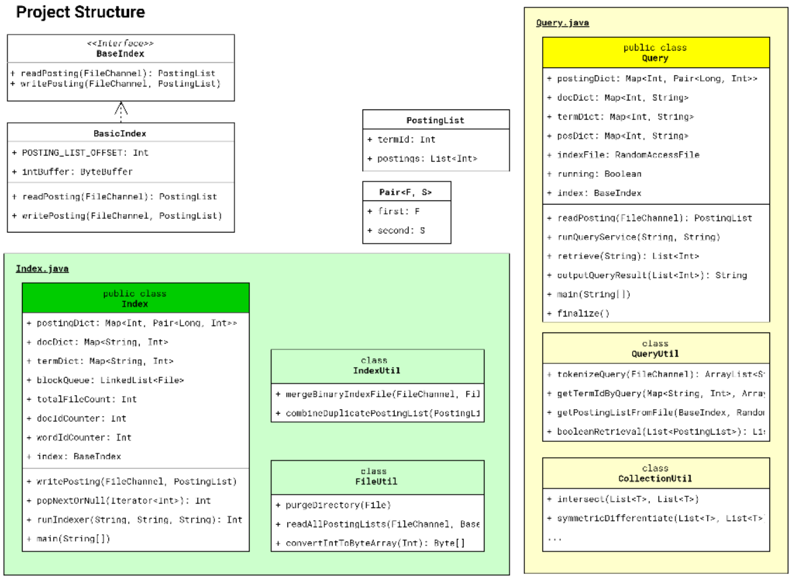
Project 1: YouGle
SUMMARY REPORT

Members
1. Krittin Chatrinan ID 6088022
2. Anon Kangpanich ID 6088053
3. Tanawin Wichit ID 6088221

Architecture Summary

The organization of the project can be divided into 2 parts: indexing and querying. Each part contains nested inner Util Classes which provide methods that facilitates the process as shown below.

Project Structure


<<Interface>>
BaseIndex
+ readPosting(FileChannel): PostingList
+ writePosting(FileChannel, PostingList)
BasicIndex
+ POSTING_LIST_OFFSET: Int
+ intBuffer: ByteBuffer
+ readPosting(FileChannel): PostingList
+ writePosting(FileChannel, PostingList)
PostingList
+ termId: Int
+ postings: List<Int>
Pair<F, S>
+ first: F
+ second: S
Index.java
public class Index
+ postingDict: Map<Int, Pair<Long, Int>>
+ docDict: Map<String, Int>
+ termDict: Map<String, Int>
+ blockQueue: LinkedList<File>
+ totalFileCount: Int
+ docIdCounter: Int
+ wordIdCounter: Int
+ index: BaseIndex
+ writePosting(FileChannel, PostingList)
+ popNextOrNull(Iterator<Int>): Int
+ runIndexer(String, String, String): Int
+ main(String[])
class IndexUtil
+ mergeBinaryIndexFile(FileChannel, Fil
+ combineDuplicatePostingList(PostingLi
class FileUtil
+ purgeDirectory(File)
+ readAllPostingLists(FileChannel, Base
+ convertIntToByteArray(Int): Byte[]
Query.java
public class Query
+ postingDict: Map<Int, Pair<Long, Int>>
+ docDict: Map<Int, String>
+ termDict: Map<Int, String>
+ posDict: Map<Int, String>
+ indexFile: RandomAccessFile
+ running: Boolean
+ index: BaseIndex
+ readPosting(FileChannel): PostingList
+ runQueryService(String, String)
+ retrieve(String): List<Int>
+ outputQueryResult(List<Int>): String
+ main(String[])
+ finalize()
class QueryUtil
+ tokenizeQuery(FileChannel): ArrayList<S
+ getTermIdByQuery(Map<String, Int>, Array
+ getPostingListFromFile(BaseIndex, Rando
+ booleanRetrieval(List<PostingList>): Li
class CollectionUtil
+ intersect(List<T>, List<T>)
+ symmetricDifferentiate(List<T>, List<T>
...

Indexing

KEY STEPS

Indexing involves steps as follows. Starts from `Index.runIndexer()`

1. For each block, get the document files inside.
   - ➢ For each file inside, the program records it into `docDict: Map<String, Int>`, constructs `blockPostingLists: Map<Integer, Set<Integer>>` to temporary store the Posting Lists with a mapping between `termId` and a `TreeSet` of `docId` and read it line-by-line.
     - ➢ For each line, the program splits words.
       - ➢ For each word, the program records it down to `termDict: Map<String, Int>`
   - ➢ After a block processing is finished, it constructs binary index file for that block as well as processing `docFreq` inside `postingDict: Map<Int, Pair<Long, Int>>.`
2. After every block got its index files, Block Merging process begins.
   - ➢ For each pair of blocks inside `blockQueue: LinkList<File>` (The following is located in a static method `IndexHelpers.mergePostingList()`)
     - ➢ Read a pair PostingLists (`p1` & `p2`) at a time to save memory from both files (`bf1` & `bf2`). Compare both `termId`. Merge them if both `termId` are equal.
       - ➢ Write the array of merged PostingList into merge file (`mf`) as well as updating `bytePosition` inside `postingDict: Map<Int, Pair<Long, Int>>.`
3. **Write `termDict`, `docDict`, and `postingDict` into files**
4. Indexing process is finished

