TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Giorgi Khundzakishvili 182474IVSM

GREEN COMPUTING BASED SOLUTION FOR MONGODB ADMINISTRATION

Master's Thesis

Supervisor
Sadok Ben Yahia
Professor

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Giorgi Khundzakishvili 182474IVSM

KESKKONNASÄÄSTLIK LAHENDUS MONGODB HALDUSEKS

Magistritöö

Juhendaja
Sadok Ben Yahia
Professor

Tallinn 2020

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author:	Giorgi Khundzakishvili
Date:	14.05.2020

Abstract

MongoDB is a modern NoSQL, document-oriented database. Its schema-less nature has many advantages as well as disadvantages. In this thesis, we propose solutions to the problem related to the efficient discovery of the fields to be indexed. The process is based on the initial query set and is shown to equate the extraction of a minimum transversal of a hypergraph. The problems are solved by considering the impact on the environment. The impact on the environment is assessed using the benchmarks, which monitor the overall efficiency throughout the queries. Besides, the implemented algorithm of index field set discovery employs known practices and knowledge from non-relational databases for better performance and efficiency. The solutions are the building blocks for various applications including production environments, data warehouses, and large scale systems.

Annotatsioon

MongoDB on kaasaegne NoSQL, dokumentidele orienteeritud andmebaas. MongoDB skeemideta andmestruktuuril on palju eeliseid ja puudusi. Selles lõputöös pakume välja lahendused tõhusamaks indekseeritavate väljade tuvastamiseks. Toiming põhineb esialgsel päringukogumil ja võrdub minimaalse hüpergraafi läbimisega. Probleemide lahendamisel võetakse arvesse mõju keskkonnale. Keskkonnamõju hinnatakse võrdlusaluste abil, mis jälgivad päringute üldist tõhusust. Lisaks, rakendatud indeksväljade komplekti tuvastamise algoritm kasutab mitte-relatsioonilistest andmebaasidest teadaolevaid meetodeid ja teadmisi, et saavutada parem jõudlus ja efektiivsus. Lahendused on ehituskivid erinevate rakenduste, sealhulgas tootmiskeskkondade, andmeladude ja suuremahulistele süsteemidele.

List of abbreviations and terms

SQL	Structured Query Language
RAM	Random-access memory
CPU	Central Processing Unit
DS	Disjunctive Support
MT	Minimal Transversal
CM	Current Minimum
OS	Operating System

Table of Contents

List of Figures

List of Tables

1. Introduction

MongoDB (www.mongodb.com) is a document-oriented database, which is at the top of the charts of non-relational databases. The increasing trend of using MEAN (MongoDB Express Angular NginX) Stack for modern applications, made MongoDB a leading NoSQL solution. NPM registry (npmjs.com) shows this trend quite clearly. As of October 2019, MongoDB driver for NodeJS has hit 1.2M downloads per week. There has been a lot of early adaptations of the technology itself. The biggest challenge for the early adopters was a transition from Structured Query Languages to NoSQL.

MongoDB has become very appealing for new technology lovers. However, many of them anticipated the results of using schema-less design after years of experience in SQL. In Bonnet et al. (2011), the authors talk about the weaknesses of the MongoDB database, one of them is familiarity with already well-developed SQL database solutions. The database has no schemas, there is a freedom to create any document structure as long as it represents a valid BSON document. Getting started with MongoDB might seem a very easy task, but as time passes and the complexity of database increases, flaws of unstructured documents show off their weaknesses. It is becoming easier and easier to add more fields since documents do not follow schema. An increased amount of fields can be hard to maintain and one more thing that is hard to maintain is performance. Querying using different fields can yield different execution times. Thus, having many fields and many different combinations of fields in queries, it's not easy to optimize all of them while maintaining the overall performance.

So one of the biggest problems, caused by the complexity of the schema-less design, we cover in this thesis, is the difficulty to choose the right set of fields to index. This is based on our current test database usage, but it is generalized for different databases. We thoroughly discuss these issues later in this chapter.

1.1 Research

This is a case study, dealing with problems of MongoDB with datasets mimicking the real-world scenarios. We will be using a quantitative approach During the research, we will try out mainstream tools and record the results of exploration. We will concentrate on the details that are beneficial for the study and propose improvements over the number of different cases and implementations. We will deal with finding ways to improve different aspects such as performance, runtime, energy consumption.

We answer the following questions:
RQ1. How to improve the efficiency of MongoDB by taking energy consumption into consideration?
RQ2. How to decrease the complexity of indexing while optimizing the query execution?

1.2 Non-relational and NoSQL

The last decade was very active in the adaptation of different technologies. One of the biggest trends in Databases is Non-relational Databases. Non-relational is the type of database that does not use tables and has storage strategy best fitted for its purpose. Examples of non-relational structures can be an entry that contains only keys and values, or values representing a combination of multiple key-value pairs. But what is NoSQL? NoSQL - database type, that does not rely mainly on SQL language and has its own syntax. We might meet the same database management system mentioned as Non-relational and NoSQL in different sources. Actually, those two can be the same. The database might be NoSQL and support structured query language, although this does not mean that the core works the same way as RDBMS (Relational database).

1.3 SQL vs NoSQL, which one is better?

SQL databases have been around for a long time. They are stable, have a schema, data can be retrieved, and mixed from different tables easily (Joins). However, Big data tends to have different needs from the databases. Lately, large applications have been adapting NoSQL solutions. The main reason is data inconsistency. In some cases, you need to store and fetch data, no matter what, even if it contains extra unnecessary fields and data structures (for the specific jobs). Literature Jung et al. (2015) suggests, that flexibility and scalability are the key features that make NoSQL (specifically MongoDB in this case) appealing alternative to well-established SQL.

So, anyway, should we abandon RDBMS and choose NoSQL? There is no necessity to fully switch to it. Those two different types of databases can be used together as well ((Ongo and Kusuma, 2018)). However, in Chickerur et al. (2015), authors underscore the fact that in some scenarios, such as big data applications, NoSQL databases can work better than some implementations of SQL databases. By and large, We do not have to switch anywhere as long as current technology choices fulfill the application requirements.

1.4 Schemaless Design

Since MongoDB does not have schemas, it is very flexible in terms of entry structure. However, everything comes with a cost. Having no schema can easily become a time-waster. For example, it is hard to find what fields/keys are used in the collection. There has been research carried out for schema discovery (Vokorokos et al. (2017)) and validation (Prabagaren (2014)). The first research is dedicated to finding the schema details and improve the state of the art with new tools. However, in Prabagaren (2014), the authors chose a different approach, relying on the fact that if we need validation, then we should have rules (similar to the schema) in place. This option does not cover the case, when the database is already deployed and populated with data, however, it makes sure that data will have the desired structure in the future. Our research is oriented to the first case because we examine databases with complex collections.

1.5 Energy Efficiency

The energy consumption has become a topic that shifts the industry from energy-hogging infrastructures to more environment-friendly solutions. In Song et al. (2019), the authors mention that reducing energy consumption leads to less performance. However, they also claimed that this is not true in every case. There are different ways to achieve better energy efficiency. They suggest, that we can divide those methods into two categories, hardware optimization, and Software optimization. Former is dedicated to dynamically switching hardware components based on usage (e.g. Turn on and off the machine or its components when underutilized) (Orgerie et al. (2014), Rossi et al. (2017)). The latter can be used in monoliths, single machine setups Zeng et al. (2005), although it is quite popular in distributed systems as well Kansal and Chana (2016). Since we try to improve the overall query execution efficiency, we are dedicated to the software optimization method.

1.6 Hypergraphs

Definition 1.6.1. Hypergraph (Trabelsi et al., 2019)
Let the couple $\mathcal{H} = (\chi, \xi)$ where $\chi = \{x_1, x_2, x_3, \ldots, x_n\}$ is finite set and $\xi = \{e_1, e_2, \ldots, e_m\}$ is finite family of χ. H is a hypergraph such that :

$$\cup_{i=1}^{m} e_i = \chi \mid \forall i \in \{1, \ldots, m\}, e_i \neq \emptyset \tag{1.1}$$

The elements $x_1, x_2, \ldots, x_n$ are called *vertices* or *nodes*, whereas the sets $e_1, e_2, \ldots, e_m$ are the hyperedges of the hypergraph.
Figure 1.1 illustrates a hypergraph $\mathcal{H} = (\chi, \xi)$ of order 8 and size 15 such that $\chi = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $\xi = \{\{1, 2\}, \{2, 3, 7\}, \{3, 4, 5\}, \{4, 6\}, \{6, 7, 8\}, \{7\}\}$

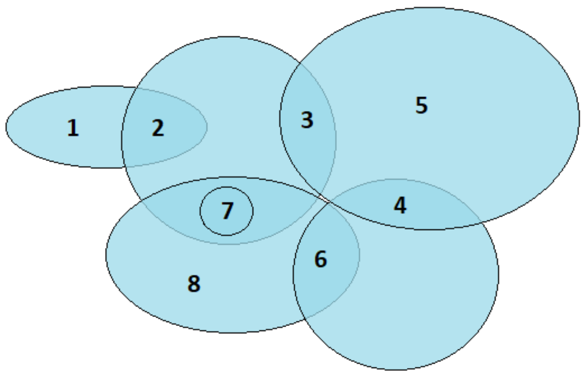


Figure 1.1: Example of a hypergraph

A hypergraph $\mathcal{H} = (\chi, \xi)$ can be represented by an incidence matrix which rows represent the vertices and columns represent the hyperedges of $\mathcal{H}$ in order that:

$$\mathcal{IM}_{\mathcal{H}}[e_i, x_j] = \begin{cases} = 1 & \text{if } x_j \in e_i \\ = 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

Figure 1.2 depicts the matrix incidence of the hypergraph given by Figure 1.1.

$$\begin{array}{c|cccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ \hline e_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ e_2 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ e_3 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ e_4 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ e_5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ e_6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figure 1.2: Incidence matrix of the hypergraph given by Figure 1.1

Definition 1.6.2. Transversal (Trabelsi et al., 2019)
Let $\mathcal{H} = (\chi, \xi)$ be a hypergraph. A set $\mathcal{T} \subset \chi$ is a transversal of $\mathcal{H}$ if it meets every hyperedge, which is described by the following expression:

$$\forall i \in \{1, 2, \dots, m\}, \mathcal{T} \cap e_i \neq \emptyset \tag{1.3}$$

Extracting hypergraph minimal transversals has been shown to have myriad of applications in computer science, e.g., database theory, logic, and AI, to cite but a few Jelassi et al. (2015).

