# 

# **23-1 Database System Team Project**

phase2

21900156 김예준 / 22000796 함상훈 / 22100579 이진주

| [**1. Introduce**](#_119lihs4yz3u) 2  [**2. Denormalization 4**](#_9pb3ar1vuvxz)  [2.1 N\_finalUser denormalization 5](#_ju06r5pz84lu)  [2.2 N\_document denormalization 6](#_ru5cs1gq2eaa)  [2.3 N\_board - board denormalization 7](#_cfgqdbqemyoa)  [2.4 N\_savedDoc - savedDoc denormalization 7](#_guv9hlf6s9y1)  [2.5 N\_frequency and N\_similarity from source DATA 8](#_c4smyfvqevxc)  [2.6 N\_tfword\_cnt : table from task5 8](#_whpuwwpc0l66)  [**3. indexing 9**](#_8nvcry5l596j)  [3.1 N\_document 9](#_u9z1j9wyo7vm)  [3.2 N\_savedDoc 10](#_aix1sbzee0bu)  [3.3 N\_frequency 10](#_p0iy0rc2149j)  [3.4 N\_similarity 11](#_oixrlno477ox)  [3.5 N\_tfword\_cnt 11](#_mhzi3lvgfdxv)  [**4. task solutions 12**](#_jvulladu0m3o)  [**5. strategies 23**](#_z7c3io5fijot)  [**6. summary of DB 25**](#_2r9b9es1tl1y)  [6.1 Database size 25](#_exw02iknojda)  [6.2 Table size 25](#_xv10hrsizkyg) |
| --- |

# **1. Introduce**

Phase 2 of the team project provides a new challenge for row data in KUBIC, which was covered in phase 1, with time complexity as a priority.

In phase2, we consider the new data tables together.

* frequency: contains the TF-IDF(term frequency-inverse document frequency) scores for selected words for each document. 877490 records and 4 columns. We can measure how relevant a word is to a document in a collection of documents by them.
* similarity: Cosine similarity analysis of the service documents(between each pair of data instances). 10,000,000 records and 3 columns. We use it to measure the similarity between each pair of documents.

source of this phase

|  |
| --- |

The purpose of our assignment is to optimize the database using denormalization and indexing techniques. As databases grow in complexity and size, it becomes crucial to find ways to improve their performance and efficiency. Denormalization involves strategically duplicating and storing data in multiple places to reduce the need for complex joins and improve query performance. Indexing, on the other hand, involves creating data structures that allow for quick and efficient data retrieval.

In this assignment, we aim to explore the benefits of denormalization and indexing in optimizing our database which could have been traded off the normalization process. By implementing denormalization, we can reduce the number of tables and simplify the data model, leading to faster query execution. Additionally, by carefully creating indexes on frequently accessed columns, we can significantly improve the speed of data retrieval operations.

Through this assignment, we hope to demonstrate the advantages of denormalization and indexing in enhancing the performance of our database and make best performance in solving the given tasks. By optimizing our database, we can achieve faster query execution times.

We aim to define a database and provide queries with optimal performance for the given 11 tasks after denormalization and indexing. We proceeded according to the procedure below.

First, we denormalized the normalized tables and added indexes to make them in an optimal form for better performance.

Then, based on the resulting database, we wrote some queries that solve each of the given tasks.

The written queries are modified by utilizing various strategies to result in more efficient time performance.

To explicitly evaluate the improvement measure, we first designed 10 queries from the given questions. We computed the average query speed for each query out of 5 times executions from the schema constructed in our last assignment.

