Mongo Documentation

Mercia Malan

<2019-06-13 Thu>

Contents

1 M201

1.1 Hardware:

Database designed around the usage of memory

- aggregation, index traversing, write operations, query engine, connections(1mb per established connection)

By default mongo will try to use all cpu cores.

Page compression, data calculation, aggregation, map reduce requires cpu cores

Writing constantly to the same document - each document will block writes on that document, thus multiple writes to the same document won't be increased by concurrency

1.1.1 Persisting data

- higher IOPS your server provides, the faster your writes and reads will be

Mongo will benefit from some, but not all raid architectures - such as 10. Discourage using raid 5 or 6 or 0. Raid 10 provides redundancy across drives and also optimised for read and write

Mongo can use multiple disks

1.1.2 Network hardware

Faster and larger bandwidth, the faster. Firewalls, load balances etc also play a role in network latency. Make sure write & read concern and read preference aligns with network architecure

1.2 Indexes:

index keeps a reference to the document Also uses b-tree for storing index e.g.: lastname: 1

acosta –> {lastname: acosta . . . } bailey –> {lastname: bailey . . . }

writes, updates and deletes slows down the more indexes there are, thus try to not have many irrelevant indexes. Each write might need to rebalance b-tree

1.2.1 How is data stored on disk

1. WiredTiger Creates a file for each collection and each index Can run mongod with –directoryperdb and then it will create a sub directory per db Can run with –WiredTigerDirectoryForIndexes and inside each db sub directory it will create a "collections" and an "index" folders. This is useful if you want to create symbolic links to different disks.

1.2.2 Single Field Index

Can specify index with dot notation for sub documents. It's bad practice to create an index on a subdocument field - rather create it on the nested field inside the sub document by using dot notation

1.2.3 Sorting and indexes

When running a query with a sort on the field that is indexed, it will do an IXSCAN (index scan), because it uses the index keys to help with sorting. IXSCAN can either be forward or backward depending on the sorting direction vs the index. e.g. if the index is ascending ({name: 1}) and sort is ascending (.sort({name:1}), forward ixscan would be used. if it was .sort({name: -1}), it would use a backward ixscan.

1.2.4 Compound Indexes

For compound indexes it is still stored flat. e.g. on lastname and firstname, it will have index keys such as ("last","first") (such as in a telephone book). Compound indexes has index prefixes, which will be the first key, the first and second keys, the first, second and third keys etc. E.g.: {$\text{last}_{\text{name}}$: 1, $\text{first}_{\text{name}}$: 1, age:1} Prefixes: {$\text{last}_{\text{name}}$: 1}, {$\text{last}_{\text{name}}$: 1, $\text{first}_{\text{name}}$: 1} Prefixes can also be scanned forward and backward, as long as the direction is either exactly what it is in the index, or the exact opposite

1.2.5 Multi-key Indexes

When your index key is inside an array, it will create a key per value in the array. Can only create one multi-key indexes. (ie only one index that is in an array} Multi-key indexes do not support covered queries E.g.

```
{
  productName: ["x","y","z"],
  stock: [
      {name: "a", qty: 1},
      {name: "b", qty: 2},
      {name: "c", qty: 2},
  ]
}
```

Can have an index on productName or on stock.qty, but not on both. Index on productName will create 3 index keys: x,y,z.

1.2.6 Partial Indexes

Create an index on only a subsection of the data, for example:

```
{
    "name" : "some restaurant",
    "cuisine" : "Indian",
    "stars" : 4.4,
    "address": {
        "street": "123 str",
        "city": "New York"
    }
}

db.restaurants.createIndex(
    { "address.city": 1, "cuisine": 1 },
    { partialFilterExpression: { 'stars': { $gte: 3.5 } } }
)
```

1.2.7 Text Indexes

```
{
    productName: "MongoDB long sleeve t-shirt",
    category: "Clothing"
```

```
}

db.restaurants.createIndex(
    {productName: "text" }
)

//leverages mongo's full text search capabilities:
db.products.find({ $text: {$search: "t-shirt" }})
```