Definition 1.6.3. Minimal transversal (Trabelsi et al., 2019)
A transversal $\mathcal{T}$ is minimal if it doesn't exist $\mathcal{T}^{'}$ a subset of $\mathcal{T}$ where $\mathcal{T}^{'}$ is a transversal.

$\mathcal{M}_{\mathcal{H}}$ denotes in the sequel the set of minimal transversals defined on $\mathcal{H}$. From the hypergraph, given in Figure 1.1, the following set of minimal transversals is drawn: $\mathcal{M}_{\mathcal{H}} = \{\{1,4,7\},\{7,4,2\},\{7,6,3,1\},\{7,6,3,2\},\{7,6,5,1\},\{7,6,5,2\}\}$. The minimal cardinality of transversal is denoted by $\tau(\mathcal{H})$ of the hypergraph $\mathcal{H}$:

$$\tau(\mathcal{H}) = \{min|\mathcal{T}|, \forall \mathcal{T} \in \mathcal{M}_{\mathcal{H}}\} \tag{1.4}$$

Therefore, the smallest minimal transversal of the hypergraph $\mathcal{H}$ shown in figure 1.1 includes, in terms of cardinality, three vertices, i.e. $\tau(\mathcal{H}) = 3$.

The minimal traversal can be defined by the concept of an *essential itemset* that could be extracted from an extraction context. In the following, let us start by highlighting the close connection between the latter and a hypergraph.

Definition 1.6.4. Extraction context (Trabelsi et al., 2019)
We can formally represent a database as an extraction context $\mathcal{K} = (\mathcal{O}, \mathcal{I}, \mathcal{R})$. It is a triplet where $\mathcal{O}$ and $\mathcal{O}$ are respectively finite sets of objects(transactions) and attributes(items) and $\mathcal{R}$ is a binary(incidence) relation between $\mathcal{O}$ and $\mathcal{I}$ (i.e $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{I}$). Taking as an example the hypergraph of Figure 1.1, the corresponding extraction context is shown in Table 1.1.

Definition 1.6.5. Disjunctive support of an itemset. Trabelsi et al. (2019)
The disjunctive support of an itemset X is the number of transactions that contain at least one of the items of X.

$$Supp(\vee X) = |\{o \in \mathcal{O} | (\exists i \in \mathcal{I}, (o, i) \in \mathcal{R})\}| \tag{1.5}$$

An essential itemsets is characterized by a disjunctive support and is defined as follows.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
e_1	×	×						
e_2		×	×					
e_3			×	×	×			
e_4				×		×		
e_5						×	×	×
e_6							×	

Table 1.1: Extraction context of the hypergraph depicted in Figure 1.1

Definition 1.6.6. Essential itemset (Casali et al., 2005)

Let $\mathcal{H} = (\chi, \xi)$ be a hypergraph and $X \subseteq \chi$. X is said to be an essential set of vertices if and only if:

$$Supp(\vee X) > max\{Supp(\vee X \setminus \{x\}) | x \in X\} \tag{1.6}$$

It is important to remind that the essential itemsets extracted from a hypergraph satisfy the order property of essential itemsets. If I is an essential itemset, then $\forall$ $I_1 \subset \mathcal{I}$, I_1 is also an essential itemset. Thus, the notion of minimal transversal can be redefined through the support of a set of vertices and the notion of the essential itemset (Jelassi et al., 2014; Ghabry et al., 2018), according to proposition 1.6.1.

Proposition 1.6.1. *Minimal Transversal (Trabelsi et al., 2019)*
A subset of vertices $\mathcal{X} \subseteq \chi$ is a minimal transversal of the hypergraph $\mathcal{H}$, if $\mathcal{X}$ is essential and if its disjunctive support is equal to the number of hyperedges of $\mathcal{H}$, i.e., X is an essential set such that $Supp(\vee X) = |\xi|$.

In our case, hypergraphs play a huge role in the optimal index set generation algorithm, because the algorithm will be based on hypergraph theory. We can assume that the edge is a query executed by the user, and the node is the field used in a query. As a result, we get a query set in the form of a hypergraph structure.

2. Related Work

2.1 Schema Discovery

In Vokorokos et al. (2017), work is dedicated to gain performance improvements over existing solutions. However, as we know, lots of resources are spent during schema analysis. So, our research is oriented to reduce resource utilization during the discovery of schema-based on the needs of the application. For example, if we work on enterprise applications, we need a schema to be relevant to the latest changes, but in the case of a data warehouse, every document is important. We can use this information to optimize the process of schema discovery.

2.2 Creating Indexes

Creating an index is not a problem in MongoDB. It supports different types of indexing strategies. Although, there is a problem regarding the size and amount of indexes. In Kanoje et al. (2015) authors list indexing capabilities as a disadvantage because indexes take a lot of RAM and can cause serious performance issues. At the same time, in Mahajan and Zong (2017), research shows that index is beneficial for both, performance and energy efficiency. We aim to create indexes based on usage patterns found in queries executed on the given database. After that, extracting fields used in queries and present it as a hypergraph.

At this stage, edges are queries, and nodes represent fields accordingly, in a hypergraph. Then, we proceed with minimal transversals finding (Berge (1984)).

2.3 Optimization

Obviously, the algorithms have to be optimal, because the data we process can go beyond multiple gigabytes, thus we need to use the right strategy to develop algorithms discussed in sections 2.1 and 2.2. In Vokorokos et al. (2017) the authors describe the differences between available methods of Mongodb data processing. They try to overcome performance problems with the use of the aggregation framework, MapReduce framework, and local algorithms. They narrowed down the options to local algorithms and aggregation framework because results given by MapReduce were too slow to keep for implementation.

We are going to use the same approach and use the local algorithms. Besides, we will cover "find" queries and different operands that are being used during querying/filtering the data. Although we will not cover aggregations, since some of the pipeline stages get quite complicated and need manual optimization, or can be impossible to optimize using indexes[1].

2.4 Measuring Energy Efficiency

In Song et al. (2019), the authors consider different factors for measuring such as CPU utilization, idle time, RAM, disk, network. Measurements are then converted in the same units to make them easy to cooperate, called *Resource Quantity.* The latter is "a measurement of the number of computing resources allocated to a task or consumed by a task. Quantity of both allocated and consumed resources is time relevant."

On the other hand, In Mahajan and Zong (2017), the authors propose three metrics: Speedup, Powerup, and Greenup.

$$\text{Speedup} = \mathrm{T}_{\phi}/\mathrm{T}_{\mathrm{o}}$$
$$\text{Greenup} = \mathrm{E}_{\phi}/\mathrm{E}_{\mathrm{o}}$$
$$\text{Powerup} = \mathrm{P}_{\mathrm{o}}/\mathrm{P}_{\phi}$$

T_{ϕ} is the execution of query before optimization, while T_{o} is the execution time of optimized one. Same way, E_{ϕ} the energy consumption before optimization and E_{o} after. Also, "Powerup is defined as the ratio of the average power consumed by the optimized query over the average power consumed by the un-optimized query".

They are using given metrics to evaluate the impact of optimizations on performance, power, and energy efficiency. They Have highlighted 4 different cases:

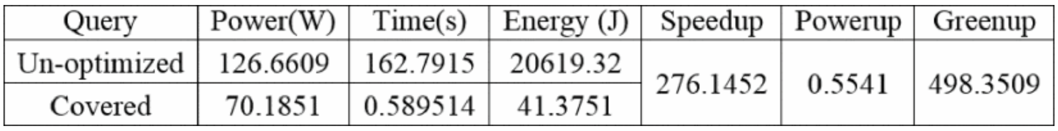
- Speedup > 1, Powerup < 1, and Greenup > 1 (Case 1): Optimized version performs better while consumes less power. We do not lose the performance, which is the best case we can achieve.
- Speedup > 1, Powerup $= 1$, and Greenup > 1 (Case 2): The power consumption stays the same as before. The optimization takes care of reducing execution time itself.

[1]Aggregation Pipeline - MongoDB Manual: Pipeline Operators and Indexes `https://docs.mongodb.com/manual/core/aggregation-pipeline/#pipeline-operators-and-indexes` - accessed 25-April-2020

- Speedup > 1, Powerup > 1, and Greenup > 1 (Case 3): The optimized version has significantly higher performance while it consumes more power. However, the increased energy consumption comes with higher benefits and we still save energy.

- Speedup < 1, Powerup < 1, and Greenup > 1 (Case 4): The performance is reduced, but the power consumption is significantly lower. The result is less performance, but higher efficiency which contributes to energy usage reduction.

They have great results in Covered queries[2]. A covered query is using the full potential of an index. Since it contains all the fields requested by the query, we are able to avoid scanning the documents.