# 

# **2. Denormalization**

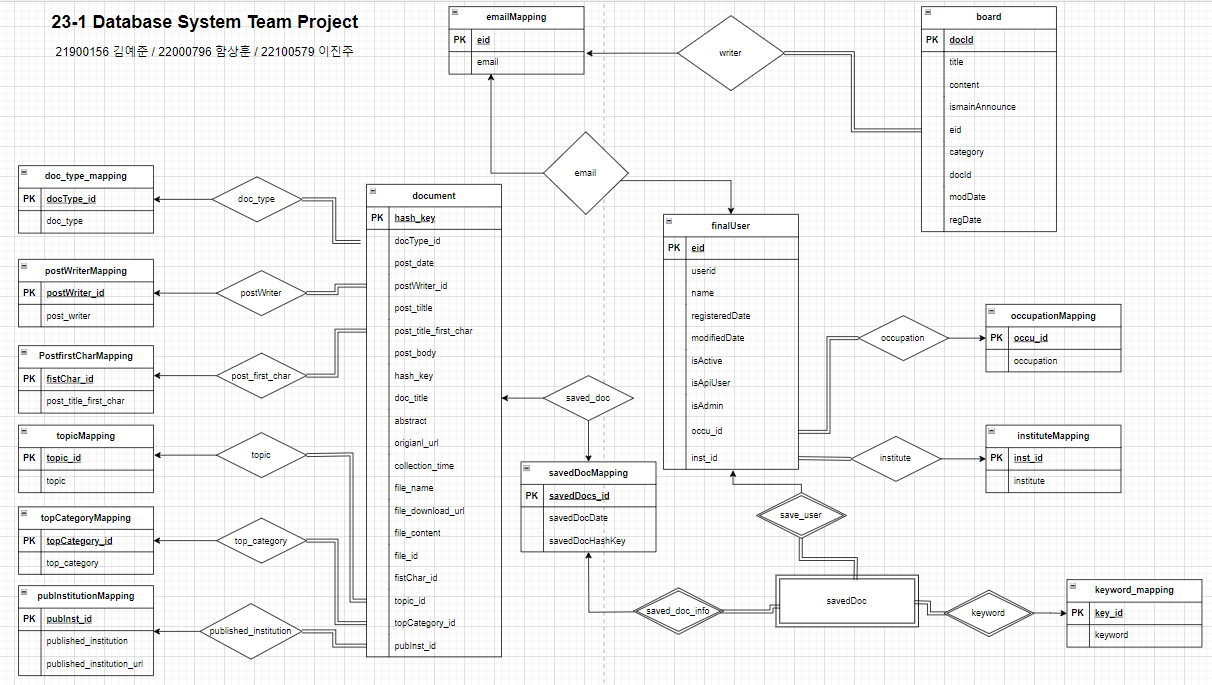


Figure 1. ER diagram constructed in the previous assignment (Phase 1)

