CSE 510: Database Management System Implementation - Project Phase 2

Minibase Skyline Operator

Group Members

1. Kunj Patel (khpatel8@asu.edu , 1213184152)
2. Khushal Modi (kcmodi@asu.edu, 1219446332)
3. Mishal Shah (mshah31@asu.edu, 1217195421)
4. Monil Nisar (mnisar2@asu.edu, 1217111805)
5. Samip Thakkar (sthakka2@asu.edu, 1217104967)
6. Saurabh Gaggar (sgaggar1@asu.edu, 1219666643)

Abstract

After exploring the building blocks of the minibase in Phase – I, we move onto the next phase which is adding more functionalities to the Minibase DB. As a part of Phase - II, we will be adding the feature to retrieve skyline on n-dimensional dataset. Skyline is defined as a subset of the dataset that has the most optimal combination of some attributes of the data. Skyline is useful in domains where certain properties of the data are preferred over the others. We will implement multiple algorithms to find skylines for various data organizations like unsorted, sorted and indexed data over different attributes. Implementing skyline queries will move us a step forward towards our goal of implementing a user preference sensitive Database Management System to support e-commerce applications.

Keywords

Skyline, Extending DBMS, BlockNested Skyline, BTree, Combined Btree, Indexing

1. Introduction:

1.1. Terminology

- 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].

1.2. Goal Description

After executing various tests and understanding the building blocks of Minibase in phase 1, we are adding new features to the Minibase DB to provide support for skyline computation. The primary goals are:

- Implement tuple comparison methods to determine if a tuple dominates the other tuple.
- Implement tuple comparison method to determine which tuple is greater than other based on sum of preference attributes.
- Modify tuple sorting based on preference attributes
- Compute skyline on B Trees using BTreeSky iterator based on preferences of individual BTree attributes as well as combined.
- Implementing a program to store and index the data. The constraint will be the number of pages.
- Modify the Minibase disk manager to count the number of reads and writes.

1.3. Assumptions

- The structure of the input data file will be the same as the test sample provided on piazza. This means that the first line of the data contains the number of attributes in the dataset.
- The data provided for testing will be in permitted length (if there are limit constraints while implementation, the length of value should not exceed the limit).
- Java float variables will be used for storing floating point numbers.

2. Implementation Details

2.1.1. Tuple comparison method: Dominates

This is part 1 of task1 which requires to create a new comparison method between two tuples named Dominates. The input of the function has two tuples, along with the types, the length of input and input strings. The comparison is done on the basis of preference attributes. The domination is decided based on the sum of preferred attributes, rather than all attributes.

The sum of the tuple values of preferred attributes is calculated for both tuples (say t1 and t2). If the sum for preferred attributes of t1 is greater than that of t2, it returns 1, else it returns 0. This method is specifically used for calculating if one particular tuple dominates another or not.

2.1.2. Tuple comparison method: CompareTupleWithTuplePref

This is part of task 2. Unlike Dominates comparison method, which focuses more on whether the first tuple dominates another or not, this comparison method focuses on determining which of the two tuples dominate another over the other. The basic logic for calculating dominance is the same as Dominates. We calculate the sum of the tuple values based on the preference list. For tuples t1 and t2, we return

-1, if t1 is smaller
0, if t1 and t2 are equal
1, if t1 is greater

2.2. Modify Sort using CompareTupleWithTuplePref method

This task requires me to modify the sorting of the tuple. Rather than sorting based on one specific field, this task involves using the Compare Tuple with preference method defined in the previous task to perform sorting. For given tuples in a data file, along with the preference list, we first get the comparison using tuple preference method. The sum of the attributes of tuple is calculated based on the preference list, and the sorting is done on the basis of the sum calculated.

2.3. Implement NestedLoopsSky, BlockNestedLoopsSky, and SortFirstSky iterators

2.3.1 Nested Loop Skyline

NestedLoopsSky is the brute force ignorance approach to derive the skyline for a given dataset and a list of preferred attributes. We run a nested loop on a heap file containing the entire dataset to compare each data element with every other element to verify if it belongs to the skyline. This

is a naive approach to calculate the skyline. As mentioned in Section 3.3 of "The Skyline operator"[1], this is a very inefficient approach.

