CSE 510: Database Management System Implementation - Project Phase 3

User Preference Sensitive Minibase

Group Members	
Khushal Modi	kcmodi@asu.edu (1219446332)
Monil Nisar	mnisar2@asu.edu (1217111805)
Saurabh Gaggar	sgaggar1@asu.edu (1219666643)
Mishal Shah	mshah31@asu.edu (1217195421)
Kunj Patel	khpatel8@asu.edu (1213184152)
Samip Thakkar	sthakka2@asu.edu (1217104967)

Abstract

After exploring the building blocks of the minibase in Phase-1 and implementing various skyline computation operators in Phase-2. In this phase we add more features to make Minibase user preference sensitive. We have implemented Clustered B-Tree Index, unclustered and clustered variants of the linear Hash Index, a groupBy operator which can run MIN(), MAX(), AVG() and SKYLINE() aggregations over input data, hash-based and index nested loop based join operator as well as Top-K join operators using hash and NRA designs. Two types of user preference operators are implemented are skyline and top-k joins. Skylines finds all the non-dominated values across all possible scoring functions, top-k joins return the highest k values for a given score function. A linear hashing based hash index has been implemented which adds buckets one by one if required and doesn't need doubling, this provides predictable and similar insert and delete times.

Keywords

User Preference sensitive DBMS, Linear Hash. B-tree, Joins, Top-K Joins, NRA Join, Group By

Table of Contents

1. Introduction:

1.1. Terminology

- Tuple: A record in a database
- Heap File: Unsorted records stored in the page format, that helps in efficient retrieval.
- BTree: It is like a tree data structure, but it is self-balancing in nature. This property makes the indexing of the files efficient and retrieval fast.
- B+ Tree: It is like B-tree, but the storage can be done only on leaf nodes (unlike BTree, where internal leaves can help in storage).
- Skyline: "*Given a set of points, the skyline comprises the points that are not dominated by other points. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension*." (Efficient Progressive Skyline Computation, 2001)[3].
- Hash index: An **array** of N buckets or slots, each one containing a pointer to a row. Hash indexes use a hash **function** F(K, N) in which given a **key** K and the number of buckets N , the **function** maps the **key** to the corresponding bucket of the hash index.
- BTree index: A **b-tree index** stands for "balanced tree" and is a type of **index** that can be created in relational databases. It's the most common type of **index** that I've seen in Oracle databases, and it's the default **index** type.
- Join Operator: The ***JOIN*** operator specifies how to relate tables in the query. The JOIN operator is one of the set operations available in relational databases.
- HashIndexWindowedScanner: It returns an object of Iterator class which will provide an Iterator for scanning each window for the hash.

1.2. Goal Description

After adding new features to the Minibase DB to provide support for skyline computation in phase II, we are adding new features to the Minibase DB to provide support for the following:

- Implement clustered B-tree index structure
- Implement unclustered and clustered linear hash index structure
- Extend Minibase with a GroupBySort and GroupByHash operator to aggregate using MIN(), MAX(), AVG() and SKYLINE() function over input data.
- Implement hash-based and index nested loop joins.

- Implement Hash-based top-K join which equi-joins two relations on the given two join attributes.
- Implement TopK_NRA join iterator which computes the top-k highest valued tuples using the NRA algorithm.

1.3. Assumptions

- Group-by attributes can be numeric as well as string whereas aggregation attributes can be numeric only.
- The output attributes for join operation would not have duplicate attributes (the ones which we use to join the relations).
- If HashIndex is present on the join attribute, we will prefer it for indexNestedJoins for task 4a.
- We only have equality join in HashJoin for task 4b.
- The HashJoin will need at least 6 pages to run as it opens some index and heap files at concurrently.

2. Implementation Details

2.1 Clustered BTree Index

A clustered index is a type of index in which table records are physically reordered to match the index. In other words, a clustered index imposes some kind of order on the underlying data file. B+ trees clustered index is a type of B+ tree index structure where the leaf nodes store the information about the data page of the table. In a clustered index, only one pointer per data page is stored in the index leaf page whereas in an unclustered index, one pointer per record is stored in the leaf pages. Hence, clustered indexes are shorter than unclustered indexes since the fanout increases in the leaf pages. Also, since the clustered index imposes an order on the underlying data file, when a record is inserted in the table, it is inserted in a manner such that the order of the data file is maintained correctly. Because of this, we can only have one clustered index on the table whereas we can have multiple unclustered indexes on a table.

As a part of task 1, we are creating a clustered index on the database tables. We are using the existing unclustered btree code as our index file and we have created a new heapfile derivative, ClusteredHeapfile, which uses the underlying clustered index for insert, delete and scan operations. One one pointer per data file page is stored in the index file leaf page as per the clustered index definition. Below diagram shows how our clustered index is maintained.