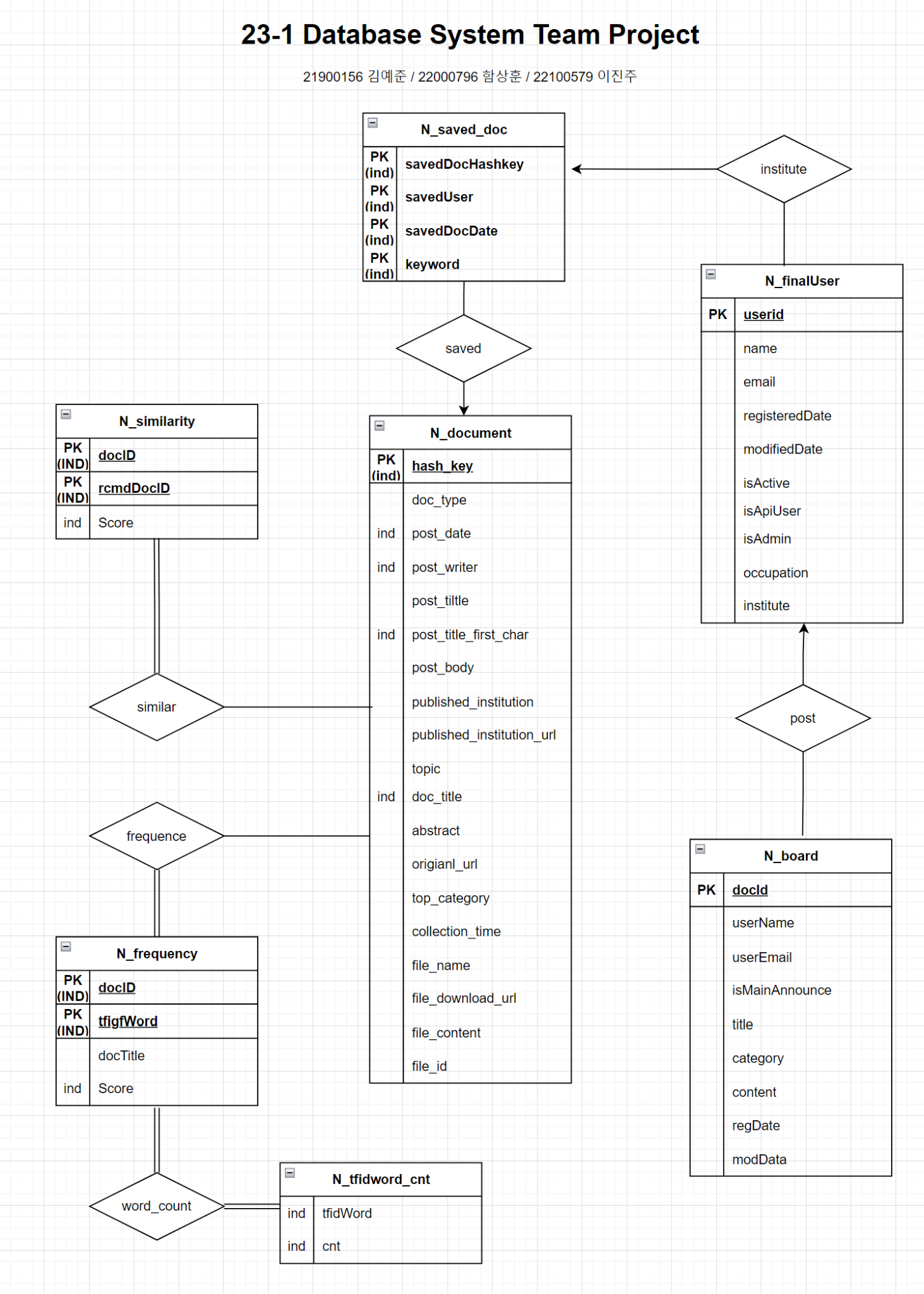


Figure 2 . New ER diagram constructed in this assignment by denormalized

We denormalized the tables that were normalized in phase 1(Figure 1) and returned them to the four tables of the big topic. The denormalized tables are Figure 2.

**N\_finalUse**r: This is where data about the user is stored.

**N\_document**: Overall data about the document is stored.

**N\_board**: Data related to the board is stored

**N\_savedDoc**: Stores data for documents bookmarked by the user.

In addition to these four tables, we imported the frequency and similarity tables added in phase 2, named **N\_freqency** and **N\_similarity**, respectively, and used them together.

Using the N\_frequency table, we created an additional table, **N\_tfiword\_cnt** , for use in several future tasks. This table is useful in solving given tasks and it will be explained further later.

The following DDL queries were used to complete each table by denormalization.

## **2.1 N\_finalUser denormalization**

| create table N\_finalUser  as select \* from finalUser;  alter table N\_finalUser add column email varchar(255);  update N\_finalUser n, emailMapping x  set n.email = x.email  where n.eid = x.eid;  alter table N\_finalUser drop eid;  alter table N\_finalUser add column occupation varchar(255);  update N\_finalUser n, occupationMapping x  set n.occupation = x.occupation  where n.occu\_id = x.occu\_id;  alter table N\_finalUser drop occu\_id;  alter table N\_finalUser add column institute varchar(255);  update N\_finalUser n, instituteMapping x  set n.institute = x.institute  where n.inst\_id = x.inst\_id;  alter table N\_finalUser drop inst\_id; |
| --- |

Firstly, we considered combining “finalUser”, “emailMapping” ,“occpuationMappping” and “instituteMapping” tables. For denormalizing this tables, we proceeded in the form of adding columns and then adding rows according to the conditions and dropping column which is executed like JOIN. By combining, we make it to access information about “finaluser” more quickly. Information of “finaluser” is same with attributes of “N\_finalUser” in Figure 2.

