Project Phase 3

CSE 510: Database Management Systems Implementation

Spring 2021

Implementation of various Relational Database Management System functionalities

Riya Jignesh Brahmbhatt
Rutva Krishnakant Patel
Venkata Satya Sai Dheeraj Akula
Nithya Vardhan Reddy Veerati
Haard Mehta
Karthik Nishant Sekhar

Contents

Abstract

In the phase 3 for the project, we have further extended Minibase to include relational database management system functionalities such as GroupBy on various aggregation functions, Joins using different algorithms, TopKJoins using the naive approach as well as the NRA algorithm. Minibase only supported unclustered BTree indexes. We have further extended it to support clustered BTree index as well as clustered and unclustered hash index. Finally, we have integrated all the functionalities of phase 2 (skyline computation) as well as phase 3 into a query interface as a part of the last task. Lastly, we have tested all the functionalities on the data files provided and analysed the number of reads and writes. This phase includes the implementation of a persistent database.

Keywords

Clustered BTree index, Clustered hash index, Unclustered hash index, GroupBy operator, aggregation attribute(s), Join operator, Join attribute, Index Nested Loop Join, Hash Join, Join conditions, NRA based Top K Joins, Persistent Database, Query interface, Target utilization, Hash buckets, Merge attribute

Introduction

This phase is divided into 6 tasks. Task 1 constitutes the implementation of clustered BTree index which has the functionality of insertion deletion and scan of the index. Task 2 implements the clustered and unclustered hash index which has insertion, deletion and scan. It uses the concept of target utilization of the hash buckets. Task 3 implements the group_by operator using both the sorting and hashing based techniques on the max, min, avg and skyline aggregation functions. Task 4 constitutes the implementation of index nested loop join and hash join, which was previously absent in minibase. Task 5 implements the Top K Join using the naive approach as well as the NRA algorithm. Task 6 is a query interface which can perform various relational database management system functions like create table, compute skyline, perform join etcetera. On all the aforementioned tasks, we have computed the number of reads and writes on the disk and analysed the results.

- Terminology

1. Hash buckets- It is a storage unit. Tuples which have the same hash value on a particular key belong to the same bucket.
2. Target utilization- The percentage space utilized in a page before splitting it

3. Join attribute- The attribute in the inner and outer relation on which the join has to be performed
4. Merge attribute- The attributes on the inner and outer relation that have to be combined to form a single attribute. In case of TopKJoin, we merge attributes from both the relations by computing the average.
5. Aggregation attribute- The attribute on which the aggregation function has to be applied after grouping the values of the group attribute.
6. Aggregation function- The function to be performed on the groups in the case of group_by operator. Minibase will support four aggregation functions, namely, max, min, avg and skyline.
7. n_pages- The number of buffer pages available
8. Persistent database- A persistent database is stable as well as recoverable
9. NRA Top K Joins- No Random Access based Top K Join algorithm
10. Join conditions- The conditions on which a join can be performed between two relations. These are >, <, ≥, ≤, =.

- Goal Description
    - Task 1

      This task implements the clustered BTree index on a given attribute. It can be made at the time of table creation. For example, the query

      create_table CLUSTERED BTREE 2 Customers

      will create a clustered btree index on the second attribute of the input data file customers at the time of table creation. A clustered index has a key and a pointer to a page pair. A scanner created on the clustered index will scan all the tuples one by one. It also facilitates deletion of records.

    - Task 2

      This task is divided into two subtasks.
      Task 2a- Create an unclustered linear hash index on the specified attribute. Each page in the hash bucket will have a <*key, tuple_pointer*>

pair to the tuples in the datafile. Once a table has been created, the following query will create a unclustered linear hash index on attribute number 2 of the table *Customers*

create_index HASH 2 Customers

This task also facilitates insertion of a record in the table as well as the index file. This is done by keeping track of the split pointer and hash bucket target utilization. It also has the delete functionality, where we can specify the record to be deleted. Scanner will scan and give a tuple as output one by one.

Task 2b- The goal is to create a clustered hash index on the specified attribute at the time of table creation. Each hash bucket page will contain a <*key, page_pointer*> pair to the pages in the datafile. The following query will create a clustered linear hash index on attribute 2 of the table *Customers:*

create_table CLUSTERED HASH 2 Customers

This task also has a scanner to scan the tuples one by one. It facilitates insertion of a record as well as deletion.

- Task 3

Task 3 is divided into two subtasks and it implements the group_by operator on the specified group by attribute and applies the aggregation function on the specified aggregation list. Min, max, avg and skyline are the four aggregation functions. Min, max and avg will give one tuple as output for each group of the group by attribute whereas the skyline function can have multiple aggregation attributes as well as multiple tuples in the output per group.

Task 3a- This task uses the sorting functionality of minibase and creates a temporary heapfile sorted on the group by attribute using Sort.java. It then calculates the aggregation function based on the specifications from the user. Consider the following query as example:

GROUPBY SORT MAX 1{2} Customer 20

This query will group the first attribute of the Customer table and find the maximum value of the second attribute for each group of the first attribute. The n pages to perform sorting are 20.

Task 3b- This task uses the hash index based technique(hash index is formed on the group by attribute) to perform group_by because the scan of a hash index will give all the same valued group by tuples together in the output. Consider the following query:

GROUPBY HASH MIN 2{3} Customer 20

This query will scan the existing/created hash index and find the min value of the third attribute for each group of the second attribute on the Customer table.

- Task 4

This task is divided into two subtasks. Minibase does not have the functionality for index nested loop join as well as hash join.