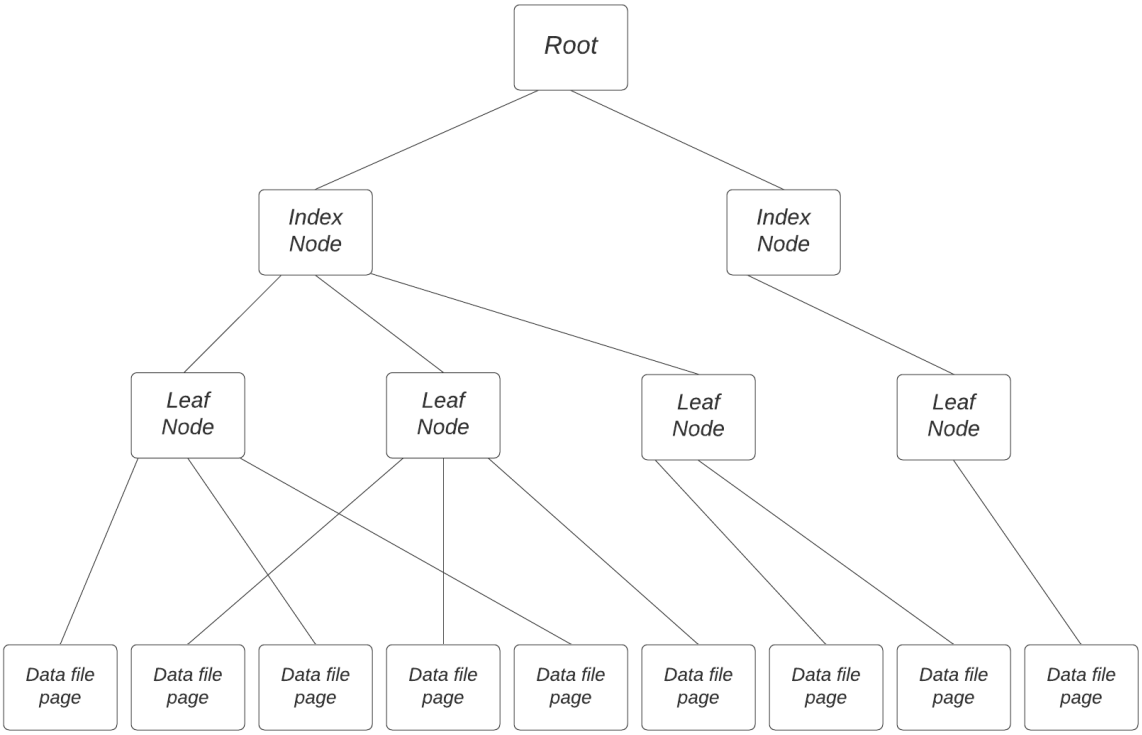
As shown in the diagram example below, our index nodes can handle 3 page pointers and it shows the structure of the data file and index file. It is not necessary that the data file is sorted or sequential but the scan on the leaf pages of the index will provide sorted access in our design. So for any kind of range access or equality search, we first traverse the tree to find the appropriate data page to start our search and then start traversing the data file to search for our required records. Unlike in a random ordered heap file, where we only use the directory to find an empty slot in the data page for insert, insert is a bit costly sometimes in clustered index since we need to traverse the entire tree to find the appropriate position for where we need to insert. Also, insert might lead to splitting of data pages which inturn needs an update in the index. Hence, clustered index is useful for when equality queries and range queries on the tables.

Clustered Heapfile data structure uses the structure of BTSortedPage as its datapages which keeps the page packed and sorted. So anytime an insert is done on the page, it performs an insertion is made on the page, it performs an insertion sort. Similarly, when a record is deleted from the page, it packs the page to keep the structure intact and ready for next insertion. This supports our aim of keeping the records in a page in a certain order and allowing easy insertion and deletion. Due to this, after every insertion, deletion or a merge operation, we update the unclustered indexes of the table because the BTSortedPage structure might change the record ids of all the records of the datapage where the insertion, deletion or merge happened. Hence, we keep a track of all the rids which were changed due to insert, delete or merge operations.

We have implemented the following operations on our Clustered Heapfile as a part of this project →

a. Insert

b. Delete
c. Merge
d. Ascending Ordered Scan
e. Descending Ordered Scan



2.1.1 Clustered BTree Index → Insert

Let's say we want to insert a Tuple *t* with Key *k* in the Clustered Heapfile *h* having a clustered index *Ind* on it. The following algorithm is used to insert this record to the data file →

Step 1 → Start a scan on *Ind* for all keys greater than or equal to *k*.

Step 2 → If the first element in the scan is null, that means that this is the highest key we have seen so far.
Step 2.1 → Check if the datapage of the last index leaf page entry has space to accommodate this record, if so, insert *t* on that data page and update the key in the *Ind* accordingly. Also, update the directory of *h* accordingly. Jump to *Step 4*.

Step 2.2 → Since the last data page does not have space to accommodate this record, we insert a new data page into the heap file and insert (*k*, *rid*) pair into the index. Also, update the directory of *h* for this new data page. Jump to *Step 4*.

Step 3 → If the first element of the scan is not null, it means that the scan returns with a key pointer greater than or equal to the key to be inserted, *k*.

Step 3.1 → If the datapage of the scan element has space to accommodate the record, insert the record *t* in that page and update the index accordingly. Also, update the directory of *h* accordingly. Jump to *Step 4*.

Step 3.2 → If the datapage of the scan does not have space to accommodate t, we create a new data page, we might have to insert *t* in between the data page and split the page and update the index with the new keys and page pointers. Also, update the directory of *h* accordingly. Jump to *Step 4*.

Step 4 → Update the unclustered indexes on the table with the newly added entry and also update the old records which were moved due to splitting. For this, we keep a list of records that were moved during the insert operation.

Step 5 → Perform the *Merge* operation after all the inserts are done to maintain the utilisation and avoid unnecessary page usage.

2.1.2 Clustered BTree Index → Delete

Let's say we want to insert a Tuple *t* having key *k* from the Clustered Heapfile *h* having a clustered index *Ind* on it. The following algorithm is used to delete this record to the data file →

Step 1 → Start a scan on *Ind* for all keys greater than or equal to *k*.

Step 2 → If the scan returns null, then key *k* is not present in the clustered heapfile *h.*

Step 3 → If the scan finds an element in the index leaf page which is greater than or equal to key *k*, start scanning the individual data pages of the entries in the index leaf pages until we reach a page where the lowest key is greater than the key *k* to be deleted.

Step 3.1 → Scan the data page and see if we find a tuple matching to our tuple *t.* If so, delete that record and update the directory of *h* accordingly. If the data page becomes empty after deletion, delete that page from the clustered heap file and update the directory of *h* accordingly. While doing this entire operation, keep a track of all the record ids getting changed. Repeat this step until we reach a suitable breaking condition.

Step 4 → Update the unclustered indexes on the table with the newly deleted entry and also update the old records which were moved due to the deletion. For this, we keep a list of records that were moved during the insert operation.