## **2.2 N\_document denormalization**

| create table N\_document  as select \* from document;  # postTitleFirstChar------------------------------------  alter table N\_document add post\_title\_firt\_char varchar(1);  update N\_document n, PostfirstCharMapping x  set n.post\_title\_first\_char = x.post\_title\_first\_char  where n.firstChar\_id = x.firstChar\_id;  alter table N\_document drop firstChar\_id;  # topic------------------------------------  alter table N\_document add topic varchar(255);  update N\_document n, topicMapping x  set n.topic = x.topic  where n.topic\_id = x.topic\_id;  alter table N\_document drop topic\_id;  # topCategory------------------------------------  alter table N\_document add top\_category varchar(255);  update N\_document n, topCategoryMapping x  set n.top\_category = x.top\_category  where n.topCategory\_id = x.topCategory\_id;  alter table N\_document drop topCategory\_id;  # publishedInstitution------------------------------------  alter table N\_document add published\_institution varchar(255);  alter table N\_document add published\_institution\_url varchar(255);  update N\_document n, pubInstitutionMapping x  set n.published\_institution = x.published\_institution  where n.pubInst\_id = x.pubInst\_id;  update N\_document n, pubInstitutionMapping x  set n.published\_institution\_url = x.published\_institution\_url  where n.pubInst\_id = x.pubInst\_id;  alter table N\_document drop pubInst\_id;  # postWriter------------------------------------  alter table N\_document add post\_writer varchar(255);  update N\_document n, postWriterMapping x  set n.post\_writer = x.post\_writer  where n.postWriter\_id = x.postWriter\_id;  alter table N\_document drop postWriter\_id; |
| --- |

## Secondly, we considered combining “PostfirstCharMapping”, “topicMapping” ,“topCategoryMapping” and “pubInstitutionMapping” and “postWriterMapping” tables. For denormalizing this tables, we proceeded in the form of adding columns and then adding rows according to the conditions and dropping column which is executed like JOIN. By combining, we make it to access information about “documents” more quickly. Information of “document” is same with attributes of “N\_document” in Figure 2.

## **2.3 N\_board - board denormalization**

| CREATE TABLE N\_board as(  SELECT docId, title, content, isMainAnnounce, board.eid as eid, category, modDate, regDate, email  FROM DB16.board left join DB16.emailMapping on board.eid = emailMapping.eid); |
| --- |

## Thirdly, we considered combining “emailMapping” and “board” tables. For denormalizing this tables, we proceeded using JOIN. By combining, we make it to access information about “board” more quickly. Information of “board” is same with attributes of “N\_board” in Figure 2.

## **2.4 N\_savedDoc - savedDoc denormalization**

| CREATE TABLE N\_savedDoc as (  SELECT eid, savedDocDate, savedDocHashKey, keyword  FROM DB16.savedDoc natural join DB16.savedDocsMapping natural join DB16.keyword\_mapping natural join DB16.finalUser); |
| --- |

## Fourthly, we considered combining “savedDocMappingMapping” and “savedDoc” tables. For denormalizing this tables, we proceeded using Natural JOIN. By combining, we make it to access information about “savedDoc” more quickly. Information of “savedDoc” is same with attributes of “N\_savedDoc” in Figure 2.

## **2.5 N\_frequency and N\_similarity from source DATA**

| create table N\_frequency as  select \* from DATA.frequency;  create table N\_similarity as  select \* from DATA.similarity; |
| --- |

## Fifthly, we have to use the given ‘similarity’ table that contains the similarity between each pair of data instances and the given ‘frequency’ table that contains the TF-IDF (term frequency-inverse document frequency) scores for selected words for each document. For using those tables, we bring all of the datum from ‘similarity’ table and ‘frequency’ table on our schema by using ‘select \*’ clause.

## **2.6 N\_tfiword\_cnt : table from task5**

| CREATE TABLE N\_tfiword\_cnt AS(  select tfidfWord, *count*(tfidfWord) from N\_frequency group by tfidfWord  order by *count*(tfidfWord) desc); |
| --- |

# Lastly, we create the table ‘N\_tfiword\_cnt’ that contains tfidfWord and the number of tfidfWord used which is arranged in descending order. After looking at the given tasks, we decided that we would need this table, so we made it .