Task 4a- Index Nested loop join is implemented. The user gives the join condition, outer table name, join attribute for the outer table, inner table name and join attribute for the inner table as the input. Both the relations are joined on the join condition and the resultant tuple, which has the attributes of both the relations, is displayed one by one. Consider the following query:

JOIN INLJ Customer 2 Products 5 > 20

This query, for every tuple in the table Customer, will “compare” attribute 2 to the fifth attribute of table Product and join the ones where the value of the fifth attribute is less than the value of the second attribute.

Task 4b- Hash join is implemented which has the equi join condition. The user input is the outer table name, join attribute for that table, inner table name and join attribute for the inner table. The output will have the attributes of both the relations and they will have equal values on the join attributes. Consider the following query:

JOIN HJ Customer 5 Product 2 = 10

The result of the above query will have all the attributes of both the relations but the attribute number five of the table Customer and attribute number 2 of the table Product will be equal. The join is performed based on hashing the join attributes. When the same hash function is applied to both the join attributes, the tuples with the same value will be inserted in the same hash bucket. Then the corresponding hash buckets of the inner and outer are joined using Nested Loop join.

- Task 5

  This is divided into two subtasks. The goal is to implement the Top K Joins operator, first using the naive approach, then using the NRA algorithm.

  Task 5a- This task implements the Top K Joins using the naive approach. The outer and the inner relation are joined on the join attribute of both the relations using the Hash Join operator of task 4b. An attribute is added to the joined tuples which stores the average of the merge attributes of the inner and the outer relations. Then these tuples are sorted on the average attribute in descending order. Finally, the top k tuples are returned as the result.

  Task 5b- This task implements the Top K Joins using the NRA algorithm which maintains the lower limit, upper limit and threshold. Unlike the naive algorithm, there are no random accesses. The user inputs the names of both the tables, their respective join and merge attributes to compute the top k results.

- Task 6

  For this task, a query interface is implemented where the user can open and close a database, create tables and indexes, insert and delete data, and call other queries on the database. The queries include Skyline, GroupBy, Join, and TopKJoin with their various implementations.

- Assumptions

  - The target utilization for hash buckets is 80% in our code
  - In Top K joins, the merge attribute is either an integer or a real number, because we compute the average of the two merge attributes

- In the implementation of the group_by operator, we have assumed that the aggregation attribute will be an integer or real number because we have to compute the max, min, avg or skyline of the aggregation attribute(s).
- In the hash join implementation, we have used the modulus 4 hash function on the join attribute. We have used the following hash function if the join attribute is of type string

```
hash = 7;
for (int i = 0; i < (temptuple.getStrFld(key)).length();
i++) {
      hash = hash * 31
     temptuple.getStrFld(key).charAt(i);
 }
 hash = hash%N;
```

Description of the proposed solution/Implementation

Task 1 Clustered BTree Index

For clustered B Tree we are using the built in unclustered B Tree index and instead of setting the RID of a tuple we are storing the PID of the page in there. For storing a page and getting access to the page we used our custom version of heap file so that its size is fixed to the length of a single page. The max element of a page is stored in the index file so that while retrieving the record we can get faster access to the tuple by doing a range search. Clustered B Tree takes care of scenarios where the new tuples that get inserted push away the old tuples to new pages, when the indexes are repeated, when the repeated indexes overflow to a new page etc.
The worst case scenario would be one when we are inserting tuples in descending order then all the tuples would be pushing each other in the heap file. So we provide a bulk load function which sorts the records and then inserts them into the Clustered BTree.

Insert (Tuple t):
Insert is a recursive function that does a range search on the index to determine the bucket in which to place the incoming tuple. Insert takes care of overflow tuples that get pushed off the bucket which would be recursively checked to determine the bucket in which to place them. When we insert a new tuple there may be a case that the max element of the page is not modified and there may be a case that the max element of the page got modified. In the second scenario we need to update the index.

Delete (Tuple t):
Delete removes the tuple from the appropriate bucket by doing a range search, In this case also the max element of the bucket may change which leads to updating the index file. If all the elements of the page are removed then the leaf node from the index file is removed and the present bucket is deleted from the memory.

Delete_multiple (Tuple t)
Delete_multiple is a wrapper around the delete function to remove multiple instances of the same tuple.

BulkLoad (String h)
BulkLoad takes a heap file as input and sorts the data then inserts them into the clustered btree.

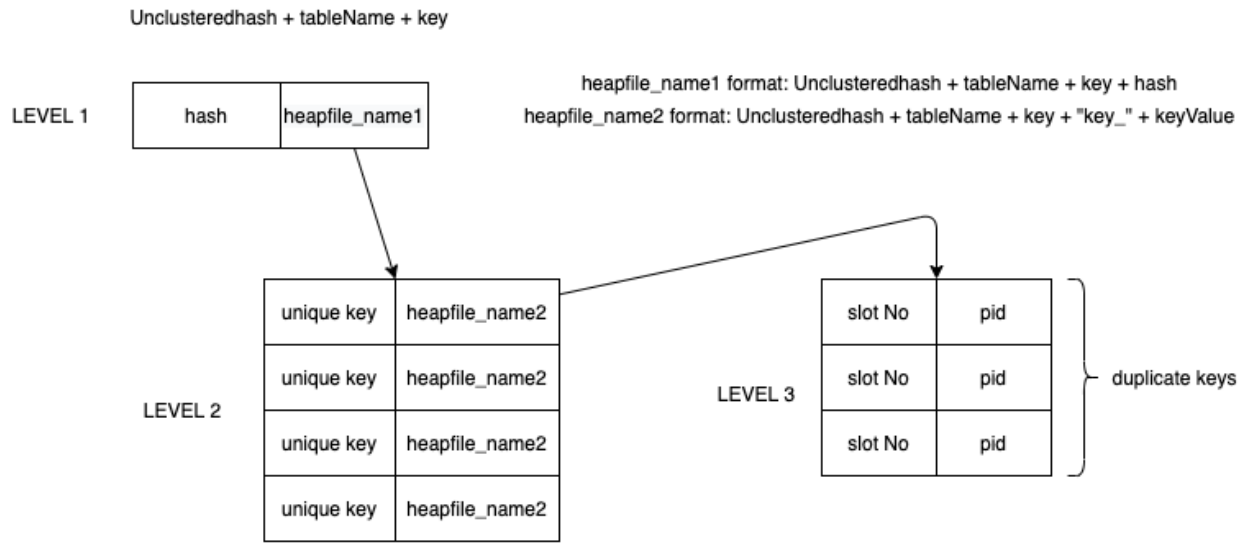
get_next(RID rid)
get_next(rid) is a function similar to what a heapfile provides. Its function is to provide the next record from the clustered B tree and modify the rid such that it points to the next record.