Our implementation of NestedLoopsSky accepts the relation name as a parameter, which is then used to open a heapfile and scan the stored data. For each element scanned, we open another scan on the same heapfile and compare each Tuple in the inner scan with the outer scan's data element using TupleUtils.dominates function. Since we have two Heap files and two Scans open in the skyline computation, the buffer manager fails with n_pages less than four.

In this phase, we have implemented the Nested Loops Skyline with an additional feature to control the amount of buffer memory the skyline computation can use. Hence, in cases of large datasets and small buffer memory allowance, there are a lot of buffer page replacements and disk reads which inturn adds onto the run time of the iterator. Also, the disk read might have a lot of random access cost since the algorithm can read data from any part of the heap data file. On a positive note, the skyline computation does not require any kind of write operation to the disk since it does not require any kind of preprocessing before the iteration begins and only calculates one skyline element at a time.

2.3.2 Block Nested Loop Skyline

BlockNestedLoopsSky is an improvisation over NestedLoopsSky which has the worst case complexity of $O(n^2)$ but a better average runtime. We have implemented a modified version of the Block Nested Loop Skyline algorithm given in Section 3.3.1 of "The Skyline Operator"[1].

In Block Nested Loop Skyline, we have divided the memory allowance between the main memory window and the buffer manager equally i.e. in case of n_pages being 10, 5 pages will be allotted to the buffer manager and 5 pages will be allotted to the window. Also, we keep a temporary heap file to store the skyline candidates just in case the window runs out of memory. Now, similar to the Nested Loop Skyline algorithm, we start by scanning the records in the data heapfile. But now, instead of comparing the record (data element, d) with all the other records, we compare it with all the records in the window and the temporary heap file. There are 3 possible cases in this scenario :

1. *d* is incomparable to all the records in the window and the heap file. In this case, we add *d* to the window. If the window is full, we add *d* to the temporary heap file.
2. Some element in the window or the temporary heap file dominates *d*. In this case, *d* is not a part of the skyline.
3. *d* dominates one or more data elements in the window or the temporary heap file. In this case, *d* is added to the skyline candidates, and the tuples in the window or the temporary heap file which are dominated by *d* are removed.

Our implementation of Block Nested Loop requires at least five buffer manager pages which means it requires at least ten pages of allowance. Anything less than 10 fails due to insufficient buffer memory. We have two heap files, two scans and a file write operation going on inside the BlockNestedSky computation and hence it requires a certain number of buffer manager pages at any given time.

Block Nested Loop Skyline has a much better average runtime than Nested Loop Skyline. In terms of number of disk reads/writes, it performs much better and because it needs only one pass over the data, it requires a much lesser amount of random disk accesses. After the first iteration, Block Nested Loop will work on the shorter temporary heap file and the main memory window. Compared to the Nested Loop algorithm, Block Nested Loop runs few disk writes to store the temporary heap file data but overall it is much more efficient and faster than the naive Nested Loop algorithm. The difference between the number of disk reads between both the algorithms can be seen in the output files.

2.3.3 Sort First Skyline

SortFirst skyline is an efficient implementation over NestedSkyline. This approach uses sorting to calculate skyline in a manner that limits comparison of dominance of tuples by great numbers. It reduces the comparison by leveraging the fact that the tuple that occurs late in sorted order cannot dominate the one that occurred earlier. Thus we would only compare the tuples that occur after.

This algorithm also reduces the disk I/O by storing the skyline in memory. The outer loop will traverse the sorted data on disk and the inner loop will scan the buffer in memory. The problem arises when the buffer becomes full. Then we have to store the tuples on a temporary heap on disk and then again use SortFirst or a different algorithm to calculate skyline from the heap file.

We are allocating 50% of the n_pages to the sort operation and the rest to the in-memory buffer. Hence for lower thresholds n_pages = 5 and 10, buffer manager fails to allocate enough memory to SortFirstSky for larger datasets. However for larger thresholds n_pages > 20 - we are able to pre sort tuples and subsequently run the SortFirstSky algorithm to get the skyline objects.

We have used Block Nested to calculate the skyline on the heap file. Since SortFirstSky uses the Minibase sort operator, and also requires BlockNestedLoopsSky, the algorithm fails for low memory allocation (n_pages). This is reflected in the output file generated for SortFirstSky.