# 

# **3. indexing**

We set up appropriate indexes on the tables to improve performance on certain search queries.

The indexes were set on the primary key of each table by default, and as we wrote the queries to solve the tasks, we set additional indexes on the appropriate columns for each query.

We find that queries with ‘group by’ are slow to execute. Therefore, using sorted data through index is expected to speed up execution.

Similarly, ‘order by’ was often used in queries, which could also be expected to improve performance by using index.

Frequently referenced data could also be expected to improve performance using index, but in this given task, frequently referenced data are usually pk, so index was automatically generated.

## **3.1 N\_document**

(PK) hash\_key

post\_date

post\_title\_first\_char

doc\_title, post\_writer

| create index doc\_index1 on N\_document (hash\_key);  create index doc\_index2 on N\_document (post\_date);  create index doc\_index3 on N\_document (post\_title\_first\_char);  create index doc\_index4 on N\_document (doc\_title, post\_writer ); | |
| --- | --- |

In order to further optimize the N\_document table, we have created four indexes: doc\_index1, doc\_index2, doc\_index3, and doc\_index4. Each index is designed to improve the efficiency of specific types of queries by creating a data structure that allows for quick data retrieval based on the indexed columns. The index doc\_index1 has been created on the column hash\_key of the N\_document table. This index is particularly useful when querying for specific documents based on their hash\_key values. The index doc\_index2 has been created on the column post\_date of the N\_document table. This index is beneficial for queries that involve filtering or ordering documents based on their post\_date values. The index doc\_index3 is built on the column post\_title\_first\_char of the N\_document table. This index is valuable when searching for documents based on the first character of their post\_title. The index doc\_index4 is a composite index created on the columns doc\_title and post\_writer of the N\_document table. This composite index is useful for queries that involve filtering or sorting documents based on both the doc\_title and post\_writer values simultaneously.

## **3.2 N\_savedDoc**

(PK) savedDocHashKey, email, savedDocDate, keyword

| create index savedDoc\_index on N\_savedDoc (savedDocHashKey, email, savedDocDate, keyword); | |
| --- | --- |

To further optimize the N\_savedDoc table, we have created an index called savedDoc\_index. This index is designed to improve the efficiency of queries involving multiple columns: savedDocHashKey, email, savedDocDate, and keyword. The purpose of this index is to enable fast data retrieval based on combinations of these columns. By creating this index, the database engine can efficiently locate and retrieve saved documents based on the values stored in these columns.

## **3.3 N\_frequency**

(PK) docID, tfidfWord

Score desc

| create index freq\_index1 on N\_frequency (docID, tfidfWord);  create index freq\_index2 on N\_frequency (Score desc); | |
| --- | --- |

In order to optimize the N\_frequency table, we have created two indexes: freq\_index1 and freq\_index2. The index freq\_index1 has been created on the columns docID and tfidfWord of the N\_frequency table. This index is particularly useful for queries that involve searching or filtering frequency records based on both the document ID (docID) and the TF-IDF word (tfidfWord). The index freq\_index2 has been created on the column Score in descending order (desc) of the N\_frequency table. This index is valuable for queries that require sorting frequency records based on the Score column in descending order.

## 

## **3.4 N\_similarity**

(PK) docID, rcmdDocID

Score desc

| create index simil\_index1 on N\_similarity (docID, rcmdDocID);  create index simil\_index2 on N\_similarity (Score desc); | |
| --- | --- |

In order to optimize the N\_similarity table, we have created two indexes: simil\_index1 and simil\_index2. The index simil\_index1 has been created on the columns docID and rcmdDocID of the N\_similarity table. This index is particularly useful for queries that involve searching or filtering similarity records based on both the source document ID (docID) and the recommended document ID (rcmdDocID). The index simil\_index2 has been created on the column Score in descending order (desc) of the N\_similarity table. This index is valuable for queries that require sorting similarity records based on the Score column in descending order.