getRecord(RID rid)
getRecord does a range search on the clustered b tree to find the record and return the record if it is found in the heap file

Task 2: Hash Index

Task 2a Unclustered Linear hash index

The unclustered linear hash index maintains a set of hash buckets (Here we have considered each hash bucket as a Heapfile). Each bucket consists of <key, tuple pointer>.
For our implementation, we have formed a three level heap file structure for Index files. The data file is implemented using Heapfile.



UNCLUSTERED LINEAR HASH INDEX STRUCTURE

The following functionalities are implemented in unclustered linear hash index:

Insert(): This will insert all the records from the datafile to the table name and the key attribute specified by the user. For insertion, we have used static hashing by predefining the number of buckets (N). We have hardcoded the target utilization as 80%.

We are using the normal hash function i.e. hash = keyValue%N for static hashing. We have used the hash function mentioned in the assumptions above, for any attribute of type string.

The number of buckets such that the target utilization is maintained is calculated by the below mentioned formula.

```
temp =
(HFPage.MAX_SPACE-HFPage.DPFIXED)/(t.size()+HFPage.SIZE_OF_SLOT);
N = Math.round((_noOfRecords)/(temp*_targetUtilization/100))+1;
```

This way, when we have records in bulk from beforehand, we don't have to perform linear hashing and maintain split pointers.

InsertRecord(): This will take a tuple as an argument that we want to insert in an already created index. Before every insertion, we maintain a Split Pointer that will be initialized to 0 and the number of buckets. As discussed in the class, we will increase the split pointer and add a bucket whenever the page utilization exceeds the target utilization followed by rehashing the elements in the bucket that split pointer initially pointed to.

Delete(): In our implementation, we pass the tuple that is to be deleted in the argument.

For deletion there will be several cases:

1) Only one clustered index exists on the table
2) One or many unclustered index exists on the table
3) Several unclustered indexes exist on the table and one clustered index exists on the table.

-First case, we will discuss in the clustered hash index section of the report.
-For the second case, if only unclustered index exists then, normally we first scan the tuple/record we want to delete. But here, because of the naming convention, we can directly access the file without scanning and get the list of the RIDs for all the tuples from the index file, delete them (using the list of RIDs) from the data file and then delete the RIDs from the last level of the index file.
We delete all the heapfiles that are empty at any level and merge the buckets if it exceeds the target utilization.
-For the third case, if there exists a clustered index as well as an unclustered index then we first perform get a list of page ids from the clustered index. We then open that page and find the list of the RIDs of the tuples are equal to the tuple we want to delete. If all the RIDs are the same as the tuple to be deleted then we empty the page and delete that pageid entry from the last level heapfile in the index file. Then we delete those RIDs from the data file and move to the unclustered deletion. We do not need to delete from the data file, as it has been done during the clustered deletion part. We delete the list of RIDs from the unclustered index file.

Scanner(): For this we have implemented 2 types of scanners, one that returns all the tuples grouped together and another one returning all the tuples with a specific key value. We have used the recursive function for its implementation which is the most efficient one to perform the scans.