RESULTS

Tested with JVM max heap size = 512 MB (**-Xmx512m**)

```
SMALL DATASET
Indexing Test Result: ./index/small:
        Total Files Indexed: 6
        Memory Used: 0.51152 MBs
        Time Used: 0.043 secs
        Index Size: 2.17437744140625E-4 MBs
        Alright. Good Bye.
```

```
LARGE DATASET
Indexing Test Result: ./index/large:
        Total Files Indexed: 98998
        Memory Used: 146.332192 MBs
        Time Used: 128.287 secs
        Index Size: 55.37303924560547 MBs
        Alright. Good Bye.
```

```
CITESEER DATASET
Indexing Test Result: ./index/citeseer:
        Total Files Indexed: 18824
        Memory Used: 406.39152 MBs
        Time Used: 217.832 secs
        Index Size: 64.60990905761719 MBs
        Alright. Good Bye.
```

Querying

KEY STEPS

Querying involves steps as follows. Starts from
`Query.runQueryService()`

1. **Read `termDict`, `docDict`, and `postingDict` from the written files**
2. Tokenize the incoming user query. The result is an **`ArrayList`**`<String>` of Token
3. For each Token, find the Term Id if there are any. The result is an **`ArrayList`**`<Integer>` of Term Id
4. For each Term Id, to get **`PostingList`**, find the Byte Position from **`postingDict`**, set the position of the **`FileChannel`** of the index file to that Byte Position, and read from the index file starting from that position. Finally, the outcome is an **`ArrayList`**`<PostingList>`.
5. Sort the **`ArrayList`**`<PostingList>` by its Document Id **`ArrayList`** size.
6. To retrieve a relevant result according to **Boolean Retrieval** method, the document Id ArrayList in each PostingList must be intersected by using a static method **`Query.CollectionUtil`**`.intersect()`. The outcome is an **`ArrayList`**`<Integer>` of document Ids that contain all tokens.
   - Please note that intersecting multiple Arrays have to be done in paired manner similar to merging index files.
7. **Write filenames into the output file.**
8. Querying Process is finished

RESULTS

Tested with JVM max heap size = 512 MB (**-Xmx512m**)

```
SMALL DATASET
Query Test Result: [hello, bye, you, how are you, how are you
?]:
        Memory Used: 0.537136 MBs
        Time Used: 0.013 secs
        No problem. Have a good day.
```

```
LARGE DATASET
Query Test Result: [we are, stanford class, stanford students,
very cool, the, a, the the, stanford computer science]:
        Memory Used: 133.15348 MBs
        Time Used: 1.04 secs
        No problem. Have a good day.
```

```
CITESEER DATASET
Query Test Result: [shortest path algorithm, support vector
machine, random forest, convolutional neural networks, jesus,
mahidol, chulalongkorn, thailand, polar bears penguins tigers,
algorithm search engine, innovative product design social
media, suppawong, tuarob, suppawong tuarob, suppawong tuarob
conrad tucker]:
        Memory Used: 24.910048 MBs
        Time Used: 0.165 secs
        No problem. Have a good day.
```

QUESTIONS

QUESTION A

The trade-off of different sizes of blocks

Bigger blocks require more memory to store, but with lower write overhead. On the other hand, smaller block sizes mean there are more blocks to be merged; thus, more Disk writing overhead. Actual merging process which requires Disk read may not affect the total CPU burst time, it is independent of the size of each block.

Balancing memory and minimizing indexing time

Disk writing and reading are considered expensive in terms of time; therefore, in order to minimize time and memory usage, choosing the right ByteBuffer size is crucial. Bigger buffer can read and write faster, but at the cost of memory usage. Smaller buffer, on the other hand, can read and write slower, but lower memory usage. In our project, we utilize the maximum prime factor of the size of the data to be written. With this, we can minimize the call of Disk read and write methods; thus, lower indexing time. For example, we have 20 integers or 80 bytes to write or read, 20 has prime factors of 2, 2 and 5. We select 5 multiply by 4 bytes (An Integer has 32 bits/4 bytes) or 20 bytes as the Buffer size to reduce Disk IO overheads and to perfectly fit the data size (no remainder to avoid buffer underflow exception).

QUESTION B

Scalability limitations

1. **Sequential block indexing** – Iterating through all of the blocks sequentially could lead to a linear time complexity which is undesirable for a large system with a large dataset.
2. **Large data structures** – Maintaining large Data Structures like docDict, termDict and postingDict can be troublesome for a very large dataset due to its global scope which lives longer than looping files in one block. Therefore, as blocks are indexed, the system gets slower due to heavy Garbage Collection events, and eventually, the process gets terminated because the system runs out of memory.
3. **Slow and Redundant Index files merging algorithm** – Linear merging time complexity comes from the fact that we iterate through the file, comparing PostingList one-by-one. The algorithm is also considered redundant because of the way we move back the file pointer to where it was before reading. This is considered an overhead because of the redundancy in reading the same PostingList again.
4. **Large index file** – Index file could potentially get very large due to a large dataset. This can lead to severe fragmentations on the disk especially magnetic ones which lead to slower read and write due to seek time.