Step 5 → Perform the *Merge* operation after all the inserts are done to maintain the utilisation and avoid unnecessary page usage.

2.1.3 Clustered BTree Index → Merge

In merge operation, we go over all the data pages in the Clustered Heapfile and see if they are underutilised i.e. that 2 consecutive pages have less than 50% page utilization. In such cases, we merge the 2 datapages and update the index files accordingly. This is also a costly operation since the merge might lead to changes in rids of many records which inturn needs to be updated into the unclustered indexes and the Clustered index tree.

2.1.4 Clustered BTree Index → Scans

We have 2 types of scans on the Clustered BTree index, Ascending and Descending order scans. As the name suggests, one traverses the leaf pages from left to right and the later traverses the leaf pages from right to left. When traversing the index leaf pages, we scan the datapage for each entry in the leaf page. Scans are also costly operations because they scan the leaf pages and for each entry in the leaf page, they scan the underlying data page. Clustered index scans are used in Top-K NRA joins and Index Nested Loop Joins.

Normal Scan on a table with Clustered BTree index would directly scan the data file rather than using the index because using the index in file scan is useless since we don't need to traverse the index for any kind of order. Clustered BTree scans are only useful in cases of range scans in ascending or descending order.

Entire scan of the data file is done using the ClusteredHeapfile structure and ClusteredBTree is only used in cases of range scans and equality search.

2.2 Linear Hash Index

As part of task 2 we had to implement a Linear Hashing bases unclustered and clustered hash index for minibase. These would then be used for implementing group-by and hash-join operators.
Both the indexes are similar in the index implementation, their expanding and shrinking logic is same, the main difference is the unclustered index buckets contains <HashKey,RID> pairs and clustered index's bucket contains <HashKey,PageNo> pairs, another difference is the clustered index manages the insertion and deletion of tuples from the underlying data file.
The HashKey class represents a key, its object can be created by supplying a int,float or String value as key in the constructor. RID class stores the page and slot number information. A HashEntry object consists of a HashKey and corresponding RID. Many of these HashEntry are stored in a HashBucket depending on the hash value of the HashKey.
When an index is created it creates 2 HashBucket , numbered 0 and 1. The HashBucket are implemented as extensions of heapfiles, thus they handle overflow pages automatically as part of the heapfile implementation. The hashbucket is compacted when it is rehashed on insertions and deletions.
The metadata of the index like hashfunction H0 domain, split pointer location, number of buckets,number of entries in index, keytype of index, target utilization is stored in a separate page called HIndexHeaderPage.
The hashindex uses 2 hash functions and a split pointer to decide which buket a HashEntry is in. To hash a key, first an integer based hash of the key is calculated by using the java hashCode method, this has different implementations for integer,float and string. The integer value of the last log2(N) bits of this hash is extracted, this number gives the bucket number of the key. For 2 hash functions of domain N and 2N, we store only log2(N) in the index metadata,this is called h0 in our implementation. This value along with the split pointer location is sufficient for finding the appropriate location of any key.

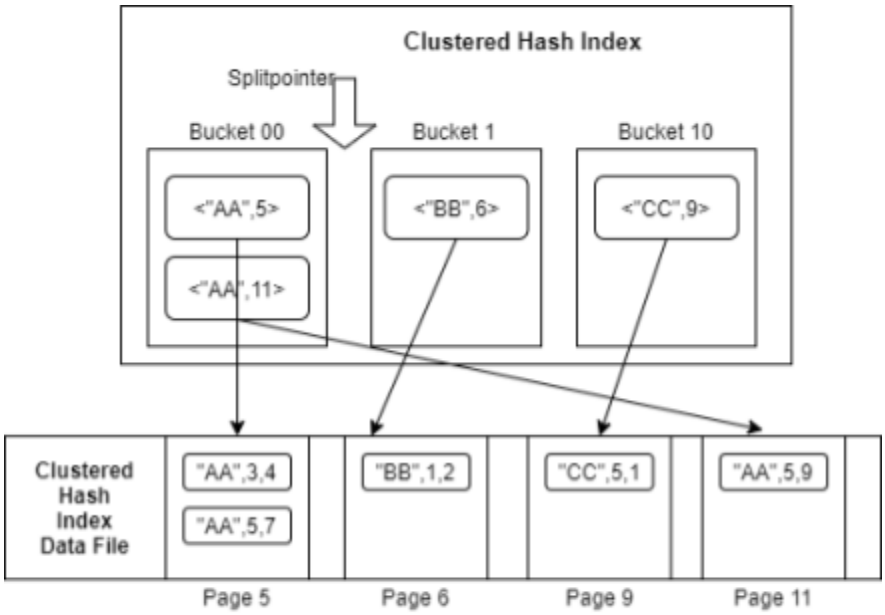
2.2.1 Unclustered Linear Hash Index

The unclustered hash index supports inserting and deletion given a HashKey and an RID value.
If an insertion is requested on the index, first the appropriate bucket number is calculated using H0 and splitpointer, then the entry is inserted in the bucket, and the current utilization of the index is calculated using the pagesize of minibase, size on entry, number of entries and buckets in the index. If this value is greater than the configured targetUtilization value then a new HashBucket is added to the index and the split pointer is moved ahead, and the hash function H0 value is updated if required.
On deletion from the hash index, the appropriate hash entry is found and deleted, and if the index utilization falls <10% of the target utilization the index is shrunk by rehashing and deleting the last HashBucket. There are always minimum 2 buckets, the index is not shrunk beyond this.