2.4. Implement BTreeSky iterator

This task required us to implement an iterator to calculate skyline points using BTree Indexes created on individual attributes.

Algorithm from Section 4.1 of “The Skyline operator”[1] was implemented to calculate the skyline. The algorithm finds out the first skyline point by scanning over the individual btree

indexes and stops when the first common tuple is found in all the indexes. It then prunes the data by copying all scanned tuples to a temporary heapfile and then runs BlockNestedLoopSky on this pruned data to find the subsequent skyline points. The class BTreeSky was implemented in the btree package, additionally DiskBackedArray was also created to support the calculation of skylines.

DiskBackedArray is a simple data structure which uses a heapfile as an array or RID elements and provides functions to insert, and search the heafile using a scan. Since minibase uses fixed length tuples, the tuples will be saved in the insertion order in the heapfile. We have used this property to use a heapfile as an array of RID elements. This allows our algorithm to work in less memory situations as all the pages of the heapfile need not be in the buffer frames. During insertion and scanning the BufferManager automatically manages the page movement in and out of memory.

The constructor of BTreeSky receives an array of IndexFile, this array has references to BTree indexes of the preference attributes. These are casted to an array of BTreeFile variables from which a scan can be created. The algorithm starts by creating a BTFileScan and a DiskBackedArray for each IndexFile received. Then for each BTFileScan we get the next key and the RID of its corresponding tuple. We then check the DiskBackedArray for the other indexes if this RID exists in them. If this RID is found in all the DiskBackedArray's then the loop breaks and this RID is a skyline element. If this RID is not found in any of the other indexes previously scanned RID, then it is added to current index's DiskBackedArray. We then proceed to the next BTFileScan and repeat this process until the first skyline element is found. All the data points which have been scanned before need to be considered further for skyline calculation and the data points which were not scanned in any of the indexes can be ignored and pruned as they will be dominated by some of the already scanned points.

We now create a new heapfile to store the pruned data elements, iterate linearly and scan from the beginning for each DiskBackedArray and keep inserting the corresponding Tuples in the pruned data file until the first skyline element is scanned.
Since the BTreeFile indexes contain only the pointer(RID) to the actual Tuple. We have to open the main data heapfile to get the corresponding Tuple for a RID.
Then we call BlockNestedLoopSkyline and pass the name of the pruned data heapfile, to find out the remaining skyline points.

Thus, the overall computation of the skyline points is quicker than running BlockNestedLoop on the entire data as we have pruned the data. This is an advantage of using indexes to calculate skylines. Although this method does require that an index is present on the attributes we wish to consider for skylines.

2.5. Implement BTreeSortedSky

This task involved finding a skyline on a dataset which has a combined index created on all the selected preference attributes. This combined BTree index provides an indirect sorted access to the data even though the data heapfile is unsorted. A combined BTree Index was created on the sum function (monotonic). The obtained sum was then multiplied with -1 (negative 1) before adding it to the BTree to have the BTree keys sorted in descending order as we are calculating MAX skylines.

Since the data was available to us in a sorted fashion via BTree scan, we were able to implement a more efficient version of BlockNestedSky loop algorithm. An in-memory window similar to BlockNestedLoop was used. First step in this implementation is to start iterating the BTree and add elements to the window array (main memory) and make sure that these tuples are not dominated by the tuples already in the window. If a tuple is not dominated by any of the window tuples then we add it to the window array. In the case when the main memory is full, we add the tuple to the disk. After one pass, we can return the first set of skylines. The ones we wrote on disk still need to be pruned. So we do this process once more to get another set of skylines.

We were also able to restrict the main memory limit to n_pages as defined in the project description by controlling the size of the memory window.

2.6. Store and index data in Minibase

2.6.1 Modify BTree to support float keys

The BTree Implementation present in Minibase supported only integer and string keys, support for floating point keys was required to support indexing the test data. This required changes to multiple files. First we created a new FloatKey class similar to IntegerKey class in the btree package.Then we added constructors in KeyDataEntry class for Float keys. Multiple utility functions present in the BT.java class had to be modified to now support AttrType.attrReal (floating point) keys. These utility functions were mainly for comparison of keys, serialization and deserialization of keys into byte-array. Finally, IndexScan and IndexUtils classes were updated to support Float keys.