Will create 5 indexes: mongodb, long, sleeve, t, shirt. Take note that this type of index will take a lot longer to write. One way to make this a bit better is to use compound indexes, like including category as first index key. Text queries logically or, so searching for like $search: "mongodb long" will do "mongodb" or "long". You can project a text score and then sort by it, which will sort according to how well it searched

```
db.products.find(
    { $text: { $search: "MongoDB long" } },
    { $meta: "textScore" }
).sort( {
    score: {$meta: "textScore" }
})
```

1.2.8 Collations

Settings for specific locale's. Can specify different collation's for indexes

```
db.foreign_text.createIndex(
    {name: 1},
    {collation: {locale: 'it'}}
)
//to use, the query must match the collation
db.foreign_text.find({name: "x"}).collation({locale: 'it'})
```

Can add strength: 1 to the collation to ignore case

1.2.9 Building Indexes

Foreground indexes will block the entire databases until build is complete. Background indexes don't block operations, but take a lot longer and will still impact queries. (to indicate background, add extra document to createIndex {"backround":true})

1. Query Plans Plan created when a query is run. Multiple will be created and best one is then chosen when running the plan Starts by looking at the indexes to see which one(s) can be used, and then tries each of them and compares. Plans are cached and cache cleared when: restart, threshold of factor of 10, index rebuilt or if index is created or dropped

2. explain() Create explainable object e.g. exp = db.people.explain(), and then do exp.find() default is 'queryPlanner' (doesn't execute query), 'executionStats' and 'allPlansExecution' (both executes query) Can store the explain output to a variable and access certain parts directly

   ```
   var exp = db.people.find({}).explain()

   exp.executionStats.executionStages
   ```

   With shards, different shards might return different winningplans

1.2.10 Force Index

Add .hint({..index..}) to force it to use index - use with caution

1.2.11 Resource allocation for indexes

RAM will be the biggest used resource for indexes

1.2.12 How to determine index size

Mongo compass, or db.stats() If there is no disk space left for indexes, they won't be created

1.2.13 How to determine how much memory indexes uses

db.col.stats({indexDetails: true}) It's quite a lot of data so rather assign it to a variable and then access certain parts, such as "indexDetails"

```
stats = db.col.stats({indexDetails: true})
stats.indexDetails
//look at further details per index e.g.
stats.indexDetails.index_name.cache //contains total bytes currently in cache
```

1.2.14 Edge cases of indexes

1.2.15 Occasional reports and needing indexes to support them

Indexes that are not being used should not exist. Indexes needed occasionally should not be in memory. Can create indexes only on a secondary to be used specifically for a report

1.2.16 right-end-side index increments

As we're inserting data, the index B-tree might always grow on the right-hand side and thus unbalanced. We can then only check the right side of the index memory to know how much memory to allocate for a specific query. E.g. if you know you're only querying recently added data, it will only use the latest indexes, and thus do not need the entire index in memory.

1.3 Optimising CRUD queries

1.3.1 Index Selection

Equality queries are better for use with indexes because it's very specific, whereas range queries are not specific So selection is better with equality. Consider this in index order - keep equalities first E.g.

```
db.col.find({"zipcode": 1000}) //equality
db.col.find({"zipcode": {$gt: 5000}}) //range query
```

1.3.2 Equality, Sort, Range

Query and index for most performance e.g.:

```
db.restaurants.find({'address.zipcode': {$gt: '5000'}, cuisine: 'Sushi' })
    .sort({stars: -1})

db.restaurants.createIndex({ "cuisine": 1, "stars": 1, "address.zipcode": 1})
```

Indexes should always be done in this order, first list equality, then sort, then range.

1.3.3 Performance Tradeoffs

Sometimes it makes sense to be a little bit less selective to prevent an in memory sort, because execution time will be less. In the example above, stars and zipcode was switched, because having the stars last makes it do an

in memory sort. In the current order it will examine more keys, but only do an index scan followed by a fetch, which has a much lower execution time than before.

1.3.4 Covered Queries

The entire query is serviced by index keys. 0 documents to be examined One way to do this is by adding a projection which only contains the index keys, this way mongo can run the query and return values by only using index. (note _id: 0).

Covered queries only work if all the fields are in the query and in the projection. Using the opposite projection by ommitting fields, even if the rest that are left over are all in the index, it will not be covered - how is mongo to know for sure that ALL documents only have the index fields left.