Query	Power(W)	Time(s)	Energy (J)	Speedup	Powerup	Greenup
Un-optimized	126.6609	162.7915	20619.32	276.1452	0.5541	498.3509
Covered	70.1851	0.589514	41.3751			

Figure 2.1: Results of covered queries presented in Mahajan and Zong (2017)

From Figure 2.1, we see that accessing just index, results in great performance and energy efficiency boost. However, covered queries are very specific, since not all the queries can be optimized this far because in many cases, access to the collection is necessary.

We aim to get better results by optimizing the resources. Since Software optimization can be challenging, there is still room for improvement. Our tests will not include idle, allocated resource utilization as in Song et al. (2019), but the approach will consider overall utilization during execution. We are going to test the effectiveness of suggested indexing by running a query set and compare mean measurements with and without indexes. We will run queries a couple of times to get a more precise result and exclude the possibilities of external interference in the form of either energy consumption, or unexpected resource utilization by the system during testing.

[2]Query Optimization - MongoDB Manual: Covered queries `https://docs.mongodb.com/manual/core/query-optimization/#covered-query` - accessed 30-April-2020

3. Automated indexing - Algorithm

3.1 Input Format

```
{
    "_id": 1192,
    "name": "John",
    "age": 37,
    "additionalInfo": {
        "hobby": "Collecting coins"
    }
}
```

Listing 3.1: MongoDB document example

```
db.collection.find({
    age:37,
    additionalInfo: { hobby: "Collecting coins" }
})
```

Listing 3.2: MongoDB query example

The part we concentrate on is the parameter of the ”find” method. The given parameter is the special type of document, called *query filter document*. The given document has fields ”age” and ”additionalInfo”. At this point, we are concentrated on fields, since fields are the entities that are being indexed. If we look at ”additionalInfo”, it has a value which is a subdocument. Subdocuments are the documents that are nested into another document as a value. The subdocument has another attribute ”hobby”.

We might think that we have three different fields, but since ”hobby” is under ”additionalInfo”, MongoDB will look for ”additionalInfo.hobby”.

As a result, we get three fields, but in reality, we have only two comparisons, because the last two fields point to the value inside the nested document. Worth noticing that mognoDB allows the given query to be written like this:

```
db.collection.find({
    age:37,
    "additionalInfo.hobby": "Collecting coins"
})
```

Listing 3.3: MongoDB query example

We see, that in listing 3.3 there is a dot between ”additionalInfo” and ”hobby”, it is called dot notation. Usage goes like this:

```
<embedded document name>.<embedded document field>
```

Also, when using a dot for joining two attributes, we are required to use double quotes around it.

So for example1, we ended up with two fields that have to match during find operation: ”age” and ”additionalInfo.hobby”.

```
db.collection.find({
    name: "John",
    age:{ $gt:35 }
})
```

Listing 3.4: MongoDB query example 2

The example in listing 3.4 introduces ”query operators”. Query operators help us to specify query conditions. In this case, age has a subdocument as a value, that has field ”\$gt”. ”\$gt” is an operation that matches every element that is greater than its value (in this case 35). So, instead of looking for a specific value, we match the range with a given condition. Despite that, we don’t index query operators, since they are special entities and don’t represent the part of the actual document(s) that has to be fetched.

As a result, in example2 (Listing 3.4) we are looking for two fields: ”name” and ”age”. This approach gives us a way of extracting a set of fields from the query. Every query is represented by a set of fields. The list of query representations is the first stage input for our algorithm.

The second stage consists of given steps:

1. Extract the fields from all the query representations
2. Create a list of unique elements from step 1
3. Enumerate list of unique elements

4. Match number of enumeration with the corresponding field in the list of query representations

The result of stage 2 is our final input for the algorithm that is looking for minimal transversals.

3.2 Theoretical aspects of minimal transversal

Let's assume that we already have the result of stage 2 from input parsing, such that each field is already represented with a unique number

Our example input will look like this:

```
1 2
2 3 7
3 4 5
4 6
6 7 8
7
```

In this example, we have 6 queries and 8 fields. each row is a representation of the query, and each number represents the field.

Another assumption is, that we know how many unique fields we have, otherwise first we have to identify all the fields used in queries, and then proceed further.

The first step will be representing the input in form of a hypergraph, but instead of chaining them and creating a graph structure, we represent our hypergraph in form of the table consisting of 1 and 0, where a row represents a query, a column represents a field.

1	1	0	0	0	0	0	0
0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	0
0	0	0	1	0	1	0	0
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0

So, we have value for every combination of tuples like <field, query> on a crossing of the corresponding field, and query. In our hypergraph, the field is a node, the query is an edge.

Let's explore the example further and explain the first row of a matrix: 1 1 0 0 0 0 0 0 The first two numbers are "ones" and describe the presence of 1 and 2 nodes in the given edge (the query representation). The rest of the numbers are "zeros" and correspond to the absence of the field that corresponds to the given column.

At this point, we already have the necessary information to start computing minimal transversals.

3.2.1 Calculating Disjunctive support of individual nodes

Disjunctive support is the number of edges being covered by the element. The disjunctive support in our case can also be represented by the sum of elements of the corresponding column in our two-dimensional array. However, this condition changes as we go further. Because of that, let's say that we have the list of fields and we need to calculate disjunctive support of given fields. The disjunctive support can be represented as the number of edges that are being covered by at least one of the nodes in the given list.

So generalizing calculation of list of items to a single item can be done by representing node as a list of a single element.

Let's call the list of such nodes the "item".

So as an example, we can represent the table of an item and its disjunctive support:

1	1
2	2
3	2
4	2
5	1
6	2
7	3
8	1

We should point out, that in the case when disjunctive support of single item equals the number of queries, it is minimal transversal. We should keep it in the list of minimal transversals, ignore it in upcoming operations, and continue with remaining items. In our case, we do not ignore any of the items, since none of them have disjunctive support of 6.

3.2.2 Calculating a maximal clique

Now we have to find the items such that a combination of their nodes does not represent space where minimal transversal can be found. Let's call it a maximal clique. For more clarity, we start building a maximal clique with item 1. As we know, at this stage item consists of only one node, so we don't address nodes individually for now. Item 1 has DS (disjunctive support) of 1. 1 is less than 6 (amount of queries that should be covered by combination). Adding 2, DS = 2 since 1 and 2, at least one of them appears in 2 queries, max-clique becomes 1,2.

```
Add 3 -> Max clique {1,2,3}, DS = 3;
Add 4 -> Max clique {1,2,3,4}, DS = 4;
Add 5 -> Max clique {1,2,3,4,5}, DS = 4;
```

After adding 5, DS has not changed, because 5 is covered by third query, which was also covered previously by 3 and 4.

```
Add 6 -> Max clique {1,2,3,4,5,6}, DS = 5;\\
Add 7 -> Max clique {1,2,3,4,5,6,7}, DS = 6;\\
```

At this point, we reached 6, which is the maximum amount for DS, meaning that a given set of items can not be max-clique because it contains subset that leads us to minimal transversal. So we remove 7 and keep it in the leaders' list. A leader is an item that can be extended by further exploring items that are a superset of a given item.
Next, we add 8 to max clique and DS equal to 5, max clique = $\{1, 2, 3, 4, 5, 6, 8\}$, leaders = $\{7\}$. Calculating the maximal clique has come to the end. We can notice that we won't be able to find any subset of max clique such that combining them will give us minimal transversal(s).

3.2.3 Build helper table for each leader

The helper table is a two-dimensional array. It consists of N+1 rows, where N is the number of elements in a leader. In our case, we have only one leader, which is 7, and our helper table will have 2 rows. The first row contains all the query numbers, which do not contain 7. The second row contains all those query numbers, where 7 is included. Building the helper table for item 7 will go like this:

Build array, consisting of two empty arrays.

```
[
    [],
    []
]
```

Take query number 1. As we see, 7th column has 0, we append 1 in first row.

```
[
    [0],
    []
]
```

Query No 2 - 7th column is 1, we append 2 in the second row.

```
[
    [0],
    [1]
]
```

... Query No 6 - 7th column - 1, append 2 in second row.

Final result:

```
[
[1,3,4],
[2,5,6]
]
```

We can see clearly, that we do not encounter 7 in 1,3,4 queries and we encounter 7 in 2,5,6. We finished building the table.

3.2.4 Exploring the leaders

Next iteration is initialized for every leader separately.

Our list of leaders consists of only one item - 7, so we continue with this item.

We need to somehow proceed with expanding our item with more nodes. The method we will use, we call *Stick and Compute*. This means we create a new item by combining elements from two items from the previous stage. We already know the elements of the previous stage, both leaders and max-clique items. We create a new list with leaders from the leaders' list that has an index greater than the leader we are exploring currently and items from max-clique. We call this array "neighbors". Our list of neighbors, in this case, is the same as max-clique from the previous stage, because we have only one leader at this moment.

$$\text{neighbors} = \{ 1, 2, 3, 4, 5, 6, 8 \}$$

We stick current leader with each item from neighbors. As a result, we get new items with different set of elements:

$$\{ 71, 72, 73, 74, 75, 76, 78 \}$$

We should notice, that 71 is shorthand for the item that contains 7 and 1 together. So, we write a set of [7,1] as an item named 71 (separator free form) for clarity. Taking the first item from a newly created list of items - 71. Calculating disjunctive support. We see that DS is 4.

It has two direct parent items - 7 and 1. Their disjunctive support is 3 and 1 respectively. We keep item since DS of 71 is larger than its both parents' DS separately (Definition 1.6.6). Same for 72, 73, 74, 75, 76.