2.2.1 Clustered Linear Hash Index

Clustered hash index support insertion and deletion given a HashKey and a Tuple. It maintains an underlying data file and does insertion into it in the appropriate page depending on the HashKey.
ClusHIndexDataFile(*heap.ClusHIndexDataFile*) is the data file which stores the tuples for the clustered hash index, it stored tuples with same key on the same page.It is implemented as extends Heapfile of minibase and provides methods to insert a tuple on a given page number and insert a tuple on a new page. For deletion this file exposes a method to delete a tuple from a given page. This data file does not deviate

from the directory based implementation of the Heapfile. To support multiple Tuples with the same HashKey, the clustered hash index may contain multiple HashKey,PageNumber entries with the same HashKey but different PageNumber in the same HashBucket.
When the clustered hash index gets an insert call for HashKey and Tuple, the bucket number is calculated using hashing and split pointer as usual. Then in that bucket all the page pointers for that key is found, and insertion is done on the page with available space for the Tuple, if no existing page has space, then insertion is done to nea page on data file and HashKey,NewPageNumber entry is inserted into the HashBucket.
Then, if required the index is extended if index utilization goes over the target.
For deletions from the clustered index, it accepts a HashKey and the Tuple to delete, and deletes all tuples which match this from the datafile which have the same HashKey as supplied. This is done by first getting the bucket number by hashing the key, then finding all page numbers of the data file which contain tuples with the same Key, and finally calling delete method of the data file with the Tuple and the page number. We notice that there are many operations to search all the same keys in a HashBucket here,but this will not have much impact as the Bucket page is already in buffer and no additional disk access is done.
Finally, similarly to the unclustered index, the last hashbucket may be deleted and entries rehashed if the index utilization falls below 10% of the target utilization.



Clustered Linear Hash Index

To support implementation of other operators several types of scans had to be implemented on the HashIndex. There is support for equality scan,which given a key finds all key,pointer pairs.

We implemented a Windowed Scanner over the HashIndex which gives us Iterators for each bucket. The iterator works by fetching RIDs from the heap file corresponding to the bucket and using the RID to get a particular record from the main heap file of the relation. We implemented *HashIndexWindowedScan.java* and *HIndexedFileScan.java* to serve the scanning heap file over the HashIndex. This will help in

HashJoins where we need to iterate over each bucket at a time.

To accommodate heap file scanning using Hash Index, we modified the IndexScan class to use a hash index. So for equality queries over a relation, we would get an iterator to return tuples with a particular key. This will help in IndexNestedLoopJoin where we only need the tuples that satisfy our condition.

2.3 Groupby Operators

2.3.1 Group By Sort Operator

This is part of task 3 subtask (a) where we had to implement group by operation based on the sorted data. First the data is sorted on the group by attribute as specified in the user interface developed as part of task 6. Now that the related data is collocated into groups using sorting, we run aggregation functions on these groups to get results for individual groups and return the results to the user.

In our implementation we initialise the aggregation variables in the GroupBySort class and define a temporary heap file to hold individual groups for skyline aggregation. We then define the sort iterator on the original heap file containing the input tuples. The sort iterator traverses the heap file in descending fashion and sorted access on the group by attribute is achieved.

Next we have defined a get_next() method on the GroupBySort iterator which returns a list of aggregation tuples for a particular group. In the get_next() method, we keep track of the next tuple in the sort iteration as well as the value of the aggregation attribute in the last tuple seen in the iteration. We keep running the aggregations(MIN, MAX or AVG) on the iterated tuples until we find a different value for the aggregation attribute. In case of SKYLINE aggregation, we keep adding the tuples to a temp heap file initially defined which holds tuples for a particular group. Once a different value for aggregation attribute is encountered we break the loop and assign the next tuple in the sort iteration to the _next variable. In case of SKYLINE aggregation we run the skyline computation on the temp heap and eventually delete and recreate the heap file for the next group. This way we efficiently run our aggregations over the input data and keep the usage of disk pages to a minimum.

At the end once groupBySort iterator returns null, we close all the internal iterators and free the buffer pages which can then be utilized further for other operations.

2.3.2. Group By Hash Operator

This is part of task 3 subtask (b) where we had to implement groupBy operator on the buckets as defined by the unclustered hash index. First the input data is collocated in hash buckets as an unclustered hash index on the group-by attribute is created before GroupBy operation. Now here different groups of data can be present in a hash bucket as two different data points can resolve to the same hash function.

Now when we run the aggregation operation on the hash buckets, we first the bucket data to bring similar groups together and then we follow the same strategy as applied in GroupBySort operator. Hence it is possible to get multiple aggregation tuples on the same hash bucket.

At the end once groupByHash iterator returns null, we close all the internal iterators and free the buffer pages which can then be utilized further for other operations.

2.4 Index Nested Loop and Hash Joins

2.4.1. Join By Index Nested Loop

The task 4a is the implementation of IndexNestedLoop joins, which are used to overcome the drawbacks of NestedLoop and BlockNestedLoops Joins. We perform join using the index present on the inner relation. Here we have two of the possible implementations, one is joining the relations using the Btree index implementation provided by minibase. And another by implementing a hash function, which also works in the same manner as Btree, but uses hash index implementations and can work for equality joins only.

We prefer having a hash based indexing in case of equality joins, as hashing is much faster and requires less I/O operations to get the final result. But in general we check if the index is present on the attribute for inner relation. If no index is found, we calculate the join using NestedLoop approach, i.e. traversing full inner relation for every tuple in outer relation.

We use the IndexScan implementation of minibase and we enhance it by adding a hash based index scan, to get candidate tuples from inner relation. For each outer relation, we do an index scan and find tuples from the inner relation. We perform join only on these relations. We perform scan on full outer relation and only selected tuples of inner relation. In this implementation, the I/O operations depends on available buffer because we utilize at least 3 pages, 1 for inner relation and 2 for storing index, to complete this operation.

2.4.2. Join By Hash

The task 4b is the implementation of hashIndex. The idea behind HashJoin is that we group the data initially into K buckets for each outer and inner relation using the common hash function. And we perform joins only on the similar buckets, thus pruning the data. This approach guarantees us that the tuples corresponding bucket will only have the identical key. This increases overhead of creating and storing the data in a temporary heap and creating hash structures. One has to keep on cleaning the unnecessary datafiles to make sure we do not waste the storage.