You can't cover a query if

- Any of the index fields are arrays
- Any of the index fields are embedded documents
- When run against a mongos if the index does not contain the shard key

1.3.5 Regex Performance

Regex is not very performant in general.

```
db.users.find({username: /kirby/})
//if there is an index on username it will be a bit faster, however for the regex it w
//have to check every index key

//to make it better, by specifying ^ to say "begins with kirby":
db.users.find({username: /^kirby/})
```

1.3.6 Insert Performance

1. Write concern w: how many members of replica set we're waiting for to acknowledge write, a number or 'majority' j: journal - should we wait for the on-disk journal to be written wtimeout: how long (in ms) we want to wait for the write acknowledgement before timing out

   Indexes affect write time, as it needs to write to the B-tree as well.

To set writeConcern in Java:

```
MongoClient mongoClient = new MongoClient();

db = mongoClient.getDatabase("test");
coll = db.getCollection("coll");

WriteConcern w = new WriteConcern();

//withW: number, 1 = primary only, 2 = primary and 1 secondary etc, or "majority"
//withJournal: true / false
w = w.withW(1).withJournal(false);

coll = coll.withWriteConcern(w);
```

2. Index Overhead Test run on the dude's imac, so just a general idea, not 100% accurate for prod clusters

Number of indexes	0	1	5
Average inserts/sec	~ 16000	~15000	~10500
Percent loss from baseline	0%	~6.3%	~34.4%

Write Concern:

Write Concern	{ w: 1, j: false, }	{ w: 1, j: true }	{ w: "majority", j: false }	{ w: "majority", j: true }
Average inserts/sec	~ 27000	~19000	~16000	~14000
Percent loss from baseline	0%	~29.6%	~40.7%	48.1%

1.3.7 Data Type Implications

Query matching: Need to specify the type if using something specific e.g. NumberDecimal - 2.34 and NumberDecimal(2.34) are not the same thing

For sorting, when one field has different values, bson types are sorted as follows:

1. MinKey (internal type)

2. Null
3. Numbers (ints, longs, doubles, decimals
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

Same grouping will happen on an index. The B-tree will actually group it by the above data types

1.3.8 Aggregation performance

1. Index Usage Some pipeline stages uses indexes, and some don't. If a stage is not using an index, none of the stages after it will use indexes

```
db.coll.aggregate([
    { ... }
], {explain: true})
```

   (a) Match Can use an index, especially if it is early in the pipeline.
   (b) Sort Keep it early in the pipeline as well for index usage to avoid in memory sorting.
   (c) limit Keep it close to the match and sort (e.g. match, limit, sort), because then it can use a top-k sorting algorithm

2. Memory Constraints Documents has 16 MB limit For this reason, use $limit and $project 100MB RAM per stage limit (can use allowDiskUse: true - last resort!) – allowDiskUsage does not work with GraphLookup

1.4 Performance of distributed systems

In sharding - define what is a good shard key by knowing how data is accessed.

Because a sharded cluster contains MongoS + config servers + shard nodes latency does have an effect. Having the mongos on the same server as the client application an help.

1.4.1 Reads on Sharded Clusters

Two types of reads in sharded clusters, depending on whether we're using the shard key in our queries or not

1. Scatter Gather - ask every shard node
2. Router queries - ask a specific node

1.4.2 Sorting in a sharded cluster (same logic for skip or limit)

mongos will send the sort query to each relevant shard, local sorting will be done on each shard, and then a final merge sort happens on the primary shard

1.5 Increasing write performance with sharding

Each shard contains chunks with a lower and upper bounds based on the shard key. It is important to make sure that data is spread evenly across chunks.

Three things to keep in mind (typically, try to avoid to have the upper and lower bounds be the same value in chunks

1.5.1 Cardinality

Number of distinct values for a given shard keys. We want a high cardinality. E.g. country will be very low. To increase cardinality one can use compound keys

1.5.2 Frequency

Even distribution of each value across shards. E.g. for surnames in SA "M" would be a LOT more than the rest.

1.5.3 Rate of change

How our values change over time. Avoid monotonically increasing or decreasing values. E.g. ObjectID is monotonically increasing. - all of the shards will go to the "last" shard close to "maxKey". Can add e.g. _id to your shard key if it's the last value in your compound key. This can be a good idea as it increases cardinality as the value will always be unique.