Now let's see item 78. DS for this item is 3. DS(78) = DS(7) = 3 and we remove it from future operations, since "essentiality condition" is violated, which translates into the fact, that disjunctive support of item should be higher than its parents' (This does not spread on single node items since they are starting points).

Also, worth noting that none of the items above have DS of 6, thus, we do not have minimal transversals during this iteration.

We are left with the given items (DS next to them):

71	4
72	4
73	4
74	5
75	4
76	4

We continue again with creating maximal clique, resulting with:

$$\text{max clique} = \{\ 71,\ 72,\ 73,\ 75\ \}$$
$$\text{leaders} = \{\ 74,\ 76\ \}$$

The reason is the following: if we extract the set of nodes from all the items in the max clique, we get 1,2,3,5,7. DS of this set is lower than 6 and we won't be able to construct minimal transversal from it.

Before going to the next iteration, we have to check whether the item is fulfilling the minimality test.

3.2.5 Minimality Test

An item is minimal if every node in item fulfills the following condition: there is at least 1 query in query set which contains only this node from the given item. This rule is the control for such item, that might have indirect parent, that has the same disjunctive support as this element. We will discuss such cases separately.

We have created a helper table for leaders during the first iteration. We assume, that we have access to the leader's helper table (the leader for which this iteration was initialized).

Helper table looks like this:

0	1, 3, 4
1	2, 5, 6

Now, by introducing a new node in the item, we have to check this node against this table:

For every query from 0 index array, we test that node is included in the query. It is required that at least one of the queries from 0 index array contains the node.

Let's check queries from the array with index 0 against node 4:

```
query 1: does not contain
query 3: contains
```

We stop checking here since we found at least one query that contains the item.

For every query from the array with an index greater than 1, we test that node is not included in the query. In this case, the requirement is, that at least one of the queries from such an i-th array does not contain a node.

Check of the queries from an array on index 1 against node 4:

```
query 2: does not contain
```

We stop checking.

As we see, we might not have to check every query, since we stop testing as soon as we fulfill the condition for at least 1 query.

We proved that the element fulfills the minimality test.

Since the element is minimal, we can proceed to the next iteration with 74, but first, we need to update the helper table.

Update helper table

The process is very similar to Minimality Check, except, we extend helper table to consider those elements that are being covered only by second node in item (node 4):

0:	1,3,4
1:	2,5,6
2:	

At the beginning of the update, array with index 2 does not contain any element.

Let's start with the array from helper with index 1. We have to only keep the queries, which does not contain node 4:

```
query 2: does not contain, we keep
query 5: does not contain, we keep
query 6: does not contain, we keep it as well
```

So far, no changes in this array. Intermediate result of helper table is as follows:

0:	1,3,4
1:	2,5,6
2:	

Now let's have a look at an array with index 0. This is a special case since this array contains those queries, that previously contained none of the nodes from the item. Every time we add a new node in the item, this array should be reduced in size (since every new leader covers more queries than its parent). So in our case, the queries that contain node 4, are removed from this array and being appended in newly added array in helper table with index 2:

```
1: does not contain, we keep,
3: contains, we move to a new array
4: contains, we move to a new array
```

Final result of the updated helper table (for item 74):

0:	1,3,4
1:	2,5,6
2:	3,4

For the next iteration (3-rd level), we have leader 74. The neighbors: 76, 71, 72, 73, 75. Here, 76 is the leader with greater index than 74 (from the previous iteration), and the rest of the items are from the maximum clique.

The "sticking" of the items beforehand was trivial, we were getting to items and joining nodes into one item. We should notice, that if two items contain the same node, we count it once in the result item since we have a set of elements. So 74 and 76 after sticking will become 746.

We can also formulate this operation differently: when we stick two items to each other, we know that the parent for both of the items is the same, so the nodes that differentiate both from each other, is the last added node in each item. In this case, those are 4 and 6. So We can say that "sticking" operation is the same as joining current leader nodes (74) with last added node of the neighbor (6).

Generated items by sticking 74 and neighbors:

$$\{746, 741, 742, 743, 745\}$$

Disjunctive support for New items:

746	5
741	6
742	6
743	5
745	5

We can already see that 741 and 742 have Disjunctive support of 6. This means those two items can be minimal transversals. This is crucial because the item might have DS of 6, but it might not pass minimality check.

Checking minimality of 741

This is the first time we encountered a candidate of the minimal transversal. If an item is a minimal transversal, it means that we do not have any queries left without coverage by at least one node. Simply speaking, we should not have any elements in the array on 0 index of helper table, that is not covered by node 1:

```
Query 1: contains node 1.
```

The rest of the arrays in helper table should contain at least one query that is not covered by node 1:

```
Checking array with index 1 (queries - 2, 5, 6):
Query 2: does not contain node 1
We skip the rest

Checking array with index 2 (queries - 3, 4):
Query 3: does not contain node 1
Skipping the rest
```

Since all the queries are covered in the helper table, we can push 741 in the minimal transversal list. The same goes for 742, since covers all the queries in the array with index 0. In addition, at least one query from other arrays where node 2 does not cover the query. We push it to minimal transversals as well.

We are left with { 746, 743, 745 } items. Checking the direct parents of each of them, we will discover, that DS of 745 is equal to DS of 74, so we ignore it.

Now we have to generate maximal clique out of items { 746, 743 }.

We do regular building of maximal clique, getting max clique = 746, 743 with DS of 5. We end building the max clique without any leaders, meaning that we end exploring the branch.

We go back to third level on next iteration where we stopped last time. We generate helper table for 76:

0:	1,3
1:	2,6
2:	4

Now we have leader 76 and neighbors 71, 72, 73, 75. This case is very important, since we had another leader as well, 74, but its index in the list of leaders is not greater than 76. For more clarification, 74 is already used with all the listed elements to form new items, so considering it again will cause duplication of items with the same set of nodes in it, that is the reason why we skipped it this time.

For example, let's assume that we included 74 in neighbors as well. We will get 764 after sticking, which is duplicate to item 746 from previous iterations, and proceeding further with it, will cause exploring the same searching space again.

Back to current iteration, we generate new items by sticking 76 and each item from neighbors and calculate their disjunctive support:

761	5
762	5
763	5
765	5

None of them has DS of 6. Also, all the combinations have DS more than its direct parents'. We switch straight to generating max clique:

```
max clique = { 761, 762 }
leaders = { 763, 765 }
```

Continue with 763.

We generate helper table for 763 by extending item 76 with node 3:

0:	1
1:	6
2:	4
3:	3

Next iteration goes to level 4. We have 763 with neighbors { 765, 761, 762 }

We generate new items with corresponding Disjunctive support:

7635	5
7631	6
7632	6

We see two elements having DS of 6: 7631 and 7632.

We check both with minimality test:

7631 - with helper table of 763 test node 1 - test shows it is passing minimality test. We push 7631 to minimal transversal list. 7632 - with helper table of 763 we test node 2 - item passes minimality test. We push 7632 to minimal transversal list.

We can start building max clique, but we have only one item left - 7635, we add it to the max clique and we do not have leaders to proceed further. We stop this branch here.

Lastly, We have to proceed with 765 We generate helper table for current item 765 by extending item 76 with node 5:

0:	1
1:	2,6
2:	4
3:	3

Generated items: { 7651 7652 }

Both of them have DS greater than 6, and they are minimal. So we push them in the minimal transversal list.

Our final result looks like this:

```
minimal transversals = { 741, 742, 7631, 7632, 7651, 7652 }
```

Sorting minimal transversals:

At this stage, we have the output from the algorithm.

The algorithm gives us an unordered list of items. We should define requirements to choose the best possible minimal transversal.

First of all, having more indexes does not necessarily translate into better performance. In MongoDB, having more than one query gives us possible more two or more query execution strategies.

The worst-case scenario is COLLSCAN - traversing (scanning) collection documents, item by item. Fetching required documents might cause scanning all the items.

Our goal is to minimize those queries that access all the documents directly.

Having an index, we can benefit a lot. An index is a representation of B-trees, that makes it possible to find the necessary element in time complexity of $O(log2(n))$. this means that we can drastically reduce execution time and the number of disk access required to select a subset of documents.

There is one difference. Having an index on multiple fields does not mean that query will use all the possible optimizations at the same time. As usual, each index is a separate strategy. The system is looking for the best approach and using the chosen strategy. When we speak about optimization that should benefit the overall performance, we should know the side effects of using too many indexes. So for most of the cases, having just one index can hugely benefit efficiency. Having too many and the query execution might start to slow down again because of the various reasons including (but not limited to):

- Lack of memory
- Lack of storage

Lack of memory
In MongoDB official manual[1], they recommend, that if we need to have fast processing, we should ensure, that index can fit in memory. Memory is being freed up every time we need more memory for processing or keeping other indexes. If there is no enough memory, we have high utilization of storage, which is not a good sign, since performance can degrade very fast if additional operations (such as reading from slow storage) are needed to search in index.

Lack of storage
In addition, based on the information given by official documentation of MongoDB [2], Creating index means having a duplicate of given information at two different places. The first place is the document itself, second - the index. When it comes to point where you have a significant amount of documents (number goes in tens of millions), the last thing you want to have is using up the storage. More documents mean more spaces needed and since index should include values from the field, even more space is needed in total.

Considering the above reasons, we go for the least amount of indexes to be needed, thus we have to pick the minimal transversals with the minimal amount of indexes.

Example:

```
minimal transversals = { 741, 742, 7631, 7632, 7651, 7652 }
```

Our set contains items with different amount of nodes. First, we have to sort them by the length ascendingly. We see, that this is already done for our set, so we continue by picking the shortest items: 741 and 742.