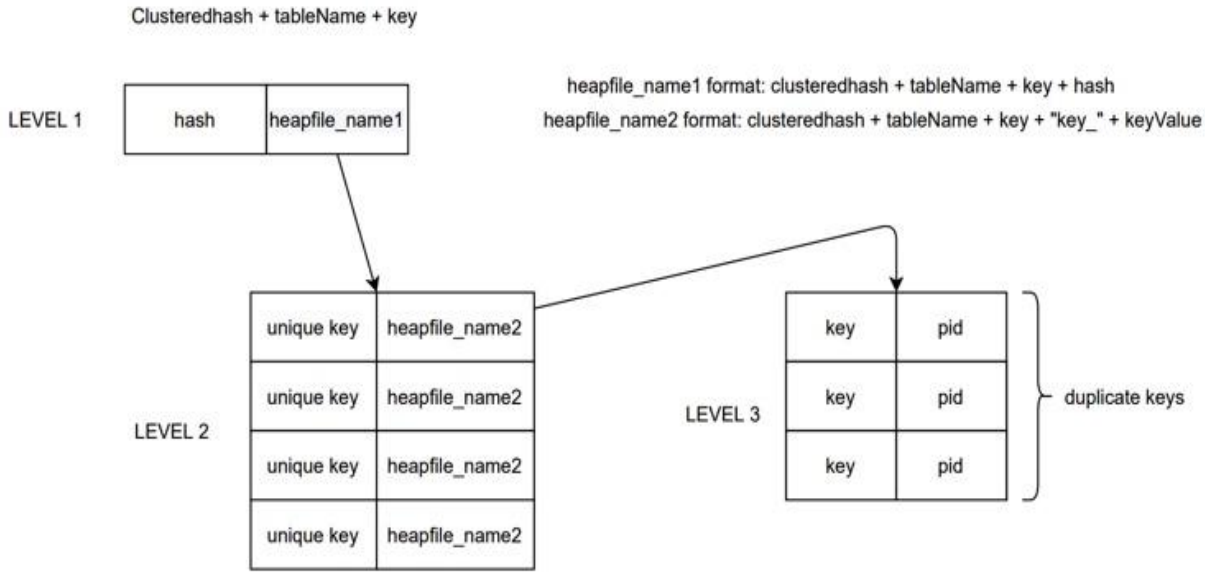
Task 2b- Clustered linear hash index

The clustered linear hash index maintains a set of hash buckets (Here we have considered each hash bucket as a Heapfile). Each bucket consists of <key, page pointer>.
For our implementation, we have formed a three level heap file structure for Index files. The data file is implemented using Heapfile.
The following functionalities are implemented in clustered linear hash index:

Insert(): In Clustered Linear Hash Index, we data file and the index file grows together. We initialize the number of buckets with 2 pages. As the tuples come, they get inserted into the index file after it checks if the page utilization is less than the target utilization. If the page utilization exceeds the threshold then it increases the split pointer and increases the number of buckets (perform the linear hashing). If the split pointer reaches the count of N then the split pointer is reset to 0 and the N changes to 2N. Along with this, we rehash the elements/tuples in the bucket that the split pointer was pointing to previously. After we insert the page id in the index, we insert the record in the datafile at that particular page id. In our implementation, we have created a method called insertRecordAtPage.

Delete(): From the 3 cases mentioned above in 2a, we will talk about the first case i.e. only clustered index exists on a table. In this case, we search for the tuple to be deleted (happens directly in our case because of the file naming convention) and search for the page ids. On opening the pages individually, we will compare the tuples inside the page with the tuple to be deleted and form an array list of all the RIDs to be deleted. We delete the list of RIDs from the data file and from the page. If the page becomes empty we delete the page and remove the pageid from the last level bucket of the index file. We keep deleting the heapfiles in the 3 level structure if they are empty.

Scan(): For this we have implemented 2 types of scanners, one that returns all the tuples grouped together and another one returning all the tuples with a specific key value. We have used the recursive function for its implementation which is the most efficient one to perform the scans.



CLUSTERED LINEAR HASH INDEX STRUCTURE

Task 3 Group By operator

The *groupby* operation is implemented on the groupby attribute and it applies the aggregation function on the aggregation attribute(s). Following are the aggregation functions:

- Min
- Max
- Average
- Skyline

Min, max and average can be applied on a single aggregation attribute, while skyline operation can be applied on a list of aggregation attributes(which act as the preference list of the skyline). For min, max and avg, there will be only one output tuple per group, whereas skyline can have multiple tuples per group. For max, min and skyline, we give all the attributes as the output, but for avg, only the group by attribute and the average for that group is in the result.

Task 3a) Group By with Sort

The GroupBywithSort iterator relies on the sorting of the input tuples on the group_by attribute. If a Btree index exists on the group_by attribute, then the tuples will be sorted in descending order and the GroupBy operator can be applied. But, if a Btree index(clustered or unclustered) does not exist, the tuples are sorted using the Sort operator provided by Minibase. Now, all the same valued tuples on the group by attribute are grouped together in the sorting order. When both clustered and unclustered BTree index exists, we prefer the clustered index due to its lesser cost while scanning. The tuples are somewhat “sorted”, so we can easily reach the page containing the tuple and then scan it.

Task 3b) Group By with Hash

The GroupBywithHash iterator will compute the result in 3 different scenarios. In this case, the tuples are not sorted as in task 3a, but as the value of the field of the group_by attribute belonging to the same group in each tuple is equal, they are together in the output of the index scan because the same hash function has been applied and so they will exist in the same hash bucket. Hence, it solves the purpose of grouping. In the first scenario, it checks if there exists a clustered hash index on the group_by attribute. If it does, it scans it. In the next scenario, if a clustered hash index does not exist, it checks for an unclustered hash index on the group_by attribute and scans it. If both exist, clustered is given priority over unclustered. Clustered is given priority

because the scanning is less expensive as compared to unclustered. In the third scenario, if neither exists, then an unclustered hash index is created and scanned.

Once we obtain the tuples in the desired order as in both 3a and 3b, the task is to apply the aggregation function on the aggregation attribute. For this, the main goal is to be able to compute it on the same group and then change the group. There is a variable that maintains the name of the group and restarts the computation of the aggregation function once the group changes. For example in max computation, the value of the max variable is declared from the first tuple. Then it is compared with all the other tuples of the group and updated if we find a greater max value, and at the end of the group, the tuple with the max aggregation attribute is inserted into the result. A similar approach is adopted for min, avg and skyline computation. All the tuples of the same group are inserted into a heap file and then the BlockNestedSkyline method is used to compute the skyline objects of each group. Once a group is done, the heapfile is re-initialised and now will contain the tuples of the new group for skyline computation. For the average computation, we keep adding the value of the aggregation attribute to a “sum” variable and once the group comes to an end, divide the sum variable by the number of tuples of that group. The number of tuples per group are maintained through the “group_count” variable.

Task 4 Join Operator

This task implements the Join operator. Minibase already has the functionality of Sort merge join and Nested Loop Join. Here we make additions for Index nested loop join and hash join.