1.5.4 Bulk Writes

Ordered bulk writes - will write in order, waiting for one operation to finish before starting the next one. Unordered will do all in parallel.

In replica sets the writes will all happen on the primary, whereas with a sharded cluster it will happen on the relevant shard, which increases time due to latency.

Unordered - parallel on sharded cluster will be faster. Keep in mind that mongos will need to deserialize each write before sending it to the relevant shard.

1.6 Reading from Secondaries

Default read preference is "primary". Can set it to secondary, secondaryPreferred or nearest Need to know that if you read from secondaries we're not guaranteed to read the most up to data

1.6.1 When is it a good idea?

Offload work - e.g. analytics / reporting Local reads in geographical spread replica sets, e.g. if users are split between east and west coast we can read from "nearest" (note they are ok to read stale data)

1.6.2 When is it a bad idea?

In general it is a bad idea Sharding increases capacity, not replication Very bad idea to read from secondaries in shards, in fact never read directly from shards

1.7 Replica sets with differing indexes

This should be used with caution, not common practice - specific cases only These nodes should not be able to become a secondary, because then the indexes will clash

Example: 1 primary, 2 secondary 1 secondary with priority 0

```
var conf = {
    "_id": "M201",
    "members": [
        {_id: 0, "host": "127.0.0.1:27000" },
        {_id: 1, "host": "127.0.0.1:27001" },
        {_id: 2, "host": "127.0.0.1:27002", "priority": 0 }
    ]
}
```

```
// running rs.isMaster() on the primary will result in 2 hosts an 1 passive
//next steps, connect to secondary
db.slaveOk()
```

To create an index on a specific secondary, start it up as a standalone node (nb use the same dbpath), and then create index, shut it down and start up replica set again

1.8 Aggregation Pipeline on a sharded cluster

Because data is distributed in a sharded cluster there is some additional work that needs to be done when performing aggregation.

When doing a match first, the chance is that all data from the match sits on one shard and thus the rest of the aggregation can happen on one shard. When just doing something like a group, work will be done on each shard, and then merged on one shard (randomly chosen).

Random shard being chosen does not apply for $out, $facet, $lookup, $graphLookup - primary shard will run the merge

Query optimiser will do some optimising such as moving a match before a sort, or combine two stages into one.

2 M310

2.1 Auditing (Chapter 3)

Audit logs are output in bson or json, with the following format:

```
{
    atype: <String>, //action type: authenticate, createIndex, createUser, addChart
    ts: { //timestamp
```

```
        "$date": <timestamp> //date and utc time
    },
    local: {
        ip: <String> , port: <int>
    },
    remote: { //incoming event
        ip: <String> , port: <int>
    },
    users: [
        { user: <String>, db: <String> }, ...
    ],
    roles: [
        { role: <String>, db: <String> }, ...
    ],
    param: <document>, //associated with atype
    result: <int> //error code
}


//example of param:
{
    atype: "authenticate",
    ...
        param: {
            user: <user name>,
            db: <database>,
            mechanism: <mechanism>
        }
}
```

2.1.1 How to enable

Can output to system log, console or file. For file we need more arguments
audit format BSON serialises to disk faster

```
mongod --dbpath /data/db --logpath /data/db/mongo.log --fork --auditDestination syslog
mongod --dbpath /data/db --logpath /data/db/mongo.log --fork --auditDestination file --
```

2.1.2 Default logging

- Schema (DDL)

- Replica Set and Sharded Cluster
- Authentication & Authorization

Note: CRUD can be audited, but is not enabled by default because it decreases performance (extra write per operation)

2.1.3 Audit filter

Sample config.yaml with added filter

```
systemLog:
  destination: file
  path: /data/db/mongo.log
storage:
  dbPath: /data/db
auditLog:
  destination: file
  format: JSON
  path: /data/db/auditLog.json
  filter: '{ atype: { $in: [ "createCollection", "dropCollection" ] } }'
```

Can either run via the config or the auditFilter parameter

```
mongod --config config.yaml --fork
mongod --auditFilter xx
```

2.1.4 DDL and DML

DDL = Data Definition Language (schema change)