## **3.5 N\_tfiword\_cnt**

cnt desc, tfidWord

| create index word\_index on N\_tfiword\_cnt (cnt desc, tfidfWord); | |
| --- | --- |

# To optimize the N\_tfiword\_cnt table, we have created an index called word\_index. This index is designed to improve the performance of queries involving the cnt and tfidfWord columns. The index is constructed to allow for efficient data retrieval based on the combination of these columns. First of indexes is the cnt column in descending order (desc) and second is the tfidfWord column. This indexing enables the database to quickly search, filter, and sort records based on both the cnt and tfidfWord values.

# 

# **4. task solutions**

We solved the task to the best of our ability, using our database which was created by the above method.

The queries used to solve each task, the results, and the execution time shown below.

**task1**

**“ On average, in which month are the most publications released (posted)?**

**Submit your solution along with the query that works on your database**

**schema.”**

| query | |
| --- | --- |
| with month as(  select *substring*(post\_date,6,2) as M  from N\_document  ),  year as(  select *count*(distinct *left*(post\_date,4)) as Y  from N\_document  )  select M, *COUNT*(\*) / (select \* from year) as avg  from month  group by M  order by avg desc  limit 1; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 55 ms, fetching: 21 ms) | 0.049253 |

**task2**

**“Find the five most important keywords (in terms of TF-IDF) in the document that was published in 2011 and has been bookmarked (saved) by the highest number of users.”**

| query | |
| --- | --- |
| with documents(docID) as (  select hash\_key from N\_savedDoc ss join N\_document dd  on dd.hash\_key = ss.savedDocHashKey  where post\_date like '2011%'  group by dd.hash\_key  order by *count*(dd.hash\_key) desc limit 1  )  select tfidfWord from N\_frequency f  WHERE f.docID in (SELECT \* FROM documents)  order by Score desc limit 5; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 17 ms, fetching: 26 ms) | 0.0089505 |

**task3**

**“Give the title of the most similar document to the document that is saved least frequently by the users in the handong.ac.kr domain.”**

| query | |
| --- | --- |
| SELECT N\_document.doc\_title  FROM N\_document  WHERE N\_document.hash\_key =(  select N\_similarity.rcmdDocID  FROM N\_similarity  WHERE N\_similarity.docID = (  SELECT N\_savedDoc.savedDocHashKey  FROM N\_savedDoc  where N\_savedDoc.email LIKE '%handong.ac.kr%'  GROUP BY N\_savedDoc.savedDocHashKey  ORDER BY *COUNT*(\*)  LIMIT 1  )  ORDER BY N\_similarity.Score DESC  LIMIT 1,1); | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 49 ms, fetching: 22 ms) | 0.048687 |

**task4**

**“Find the three most important keywords (in terms of tf-idf) in the second most frequently bookmarked (saved by the users) document amongst the articles authored by “조한범”.”**

| query | |
| --- | --- |
| with joesDoc(hash\_key, cnt) as (  select distinct s.savedDocHashKey, *count*(savedDocHashKey) over (partition by savedDocHashKey) cnt  from N\_savedDoc s join N\_document d  on s.savedDocHashKey = d.hash\_key  where post\_writer = '조한범'  order by cnt desc  )  select docTitle, tfidfWord, score, *row\_number*() over (partition by docID order by Score desc) word\_rank  from N\_frequency where docID in (  select tmp.hash\_key from (  select hash\_key, cnt, *dense\_rank*() over(order by cnt desc) as cnt\_rank  from joesDoc  order by cnt desc  ) as tmp  where tmp.cnt\_rank = 2  )  order by word\_rank limit 6; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 20 ms, fetching: 25 ms) | 0.01341725 |

**task5**

**“For all words that are used in the frequency analysis, show how many times each word has been used in the analysis (how many times each words has been used in the frequency table)”**

| query | |
| --- | --- |
| ## DDL of N\_tfiaord\_cnt table :  # CREATE TABLE T\_tfiword\_cnt AS(  # select tfidfWord, count(tfidfWord) as cnt  # from N\_frequency group by tfidfWord  # order by count(tfidfWord) desc  # );  SELECT \* FROM N\_tfiword\_cnt; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 12 ms, fetching: 28 ms) | 0.001264 |