The implementation of HashJoins has two steps. In the first step, we use a hashing method to create a hash index on inner and outer relation, we use HashIndexWindowedScanner.java to perform this. In the second step, we take the similar buckets across the two relations and perform NestedLoopJoin just on these relations, we get the original tuples using the HashIndexFileScan.java which provides us with an iterator.

2.5 Top K joins

2.5.1 Hash based Top-K Join

As part of this task, we were supposed to leverage the hash join algorithm (2.4.2) to calculate the join on the given tables and return k results. This particular implementation is the naive method to calculate top k joins. This algorithm can be divided in two steps.

The first step in this is to create a join of the two provided tables. To do this we used a hash join algorithm from our implementation. Details on this can be found in 2.4.2. For the outer relation we created a file scan which gives access to the tuple in iterative fashion. The inner relation was just passed by its name.

The second step involves iterating the results from the join and creating a new tuple. This new tuple would have an extra attribute which is the merge results from the join. We calculated the average of the two merge attributes that the user asked for. The average function is guaranteed to be monotonic and hence gave us consistency. A priority queue was used to efficiently get the top k results. The algorithm was implemented as an iterator and at every get_next() call we poll form the priority queue. Even though the approach of merging all of the tables is naive, there was some optimization since we used hash join.

2.5.2 NRA based Top-K Join

As part of this task, we were supposed to implement an algorithm that gives back users the top k results after joining the two tables. The NRA algorithm is an optimal algorithm which leverages the use of a clustered btree index which gives access to the merge attributes in descending order of their values. This hugely reduces the number of tuples we need to see in order to give the best k results.
There are two phases of this algorithm.
Before even beginning to implement this algorithm we made a check to confirm that both the tables have clustered btree index on the merge attribute. The first phase of this algorithm includes creating an iterator for accessing the tuples in descending order.
In the next step we start iterating both the relations simultaneously. For every partial object we keep a count of its lower bound and upper bound using an object we call NRABounds which helps us in deciding when to stop iterating the btree index. NRABounds object contains information about the lower, upper bounds and the corresponding tuples. At every depth we calculate the total of the scores at that level to get a sum value. If this sum value is less than the lowest of the lowest bound values and we have seen k objects then we can tell with certainty that there will not be anymore tuples in the top k results. This gives us a huge leap than the naive solution which calculates the join on the entire table and then merges the merge attributes to get the results and then return the top k results.

2.6 Query Interface and Database Persistency

As a part of task 6 we have implemented a query interface which lets the user interact with the database using the queries defined in task 6. Few examples of the queries are attached in file *interface.pdf*.

Also, we have implemented database persistency and a custom *Table.java* class in minibase to store the tables and retrieve them when the DB is reopened. A snapshot of the table is shown below.

```
>>open_database doo
Loading database doo
Replacer: Clock

Opening an existing DB...
>>output_table_info subset1
Printing the table subset1
Tablename: subset1
Table col names: [Name, Age, Address, Number, Relation, PinCode]
Table num attr: 6
Table attrypes: [attrString, attrInteger, attrInteger, attrInteger, attrInteger, attrInteger]
Table strsizes: [50, 50, 50, 50, 50, 50]
Table tuple size: 88
Unclustered btree index on attr: [false, false, false, false, false, false]
Unclustered hash index on attr: [true, true, false, false, false, false]
Clustered btree index attribute: -1
Clustered hash index attribute: -1
Number of Records in the table: 4
```

All the attributes of the table can be derived from the above shown table information. The metadata of each table in the database is stored in the database as a heapfile which is used again when the db is reopened to retrieve the information about tables in the database. *Table* class has various methods to retrieve information about indexes, data, heapfiles, index heap files, etc. for the table. This makes sure that we are not creating redundant indexes and using the existing information rather than doing redundant calculations for DB operations.

We also have some commands in addition to the commands in task 6 →

1. output_table_info tablename → outputs the table information as shown in the image above
2. destroy_database → destroys the currently open database
3. exit → exits from the UI and closes the database
4. help → command used to see all available database operations.

Additional information about the interface is given in the System Requirements section.

3. Interface Specifications

3.1 Unclustered Linear Hash Index

HIndex(String fileName, int keyType, int keySize,int targetUtilization)
Hindex.insert(HashKey keyIns, RID rid)returns void
Hindex.delete(HashKey keyDel,RID rid) returns boolean

fileName	File name of the index, if already exists open it else create new
keyType	Type of Key(int/float/string) based on global.AttrType class
keySize	Size of the key in bytes(max size for string keys)
targetUtilization	Target Index Utilization, a value between 10 and 100
keyIns/keyDel	The key of the entry to insert/delete from the hash index
rid	The rid of the entry to insert/delete

3.2 Clustered Linear Hash Index

ClusHIndex(String datafilename, String indexfileName, int keyType, int keySize,int targetUtilization)
ClusHIndex.insert(HashKey key,Tuple tup) returns insRID
ClusHIndex.delete(HashKey key, Tuple tup) returns listDelRID

datafilename	File Name of the datafile, which will store the actual tuples
indexfileName	File name of the index, if already exists open it else create new
keyType	Type of Key(int/float/string) based on global.AttrType class
keySize	Size of the key in bytes(max size for string keys)
targetUtilization	Target Index Utilization, a value between 10 and 100
key	The key of the entry to insert/delete from the hash index and datafile
tup	The tuple to be inserted/deleted from the data file
insRID	The location in the data file where the tuple was inserted
listDelRID	List of locations from the data file from where the tuple was deleted from

3.3 GroupBySort operator

```
GroupBywithSort( AttrType[] in1, int len_in1,
                                 short[] t1_str_sizes,
                                 Iterator am1,
                                 FldSpec group_by_attr,
                                 FldSpec[] agg_list,
                                 AggType agg_type,
                                 FldSpec[] proj_list,
                                 int n_out_flds, int n_pages )
```

in1	array showing what the attributes of the input fields are
len_in1	number of attributes in the input tuple
t1_str_sizes	shows the length of the string fields
am1	Iterator to iterator over the data. i.e. filescan used here for sorting
group_by_attr	Attribute on which groupBy operation will be performed
agg_list	set of attributes on which aggregation will be applied
agg_type	integer denoting max, min, avg, or skyline aggregator.
proj_list	List of projection attributes
n_pages	Max number of pages that are allowed

This subtask implements the groupBy operation on the sorted data. We input aggregation attributes, sort iterator, aggregation type, projection attributes and maximum number of buffer pages that can be allocated to the operation.