- createCollection
- createDatabase
- createIndex
- renameCollection
- dropCollection
- dropDatabase
- dropIndex

Create audit filter for DDL
Filter for getting all createIndex operations for "my application"

```
# example namespace: my-application.product
# regex: ^ = starts with, \. = . directly after
  auditLog:
    filter: '{ atype: "createIndex",  "param.ns": /^my-application\./ }'
```

DML = Data Manipulation Language (data change)
————- Chapter 3 - DML operation, still need to do chapt 1 and 2
———-

3 M320 Data Modelling

3.1 Constraints that affect your design

3.1.1 Hardware

Storing all data in RAM would be great, but it's expensive. SSD's is the next best thing but can also be expensive

3.1.2 Data

The size and security of the data needs to be kept in mind

3.1.3 Application

Network latency

3.1.4 Databse Server/MongoDB

Documents can't exceed 16MB. For reads, split frequently accessed data from less frequently accessed data. To read you also need to read the entire document into memory Document writing is acid compliant

3.2 Working Set

The total body of data that the application uses in the course of normal operations Frequently accessed data with indexes to be kept in memory if possible (ie keep working set in RAM)

3.3 The data modelling methodology

3.3.1 Phase 1: Describe the workload,getting everything you need to know about how to use your data

Logs and stats gives additional information and will be useful Create some artifacts from the data you have, e.g. size data, quantify ops, qualify ops Record your assumptions

3.3.2 Phase 2: Set up relationships

Start with data from previous phase. Determine relationships Identify, quantify, embed or link

3.3.3 Phase 3: Apply design patterns or transformations

Recognise and apply patterns needed for optimisation

If you track your assumptions as you make changes, it will be easier in the future to relect back to where you were at at the time of the change

3.4 Modelling for simplicity vs performance

Keeping related data in single collection keeps it simple, data access is just one read to the one collection, all data is together and easy to find.

3.4.1 Model for Performance

Go through all the steps of methodologies and quantify: operations per seconds latency attribution queries: e.g. can application work with data that's a little stale? Tends to see more collections when modelling for performance

Prioritise simplicity over performance

3.5 Relationships

Entities (objects in mongo) and how they are related to each other

3.5.1 Type and cardinality

- one-to-one
- one-to-many
- many-to-many

In big data, these cardinalities does not always make sense, and modelling it you can, for example, embed children into a parent since typically people don't have more than 2 children, but you can't embed your twitter followers.

We need to represent "one-to-many" in a better way

1. Modified notations Crow foot notation (one-to-zillions - based on one-to-many symbol but has 5 "fingers" instead of 3) and Numeral Notation ([min, likely, max]) This allows us to represent relationships in a more precise way.

2. One-to-many relationship modelling In mongo there are 2 main ways of modelling this relationship:

   - Embed (usually in the entity most queried) - preferred, especially if the embedded document is small
     - in the "one" side
     - in the "many" side
   - Reference (usually referencing in the "many" side) - note that mongo does not support foreign keys or cascade deletes, so if reference relationships change it needs to be manually changed in all places
     - in the "one" side
     - in the "many" side

3. Many-to-many relationship In tabular databases we create a linking table In MongoDB:

   - Embed (usually, only the most queried side is considered)
     - array of subdocuments in the "many" side
     - array of subdocuments in the other "many" side
   - Reference
     - array of references in one "many" side
     - array of references in the other "many" side

   Example: embed in the main side - results in duplication, indexing is done on the array

   ```
   //carts:
   {
   ```

```
    _id: <objectId>,
    date: <date>,
    items: [
        {qty: xx,
         item: { //copy of item from item collection
             _id: <int>,
             title: <string>
             ...
         }
        }
        ]

}


//items
{
    _id: <int>,
    title: <string>
    ...
}
```

Example: Using references

```
//items [100K]
{
    _id: <int>,
    title: <string>,
    sold_at [1,1000]: <string>

}

//stores [1000]
{
    _id: <objectId>,
    storeId: <string> //reference point
    ...
}
```

Example: Reference in the secondary side. Need a second query to get more information

```
//items
{
    _id: <int>,
    ...
}

//stores
{
    ...,
    items_sold[1,10K,100K]: <int> //references _id above
}
```