**task6**

**“Find the ten most similar documents to those of the author who holds the most representative document for keyword “개인.” + Consider only the documents who have records in the frequency and similarity tables”**

| query | |
| --- | --- |
| WITH top\_sim\_docid as (  SELECT docID  FROM N\_frequency  WHERE docID in (  SELECT docID  FROM N\_similarity  )AND tfidfWord = '개인'  ORDER BY Score DESC  LIMIT 1  )  SELECT \*  FROM N\_similarity  WHERE docID in (SELECT \* FROM top\_sim\_docid)  ORDER BY Score DESC  LIMIT 1,10; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 217 ms, fetching: 22 ms) | 0.210576 |

**task7**

**“Among the words that are used the 10th most frequently in the word frequency analysis, locate the document with the 200th highest score. Next, identify the document with the 7th highest similarity to the abovefound document. Please provide the ID, title, and author name for both documents. (You may use the UNION command to combine both results)”**

| query | |
| --- | --- |
| with 200th as (  select docID  from N\_frequency  where tfidfWord = (  select tfidfWord  from N\_tfiword\_cnt  limit 9,1  )order by Score desc  limit 199,1  ), 7th as(  select rcmdDocID  from N\_similarity  where docID in (select \* from 200th)  order by Score DESC  limit 6,1  )  select hash\_key, doc\_title, post\_writer  from N\_document  where hash\_key in (select \* from 7th)  union  select hash\_key, doc\_title, post\_writer  from N\_document  where hash\_key in (select \* from 200th); | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 77 ms, fetching: 24 ms) | 0.05481675 |

**task8**

**“Compare the topic distribution among the documents published in 2018 and 2022, respectively.”**

| query | |
| --- | --- |
| with T\_sum as(  select *sum*(case when *left*(post\_date,4) = '2018' and topic is not null and topic <> '' then 1 else 0 end) as 'sum\_2018',  *sum*(case when *left*(post\_date,4) = '2022' and topic is not null and topic <> '' then 1 else 0 end ) as 'sum\_2022'  from N\_document  )  select \* from(  select topic, (*count*(\*)/ sum\_2018) \* 100 as '2018'  from N\_document,T\_sum  where post\_date like '2018%' and topic is not null and topic <> ''  group by topic,sum\_2018  order by topic) as A join(  select topic, (*count*(\*) / sum\_2022) \* 100 as '2022'  from N\_document,T\_sum  where post\_date like '2022%' and topic is not null and topic <> ''  group by topic,sum\_2022  order by topic) as B using (topic); | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 66 ms, fetching: 23 ms) | 0.05460575 |

**task9**

**“Find the titles (post\_title) and authors (post\_writer) of the two most similar documents (regardless of the published year) to the one with the 10th longest title among those published in 2018.”**

| query | |
| --- | --- |
| with 10stlen as (  select hash\_key  from N\_document  where post\_date like '2018%'  order by *length*(post\_title) desc  limit 9,1  ), S\_DOC as (  select N\_similarity.rcmdDocID  from N\_similarity  where N\_similarity.docID = (select \* from 10stlen)  order by N\_similarity.Score DESC  limit 1,2  )  select post\_title, post\_writer  from DB16.N\_document  where N\_document.hash\_key in ( select \* from S\_DOC); | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 70 ms, fetching: 20 ms) | 0.07071275 |

**task10**

**“Among the documents whose titles start with "ㅈ”, Find the ID, title, and author name of the document with the highest tfidf importance for the keyword "관계". • Consider only the documents who have records in the frequency and similarity tables”**

| query | |
| --- | --- |
| select docTitle, post\_writer from N\_frequency f join N\_document d on f.docID = d.hash\_key  where tfidfWord = '관계' and post\_title\_first\_char = 'ㅈ'  group by docTitle, post\_writer, Score  order by Score desc  limit 1; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 50 ms, fetching: 22 ms) | 0.0416625 |