Finally, we implemented the BtreeGeneratorUtil class, this was a simple utility class to read the database heap file and create BTree indexes for each of the attributes. Individual B Trees were created by scanning the heapfile and inserting the required attribute as a FloatKey in B Tree. These BTrees would be used as input to the BTreeSky class.

2.6.2 Create BTree on combined attributes

To create a combined key on preferred attributes for BtreeSorted skyline, we created an index when the user would input the prefered attributes. This implementation also uses the Float Keys. We created GenerateIndexFiles in btree package to create BTree on combined attributes, function createCombinedBTreeIndex reads the data file and creates the combined btree. The sum of the preferred attributes is used as the key.
And for creating index in descending order, we used the same approach we did for individual attributes BTree i.e. multiplying the key with -1 and inserting it into BTreeFile. To scan we run the index scan as usual. The index created here will be provided to BTreeSortedSky.

2.7 Memory allowance (n_pages)

Since all our skylines have a feature to restrict the main memory usage, we have tweaked the existing minibase buffer manager to augment the presence of only n_pages in the buffer manager. Later, when the skyline computation is complete, we disable the feature to let the buffer manager use the total main memory available.

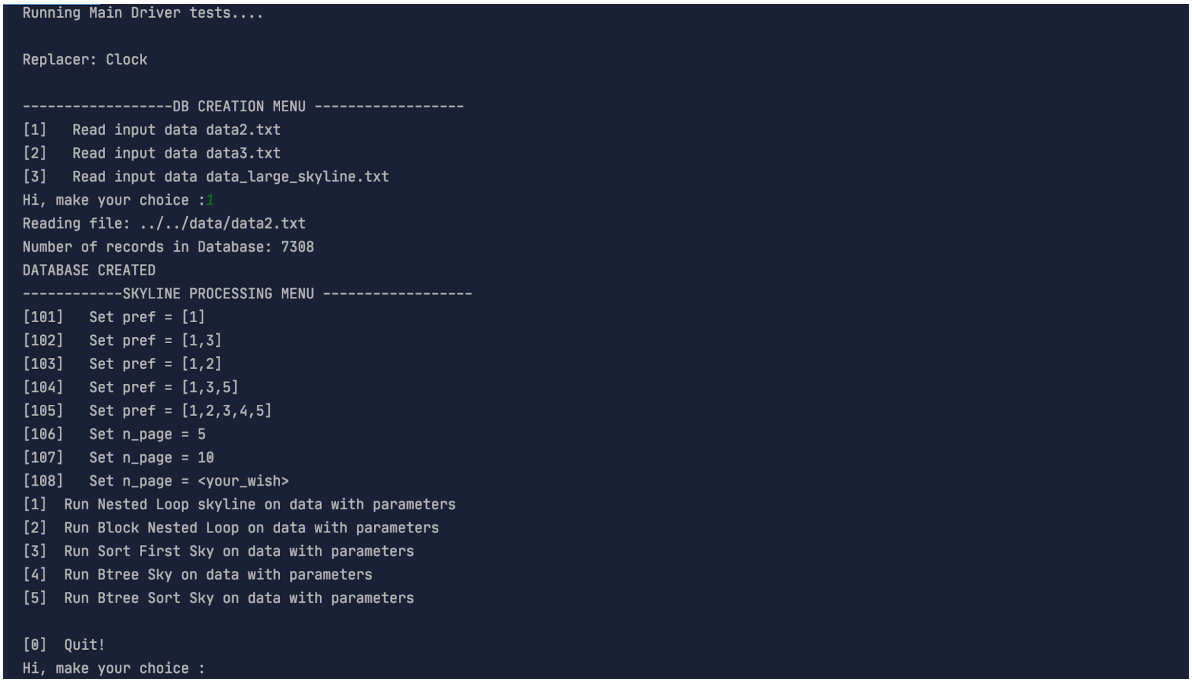
To use this feature, and as described in the specifications of Task 6, we create and store our index files immediately after getting the data file. Once the index files are ready, we will store them onto the disk and enforce the memory allowance (n_pages) in the buffer manager. This point onwards, buffer manager would not be able to use more than the given memory limit. Once the skyline computations are complete, we disable the memory limit feature to continue normal operations in the buffer manager.