4. One-to-one relationship Embed document:

   - fields at same level
   - grouping in sub-documents

   Can split it up and reference the values

   Examples:

```
{
    name: xx,
    line1: xxx,
    country: xxx
}

//recommended for one-to-one
{
    name: xx,
    address: {
        line1: xx,
        country: xx
    }
}

//have a store and store_details, have a storeId in both.
```

5. one-to-zillios relationship if the "many" is 10000 or more
   How to model? Reference in the many/zillions side. Do not embed!

```
//items [100K]
{
    _id: <int>,
    ...
}

//item_views [100B]
{
    _id: <string>,
    item_id: <int> //[0,1K, 100M]
}
```

3.6 Patterns

Pattern - e.g. design patterns

Patterns are to get the best out of your model. Applying patterns may lead to...

- Duplicating data across documents
- Accepting staleness in some pieces of data
- writing extra application side logic to ensure referential integrity (e.g. to remove links between data)

For a given piece of data..

- Should or could the information be duplicated or not?
  - Resolve duplication with bulk updates
- What is the tolerated or acceptable staleness?
  - Resolve with updates based on change streams
- Which pieces of data require referential integrity?
  - Resolve or prevent the inconsistencies with change streams or transactions

1. Attribute Pattern Polymorphis, can put different products in one collection even if they have different attributes. E.g. Coke, water and a charger in the same collection.

Have some fields that are the same accross all documents, some that has the same name but different values, and then data that are in some documents but not in others.

For optimized queries we'll need indexes, which may lead to many indexes. To make this better, use the attribute pattern.

- Identify the fields you want to transpose (which exists in some documents but not all)
- Change it to a {key: xx, value: yy} structure
- Can also add stuff, e.g. {"k":"capacity", "v":4200, "u": "mAh" } where u indicates the relationship between k and v
- Can now create an index on subdoc.k and subdoc.v

Fields that share common characterestics

```
//e.g.
{
    "title": "Dunkirk",
    ...
    "release_USA": "2017/07/23",
    "release_Mexico": "2017/08/01",
    ...
}

//using attribute pattern:
{
    "title":"Dunkirk",
    ...
    "releases": [
        {
            "k": "release_USA", "v":"2017/07/23"
        }
    ]
}

//Can now do things like:
db.movies.find({"releases.v":{$gte: "2017/07"}})
```

Attribute pattern is helpful for these problems:

- Lots of similar fields
- Want to search across many fields at once
- Fields present in only a small subset of documents

Solution

- Break field/value into a sub-document with fieldA: field, fieldB: value

Use case examples:

- Characteristics of a product
- Set of fields all having the same value type such as a list of dates

Benefits and trade-offs

- Easier to index
- Allow for non-deterministic field names
- Ability to qualify the relationship of the original field and value

2. Extended reference pattern before lookup: Application side 3.4+: $lookup, $graphLookup

   Avoid a join by embedding the join table

   e.g.

```
//orders
{
    order_id: xx,
    customer_id: xx
}


//customer
{
    customer_id: xx,
    street: xx,
    city: xx,
    country: xx
}
```

In this case we might be using orders a lot more often than customer (as aggregate root) and thus embedding some basic customer information inside orders may be a good idea. Note, we only embed the most frequently queried fields from customer, leaving the rest behind in the customer collection

```
//orders
{
    order_id: xx,
    customer_id: xx,
    shipping_address: {
        street: xx,
        city: xx,
        country:xx
    }
}

//customer looks the same as above
```

Yes, we are duplicating data but we speed up the queries. How to minimise duplication?

- For extended reference pattern, choose fields that do not change often
- Only bring the fields you need, you don't need all the information, can join to find that once in a blue moon

After a source is updated:

- What are the extended reference to changed
- When should the extended references be updated

Duplication may be better than a unique reference

E.g. In our example above, it actually works fine if the customer updates their address, because at the time of the order that was their shipping address, so we shouldn't change that. Sometimes the updates also don't need to happen immediately accross all collections.