Task 4a Index Nested Loop Join

The input for Index nested loop join is name of inner relation, name of outer relation, the join attributes for both the relations and number of buffer pages. First we check if an index exists on the join attribute of the inner relation. This index can be either hash index or btree index. Again, as mentioned previously, clustered index is given priority over unclustered index. If neither of the four indexes exist, we perform nested loop join on the entire inner and outer relation. The advantage of having an index is that for every tuple of the outer relation, we have to scan the index. The Btree unclustered index for example has the CondExpr parameter as input. In the CondExpr, we are able to specify the condition (>, <, ≥, ≤, =) as well as the join attribute, so the scanner only outputs the desired tuples and not all. So this saves computation time. The last resort is Nested Loop Join.

Task 4b Hash Join

This takes as input the inner relation name, outer relation name, join attribute for each relation and the number of buffer pages available. Hash join is implemented on the equi join condition only. Hash join is implemented by first checking if a hash index exists on the join attribute. If it does, we scan the hash index, if it does not we perform static hashing. When both the indexes exist, we prefer the clustered index over the unclustered one as explained in previous situations. For static hashing, the hash function we have used is modulus 4 on the join attribute. As mentioned in the assumptions, we have used a different hash function of the join attribute type string. Initially we create 4 buckets(heapfiles) for the outer relation and 4 for the inner relation because the result of modulus 4 will be either 0, 1, 2 or 3. As we are applying the same hash function on both the inner and the outer relations, if the join attribute value of both the relations is the same, they will be inserted in the same corresponding buckets. Due to this very reason, we then perform a nested loop join between outerBucket 1 and innerBucket1, outerBucket2 and innerBucket2 and so on. This makes the computation less expensive because if we perform nested loop join on the entire relation, then the inner loop has to run each time for every tuple of the outer relation and be compared. This method reduces the size of the inner and outer loops to a significant extent.

Task 5

Top K Joins has the inner relation, outer relation, join attributes for both, merge attributes for both and the value of k as input. Here we have used the average value of both the merge attributes to output the top k results. The output tuple will have all the attributes of both the tuples as well as an additional attribute for the average.

Task 5a Hash based Top K Joins

Here we first apply the hash join implemented in task 4b on both the inner and outer relation. Then when we get the output tuples one by one, we compute the average of the merge attributes and store it in an additional attribute. Now, using the Sort.java implementation already present in the minibase, we sort this on the average attribute in descending order. Then we give the top k tuples as the final output. This algorithm is naive and has a lot of random accesses hence it is not computationally very efficient. It is not efficient for very large databases. Buffer space required is equal to the number of database objects. Each entry is looked in the m sorted lists. The cost is linear in database size.

TASK 5B NRA Top K Joins

5A is the naive approach, which in NRA (No Random Access) Algorithm, we access sequentially the relation in parallel until we find K objects whose lower bound is higher than the threshold.
In our implementation, we are maintaining the table that stores lower_bound for relation 1 and relation 2 and the RIDs for relation 1 and relation 2. As soon as we encounter a tuple from the other relation we update the table and increment the counter. As soon as the counter is equal to K (provided by the user) we break and return the top K objects. Duplicate tuples are also handled in this implementation via a boolean variable.

Task 6

The query interface provides a way for the user to access the functionalities of the database. The interactions that are available in this interface are as follows:

- Open_database:
  The parameter passed along with this query is the name of the database. This query opens up the database if it is present or creates a new database with the name and opens it.
  To implement this, we create a new SystemDefs object. If a database file exists with the name passed, the SystemDefs object is created with 0 passed as the number of buffer pages as the database already exists. This ensures that the specific database is opened. Otherwise, if no such database exists, the SystemDefs object is created with a fixed number of pages, which creates a new database and opens it.

- Close_database:
  This query closes the database. This is done by calling the SystemDefs object's closeDB method. Before doing so, the flushAllPages method is called because this ensures that the changes made are written to the database and the data is persistent in it.

- Create_table:
  This query has 3 different scenarios.
  When the keyword "CLUSTERED" is not mentioned and the input file's name is sent as a parameter. The query creates a table in the database with the contents of the input file.
  Along with the input filename, if the keyword "CLUSTERED" is mentioned in the query, the index structure and attribute number are also passed as parameters. The index structure refers to whether the index file would have a B-Tree or a Hash index structure, and the attribute number is the attribute on which the index is created. Taking these as input, a table is created with contents of the input file

and a clustered index of the mentioned structure is created on the attribute specified by the number.
The input file is expected to be a Comma Separated File (CSV), where the first row contains the number of attributes of the file, subsequent n rows contain the mapping of the name of the attributes and their type, followed by the data itself.
To create a table, a heapfile is used. The name of the heapfile is taken to be the filename without the extension. Before adding the contents of the file into the table, we create a metadata file for this specific table, that contains 1 tuple with metadata of the table.
Firstly, the input file is opened and the number of attributes are read. Based on this number, the subsequent lines are read to map the attribute names and types. This data is put in a tuple in the following format and pushed into the metadata table, named **<tablename>_metadata.**

tablename	**attribute_names**	**attribute_types**	**Number of attributes**
customers	Name+Age+ID	STR+INT+INT	3

The attribute names and its types are concatenated to a string separated by “+” and stored. An example of the tuple is mentioned above.
Following this, tuples are created with the data to be inserted, from the input file and inserted into the table one at a time.
The table creation along with clustered index is implemented using either BTree or Hash index. These are explained in Task 1 and Task 2b respectively.