3.4 GroupByHash operator

```
GroupBywithHash( AttrType[] in1, int len_in1,
                                 short[] t1_str_sizes,
                                 HashIndexWindowScan am1,
                                 FldSpec group_by_attr,
                                 FldSpec[] agg_list,
                                 AggType agg_type,
                                 FldSpec[] proj_list,
                                 int n_out_flds, int n_pages )
```

in1	array showing what the attributes of the input fields are
len_in1	number of attributes in the input tuple
t1_str_sizes	shows the length of the string fields
am1	Hash Index scan to iterate over hash buckets
group_by_attr	Attribute on which groupBy operation will be performed
agg_list	set of attributes on which aggregation will be applied
agg_type	integer denoting max, min, avg, or skyline aggregator.
proj_list	List of projection attributes
n_pages	Max number of pages that are allowed

This subtask implements the groupBy operation on the hash buckets. We again iterate on the individual buckets and run aggregation on these bucket data. We input aggregation attributes, hash index iterator, aggregation type, projection attributes and maximum number of buffer pages that can be allocated to the operation.

3.5 Index Nested Loop Join

```
IndexNestedLoopJoin( AttrType[] in1,
                     int len_in1,
                     short[] t1_str_sizes,
                     AttrType[] in2,
                     int len_in2,
                     short[] t2_str_sizes,
                     int amt_of_mem,
                     Iterator am1,
                     String relationName
                     CondExpr outFilter[],
                     CondExpr rightFilter[],
                     FldSpec proj_list[],
                     int n_out_flds)
```

Variables	Definition
in1	Array showing what the attribute types of outer relation
len_in1	Number of attributes for outer relation
t1_str_sizes	Shows the length of the string fields for outer relation
in2	Array showing what the attribute types of inner relation

len_in2	Number of attributes for inner relation
t2_str_sizes	Shows the length of the string fields for inner relation
am1	Iterator for outer relation.
relationName	Name of inner relation name.
outFilter	A CondExpr that will provide join and other conditions for outer relation.
rightFilter	A CondExpr that will provide conditions for inner relation.
proj_list	A list of FldSpec for result tuples, (projection tuple).
n_out_flds	Number of projection attributes

This subtask implements the Join operation using Indices (Btree or Hash). We iterate on the outer relation and run search for the join attribute value in the inner index. We then use the outFilter and rightFilter to filter out conditions than the join. Also we revert to NestedLoopJoin in case we do not have any index on join attributes for inner relation.

3.6 Hash Join

```
HashJoin( AttrType[] in1,
          int len_in1,
          short[] t1_str_sizes,
          AttrType[] in2,
          int len_in2,
          short[] t2_str_sizes,
          int amt_of_mem,
          Iterator am1,
          String relationName
          CondExpr outFilter[],
          CondExpr rightFilter[],
          FldSpec proj_list[],
          int n_out_flds)
```

Variables	Definition
in1	Array showing what the attribute types of outer relation
len_in1	Number of attributes for outer relation
t1_str_sizes	Shows the length of the string fields for outer relation
in2	Array showing what the attribute types of inner relation
len_in2	Number of attributes for inner relation

t2_str_sizes	Shows the length of the string fields for inner relation
am1	Iterator for outer relation.
relationName	Name of inner relation name.
outFilter	A CondExpr that will provide join and other conditions for outer relation.
rightFilter	A CondExpr that will provide conditions for inner relation.
proj_list	A list of FldSpec for result tuples, (projection tuple).
n_out_flds	Number of projection attributes

This subtask implements the Join operation using Hashing. We iterate over the outer and inner relation to create hash buckets for each relation using the same hash function. Then we invoke NestedLoopJoin for tuples that are in the similar buckets for both relations. The will support only equality join on join attribute.

3.7 Hash based Top-K Join

```
TopK_HashJoin(  AttrType[] in1,
                int len_in1,
                short[] t1_str_sizes,
                FldSpec joinAttr1,
                FldSpec mergeAttr1,
                AttrType[] in2,
                int len_in2,
                short[] t2_str_sizes,
                FldSpec joinAttr2,
                FldSpec mergeAttr2,
                java.lang.String relationName1,
                java.lang.String relationName2,
                int k,
                int n_pages)
```

Variables	Definition
in1	Array showing what the attribute types of outer relation
len_in1	Number of attributes for outer relation
t1_str_sizes	Shows the length of the string fields for outer relation
in2	Array showing what the attribute types of inner relation

joinAttr1	Index of attribute of outer relation for joining
mergeAttr1	Index of attribute of outer relation for merging
in2	Array showing what the attribute types of inner relation.
len_in2	Number of attributes for inner relation.
t2_str_sizes	Shows the length of the string fields for inner relation.
joinAttr2	Index of attribute of inner relation for joining.
mergeAttr2	Index of attribute of inner relation for merging
k	Number of top results to return
n_pages	Max number of pages that are allowed

This subtask computes the top k joins leveraging the hash join algorithm. We call the hash join in this method and then computer the merging of attributes using average function. Then we return the k results