Extended reference pattern summary: Problem:

- Too many repetitive joins

Solutions:

- Identify fields on the lookup side
- Bring those fields into the main object

Use case examples:

- Catalog
- Mobile applications
- Real-time analytics

Benefits and Trade-offs

- Faster reads
- Reduce number of joins and lookups
- May introduce lots of dupilcation if extended reference contains fields that mutate a lot

3. Subset Pattern

   Mongo Working set is what is loaded into RAM, which is faster to access etc. What happens if the working set is bigger than RAM?

   - Add RAM
   - Scale horizontally
   - Reduce the size of the working set

   The key is to break up big documents

   E.g. movies that has all the details of the movie:

```
//movies
{
    title: x,
    complete_script: "all the stuff",
    cast: [ //[0,1000]
        {name: xx, role: xx}
    ]
}

//Split out e.g. script to movies_extra_info:
{
```

```
    movie_title: xx,
    complete_script: "all the stuff",
}
```

Can also extract a subset of the cast to a separate collection, keeping only e.g. 20 of the cast in the array

```
//movies
{
    title: xx,
    cast: [ //[0,20]
        {name: xx, role: xx}
    ]
}

//cast [0,1000]
{
    name: xx,
    role: xx
}
```

Subset Pattern: Problem:

- Working set is too big

-A lot of pages are evicted from memory

- A large part of documents is rarely needed

Solution:

- Split the collection in 2 collections:
  - most used part of documents
  - less used part of documents
- Duplicate part of a 1-N or N-N relationship that is often used in the most used sidd

Use Cases examples

- List of reviews for a product
- List of commnents on an article

- List of actors in a movie

Benefits and Trade-Offs:

- Smaller working set, as often used documents are smaller
- Shorter disk access fo r bringin in additional documents from the most used collection
- More round trips to the server
- A little more space used on disk

4. Computed Pattern Kind of Computations/Transformations applied to data:

   - Mathematical Operations
     - E.g. sum or aggregations
     - If we're doing the same sum on a collection many times, it may be better to precalculate the sums and store it in a separate collection (i.e. instead of reading the same data a million times and applying the sum, just read it from the "sum" collection)
     - Good example is viewings of movie screenings, since the views will be a lot and writes to screenings will be less. Each time we get a new screening, we add it to the count
   - Fan out .. (Do many tasks to represent one logical tasks) [Screenshot]
     - on Reads
       * Every read needs to read from several different locations
     - on Writes
       * Every write translates into writes to several documents
       * By fanning out on writes, the read does not have to fan out anymore as the data is pre-organized at write teim
       * Why use fan out on writes?
         · If the system has plenty of time when the information arrives compared to the acceptable latency of returning data on read operation, then preparing the data at write time makes a lot of sense.
         · Note that if you are doing more writes then reads so the system becomes bound by writes, this may not be a good pattern to apply.

    - A good example for this pattern is the social networking site for sharing photos
        * Copy photos that the user follows onto their profile, so that each time they load their profile it does not have to fetch all the photos from all the users they follow.

- Roll Up Operations
    - Typically in reporting of financial data
    - Running a group operation
    - Example: We have wine types which contains the type and the country. We may want to look at it by a specific category, such as country. It may make sense to then create collections of this aggregated view

When should we use computed pattern?

- Overuse resources (CPU)
- Reduce latency for read operations (If you have long read operations that depend on complex operations)

Summary: Problem:

- Costly computation or manipulation of data
- Executed frequently on the same data, producing the same result

Solution

- Perform the operation and store the result in the appropriate document and collection
- If need to redo the operations, keep the source of them

Use Case Examples

- IOT
- Event Sourcing
- Time Series data
- Frequent Aggregation Framework queries

Benefits and Tradeoffs

- Read queries are faster

- Saving on resources like CPU and Disk
- May be difficult to identify the need
- Avoid applying or overusing it unless needed (may add unwanted complexity)

3.6.1 Bucket Pattern

It is important to find the middle solution. In the case where you have a 10 mil temperature sensors that sends the data to the db every minute. We can't store each of these in it's own document because you will just have way too many documents and it will become unmanageable. If we keep 1 document per device it may reach the 16MB quickly, and become unmanageable.

So what do we do?

Perhaps create one document per device per day. Can also have one array per hour on this document. Or one document per device per hour? Then we have smaller documents and can use a sub document with the temperature data instead of an array

The concept of grouping information together is called bucketing. Bucketing is often done with a date, which makes it easy to delete archive data.