Also, since some skylines have lists/arrays/queues to store the skyline candidates in main memory, we have divided the memory allowance between them and the buffer manager. Hence, the buffer manager will not have the entire allocated memory but only a part of it which is decided by each algorithm individually in the current implementation. Now because each algorithm is implemented in a different way and using different numbers of buffer pages, some algorithms might work with a small memory allowance and some might not. This will be reflected in the output files attached with this report.

2.8. Count the number of reads and writes

As a part of this task, we have added a counter over the database to track the number of reads and writes happening on the DB. This is later used to calculate the efficiency of the skyline algorithms and see the impact of low vs high memory allowance (n_pages) on disk read and write counters.

3. Interface Specifications



This is the main interface, the first screen that appears when we run the project. The user can see the parameters like preference list, and number of pages, and can set them. The user can run the specific task by giving the corresponding choice as mentioned. The task wise interfaces are as below:

3.1. Task 1

3.1.1 Dominates Interface

```
static boolean Dominates(Tuple t1,
                         AttrType[] type1,
                         Tuple t2,
                         AttrType[] type2,
                         short len_in,
                         short[] str_sizes,
                         int[] pref_list,
                         int pref_list_length)
```

Parameter	Description
Tuple t1	Tuple one
AttrType[] type1	array showing what the attributes of the input fields.
Tuple t2	Tuple two
AttrType[] type2	array showing what the attributes of the input fields.
Short len_in	number of attributes in the input tuple
Short str_sizes	shows the length of the string fields
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes

The Dominates function in TupleUtils is used to check if Tuple t1 dominates Tuple t2. This function is used internally in skyline computations. Any user can call this function from the TupleUtils. Two tuples Are taken as input (t1 and t2), along with the attributes of input fields and length of tuples. A preference list is given, which is used to determine the dominance between two tuples. The length of preference list is also given as input to the function. The output is 1 if t1 dominates t2 or 0.

3.1.2 CompareTupleWithTuplePref Interface

```
static boolean CompareTupleWithTuplePref(Tuple t1,
                                         AttrType[] type1,
                                         Tuple t2,
                                         AttrType[] type2,
                                         short len_in,
                                         short[] str_sizes,
                                         int[] pref_list,
                                         int pref_list_length)
```

Parameter	Description
Tuple t1	Tuple one
AttrType[] type1	array showing what the attributes of the input fields.
Tuple t2	Tuple two
AttrType[] type2	array showing what the attributes of the input fields.
Short len_in	number of attributes in the input tuple
Short str_sizes	shows the length of the string fields
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes

This function is used to determine whether t1 dominates t2. Two tuples (t1 and t2) are given along with an array of their types. A preference list is given as input along with its length. The preference list helps in determining the attributes which are accounted for calculating the sum for the dominance. The function returns 1 if t1 is greater than t2, -1 if t2 is greater and 0 if both are equal.

3.2. Task 2

```
SortPref(AttrType[] in,
                 short len_in,
                 short[] str_sizes,
                 Iterator am,
                 TupleOrder sort_order,
                 int[] pref_list,
                 int pref_list_length,
                 int n_pages)
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
short len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
Iterator am1	iterator over the data file (not used in our case; passing it as null)
TupleOrder sort_order	Iterator over the sorted data
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes
int n_pages	number of pages available for the sort operation

This task performs the sorting of tuples. Rather than standard sorting on a single column, the task uses the CompareTuple logic from the previous task. The array containing the attributes is given along with the length of input. An iterator is given to read the file (not used here, so null). A preference list is given along with its length. The sorting is done based on the sum of preferred attributes.

3.3. Task 3

3.3.1 Nested Loop Skyline Interface

```
NestedLoopsSky(AttrType[] in1, int len_in1, short[] t1_str_sizes,
                Iterator am1, java.lang.String
                relationName, int[] pref_list, int pref_list_length,
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
int len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
Iterator am1	iterator over the data file (not used in our case; passing it as null)
String relationName	Name of the heapfile containing the data
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes
int n_pages	number of pages available for the skyline operation

Nested Loop is implemented as an iterator and supports all the basic functionalities of an iterator. Users can call the NestedLoopsSky object with the parameters as mentioned above and then use the get_next() function of the iterator to retrieve the next element in the skyline data set. The driver implemented as a part of Task 6 will take care of everything and provide the user with a UI to work with the skyline features.

