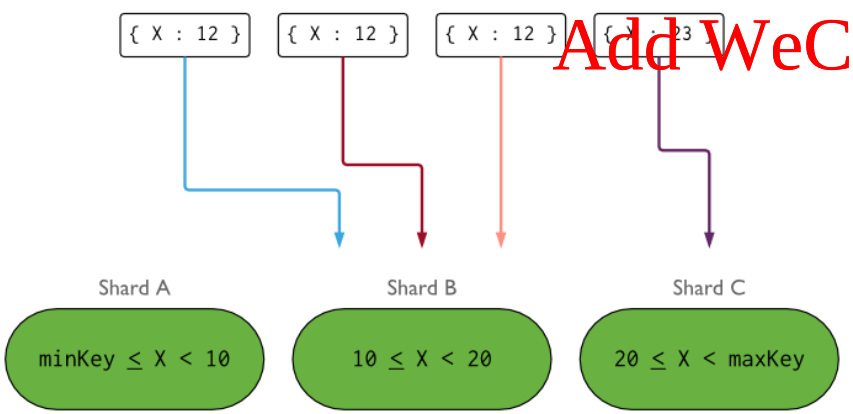
- Hash Sharding vs. Range Sharding



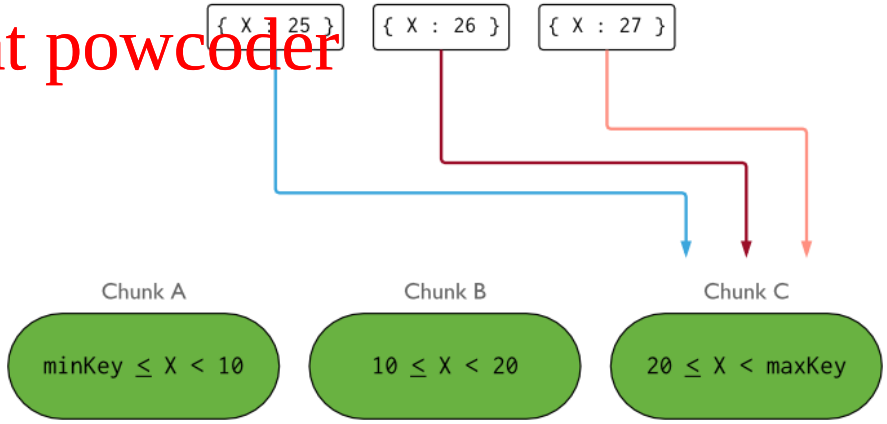
Shard key selection

- The ideal shard key should distribute data and query evenly in shards
  - High cardinality
    - Gender is not a good sharding key candidate
  - Distribution not skewed
    - Key with zipf value distribution is not a good sharding key candiddate
  - Change pattern
    - Timestamp is perhaps not a very good shard key candidate

Shard key with skewed distribution would create query hot spot



Monotonically increasing shard key would create insert hot spot



Example of good sharding key

Machine 1	Machine 2	Machine 3
Alabama → Arizona	Colorado → Florida	Arkansas → California
Indiana → Kansas	Idaho → Illinois	Georgia → Hawaii
Maryland → Michigan	Kentucky → Maine	Minnesota → Missouri
Montana → Montana	Nebraska → New Jersey	Ohio → Pennsylvania
New Mexico → North Dakota	Rhode Island → South Dakota	Tennessee → Utah
	Vermont → West Virginia	Wisconsin → Wyoming

`user` collection partitioned by field "**`state`**" as shard key

Config Server

- **Config servers** maintain the shard metadata in a config database.
  - Chunks and their locations in shard
- Config servers *do not* run as replica set, it runs **two-phase commit** protocol to ensure strong consistency among copies
  - 3 server is recommended as optimal setting
  - more instances would increase coordination cost among the config servers.

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	$shard_2$
users	{ name : 'Nessman' }	{ name : 'Ogden' }	$shard_4$
...			

`user` collection partitioned by field "**`name`**" as shard key and are stored as chunks in different shards

Routing Processes -- mongos

- In a sharded cluster, **mongos** is the front end for client request
  - When receiving client requests, the **mongos** process ***routes*** the request to the appropriate server(s) and ***merges*** any results to be sent back to the client
  - It has no persistent state, the meta data are pulled from **config servers**
  - There is no limits on the number of **mongos** processes. They are independent to each other
- Query types
  - Targeted at a single shard or a limited group of shards based on the **shard key**.
  - Broadcast to all shards in the cluster that hold documents in a collection.

Targeted and Global Operations

- Assuming shard key is field x

Operation	Type	Execution
db.food.find({x:300})	Targeted	Query a single shard
db.foo.find({ x : 300, age : 40 })	Targeted	Query a single shard
db.foo.find({ age : 40 })	Global	Query all shards
db.foo.find()	Global	Query all shards, sequential
db.foo.find(...).count()	Variable	Same as the corresponding find() operation
db.foo.count()	Global	Parallel counting on each shard, merge results on mongos
db.foo.insert(<object>)	Targeted	Insert on a single shard
db.foo.createIndex(...)	Global	Parallel indexing on each shard

Sharding Restrictions and Limitations

- When shard key is not the **_id** key, the uniqueness of the **_id** values can only be guaranteed at application level
- Certain operations are not supported in sharded environments
- Shard Key Limitation
  - Shard key cannot be multikey index, text index or geospatial index
  - Shard key is immutable
  - Shard key value in a document is immutable

http://docs.mongodb.org/manual/reference/limits/

Summary

- **MongoDB is a general purpose NoSQL storage system**
  - Lots of resemblance with RDBMS
    - Indexing, ad-hoc queries
    - It supports spatial queries
  - Single document update is always atomic
  - Latest version has support for multi-document transaction
    - Application level 2phase commit can be implemented for earlier versions
- **Key Features**
  - Flexible schema
    - Collection and Document
    - Documents are stored in binary JSON format
    - Natural support for object style query (array and dot notation)
  - Scalability
    - Sharding and Replication
  - Various consistency levels achieved through write concern, read preference and read concern property combination