If we have only one short item, we finish searching, otherwise, we have to pick the most promising in terms of efficiency. The number of executed queries become very important to factor while choosing the best one.

[1]Web page `https://docs.mongodb.com/manual/tutorial/ensure-indexes-fit-ram/` - accessed 24-April-2020

[2]Web page `https://docs.mongodb.com/manual/indexes/` - accessed 24-April-2020

Speaking of efficiency, we have queries that are being run for the average amount of times per time unit. Let's define a time unit to be a month. Each query has a different amount of executions month by month. Let's say that we have an average number of executions per month. This information will help us to find the best option out of multiple cases.

Extending the initial input of queries with the number of executions:

query #	**number of executions**	**query fields**
1	22019	1,2
2	1801	2,3,7
3	29	3,4,5
4	331	4,6
5	12	6,7,8
6	812	7

Now we have to somehow consider execution time to land on the best combination of fields to be indexed. We could just sum the number of executions for the covered queries. The problem is that every minimal transversal is covering all the queries in the list. So, Having this considered, we can still factor without ending up with the same sums for every option.

Start by summing up the number of executions for each of the fields from the node. For example - 741:

```
7 -> 812+12+1801 = 2625
4 -> 331+29 = 360
1 -> 22019
```

Next, we sum up the results of each field: 22019+2625+360 = 25004. Let's call it a score of the minimal transversal.

We do the same for 742:

```
7 -> 812+12+1801 = 2625
4 -> 331+29 = 360
2 -> 22019 + 1801 = 23820
Sum: 2625+360+23820 = 26805
```

We see, that 742 has a higher score than 741. So, our final choice is 742.

We might have questions about why we have decided to count the frequency of each query in the same calculation multiple times. For example, 1801 is repeated in node 7 and node 2 when we calculate the score of 742. The reason is still the possible expansion of strategy choices when we are executing the query. This matters, because we get the chance of landing on better efficiency during query execution.

And since we still create the same amount of indexes, despite the size of the index, the more queries each index can cover, the higher the execution strategy options.

So, the metric for determining the best possible option looks like this:

```
if the number of elements with the smallest MT is 1 -> smallest MT
else -> element where Sum of (
         the sum of query count of queries covered by
                             each item in MT items
     ) is the highest.
```

3.3 Optimize Iterations

In the previous section, we generate a list of minimal transversals, but in reality, we do not use all of them, since we only need the smallest items from the result set. This means, that our algorithm is not very efficient for our requirements. To fix this issue, we can extend our implementation by introducing the "current minimum".

Proposition 3.3.1. *Current Minimum*
The current minimum is the size of the smallest item in a set of minimal transversals during the current iteration of the algorithm.

At the beginning of the algorithm, the current minimum is equal to the number of nodes in the hypergraph. in our case, it is equal to 8. Every time we find an item that is minimal transversal, we update the current minimum with its size. If we limit the exploration and stop when we reach the current minimum, eventually, we reduce the number of required iterations to find the smallest minimal transversals.

Let us bring the example from the previous section. During iteration of 74, after we push 741 in the minimal transversal list, we set the current minimum (CM) to 3, because item 741 consists of 3 nodes. Next comes 742 and we again, set CM to 3. During this iteration, we do not build max clique, since CM is 3 and we already covered all the generated items of size 3 from the branch of item 74. We stop exploring this branch.

After this, we go to the iteration, where the leader is 76. The generated items are 761, 762, 763, and 765. Since none of the items have disjunctive support of 6, we do not have an item that can be minimal transversal. We know that CM equals 3, thus we do not continue building max clique since we do not have to explore items with 4 nodes.

As a result, the largest items we explored are equal to 3 and our list of minimal transversals consists of items with a size of less than 3 accordingly. Different scenarios might have different results, but the moment we find the smaller minimal transversal than the current minimum, we update the current minimum, meaning that our search space is reducing even further. In our example, we found only two minimal transversals and the number of iterations is reduced by 2.

In the next chapter, we will continue with the implementation of the algorithm with and without the current minimum.

4. Implementation

In the last chapter, we have introduced the algorithm of finding an optimal subset of fields for indexing based on the provided query set and frequency of execution for each of them. In this chapter, we go through the implementation and details associated with it. We will go through some optimizations and refactoring.

4.1 Implementation of MT algorithm

First of all, we start by initializing the algorithm itself.

```
const transversals = [];
const currentMinimum;

function execute(data, minimum) {
    const uniques = extractUniqueValues(data);
    currentMinimum = uniques.length;
    const table = createBooleanTable(data, unique);
    initializeRoots(table, unique, minimum);
    return transversals;
}
```

Listing 4.1: Initialization of Searching Minimal Transversals

First, we initialize global, empty array for transversals (this is a final array, which is filled by minimal transversals we find, so wherever we see variable transversals, we use this array) and declare a variable for keeping track of current minimum. Then we declare our starting point, the function "execute". Input for our algorithm is the data in the form of a two-dimensional array and the toggle "minimum". if the minimum is true, we search for current minimums, otherwise, we use the normal version. Each subarray is an edge, each element in subarray is a node. Next, we need to get a list of nodes, function "extractUniqueValues" is used for that. When we get a list of nodes, the current minimum will be set to the length of the given list. Also, we need a boolean table, two-dimensional representation of input data where each row represents query, each column - node. The crossing of row/column is true if given query contains the corresponding node (for this, we use function "createBooleanTable"). The last step is initializing root nodes (first level leaders), where we start building items consisting of 1 element. "initializeRoots" is a void function, where most of our processing will happen. Finally, we return the product of execution - minimal transversals.

We continue with function ”extractUniqueValues”.

```
function extractUniqueValues(data) {
    const uniques = new Set();
    for (let i = 0; i < data.length; i++) {
        for (let j = 0; j < data[i].length; j++) {
            uniques.add(data[i][j]);
        }
    }
    return sorted([...uniques]); // [...uniqes] is conversion of
        set "uniques" into array.
}
```

Listing 4.2: Extracting list of nodes

We define set, then we go through every node in every edge (query) to find elements and add to ”uniques”. The last part is returning the list of sorted unique nodes. The time complexity of this function in the worst-case scenario is $O(N * M)$ where N is the number of edges, M is the average number of nodes containing each node. It is also worth considering, that adding element in the set is not ”free” operation, but it depends on the size of the set itself and the number of collisions. Amortized complexity is around $O(1)$ and that’s why we ignore it. Also, using Array instead of the set would have caused increasing time complexity, since the complexity of search in the array has a linear complexity.

```
function createBooleanTable(data, uniques) {
    const result = [[]];
    for (let i = 0; i < data.length; i++) {
        const temp = [false];
        for (let j = 0; j < uniques.length; j++) {
            temp.push(false);
        }
        for (let j = 0; j < data[i].length; j++) {
            temp[data[i][j]] = true;
        }
        result.push(temp);
    }
    return result;
}
```

Listing 4.3: Creating boolean table

We start building a boolean table by creating an array with an empty array as an element. The empty sub-array is given for enumeration (This way the first edge is on index 1, second has index 2, etc.). We start building the temporary table for each query (here we start with non-empty array as well, to keep counting from 1). Every temporary table has the same length, so the first nested loop is creating an array with all elements set to false, then we toggle booleans on those indexes which equals to node numbers encountered in the current edge.

The last function from initialization is ”initializeRoots”.

```
function initializeRoots(table, uniques, minimum) {
    let neighbors = [];
    let leads = [];

    for (let i = 0; i < uniques.length; i++) {
        // step 1
        let ds = disjunctionSupport(table,[uniques[i]]);
        if(ds === table.length - 1) {
            transversals.push([uniques[i]]);
            currentMinimum = 1;
            continue;
        }
        // step 2
        neighbors.push(uniques[i]);
        ds = disjunctionSupport(table, neighbors);
        if(ds === table.length -1) {
            leads.push(neighbors.pop())
        }
    }

    //Step 3
    if (minimum && currentMinimum == 1) {
        return;
    }

    // Step 4
    for(let i=0; i < leads.length; i++) {
        const lead = leads[i];
        const leadDS = disjunctionSupport([lead])
        // array of every element starting from index i+1 in leads
        const leadNeighbors = leads.slice(i+1);
        // concatenation of neighbors and leadNeighbors
        const passedNeighbors = [...leadNeighbors, ...neighbors];
        const supports = disjunctionSupports(passedNeighbors)
        const helper = createHelper(table, lead);
        nextIteration(table, passedNeighbors, helper,[lead],
           supports, leadDS, minimum);
    }
}
```

Listing 4.4: Initialize Roots

This function generates and initializes first level items. We have split into steps.

- Step 1 - we check disjunctive support for each item (we can assume that node is already an item at this level), then we check if it is equal to the number of queries, this item is minimal transversal, we set current minimum and we keep the item.
- Step 2 - if the item was not minimal transversal, we push it in neighbors (this is already part of generating max clique). Then we do separation of leaders from neighbors.
- Step 3 - if we already found minimal transversals and we are looking for current minimums, we stop the execution (since we already found all of them).
- Step 4 - we iterate through leaders. For each leader, we calculate parameters for proceeding to the next iteration.