3.3.2 Block Nested Loop Skyline Interface

```
BlockNestedLoopsSky(AttrType[] in1, int len_in1, short[] t1_str_sizes,
                Iterator am1, java.lang.String
                relationName, int[] pref_list, int pref_list_length,
                int n_pages)
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
int len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
Iterator am1	iterator over the data file (not used in our case; passing it as null)
String relationName	Name of the heapfile containing the data
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes
int n_pages	number of pages available for the skyline operation

Block Nested Loop is implemented as an iterator and supports all the basic functionalities of an iterator. User can call the BlockNestedLoopsSky object with the parameters as mentioned above and then use the get_next() function of the iterator to retrieve the next element in the skyline data set. The driver implemented as a part of Task 6 will take care of everything and provide the user with a UI to work with the skyline features.

3.3.3 Sort First Skyline Interface

```
SortFirstSky(AttrType[] in1, int len_in1, short[] t1_str_sizes,
                Iterator am1, java.lang.String
                relationName, int[] pref_list, int pref_list_length,
                int n_pages)
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
int len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
Iterator am1	iterator over the data file (not used in our case; passing it as null)

String relationName	Name of the heapfile containing the data
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes
int n_pages	number of pages available for the skyline operation

3.4. Task 4

```
BTreeSky(AttrType[] in1, int len_in1, short[] t1_str_sizes,
                Iterator am1, java.lang.String
                relationName, int[] pref_list, int[] pref_list_length,
                IndexFile[] index_file_list,
                int n_pages)
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
int len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
Iterator am1	iterator over the data file (not used in our case; passing it as null)
String relationName	Name of the heapfile containing the data
int[] pref_list	array of the indices of the preferred attributes
int pref_list_length	number of preferred attributes
IndexFile [] index_file_list	array of index files
int n_pages	number of buffer frames available for the operation

This task implements the BTreeSky iterator which computes the skyline, based on individual preference attributes indexes. Once again, we will pass the input details, preference list details, index file list and number of pages. The Array of IndexFile has reference to BTree indexes of preference attributes.

3.5. Task 5

```
BTreeSortedSky(AttrType[] in1, int len_in1, short[] t1_str_sizes, int
               Iterator am1, java.lang.String
               relationName, int[] pref_list, int[]
               pref_list_length, IndexFile index_file,
               int n_pages)
```

Parameter	Description
AttrType[] in1	array showing what the attributes of the input fields are
int len_in1	number of attributes in the input tuple
short[] t1_str_sizes	shows the length of the string fields
int Iterator am1	Iterator to iterator over the data. Not used in this algorithm
String relationName	Name of the heapfile containing the data
Int [] pref_list	Int array containing 0s and 1s. 0 meaning that dimension is ignored. 1 meaning use that attribute in combined indeing. (NOTE: If you are using the Appdriver.Java file to test, you are not required to follow this format of pref_list. Just follow the prompts given in Appdriver)
Int pref_list_length	Length of the pref_list array.
IndexFile index_file	The file that has been indexed using combined BTree indexes on preference attribute. This is the file returned from task 6's CombinedBTreeIndex method
Int n_pages	Max number of pages that are allowed

In this task, we implement the BTreeSortedSky iterator which computes the Skylines using combined BTree on preference attributes. The input of the task is the input tuples and their data type and length. We give an iterator as input (though it is not used in this algorithm). We set the heapfile in which data is stored. The information of the preference list and index file with combined BTree is given as input. Finally, the user sets the maximum number of pages allowed.

3.6. Task 6

```
public class AppDriver implements  GlobalConst{

    public static void main(String [] argvs) {

        try{
            Driver driver = new Driver();
            driver.runTests();
        }
        catch (Exception e) {
            System.err.println ("Error encountered during running main driver:\n");
            e.printStackTrace();
            Runtime.getRuntime().exit(1);
        }finally {