- Create_index
  This query creates an unclustered index based on the parameters of the passed.
  The parameters passed are index structure, attribute number and tablename.
  The index structure refers to whether the index is based on a BTree or Hash.
  The creation of an unclustered hash index is explained in Task 2a.
  For unclustered BTree index, the implementation already exists in the minibase. We are able to create this index using this implementation.
  After an index is created on a heapfile, this information is written as a tuple in an index metadata file, specific to each table.
  This data is put in a tuple in the following format and pushed into the index metadata file, named **<tablename>_index_metadata**

index_type	index_structure	Key_attribute	Split pointer	num_buckets
UC	HASH	2	2	5

An example of the tuple is mentioned above. The split pointer and num_buckets are initially set to -10 but are then updated during static or linear hashing. (during insertion in task 2)

- Insert_data
  This query inserts data from a file into a table. It takes in parameters the name of the file containing the data and the name of the table in which the data is inserted.
  Firstly, the file is read and the schema of the file is computed. The schema of the table as well, is looked up in its metadata table. These 2 are compared and if they are equal, we go ahead with the insertion. The remaining data lines of the file are read as tuples and are inserted into the table, one at a time.

- Delete_data
  This query looks up data from the file and deletes it if found in the table. Note that, duplicates of the tuples are also deleted if present in the table. As parameters, the query is provided the name of the table and the file.
  Firstly, the file is read and the schema of the file is computed. It is compared to the schema of the table that is looked up in the metadata heapfile. If they match, the deletion is started. All the tuples from the file are read and stored in an arraylist and is compared to all the tuples in the table one at a time that is brought using the table.getNext(). If 2 tuples match, the tuple from the table is deleted. Since modifications are done on the database, the buffer pages are flushed post deletion.

- Output_table
  This query prints the tuples present in the table whose name is passed as a parameter. The schema of the table is looked up from its metadata table. This helps in initializing the tuple that can be used to retrieve the tuple data from the table (heapfile). A scan is opened on the table and it reads all the tuples one at a time and prints its contents.

- Output_index
  This query prints the keys of the index table on the basis of the table name and the attribute provided. The index metadata table is queried on the basis of the attribute number and checked whether an index exists on it. If an unclustered

index exists, the keys are output. Similarly, if a clustered index exists, the key is output. Else N/A is printed.

- Skyline
  This query outputs the skyline tuples of a given table. Here, any of the various different methodologies of the skyline implementation, namely the Nested Loop Skyline, Block Nested Loop Skyline, Sort First Skyline and BTree Skyline can be called to compute these tuples. Along with this, the parameters contain the preferred attributes. All these algorithms were implemented as a part of phase II.

- GroupBy
  This query performs group/aggregation operations on the table specified. As a part of this query, the group attribute number and the aggregation attributes are passed. This is performed on the basis of either Sort or hash. The detailed implementation of these algorithms are explained in Task 3.

- Join
  This query executes a join on the given inner and outer tables. The names of the outer and inner tables, the join condition and the number of buffer pages that can be used are provided as the parameters. The various implementations like Nested loop join, Sort merge join, hash join and index nested loop join exist and any of these can be selected. The detailed implementation of these algorithms are explained in Task 4.

- TopKJoin
  This query runs the TopKJoin on the inner and outer relations specified. This is done in 2 ways, on the basis of Hash and using the NRA algorithm. The detailed implementation of these algorithms are explained in Task 5.

Interface Specifications

- Task 1

```
public BTreeFileClustered(String filename, int indexField, int
len_in1, AttrType[] in1, AttrType key_type,  short[] str_sizes)
```

  - fileName is the datafile name for the table

  - Index field is the field on which we are creating the index

- len_in1 is the length of the total input attributes
- in1 is an array of attribute types of input tuple
- Key_type is the attribute type of index field
- str_sizes is an array of max length of string sizes in each row.

- Task 2

2a) Show below is the constructor for Unclustered Hash Index

```
public UnclusteredHashIndex(
                    String fileName,
                    Tuple inserttuple ,
                    int key,
                    AttrType[] attr,
                    Float targetUtilization,
                    Iterator am1,
                    String tableName)
```

- fileName is the datafile name for the table
- insertTuple is the tuple which will be used for the insertRecord function (to insert a particular record in an existing table)
- Key is the attribute key on which the unclustered index is formed
- Attr is the list of the attribute type read from the file that is input from the user.
- targetUtilization is the value that is hard coded for the threshold of the index utilization.
- am1 is used to iterate over the input heap file.
- tableName is the table name (same as the filename)

  Following this there are methods such as Insert, InsertRecord, DeleteUnclustered, Hashfunction.

2b) Show below is the constructor for Clustered Hash Index

```
public ClusteredHashIndex(
            String fileName, int key, AttrType[] attrType,
```

```
Float targetUtilization, Iterator am1,
String tableName)
```

- fileName is the datafile name for the table
- Key is the attribute key on which the unclustered index is formed
- AttrType is the list of the attribute type read from the file that is input from the user.
- targetUtilization is the value that is hard coded for the threshold of the index utilization.
- am1 is used to iterate over the input heap file.
- tableName is the table name (same as the filename)

Following this there are methods such as Insert, DeleteClustered, ReHashfunction.

- Task 3

  This task can be tested using the query interface in task 6.

  Following are the subtasks:

  3a) Shown below is the constructor for GroupBywithSort iterator

```
public GroupBywithSort(
     AttrType[] in1, int len_in1, short[] t1_str_sizes,
     java.lang.String relationName,Iterator am1,
     FldSpec group_by_attr,
     AggType agg_type,
     FldSpec[] agg_list, FldSpec[] proj_list,
     int n_out_flds, int n_pages)
```

  - in1 is the list of AttrType for the tuples in the input heapfile/database table.
  - len_in1 is the length of in1 mentioned above
  - t1_str_sizes is the length of the strings in the tuples of the heapfile
  - relationName is the name of the input heap file
  - am1 is used to iterate over the input heap file.