Gotchas with Buckets:

- Random insertions or deletions in buckets, if you need to frequently access data in buckets to insert or delete, this may not work
- Difficult to sort across buckets
- Ad hoc queries may be more complex, again across buckets
- Works best when the "complexity" is hidden through the application code

Summary:
Problem

- Avoiding too many documents or too big documents
- A 1-to-many relationship that can't be embedded

Solution

- Define the optimal amount of information to group together
- Create arrays to store the information in the main object

- It is basically an embedded 1-to-many relationship, where you get N documents, each having an average of Many/N sub documents

Use cases examples

- IoT
- Data warehouse
- Lots of information associated to one objects

Benefits and trade-offs

- Good balance between number of data acces and size of data returned
- Makes data more manageable
- Easy to prune data
- Can lead to poor query results if not designed properly
- Less friendly to BI tools as you will need to know the schema to be able to form your queries properly

3.6.2 Schema Versioning Pattern

Need to alter the schema a some point Have "$\text{schema}_{\text{version}}$" field in each document.

Application lifecycle for this change:

- Modify Application
  - Can read/process all versions of documents
    - Have different handler per version
    - Reshape the document before processing it
- Update all Application servers
  - Install updated application
  - Remove old processes
- Once migration completed
  - remove the code to process old versions

Document Lifecycle

- New Documents:
  - application writes them in latest versions
- Existing documents A use updates to documents
  - to transform to latest version
  - Keep forever documents that never need an update

  B . . . or transform all
  - documents in batch
    - no worry even if process takes days

Summary: Problem

- Avoid downtime while doing schema upgrades
- Upgrading all documents can take hours, days or even weeks when dealing with big data
- don't want to update all documents

Solution:

- Each document gets a "$\text{schema}_{\text{version}}$" field
- Application can handle all versions
- Choose your strategy to migrate the documents

Use case examples

- Every app that use a database, deployed in production and heavily used
- System with a lot of legacy data

Benefits and Trade-Offs:

- No downtime needed
- Feel in control of the migration
- Less future technical Debt
- May need 2 indexes for same field while in migration period

3.6.3 Tree patterns

For Hierarchical data, with direct relationship between parent and child Answers these questions:

1. Who are the ancestors of Node X?
2. Who reports to Y?
3. Find all nodes that are under Z?
4. Change all categories under N to under P

Model tree structures:

- Parent References Document contains a parent field with reference to its parent Good for question 2 and 4

```
{
    name: "office",
    parent: "swag",
}
//example query to get all ancestors
db.categories.aggregate([{
    $graphLookup: {
        from: 'categories',
        startWith: '$name',
        conncetFromField: 'parent',
        connectToField: 'name',
        as: 'ancestors'
    }
}])
```

- Child References Array of children Good for question 3

```
{
    name: "office",
    children: ["books", "stickers"]
}
```

- Array of ancestors Use an ordered array Good for question 1,2,3

```
    {
        name:  "Books",
        ancestors: ["Swag", "Office"]
    }
```

-Materialized Paths (modification of ancestors) Use a string value with ancestors with some separator Can only use if you need to know the ancestors of X Only good for question 1

```
{
    name: "Books",
    ancestors: ".Swag.Office"
}
```

Can use a single index with regular expression to find a path

```
//immediate ancestor of Y
db.categories.find({ancestors: /\.Y$/})

//if descends from X and Z
db.categories.find({ancestors: /^\.X.*Y/i})
```

Summary: Problem

- Representation of hierarchical structured data
- Different access patterns to navigate the tree
- Provide optimized model for common operations

Solution

- Different patterns
  - Child Reference
  - Parent Reference
  - Array of ancestors
  - Materialized Paths

Use Cases Exampe

- Org Charts
- Product Categories

Benefits and Trade-Offs

- Child REference
  - Easy to navigate to children nodes or tree descending access patterns
- Parent Reference
  - Immediate parent node discovery and tree updates

3.6.4 Polymorphic Pattern

Can group things by either things they have in common or by their differences E.g. if we have a customer and food from mexico and a customer and food from canada, we will probably group it by customer.

Useful when you want to create a single view. Last name and surname is the same field but different names, merge into one name.

3.6.5 Approximation pattern