        }
    }
```

Implemented an AppDriver class in the driver folder inside the src folder. This Driver runs all the necessary UI so that the user can run the skyline on given three datasets.

4. System Requirements/ Execution instructions

To be noted →

1. Currently the data set paths are hard coded in the code and if anyone wants to run the AppDriver in their own environment, they will need to edit the paths stored in dbcreationmenu() in src/driver/AppDriver.java.

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 AppDriver.java file to use the UI menu created as a part of Task 6.

4. Run `make db` command inside `src/` folder of Phase_2. 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
```

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

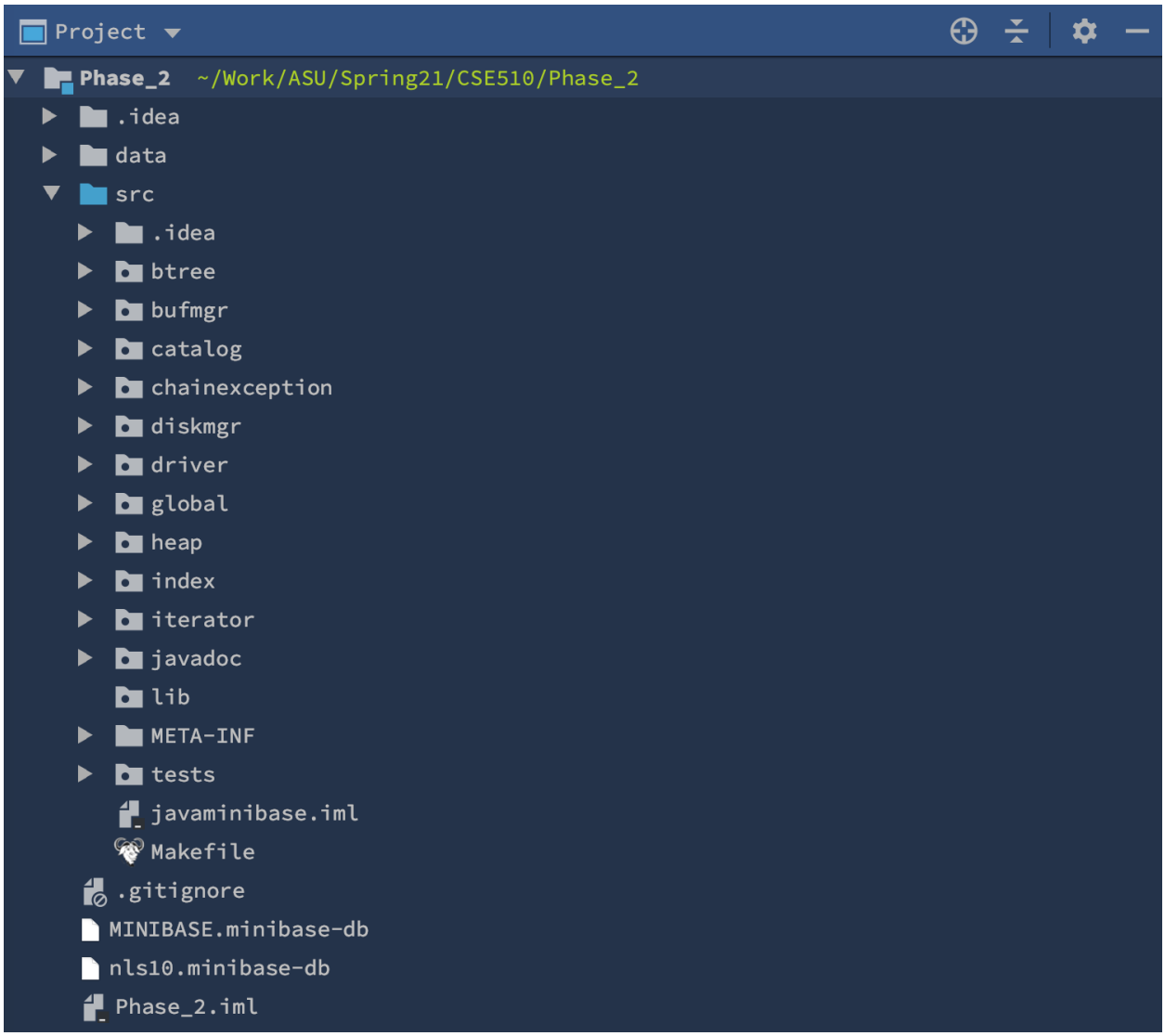
```
mish@Mishal driver % make driver
/usr/bin/javac -classpath .:.. AppDriver.java
/usr/bin/java  -classpath .:.. driver.AppDriver

Running Main Driver tests....

Replacer: Clock

------------------DB CREATION MENU ------------------
[1]   Read input data data2.txt
[2]   Read input data data3.txt
[3]   Read input data data_large_skyline.txt
Hi, make your choice :
```