- group_by_attr is of type FldSpec and contains the field number of the attribute to be grouped in the relation.
- agg_type is of type AggType and is used to specify whether the aggregate function is max, min, average or skyline.
- agg_list is the list of attributes on which the aggregation function is to be applied. If the aggregation function is min, max or average, this list will have only one attribute, otherwise it can have a list of attributes that act as the preference attributes for skyline computation.
- proj_list has the mapping for the output tuple
- n_out_flds is the number of output fields
- n_pages is the number of buffer pages available.

3b) Shown below is the constructor for GroupBywithHash iterator

```
public GroupBywithHash(
     AttrType[] in1, int len_in1, short[] t1_str_sizes,
     java.lang.String relationName, Iterator am1, FldSpec
     group_by_attr, AggType agg_type, FldSpec[] agg_list,
     FldSpec[] proj_list, int n_out_flds, int n_pages)
```

The parameters are the same as explained for task 3a.

- Task 4

This can be tested using the query interface implemented as a part of task 6. Following are the subtasks:

4a) Shown below is the constructor for IndexNestedLoopJoins

```
public IndexNestedLoopJoins(
     AttrType[] in1, int len_in1, short[] t1_str_sizes,
     AttrType[] in2, int len_in2, short[] t2_str_sizes,
     int n_pages, String relationName1,
     String relationName2, FldSpec[] proj_list,
     int n_out_flds, String inner_index_name, CondExpr[]
     condexpr)
```

- in1 is the list of AttrType for the tuples in the input heapfile/database table for the outer relation.

- len_in1 is the length of in1 mentioned above
- t1_str_sizes is the length of the strings in the tuples of the heapfile for outer relation
- in2 is the list of AttrType for the tuples in the input heapfile/database table for the inner relation.
- len_in2 is the length of in2 mentioned above
- t2_str_sizes is the length of the strings in the tuples of the heapfile for inner relation
- n_pages is the number of buffer pages available.
- relation1 is the name of the input heap file for outer relation
- relation2 is the name of the input heap file for inner relation
- proj_list specifies the structure of the output tuple
- n_out_flds is the number of output fields
- inner_index_name is the name of the index file on the index file

4b) Shown below is the constructor for Hash Join

```
public HashJoins(
   AttrType in1[],
   int len_in1,
   short t1_str_sizes[],
   AttrType in2[],
   int len_in2,
   short t2_str_sizes[],
   int n_pages,
   String relation1,
   String relation2,
   CondExpr outFilter[],
   CondExpr rightFilter[],
   FldSpec proj_list[],
   int n_out_fld)
```

- in1 is the list of AttrType for the tuples in the input heapfile/database table for the outer relation.
- len_in1 is the length of in1 mentioned above
- t1_str_sizes is the length of the strings in the tuples of the heapfile for outer relation
- in2 is the list of AttrType for the tuples in the input heapfile/database table for the inner relation.
- len_in2 is the length of in2 mentioned above
- t2_str_sizes is the length of the strings in the tuples of the heapfile for inner relation
- n_pages is the number of buffer pages available.
- relation1 is the name of the input heap file for outer relation
- relation2 is the name of the input heap file for inner relation
- outFilter specifies that the join has to be made on the equality condition. It has the join attribute of the inner relation and the join attribute of the outer relation. It also specifies that they are of type symbol.
- rightFilter can be passed as null
- proj_list has the mapping for the output tuple
- n_out_flds is the number of output fields

- Task 5

  5a and 5b) Shown below is the constructor for NRA Join as well has Hash based Top K join

```
public NRAJoin(
        AttrType    in1[],
        int     len_in1,
        short   t1_str_sizes[],
        AttrType    in2[],
        int     len_in2,
        short   t2_str_sizes[],
        int     n_pages,
```

```
String relation1,
String relation2,
FldSpec join_attr1,
FldSpec merge_attr1,
FldSpec join_attr2,
FldSpec merge_attr2,
int k)
```

- in1 is the list of AttrType for the tuples in the input inner heapfile/database table.
- len_in1 is the length of in1 mentioned above
- t1_str_sizes is the length of the strings in the tuples of the heapfile
- in2 is the list of AttrType for the tuples in the input outer heapfile/database table.
- len_in2 is the length of in2 mentioned above
- t2_str_sizes is the length of the strings in the tuples of the heapfile
- relationName1 is the name of the input inner heap file.
- relationName2 is the name of the input outer heap file.
- joinAttr1 is the join Attribute of the Relation 1.
- mergeAttr1 is the merge Attribute of the Relation 1
- joinAttr2 is the join Attribute of the Relation 2.
- mergeAttr2 is the merge Attribute of the Relation 2
- K is the value of the K in top K (top k number of tuples to be retrieved)

- Task 6

  This is a persistent database query interface which can be run using the “final_integrate.java” from the testsphase3 package. It has various queries such as open_database, create_table, create_index, JOIN, GROUPBY, etcetera that can be implemented.

Output

- Task 1 creates a clustered BTree index with the functionality to insert, delete and scan the tuples. It can be made at the time of table creation.
- Task 2 creates a clustered and unclustered hash index with the functionality to insert, delete and scan.
- Task 3 implements the group_by operator and gives either the max, min, avg or skyline tuples as output for every group of the group by attribute. It takes the group by attribute, aggregation attribute, aggregation function, table name as the input.
- Task 4 implements the index nested loop join and hash join. So it takes the inner and outer relation as the input along with the join attributes for both and the join condition. Hash join is implemented only on equi join condition. The output tuples will contain the attributes of both the relations and the join attributes will be related to each other as specified by the join condition.
- Task 5 returns the top k tuples after joining them in 5 a. It returns the top k tuples without random access in 5 b. The input is the inner relation, outer relation, join attributes and the merge attributes along with the value of k.
- Task 6 is the query interface to execute all the above tasks along with the tasks of phase 2.