3.8 NRA based Top-K Join

```
TopK_NRAJoin(  AttrType[] in1,
                int len_in1,
                short[] t1_str_sizes,
                FldSpec joinAttr1,
                FldSpec mergeAttr1,
                AttrType[] in2,
                int len_in2,
                short[] t2_str_sizes,
                FldSpec joinAttr2,
                FldSpec mergeAttr2,
                java.lang.String relationName1,
                java.lang.String relationName2,
                int k,
                int n_pages)
```

Variables	Definition
in1	Array showing what the attribute types of outer relation
len_in1	Number of attributes for outer relation
t1_str_sizes	Shows the length of the string fields for outer relation

in2	Array showing what the attribute types of inner relation
joinAttr1	Index of attribute of outer relation for joining
mergeAttr1	Index of attribute of outer relation for merging
in2	Array showing what the attribute types of inner relation.
len_in2	Number of attributes for inner relation.
t2_str_sizes	Shows the length of the string fields for inner relation.
joinAttr2	Index of attribute of inner relation for joining.
mergeAttr2	Index of attribute of inner relation for merging
k	Number of top results to return
n_pages	Max number of pages that are allowed

This subtask computes the top k joins leveraging the clustered index functionality. We iterate over the relations simultaneously and then keep bounds (lower and upper) of objects to decide if we can stop iterating early. This generated an optimal join.

3.9 Clustered BTREE Index

```
ClusteredBTreeFile(  String filename,
                     int key_type,
                     int key_size,
                     int delete_fashion)
```

filename	Filename of the btree
key_type	Type of keys that will be stored in the btree
key_size	Size of the key to be stored in the btree
delete_fashion	0: Naive Delete, 1: Full Delete

Once a ClusteredBTreeFile is created, one can use the ClusteredHeapfile to insert/delete the tuples in the ClusteredBtreeFile.

4. System Requirements/ Execution instructions

To be noted →

1. Please change the path of the data folder in src/global/Table.java folder.

2. There are certain commands executed in the code which are OS specific and users might see errors when those commands are executed. Please ignore those errors.

3. Once the user has given the correct dataset path, he can compile and run the AppDriverPhase3.java file to use the UI menu created as a part of Task 6.

4. Run `make db` command inside `src/` folder of Phase_3. This will compile the necessary files and create class files. You should see the following output:

```
mish@Mishal src % make db
make -C global
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
Note: PageId.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
make -C chainexception
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
make -C btree
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
make -C bufmgr
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
make -C diskmgr
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
Note: DB.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
make -C heap
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
make -C index
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
Note: IndexUtils.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
make -C iterator
/usr/bin/javac -classpath /usr/lib/classes.zip:.:..:.:.. *.java
make -C driver
```

```
/usr/bin/javac -classpath .:.. AppDriver.java
make -C hashindex
```

5. Next step will be to run the main driver using the following command inside the src/driver folder.

```
mish@Mishal driver % make driver3
/usr/bin/javac -classpath .:.. AppDriverPhase3.java
/usr/bin/java  -classpath .:.. driver.AppDriverPhase3
```

Command: open_database dbname--> opens a database with the given name. If the db does not exist, creates the db from scratch

Command: close_database--> Closes the current database and make all the files in the db persistent

Command: create_table [CLUSTERED BTREE/HASH ATT_NO] FILENAME--> Create a new table and populate it with data from the given file, with the following format

1. First row of the file contains the number, n, of attributes
2. The next rows contains the attribute names and types (INT/STR)
3. The rest of the rows contains the data tuples

If CLUSTERED BTREE or CLUSTERED HASH is specified, then create a clustered index on the file on the attribute specified by ATT_NO (first attribute at index 1 and second at indexx 2 and so on.

Command: create_index BTREE/HASH ATT_NO TABLENAME--> Create and unclustered index on the file on the attribute specified by ATT_NO. If an index already exists, then the operation returns without any index creation. once the index is created, it needs to be maintained with the insertions and deletions

Command: insert_data TABLENAME FILENAME--> Insert data to the given table from the given file

Command: delete_data TABLENAME FILENAME--> Delete data from the given table those data that appear in the give file.

Command: output_table TABLENAME--> Output all the tuples in the given table.

Command: output_index TABLENAME ATT_NO--> Output the keys in the (clustered or

unclustered) index of the table for the given attribute. If there is no index at the given attribute, then the operation outputs N/A.

Command: skyline NLS/BNLS/SFS/BTS/BTSS {ATT_NO1, ...ATT_NOh} TABLENAME NPAGES [MATER OUTTABLENAME]--> Output the skyline of the given table for the given h attributes. If MATER is specified, then materialize the results by creating a new table with specified outputtable name.

Command: GROUPBY SORT/HASH MAX/MIN/AGG/SKY G_ATT_NO{ATT_NO1,...ATT_NOh} TABLENAME NPAGES [MATER OUTTABLENAME]--> Output the result of the groupby/aggregation operation. G_ATT_NO specifies the groupby attributes. ATT_NO1 through ATT_NOh specifies the aggregation attributes. If MATER is specified, then materialize the results by creating a new table with the specified output table name

Command: JOIN NLJ/SMJ/INLJ/HJ OTABLENAME O_ATT_NO ITABLENAME I_ATT_NO OP NPAGES [MATER OUTTABLENAME]--> Output the result of specified join operation on the given out and inner relations. The join condition is specified by two given attributes and operator OP which belongs to {=, <=, <, >, >=}. If MATER is specified, then materialize the results by creating a new table with the specified output table name.

Command: TOPKJOIN HASH/NRA K OTABLENAME O_J_ATT_NO O_M_ATT_NO ITABLENAME I_J_ATT_NO I_M_ATT_NO NPAGES [MATER OUTTABLENAME]--> Output the result of the specified top-K-join operation on the given out and inner relations. The join condtion is specified by the two given attributes and the selection of the top-K results are performed based on the specified merge attributes. If MATER is specified, then materialize the results by creating a new table with the specified output table name.

5. Related Work

We have implemented skyline and top-k queries as separate operators. In the paper “Top-k Skyline: A Unified Approach”[7] the authors propose an operator which combines these two to provide multicriteria top-k queries. They have provided a naive implementation by applying skylines on output of SQLf queries.