We encountered the function that calculates disjunction support.

```
function disjunctionSupport(table, elements) {
    let ds = 0;
    for (let i = 1; i < table.length; i++) {
        for (let j = 0; j < elements.length; j++) {
            if(table[i][elements[j]]) {
                ds ++;
                break;
            }
        }
    }
    return ds;
}
```

Listing 4.5: Disjunction Support Of elements

The implementation is optimized for checking disjunctive support for a set of nodes (same as item). In the worst-case scenario, calculating DS based on the table has a time complexity of $O(N * M)$ where N is the number of queries, M is the number of nodes. Amortized best case scenario can be $O(N * log_2(M))$ because as long as we have one encounter in the current edge from given nodes, we stop checking the other nodes.

```javascript
function createHelper(table, field) {
    const helper = [[], []];
    for (let i = 1; i < table.length; i++) {
        if (table[i][field]) {
            helper[1].push(i);
        } else {
            helper[0].push(i);
        }
    }
    return helper;
}
```

Listing 4.6: Create Helper

Additionally, have a glance at creating the helper. As we know already, a helper is a two-dimensional array, which represents the absence of nodes from the item (0 index sub-array), or presence of encounter of one and only node from the item. The algorithm for creating an initial helper is very simple. Time complexity - $O(N)$ where N is the number of edges.

The most important function, where most of the processing is happening, is "nextIteration".

```javascript
/**
 * @param {boolean[][]} table
 * @param {number[]} uniques - neighbors of current leader (item)
 * @param {number[][]} helper
 * @param {number[]} currentFields - nodes of current leader (item)
 * @param {number[]} dss - disjunctive support of neighbors
 * @param {*} currentDs - disjunctive support of current leader
 * @param {boolean} minimum
 */
function nextIteration(table, uniques, helper, currentFields, dss,
   currentDs, minimum) {
    const newDss = [], leads = [], leadsDss = [];
    const neighbors = [...currentFields]; // Copy nodes from
       current item
    const neighborsDss = [];

    // step 1
    for (let i = 0; i < uniques.length; i++) {
        const a = [...currentFields, uniques[i]];
        newDss.push(disjunctionSupport(table, a));
    }

    //step 2
    for (let i = uniques.length -1; i >= 0; i--) {
        if(newDss[i] === currentDs || newDss[i] === dss[i]){
            uniques.splice(i,1);
            newDss.splice(i,1);
            continue;
        }
        if(newDss[i] === table.length - 1) {
            const isMinimal = minimalityCheck(table, helper, uniques[i]);
            if (isMinimal) {
                transversals.push([...currentFields, uniques[i]]);
                currentMinimum = currentFields.length + 1;
            }
            // in this case, splice is used to remove element on index i.
            // array size is decreased accordingly during this operation.
            uniques.splice(i, 1);
            newDss.splice(i, 1);
        }
    }
    // step 3
    if(minimum && currentFields.length + 2 > currentMinimum ) {
        return;
    }
    // step 4
    for (let i = 0; i < uniques.length; i++) {
            neighbors.push(uniques[i]);
            neighborsDss.push(newDss[i]);
        let ds = disjunctionSupport(table, neighbors);
        if(ds === table.length -1) {
            // pop method removes last element and returns it.
            leads.push(neighbors.pop());
            leadsDss.push(neighborsDss.pop());
        }
    }

    // step 5
    for(let i = 0; i < leads.length; i++) {
        const lead  = leads[i];
        const futureHelper = minimalityCheckWithExpansion(table, helper, lead);
        // if empty array, not minimal.
        if (futureHelper.length === 0) continue;
        const futureFields = [...currentFields, lead];
        const leadDs = leadsDss[i];
        const newNeighbors = leads.slice(i+1);
        const newNeighborsDss = leadsDss.slice(i+1);
        // Concatenate newNeighbors and elements from neighbors
           coming after passed leader nodes
        const passNeighbors = [...newNeighbors, ...neighbors.slice(
           currentFields.length)];
        //Concatenate disjunctive support of neighbors and
           newNeighbors
        const passDss = [...newNeighborsDss, ...neighborsDss];
        nextIteration(table, passNeighbors, futureHelper,
           futureFields, passDss, leadDs, minimum);
    }
}
```

Listing 4.7: Process Next Iteration

We split this function into 4 steps:

- Step 1 - for each unique item in passed neighbors, stick item with the current leader and calculate disjunctive support. Save the result in ”newDss”.
- Step 2 - check new items. Remove items that have the same disjunctive support as its direct parents. If an item has disjunctive support equal to the query amount, we check on minimality. if it is minimal, we add to minimal transversals, set the current minimum, and remove the item.
- Step 3 - if we are looking for current minimums and generated elements on the next iteration is larger than the current minimum, we stop.
- Step 4 - generate max clique, separate leaders with corresponding disjunctive supports.
- Step 5 - extend helpers to consider a new node in the item. Check on minimality, if positive, initialize next iteration for the given leader.

Worth noticing, that we are not keeping the whole item in neighbors, neither in leaders. We have only the latest added notes (last nodes) kept. This way, we save resources on two different operations: when we stick and compute, we need only the last element from neighbors. This is caused by the nature of iterations. Every next iteration contains extended items based on the last iteration leader. Because of that, every neighbor differs from the leader by the last node. this way, we cut the cost of keeping duplicate information and initializing full items. we only need full items for minimality check and for next iteration.

We introduce two new functions, ”minimalityCheck” and ”minimalityCheckWithExpansion”.

```javascript
function minimalityCheck(table, helper, current) {
    let i;
    // step 1
    for (i = 0; i < helper[0].length; i++) {
        if (table[helper[0][i]][current]) {
            break;
        }
    }
    if (i === helper[0].length) {
        return false;
    }
    // step 2
    for (i = 1; i < helper.length; i++) {
        let j = 0;
        for (; j < helper[i].length; j++) {
            if (!table[helper[i][j]][current]) {
                break;
            }
        }
        if (j === helper[i].length) {
            return false
        }
    }
    return true;
}
```

Listing 4.8: Minimality Check

”minimalityCheck” is plain check, where we return boolean. True means that it is minimal, otherwise it is false. Step 1 is checking those edges, which do not contain any node from the leader item we are extending. During Step 2, we check the rest of the edges from the helper table. As we discussed, every sub-array starting from index 1, should contain at least 1 element that is not covered by the node that is extending given item.
The worst time complexity of the function can be $O(N)$ where N is the total amount of edges currently presented in the helper table.

```javascript
function minimalityCheckWithExpansion(table, helper, current) {
    const result = [[]];

    for (let i = 1; i < helper.length; i++) {
        result.push([]);
        let j = 0;
        for (; j < helper[i].length; j++) {
            if (!table[helper[i][j]][current]) {
                result[i].push(helper[i][j]);
            }
        }
        if (result[i].length === 0) {
            return [];
        }
    }

    result.push([]);
    for (let i = 0; i < helper[0].length; i++) {
        if (!table[helper[0][i]][current]) {
            result[0].push(helper[0][i]);
        } else {
            result[result.length - 1].push(helper[0][i]);
        }
    }

    return result;
}
```

Listing 4.9: Minimality Check and update helper at once

Compared to regular minimality check, implementation of ”minimalityCheckWithExpansion” has a couple of interesting features. First of all, it contains all the logic from the minimality check, except it does not finish execution by returning true or false. We are building the new helper table based on the previous one. We check every query the same way we checked in the minimality check. If we see, that minimality check is leading us to negative, we return an empty array. The empty array is the sign that the minimality check did not finish with a positive result. Otherwise, we continue building helper table.
The worst time complexity is still equal to O(N), but since we have to build the helper table, we don't stop checking sub-arrays on the first query that satisfies the minimality check criteria. Instead, we have to traverse all the sub-arrays and build a new table. That is the reason why we have two different functions. The standard minimality check is very lightweight in terms of memory usage. On the other hand, ”minimalityCheckWithExpansion” is beneficial in terms of performance, since we join two different functionalities: minimality check and building helper table.

5. Execution and Benchmarks

5.1 Test Process

Our goal is to check the whole process. The steps are as follows:

- Generate test database
- Generate a set of test queries
- Execute queries on test database before applying the indexes [test 1]
- Execute an efficient index set suggestion algorithm with generated queries
- Apply suggested indexes to the database
- Execute queries on test database after applying indexes [test 2]
- Evaluate Test 1 and test 2 execution statistics and compare the final results

First of all, we need data to populate the collection in the database. It will be easy to find a ready dataset, but since we need generalized data, it is becoming very hard to find such data. On top of that, such data will need formatting and structuring to match the MongoDB document structure. Another option is to generate new data and perform tests on it. We choose the latter option.

```
function generateDocument() {
    return {
        index: faker.random.number(),
        uuid: faker.random.uuid(),
        isActive: Math.random() < 0.85,
        pricePaid: faker.random.number(72),
        discountAmountPercentage: faker.random.number({min: 0, max:
            85}),
        picture: generatePicture(),
        age: faker.random.number(80),
        sectionColor: faker.commerce.color(),
        username: faker.internet.userName(),
        name: {
            first: faker.name.firstName(),
            last: faker.name.lastName()
        },
        company: {
            name: faker.company.companyName(''),
            uuid: faker.random.uuid(),
        },
        email: faker.internet.email(),
        phone: faker.phone.phoneNumber('(###) ###-###-####'),
        createdAt: faker.date.recent(3650),
        seatNumbers: generateRandomNumberList()
    };
}
```

Listing 5.1: Document Generator Example

To generate data, we need to have a document structure. We notice that the document in our example is quite complex since we have various types and structures. We utilize sub-documents, list (seatNumbers for example), strings, numbers. We are using a utility called faker.js[1], which helps us generate random data from a given category. For example, faker.commerce.color is a function that generates a random color. Worth noting, that we also include a universally unique identifier (uuid) (Leach et al., 2005) as the field. The reason for adding uuid, is that the field will have unique value, and the cardinality of the field will be very high. Indexing high cardinality field results in better performance [2].

We populate collection using documents produced using a generator in Listing 5.1. The biggest benefit of a generator is the ability to produce different results every time we execute it.