Result Interpretation

Task 1

The following table summarizes the results obtained on the six data files provided, and inserting them into the clustered B tree.

File Name	Reads and Writes
r_sii2000_1_75_200	R:251

	W: 554
r_sii2000_10_10_10_dup_inv	R: 244 W: 2675
r_sii2000_10_10_10_dup	R: 254 W: 555
r_sii2000_10_10_10	R: 245 W: 2677
r_sii2000_10_75_200	R: 250 W: 553
r_sii2000_1000_75_200	R: 251 W: 554

The reads and writes are almost similar for all the files because first we are bulk loading and then inserting them in linear fashion. But we see a huge spike in writes when the files are in reverse order, because we need to sort them first.

Task 3

The following table summarizes the results obtained on the six data files provided, on all the four aggregation functions:

Assumptions:

- The first column is the group_by attribute.
- In case of min, max and average, the second column is the aggregation attribute.
- In the case of skyline, the aggregation attribute list contains columns 2 and 3.

The following table shows reads/writes for GroupBywithSort iterator

	Max/min/avg	Skyline
r_sii2000_1_75_200	R: 378 W: 299	R: 499 W: 421
r_sii2000_10_10_10_dup_inv	R: 396 W: 317	R: 1809 W: 1274
r_sii2000_10_10_10_dup	R: 395 W: 315	R: 1798 W: 1268
r_sii2000_10_10_10	R: 397 W: 315	R: 403 W: 322
r_sii2000_10_75_200	R: 396 W: 315	R: 396 W: 315

r_sii2000_1000_75_200	R: 445 W: 366	R: 1179640 W: 242260

Skyline computations have more reads and writes after we access the tuples because we need to call the block nested loop which requires more random accesses.

The following table shows reads/writes for GroupBywithHash iterator

	Max/min/avg (Creating unclustered hash index)	Skyline (Creating unclustered hash index)
r_sii2000_1_75_200	R: 340921 W: 8635	R: 341212 W: 8875
r_sii2000_10_10_10_dup_inv	R: 235650 W: 9157	R: 239357 W: 11065
r_sii2000_10_10_10_dup	R: 235623 W: 9181	R: 239265 W: 11057
r_sii2000_10_10_10	R: 234806 W: 9062	R: 235414 W: 9364
r_sii2000_10_75_200	R: 235638 W: 9246	R: 236372 W: 9571
r_sii2000_1000_75_200	R: 709451 W: 20517	R: 235335 W: 32322

System requirements/installation and execution instructions

The libraries required : The minibase packages, java.io, java.util.ArrayList, java.lang

The user should execute “final_integrate.java” in the “testsphase3” package to be able to access the query interface. The query interface facilitates the various queries like skyline computation, index creation etcetera and can enable the user to test all the queries.

Related work

The NRA algorithm for task 5 b was implemented using the concept in [1]. The concepts for the join operator, Top K Join operator, clustered BTree index, Hash index(clustered and unclustered) were discussed in class. The group by operator can be easily understood using the concept of sorting and hash index as discussed in class.

Conclusions

This phase gave us an in-depth understanding of the various functionalities of a Relational Database Management System and how to implement them. This includes Join operator, Group by operator, Top K Joins operator among others. In task 1, we implemented the clustered BTree index which was absent from the minibase. This includes scanning, insertion and deletion. In task 2, we implemented the linear hash index(both clustered and unclustered) using the concept of target utilization and split pointer. This task includes scanning, insertion and deletion as well. Task 3 consists of the implementation of the group by operator using both the sorting and hashing based techniques. The main requirement here was to obtain all the tuples belonging to the same group, together for aggregation function(max, min, avg and skyline) computation. Task 4 included the implementation of index nested loop join and hash join. In task 4a, the presence of an index on the inner relation made computation less expensive and in task 4b, the use of the same hash function on both the inner and outer relation reduced the computation time by dividing the tuples into hash buckets. In task 5 we implemented the Top K Join operator using both the naive and the NRA approach. In the naive approach we use hash join initially, then compute the average using the merge attributes and return the top k averages. But this has a lot of random accesses. We can reduce those using the NRA “No Random Access” algorithm because it computes the threshold and keeps reducing the gap between the lower and upper bounds. Finally, in Task 6, we have compiled all the functionalities into a query interface. It facilitates creation, deletion, insertion, join etcetera. This makes sure that the database is persistent.

Bibliography

1) *Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal aggregation algorithms for middleware. In Proceedings of the twentieth ACM SIGMOD-SIGACTSIGART symposium on Principles of database systems (PODS ’01). Association for Computing Machinery, New York, NY, USA, 102-113. DOI:https://doi.org/10.1145/375551.375567*

2) *A Survey of Top-k Query Processing Techniques in Relational Database Systems. Ilyas, Beskales and Soliman.*

Appendix

Roles and responsibilities

- Task 1- Dheeraj

- Task 2 a- Rutva
- Task 2 b- Rutva
- Task 3 a- Riya
- Task 3 b- Haard
- Task 4 a- Dheeraj
- Task 4 b- Riya
- Task 5 a- Haard
- Task 5 b- Rutva
- Task 6- Nithya
- Report- Riya, Rutva, Haard, Nithya, Dheeraj