Linear Hashing was invented in the 80s, since then substantial work has been done to address some of its issues, mainly the existence of overflow pages. The authors in [4] have proposed a solution to solve this issue of overflow pages by recursively hashing the buckets with overflow pages into another linear hash index. This new hash index is separate from the first one, furthermore, if there are still overflow pages use another linear hash index, thus giving rise to “levels” of indexes.

In our project we have implemented top-k join operators based on hash and NRA techniques. But there is an issue that if we join multiple tables using existing algorithms like NLJ and sort merge join then for getting the ranked top k tuples we might need to sort again at the last step. The authors in [5] implement a join algorithm for finding top-k values while considering the individual orders of the input while joining. Thus the final top-k ranked join output doesn't need to be sorted unnecessarily.

Our project involves building b-tree and hash based indexes. Certain DBMS products like Oracle allow creation of function based indexes[6]. This is advantageous in scenarios where the query for a table is on the result of a function. In this case if a normal b-tree or hash based index is created it may not be used in the query plan. Thus having an index build specially for the function can optimize these types of queries. This also gives the DBMS user the control of index rather than the DBMS deciding and creating a default type of index.

6. Conclusion

We have added features to minibase to make it user preference sensitive by using skyline and top-k join operators. Additionally, operators like group-by and join operators have been added, thus giving various query plan alternatives for a query. Also, we have built a command line based query interface to interact with minibase and run these operators and queries. DB persistence is implemented using the *Table.java* class which helps us to keep all the metadata about the tables stored in the DB. It also helps us to fetch and use the table info anywhere in the database.

We observed the advantages and disadvantages of having a clustered index on the table. When we have a clustered index on the table, it is easy to get a faster sorted access but it is equally costly to update and maintain the clustered index. The updates in clustered index also leads to the updates in unclusterd index of the table and hence sometimes, it proves to be a very costly operation.

Index nested loop joins(INLJ) and hash joins operators have been implemented in addition to the already present nested loop join and sort merge join. We observe that the I/O usage of INLJ is less than Nested loop join when a clustered btree index is used. But when an unclustered btree index is used then page I/O's is more as there are too many index accessed and for each index access we have to get the tuple from the data file. Thus we can see that index nested loop join using clustered index performs better than unclustered index. This is the expected result.

Similarly while calculating top-k joins output, we see that the disk I/Os is much more for hash based implementation over NRA based implementation, this is because NRA is able to find top-k values in a single pass of the data using sorted access using a clustered index.

We also observed that with fewer available buffer frames for the operators, the disk I/Os increase as there are multiple page writes and read from disk.

With groupby, out observation and implementation led to more disk I/Os in groupby Hash as compared to groupby Sort. This is mainly due because groupby Sort gets a sorted access to the data which is easily grouped than on the Hash groupby which needs an index for grouping.

Bibliography

1. S. Borzsony, D. Kossmann and K. Stocker, ”The Skyline operator,” Proceedings 17th International Conference on Data Engineering, Heidelberg, Germany, 2001, pp. 421-430, doi: 10.1109/ICDE.2001.914855.
2. Chomicki J., Godfrey P., Gryz J., Liang D. (2005) Skyline with Presorting: Theory and Optimizations. In: Kopotek M.A., Wierzcho S.T., Trojanowski K. (eds) Intelligent Information Processing and Web Mining. Advances in Soft Computing, vol 31. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-32392-9_72
3. K.L. Tan, P.K. Eng and B. C. Ooi. “Efficient Progressive Skyline Computation.” 27th Int. Conference on Very Large Data Bases (VLDB), Roma, Italy, 301-310, September 2001.
4. K Ramamohanarao and R Sacks-Davis. 1984. Recursive linear hashing. ACM Trans. Database Syst. 9, 3 (Sept. 1984), 369–391. DOI:https://doi.org/10.1145/1270.1285
5. Ilyas, Ihab & Aref, Walid & Elmagarmid, Ahmed. (2004). Supporting top-k join queries in relational databases. The VLDB Journal. 13. 10.1007/s00778-004-0128-2.
6. Function Based Index, Oracle 11.2 documentation, https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/adfns_indexes.htm#ADFNS00505
7. Goncalves M., Vidal ME. (2005) Top-k Skyline: A Unified Approach. In: Meersman R., Tari Z., Herrero P. (eds) On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops. OTM 2005. Lecture Notes in Computer Science, vol 3762. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11575863_99

Appendix

Roles of group members:

Name	Role
Kunj Patel	Implemented Top K join algorithms (NRA and HASH based)
Khushal Modi	Implemented clustered btree, query interface and database persistency
Mishal Shah	Implemented Groupby Sort and Groupby Hash
Monil Nisar	Implemented Joins & Hash Index Windowed Scanner.
Samip Thakkar	Implemented Group by Hash and Hash index windowed scanner
Saurabh Gaggar	Implement unclustered and clustered linear hash index

Output on Test Data

Please find the sample query runs in file *output_sample.pdf*. It demonstrates outputs of some of the queries in the interface..
Also, please use the text files in submission to analyse the output of our queries on various datasets provided for testing.

1. *All_Results/GroupByHash_Results* Contains results for groupby hash operations ran on the given datasets.
2. *All_Results/GroupBySort_Results* Contains results for the groupby sort operations ran on the given datasets
3. *All_Results/btree_output_allfiles* Contains results for the BTrees indexes on given datasets.
4. *All_Results/hashindex_output_allfiles* Contains results for the Hash indexes on the given datasets.
5. *All_Results/topKjoin_results* Contains the results for topKJoin operations performed on the given datasets.
6. *All_Results/Skyline_results* Contains the results for the skyline operations on the given datasets.

Apart from all the above datafiles, we have attached small sample files as well for better analysis of the results. Also, output_sample.pdf contains some samples of the queries ran on the datasets.