Next, we generate a set of queries. The first step is to generate proper field names for queries while we have nested queries. For example, if we have "name" sub-document with fields "first" and "last", we convert them into a dot-separated format such as "name.first" and "name.last". Then, we need sample value for each field that we are searching for. Sample values are unique as well and we generate them as well. Before we start creating queries, we need to create random subsets of fields. We decide, that each query can have a minimum of two, up to 10 fields.

[1]faker.js - generate massive amounts of fake data in the browser and node.js - `https://github.com/marak/Faker.js/` - accessed 30-April-2020

[2]Why low cardinality indexes negatively impact performance - `https://www.ibm.com/developerworks/data/library/techarticle/dm-1309cardinal/index.html` - accessed 30-April-2020

```
function generateQueryObjectFromSelectedFields(fields) {
    const result = {};
    for (field in fields) {
        if(field.type === 'date')
            result[field.name] = generateDate();
        if(field.type === 'string')
            result[field.name] = generateWord(field.name);
        if(field.type === 'number')
            result[field.name] = generateNumber();
        if(field.type === 'array')
            result[field.name] = generateNumber();
        if(field.type === 'boolean')
            result[field.name] = generateBoolean();
    }
    return result;
}
```

Listing 5.2: Generate query from selected fields

As we see, for every chosen field, we generate value that has the type of the selected field. We insert generated value into an object with the given field name. We will notice, that for array type, we generate number value, because we have only one field in documents with array as a value, and the type of values in the array is number.

```
function generateBoolean() {
    // Math.random returns float value n where 0 <= n <= 1
    return Math.random() < 0.85;
}
function generateDate() {
    const date = faker.date.recent(3650);
    return {$gte: date};
}
function generateWord(field) {
    if(field === 'name.first'){
        return faker.name.firstName()
    }
    if(field === 'name.last'){
        return faker.name.lastName()
    }
    let word = faker.random.word();
    if(generateBoolean()){
        return word;
    }
    return {$regex: word};
}
function generateNumber() {
    const number = faker.random.number({min:0, max:200});
    if(generateBoolean()) {
        return number;
    }
    return {$gte: number};
}
```

Listing 5.3: Generate value for query field

Let us follow the functions provided in Listing 5.3 and describe the behavior of each of them:

- generateBoolean - returns boolean with the possibility of 85 % that the result will be "true"
- generateDate - returns random date from the last 10 years. Worth noting that we wrap value with range operator to filter dates created after given date.
- generateWord - this function differs from others since it takes field name into consideration and generates the corresponding type of value (name or last name). If it does not match selected field names, we generate a random word. also, around 15 % of cases will utilize regular expression to find documents that contain value as part of the text.
- generateNumber - returns non-negative number which is up to (inclusive) 200. Also, 15 % of cases utilize range to filter values greater or equal to generated value.

We see the usage of regular expressions for string type fields. This is intentional since we need tests to be as close to real-world scenarios as possible. The regular expressions have a big downside, it can not be optimized during query execution and can become a serious performance issue. However, there are times when we have no other choices and unfortunately such cases appear from time to time. There are also other operations/operators that can harm the performance, but since we can find many different combinations of such cases, we decided to keep only one of them (regular expression). The final result will be the same nevertheless, all the cases will harm performance to some degree.

After creating the queries, we run the first test. We run every query before indexing the fields. Then we run our index suggestion algorithm and get recommended fields for indexing. We build indexes for recommended fields. Now we have optimized collection and we run the same queries again. Finally, we calculate the average execution time before and after building indexes.

5.2 Test Results

The tests were executed on the system described below:

```
OS: Windows 10 Pro version 1909
Processor: Intel Core i7-7500U (2.7GHz)
Core count: 2 cores (4 threads)
RAM: 12 GB
NodeJS Version: 13.11.0
MongoDB Server Version: 4.2
IDLE Power Usage: 2.5 W
```

We generated a database with 3 million documents. The described test process in 5.1, is giving different results every time we execute the flow because queries are generated during the execution. The results varied and were different for every execution. We recorded different cases and selected average case that was also closer to real-world scenarios.

We used the metrics described in section 2.4, the metrics that are used in Mahajan and Zong (2017). The selected metrics are **Speedup, Powerup, Greenup**.

We also monitored load and energy consumption while tests were running. The results are provided in table 5.1.

Table 5.1: Test execution results (Given average values).

	Power	**Time**	**Energy**	**Speedup**	**Powerup**	**Greenup**
Before	11.61 W	1859.7 ms	21.59 J	11.77	0.96	12.2
After	11.2 W	158.05 ms	1.77 J			

From the results, we can see, that overall execution time has been reduced **11.77** times. The most important metric is Greenup since our primary goal was energy efficiency. After optimization, energy efficiency (Greenup) increased **12.2** times.

In addition, we recorded the execution time of each query, which is provided in Figure 5.1. The latter shows even more insights of the execution process. We see the spikes on multiple occasions. This is caused by the usage of regular expressions. Two of these spikes are showing even slower execution times than the ones before optimization.

For illustrative examples, we modified "generateWord" function which is presented in listing 5.4.

```
function generateWord(field) {
    if(field === 'name.first'){
        return faker.name.firstName()
    }
    if(field === 'name.last'){
        return faker.name.lastName()
    }
    let word = faker.random.word();
    return word;
}
```

Listing 5.4: function "generateWord" (modified)

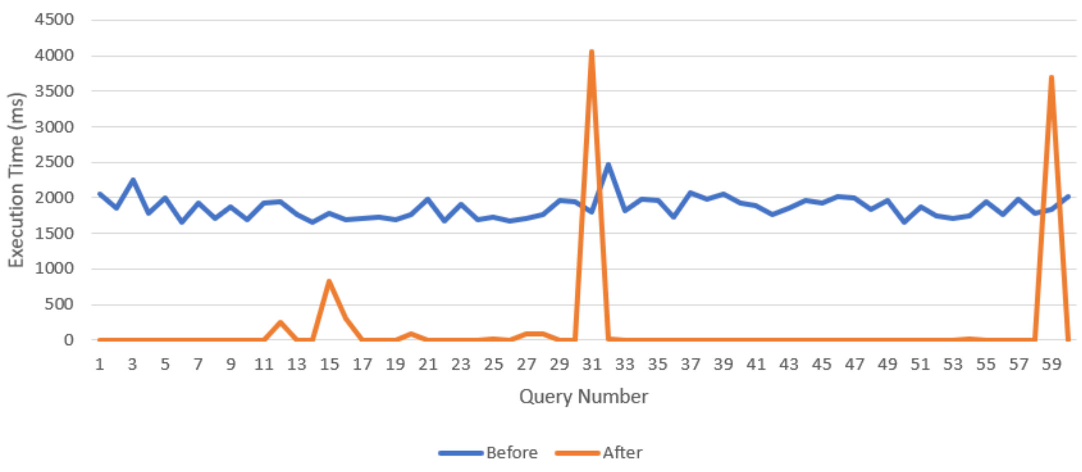


Figure 5.1: Comparison of individual query execution times

After the change provided in listing 5.4, we excluded the possibility of generating a query that includes regular expression. This helped us to have close to the best-case results. The results are proof that the correct set of indexes plays a huge role in energy efficiency. The results have shown that after optimization, average execution time went down to **2.82 ms** compared to before optimization average of 1896.07 ms. This has affected energy efficiency drastically, achieving the Greenup score of **982.7** .

Both of the tests show the improvement in energy efficiency. However, our benchmarks, no matter how close it is to real-world scenarios, can be completely different in some cases. There is always the possibility that queries consist of operands that can not be optimized by creating the indexes. If such cases are very high, the final result might not be impressive. Also, different datasets will result in different execution times. There are many other variables such as hardware, data size, query count, number of fields in each query, document structure, etc. that can affect the final results.

On top of regular tests, we tested the algorithm for finding minimal transversals. As we already know, the algorithm has two modes:

- Full - look for all the minimal transversals
- Current Minimum - reduces search space by utilizing the current minimum (3.3.1).

Our test case input for checking the performance:

```
{ { 1, 2, 3 },  { 3, 4, 5 },  { 5, 6, 7 },  { 7, 8, 9 },
  { 9, 10, 11 },  { 11, 12, 13 },  { 13, 14, 15 },  { 15, 16, 17 },
  { 17, 18, 19 },  { 19, 20, 21 },  { 21, 22, 23 },  { 23, 24, 25 },
  { 25, 26, 27 },  { 27, 28, 29 },  { 29, 30, 31 },  { 31, 32, 33 },
  { 33, 34, 35 },  { 35, 36, 37 },  { 37, 38, 39 },  { 39, 40, 41 },
  { 41, 42, 43 },  { 43, 44, 45 } }
```

We compared the Power usage in watts and got very similar results for both modes. However, we saw a huge boost in execution time.

Table 5.2: Algorithm execution result for test case (Given average values).

	Power (W)	**Time (ms)**	**Energy (J)**	**Speedup**	**Powerup**	**Greenup**
Full	11.18	4265	47.68	25.38	1	25.38
Curr. M		168	1.88			

Based on table 5.2, we achieve both **Greenup** and **Speedup** equal to **25.38** when we go for mode "Current Minimum". We can call this result the real world average scenario.

We want to see the importance of Current Minimum. We use the generated input file from the **Hypergraph Dualization Repository**[3] provided by Keisuke Murakami and Takeaki Uno. The chosen dataset **"ac 50k"** contains 32207 rows and 336 unique nodes. File size - 38.83 MB. We see this dataset as an interesting challenge since it can easily show the difference we get from using "Current Minimum" mode over the "Full" mode.