6. The directory structure is shown below.



5. Related Work

There has been a lot of research work done on finding interesting data points and computing skylines using various algorithms. A Divide and Conquer based algorithm and some optimizations are described in Section 3.4 of “The Skyline operator”[1]. This algorithm divides the entire dataset into m partitions, and computes a skyline for each partition. Then the partitions are merged one by one and the dominated points are removed, the points remaining after all the merges will be the skyline. Downside of this algorithm is that it needs to be executed completely to get the first skyline point.

There is a substantial work done on skyline computation on streams of data. For example, Lena Rudenko and Markus Endres in their paper [5] present a Stream Lattice Skyline (SLS) algorithm for efficient skyline computation on real time unbounded streams. SLS works by constructing and maintaining a Better-Than-Graph, it doesn’t rely on tuple to tuple dominates comparison. The skyline can be found by doing graph traversals on this data structure.

The work we have done in this phase of the project is on static data from a single table. However, there are algorithms for calculation of skylines on data using joins from multiple tables presented in Mithila Nagendra’s dissertation [4]. Here the author improves upon previous two ways skyline-join query algorithms by reducing the tuple to tuple dominates check by developing a pruning strategy. The inner and outer tables data is divided into dominance layers and regions and a trie based data structure to keep track of the regions dominance to the different layers in the outer table.

6. Conclusion

We have extended minibase and added various methods to compute the skyline. We were able to see a difference in runtimes and disk I/O of different skyline algorithms. With a smaller amount of memory available, we see that there are more number of replacements in the buffer manager which leads to more number of reads and writes which in turn significantly impacts the latency of the algorithm.

NestedLoopsSky, being the naive approach, takes more time to run since it does a lot of read operations on the disk. On the other hand, BlockNestedLoopsSky which uses an in-memory window to speed up NestedLoopSky has better efficiency and fewer disk I/O. One step further, SortFirstSky, provides the BlockNestedLoopsSky with a sorted heap file which speeds up the computation since a lesser number of tuple comparisons are required to verify if a tuple belongs to the skyline.

We also implemented algorithms which compute skyline using indexes on the data, BTreeSky uses individual indexes per attribute and BTreeSortedSky uses a combined BTree on all attributes. The tuple to tuple comparisons are fewer in these functions as compared to NestedLoop and BlockNestedLoop, but their real world use depends on whether such indexes are available beforehand.

The skyline computation functions implemented in this phase will be a building block for implementing a user preference sensitive DBMS to e-commerce applications in future phases.

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. M. Nagendra, “Efficient Processing of Skyline Queries on Static Data Sources, Data Streams and Incomplete Datasets,” Arizona State University, 2014. https://repository.asu.edu/items/27470
5. Rudenko L., Endres M. (2018) Real-Time Skyline Computation on Data Streams. In: Benczúr A. et al. (eds) New Trends in Databases and Information Systems. ADBIS 2018. Communications in Computer and Information Science, vol 909. Springer, Cham. https://doi.org/10.1007/978-3-030-00063-9_3

Appendix

Roles of group members:

Name	Role
Kunj Patel	Implemented Task 5 (BTreeSortedSky) and also wrote tests for it
Khushal Modi	Implemented Nested Loop Sky, Block Nested Loop Sky and buffer manager feature to support n_pages
Mishal Shah	Implemented modified Sort, counting number of reads and write and SortFirstSky
Monil Nisar	Implemented modified Sort, Task 6(store data and index) and documented the report
Samip Thakkar	Implemented Tuple comparisons and documented the report
Saurabh Gaggar	Implement BTreeSky iterator, add float keys support to BTrees