**task11**

**“Show and compare the yearly distribution of the document counts whose post\_body contains “강경” and that of “대화””**

| query | |
| --- | --- |
| with Ysum as (  select *left*(post\_date,4) as year  from N\_document  where post\_body like '%강경%' and post\_body like '%대화%'  ),Dcount as(  select *count*(\*) as document\_conunt  from N\_document  where post\_body like '%강경%' and post\_body like '%대화%'  )  select year , (*count*(\*)/document\_conunt) \* 100  from Ysum,Dcount  group by year,document\_conunt  order by year; | |
| result | |
|  | |
| execution time(output) | duration(profiling) |
| (execution: 392 ms, fetching: 20 ms) | 0.380893 |

Regarding the execution time, since the duration of the task was such that there were always an unspecified number of concurrent users on the server, there was an error in the execution time for each run, so we performed about 10-20 iterations per task and chose the lowest execution time to record.

# **5. strategies**

When we first tackled the task, we focused on getting a result somehow.

After obtaining the results, we went through the process of modifying the query to find and replace a statement that gives the same result but is more efficient in terms of execution time.

In order to determine which one has a lower execution time, we did theoretical investigations and direct experiments.

Here are some of the cases we found that were replaced to improve performance.

| **LIMIT n OFFSET m -> LIMIT p, q** |
| --- |
| We've tried writing and executing queries for the same result myself and found that there is a performance benefit to not using offset and taking the form of limit p,q. |

| **= -> IN** |
| --- |
| We've seen by reference that there are benefits to using an IN statement instead of using the = operator in a WHERE clause.  ref: [MySQL IN절을 통한 성능 개선 방법](https://jojoldu.tistory.com/565) |

| **TEXT type -> VARCHAR** |
| --- |
| Since the TEXT type is referenced once more to get the actual data, it is less time-efficient than VARCHAR, so we converted it to VARCHAR type. |

| **Minimize DISTINCT** |
| --- |
| DISTINCT statements cause a performance penalty, so we've removed them unless absolutely necessary.  ref: [SQL 성능을 높이는 5가지 방법](https://dataonair.or.kr/db-tech-reference/d-lounge/expert-column/?mod=document&uid=53567) |

| **NATURAL JOIN, JOIN … USING, JOIN ON, -> JOIN ON** |
| --- |
| While NATURAL JOIN, JOIN ... USING, JOIN ON can perform the same function  Based on GoogleAi Bard's answer, we decided that it is more efficient to use JOIN... ON.  ref: |

| **COUNT(\*), COUNT(column), COUNT(DISTINCT(column)) ->COUNT(\*)** |
| --- |
| When using COUNT, there was a performance benefit to using it with the \* symbol rather than a specific column.  ref: [[MySQL] COUNT의 잘못된 인식과 속도 차이](https://yjh5369.tistory.com/entry/MySQL-COUNT%EC%9D%98-%EC%9E%98%EB%AA%BB%EB%90%9C-%EC%9D%B8%EC%8B%9D%EA%B3%BC-%EC%86%8D%EB%8F%84-%EC%B0%A8%EC%9D%B4) |

# 

# **6. summary of DB**

As a result, our database which was used to generate all the results, has the following size.

Since we did not delete the normalized tables in phase 1 for stability, we supplemented the query with the naming policy of the tables (starting with ‘N\_’ ) that we applied earlier for accurate size measurement in phase 2.

## **6.1 Database size**

**: 478208.0 KB**

| query |
| --- |
| SELECT table\_schema AS 'DB16',  *ROUND*(*SUM*(data\_length+index\_length)/1024, 1) AS 'Size(KB)'  FROM information\_schema.tables  WHERE table\_schema = 'DB16'  AND TABLE\_NAME LIKE 'N\\_%'; |
| result |
|  |

## **6.2 Table size**

| query |
| --- |
| select table\_schema,  table\_name,  *round*(data\_length / (1024), 1) as 'data(KB)',  *round*(index\_length / (1024), 1) as 'idx(KB)'  from information\_schema.tables  where table\_type = 'BASE TABLE'  and table\_schema = 'DB16'  AND TABLE\_NAME LIKE 'N\\_%'; |
| result |
| Although the index was used, we guess it was not reflected in the information schema. |