[3]Hypergraph Dualization Repository - `http://research.nii.ac.jp/~uno/dualization.html` - accessed 13-May-2020

Table 5.3: Algorithm test results with large data (Given average values).

	Power (W)	**Time (ms)**	**Energy (J)**	**Speedup**	**Powerup**	**Greenup**
Full	11.23	580,132	6,514.9	1,611.48	1.004	1,605.76
Curr. M	11.27	360	4.06			

The results from Table 5.3 shows how much we can benefit from producing only what we need for the given task. In our case, results from both of them modes are processed the same way, resulting in the same output, so we do not lose anything by optimizing the output.

So, let us follow the process of extracting MT from the input to compute the indexed fields set. First, with full mode, we have to collect all the MT-s and filter the smallest ones. Thanks to the new mode "Current Minimum", we are able to extract the smallest ones straight and speed up the discovery process by cutting down the iterations of looking for the bigger MT-s. Thus, the added value of "Current Minimum" is reducing (or fully omitting) the iterations that do not lead to the smallest MT-s.

Finally, to see the broader picture, we have conducted the tests using inputs from different groups of datasets from the Hypergraph Dualization Repository. The results (see table 5.5) are impressive, since "Current Minimum" mode has potential to show significant efficiency improvement (win1600.dat, lose800.dat, dualmatching28.dat) which, in our case, can be improved up to **15,250.2** times, averaging **935.94**. Again, we notice, that the result may vary based on input, some inputs might show little to no difference between two modes (matching44.dat, TH160.dat). Worth noting, that table 5.5 does not contain Powerup and power metrics, since power consumption stayed the same (11.47 W) throughout the testing process (constant power usage does not affect Greenup score). However, all the metrics are available in the summary table (see table 5.4).

Table 5.4: Algorithm test full execution summary (Given average values).

	Power (W)	**Time (ms)**	**Energy (J)**	**Speedup**	**Powerup**	**Greenup**
Full	11.47	68,171	7,819.2	3.08	1	3.08
Curr. M		22,123	2,537.4			

Table 5.4 shows the average time that was taken to execute each test in Table 5.5. Based on this data, we see the overall Greenup score of **3.08** which denotes the **208%** better efficiency of "Current minimum" mode over the Full mode.

Table 5.5: Algorithm test results extended (Given average values).

input	**Full (ms)**	**Current Minimum (ms)**	**Greenup**
ac_70k.dat	26,932	71	379.32
ac_110k.dat	570	8	71.25
ac_150k.dat	43	3	14.33
bms2_200.dat	1,903	40	47.58
bms2_400.dat	122	19	6.42
bms2_800.dat	14	4	3.5
dualmatching20.dat	144	2	72
dualmatching24.dat	2,074	7	296.29
dualmatching28.dat	60,783	26	2,337.81
lose100.dat	146	3	48.67
lose400.dat	7,979	6	1,329.83
lose800.dat	49,202	11	4,472.91
matching20.dat	24	26	0.92
matching34.dat	582	606	0.96
matching44.dat	28,983	29,132	0.99
p95_8000.dat	8,527	1,788	4.77
p95_32000.dat	236,746	17,607	13.45
p98_8000.dat	1,625	224	7.25
p98_32000.dat	27,262	14,214	1.92
p98_128000.dat	382,589	70,741	5.41
p99_16000.dat	2,814	610	4.61
p99_64000.dat	37,738	15,722	2.4
p99_256000.dat	237,217	143,903	1.65
p9_4000.dat	1,6135	909	17.75
p9_8000.dat	68,141	2,152	31.66
SDFP16.dat	20	1	20
SDFP30.dat	46,779	12	3,898.25
SDTH42.dat	94	9	10.44
SDTH102.dat	14,362	30	478.73
SDTH162.dat	484,174	190	2,548.28
TH40.dat	102	93	1.1
TH160.dat	477,350	476,112	1
win100.dat	63	4	15.75
win800.dat	12,245	9	1,360.56
win1600.dat	152,502	10	15,250.2

6. Conclusion

In this thesis, we thoroughly described and implemented the index suggestion algorithm for MongoDB databases with two modes. The algorithm produced the smallest set of fields that can be indexed. With suggested fields, we covered all the queries provided by the user. Hypergraphs played a huge role in building the algorithm, plus our optimizations led us to better, optimal, efficient execution. We then developed a test process that helped us in evaluations of the results and provided execution statistics before and after building indexes. We ran our test process multiple times, from the beginning till the end. We gathered results and presented two important cases, best-case scenario, and results closest to a real-world scenario. We also tested algorithm modes and compared the efficiency.

We can draw a number of interesting conclusions from the research. First of all, using indexes matter a lot for energy efficiency. Performance and efficiency gains may vary a lot, but our tests show that we can gain 12.2 times better efficiency after applying suggested indexes to the database. Also, in some specific cases, this number can go as high as 982.7 . The algorithm test has shown that choosing the "Current Minimum" mode can result in up to 15250 times better efficiency (Greenup score).

Bibliography

Claude Berge. *Hypergraphs: combinatorics of finite sets*, volume 45. Elsevier, 1984.

L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 483–488, Aug 2011. doi: 10.1109/DEXA.2011.71.

A. Casali, R. Cicchetti, and L. Lakhal. Essential patterns: A perfect cover of frequent patterns. In *Proceedings of the 7th International Conference on DaWaK*, pages 428–437, Copenhagen, Denmark, 2005.

S. Chickerur, A. Goudar, and A. Kinnerkar. Comparison of relational database with document-oriented database (mongodb) for big data applications. In *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*, pages 41–47, Nov 2015. doi: 10.1109/ASEA.2015.19.

Issam Ghabry, Sadok Ben Yahia, and Mohamed Nidhal Jelassi. Selection of bitmap join index: Approach based on minimal transversals. In Carlos Ordonez and Ladjel Bellatreche, editors, *Big Data Analytics and Knowledge Discovery - 20th International Conference, DaWaK 2018, Regensburg, Germany, September 3-6, 2018, Proceedings*, volume 11031 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2018. doi: 10.1007/978-3-319-98539-8_23. URL `https://doi.org/10.1007/978-3-319-98539-8\_23`.

M.N. Jelassi, C. Largeron, and S. Ben Yahia. Concise representation of hypergraph minimal transversals: Approach and application on the dependency inference problem. In *9th IEEE International Conference on Research Challenges in Information Science, RCIS 2015*, pages 434–444, 2015.

Mohamed Nidhal Jelassi, Christine Largeron, and Sadok Ben Yahia. Efficient unveiling of multi-members in a social network. *J. Syst. Softw.*, 94:30–38, 2014. doi: 10.1016/j.jss.2013.06.061. URL `https://doi.org/10.1016/j.jss.2013.06.061`.

M. Jung, S. Youn, J. Bae, and Y. Choi. A study on data input and output performance comparison of mongodb and postgresql in the big data environment. In

2015 8th International Conference on Database Theory and Application (DTA), pages 14–17, Nov 2015. doi: 10.1109/DTA.2015.14.

S. Kanoje, V. Powar, and D. Mukhopadhyay. Using mongodb for social networking website deciphering the pros and cons. In *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pages 1–3, March 2015. doi: 10.1109/ICIIECS.2015.7192924.

Nidhi Jain Kansal and Inderveer Chana. Energy-aware virtual machine migration for cloud computing-a firefly optimization approach. *Journal of Grid Computing*, 14(2):327–345, 2016.

Paul Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. 2005.

D. Mahajan and Z. Zong. Energy efficiency analysis of query optimizations on mongodb and cassandra. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, Oct 2017. doi: 10.1109/IGCC.2017.8323581.

G. Ongo and G. P. Kusuma. Hybrid database system of mysql and mongodb in web application development. In *2018 International Conference on Information Management and Technology (ICIMTech)*, pages 256–260, Sep. 2018. doi: 10.1109/ICIMTech.2018.8528120.

Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, March 2014. ISSN 0360-0300. doi: 10.1145/2532637.

G. Prabagaren. Systematic approach for validating java-mongodb schema. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pages 1–4, Feb 2014. doi: 10.1109/ICICES.2014.7033768.

Fábio D. Rossi, Miguel G. Xavier, César A.F. De Rose, Rodrigo N. Calheiros, and Rajkumar Buyya. E-eco: Performance-aware energy-efficient cloud data center orchestration. *Journal of Network and Computer Applications*, 78:83 – 96, 2017. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2016.10.024.

Jie Song, Zhongyi Ma, Richard Thomas, and Ge Yu. Energy efficiency optimization in big data processing platform by improving resources utilization. *Sustainable Computing: Informatics and Systems*, 21:80 – 89, 2019. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2018.11.011. URL `http://www.sciencedirect.com/science/article/pii/S2210537917300379`.

Shaima Trabelsi, Mohamed Taha Bennani, and Sadok Ben Yahia. A new test suite reduction approach based on hypergraph minimal transversal mining. In Tran Khanh Dang, Josef Küng, Makoto Takizawa, and Son Ha Bui, editors, *Future Data and Security Engineering*, pages 15–30, Cham, 2019. Springer International Publishing. ISBN 978-3-030-35653-8.

L. Vokorokos, M. Uchnár, and A. Baláž. Mongodb scheme analysis. In *2017 IEEE 21st International Conference on Intelligent Engineering Systems (INES)*, pages 000067–000070, Oct 2017. doi: 10.1109/INES.2017.8118530.

Heng Zeng, Carla Schlatter Ellis, and Alvin R Lebeck. Experiences in managing energy with ecosystem. *IEEE Pervasive Computing*, 4(1):62–68, 2005.