QUESTION C

Improvement Ideas

1. **Introducing Parallelism** – Iterating through all blocks linearly consume time. Having multiple threads could speed up the process but at the expense of more ram usage and more coding. Parallelism can make this program very scabable to a very large dataset.
2. **Maintaining block-level Large data structures** – Instead of storing docDict, termDict and postingDict globally we should store them after finishing one block like binary index file to save memory when the operation runs long.
3. **Index files merging algorithm with Parallelism** – Using multiple threads to divide up the merging process can boost up the speed, but again with more memory used.
4. **Choosing the right file system** – Index file could get very large; therefore, choosing the right filesystem can potentially boost up the performance.

BONUS: RANKED QUERY

Implementation Details

We have implemented a Ranked Query inside RankedQuery.java. The file contains the whole bonus part because ranking involves heavily in both Indexing and Querying. In Indexing, we precalculate the score as well as collect term frequency for each term in each document; therefore, this requires a big data structure to accommodate which is `docTermFrequency: Map<Integer, Map<Integer, Integer>>`. The map object is suitable for the work because if there is no frequency for a term in a doc, the entry in the map will not present, so it could reduce memory usage. However, this certainly requires more computation power and time. More data means we have to write more; therefore, we write the entirety of `docTermFrequencyMap` to **score.matrix** file as well as vector norms. Then in the query, we load the entirety of it into the memory again. (We know how to make it more efficient by using Byte Position number.) We later process the user query and get associated term IDs. Next, we create the document vector for the query itself. We get associated doc IDs inside `PostingList` from the Index file via `postingDict`. Then we calculate the similarity between the related document vectors and query vector. And eventually, return the top-10 result in a ranked manner.

Score Calculation

The calculation of similarity is referenced from Asst. Prof. Dr. Suppawong Tuarob's week 4 lecture slide. To calculate the score we choose TF-IDF with the scheme **ltc.ltc** with cosine similarity. All calculation methods are located inside RankingMathHelper class.

SCORE (WEIGHT)	INVERSE DOCUMENT FREQUENCY
$w_{t,d} = tf_{t,d} \times idf_t$	$idf_t = \log_{10} \frac{N}{df_t}$
TERM FREQUENCY	**VECTOR NORM**
$tf_{t,d} = 1 + \log_{10} tf_{t,d}$	$\|\|x\|\|_2 = \sqrt{\sum_i x_i^2}$

CONSINE SIMILARITY

$$\cos(\vec{q}, \vec{d}) = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

Example Results

Full version available in the sent zip file inside ./output/bonus/citeseer
We also include testcase which reflect Aj. Suppawong Slide Week 4 Homework in ./output/bonus/week4Homework.

***Figure:** A test query with ranked result and term frequency table*
Query: **"suppawong tuarob"**
Dataset: Citeseer

```
Query: [suppawong tuarob]
FINAL DOCUMENT RANKING
Rank #1     with Score = 0.183140 is 29/10.1.2.1.2.txt      (#15307)
Rank #2     with Score = 0.166530 is 29/10.1.2.1.24.txt     (#15315)
Rank #3     with Score = 0.150811 is 29/10.1.2.1.15.txt     (#15302)
Rank #4     with Score = 0.126122 is 29/10.1.2.1.218.txt    (#15312)
Rank #5     with Score = 0.105930 is 29/10.1.2.1.19.txt     (#15306)
Rank #6     with Score = 0.092919 is 29/10.1.2.1.7.txt      (#15321)
Rank #7     with Score = 0.090536 is 29/10.1.2.1.21.txt     (#15309)
Rank #8     with Score = 0.089452 is 29/10.1.2.1.4.txt      (#15318)
Rank #9     with Score = 0.088664 is 29/10.1.2.1.9.txt      (#15324)
Rank #10    with Score = 0.085038 is 29/10.1.2.1.22.txt     (#15313)
```

RANK	1	2	3	4	5	6	7	8	9	10
Tf of **"suppawong"**	3	9	2	0	2	2	2	1	2	2
Tf of **"tuarob"**	2	8	6	7	13	2	6	1	4	11
Norm	28.65	46.71	41.13	29.33	64.91	56.46	68.52	45.08	59.10	71

Analysis

As demonstrated by the result above, the document with an ID of 15307 is ranked as the first one because of the TF-IDF of words that matched with the query is high and low norm value. Although, the others such as the document with an ID of 15312 or an ID of 15302 will have higher TF-IDF value of some words, the main factor comes from the norm value because the denominator of the product of the norm of document and the norm of the query gives the bigger different score than the numerator from dot product between TF-IDF value of word in document and TF-IDF value of word in query.