Skyline Computation- Project Phase 2

CSE 510: Database Management System Implementation

Spring 2021

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

Abstract

In this phase, we have worked towards implementing various methods to compute the skylines on a given dataset. There are 3 methods to design the skyline, namely- Nested Loop Skylines, Block-Nested Loop Skylines, Sort-First Skylines. Later, we used sorted B+ Trees to implement the skyline functions on the data. In the end, we conduct a performance analysis on the given methods.

Keywords

Database Management Systems, OpenSourceDatabase, MiniBase Database, Skylines, NestedLoop Skylines, BlockNestedLoop Skylines, Buffer Manager, Disk Space Manager, Heap File Management, B+ Tree, Java

Introduction

In this phase of the project, we start implementing various methods of skyline operators and integrate them to run with the minibase database management system. There are various tasks designed in order to accomplish the goal implemented to work by maximizing the preference attributes.

In phase 1 of this project, we ran various tests so as to understand the specifications of the minibase database management system. Using the understanding of Buffer Manager, Disk Space Management System, Heap File structure, B+ Tree indexing from phase 1, we have made modifications to the in-built methods of minibase and added some functions from task 1 through task 7.

We start adding the additional functionality by first creating a `Dominates` function that checks if a given tuple dominates another tuple and returns a boolean output. Moreover, we also create a tuple comparison method `CompareTupleWithTuplePref` which returns boolean output after comparing tuples on the sum of their preference attributes. In task 2, we modify the `Sort` function in a way that the tuples are sorted based on a monotonic function, here, the sum of their preference attributes. Later in task 3, we start developing methods to compute the skylines. We make a method to compute skyline using `NestedLoopsSky`, `BlockNestedLoopsSky`, and `SortFirstSky`. Once these methods are created, we use them in task 4 and task 5 to generate skylines from sorted B+ Trees. In task 6, we integrate all the previous tasks to compute skylines using the previously created methods. Here, we let the user decide the attributes to make the skylines and also the number of memory pages for the operation. Finally, in task 7, we make use of the given class and take counts of the number of reads and writes.

Task 1

In this task, the aim is to create a tuple comparison method `Dominates,` that returns 1 if tuple t1 dominates tuple t2 in the list of preference attributes. This is a simple tuple comparison method, one of the starting steps towards computing the skyline for the given data table, on the set of preference attributes.

The function `Dominates` checks whether tuple1 is better than tuple2 on the preference attributes. We have a flag that indicates the situation. It is initially set to 0. We iterate through the list of preference attributes, and first, check the type of attributes in the tuples. We start a loop on the preference columns of the tuples, initially checking if the attribute type of each column in tuple1 matches that of its corresponding column in tuple2. For the main domination logic, we would want the data to be able to rank higher or lower than others, and hence we consider columns that are of type Integer and Float(Real). In the loop, using simple if-else logic, we check if the value in a preference list attribute in tuple1 is greater than that in the corresponding preference list attribute in tuple2. If this condition holds true, we set the flag as 1. But if the converse condition holds true in that at least one value in the preference attribute in tuple1 is smaller than the core value in preference list attribute in tuple2, we set the flag as -1 and return 0, stating that the tuple1 does not dominate tuple2. After all the passes, we check if the flag variable is 1 so as to align the code to match the logic of Dominates function that if tuple1 dominates tuple2 in the list of preference attributes, return 1, else return 0.

Next in this task, we implement the function `CompareTupleWithTuplePref` which compares tuples based on the sum of their preference attributes. First, we iterate over the length of the preference list and check the data types of each preference attribute in the tuple. If the attributes are Integer or Float(Real), we compute the sums of each preference attribute for the individual tuples and store them in their respective sum variables. Finally, we check which tuple has a higher sum of preference attributes. If the sum of preference attributes for tuple1 is greater than that of tuple2, we return 1, vice versa if the opposite is true, we return -1. If both the sums are equal, we return 0.

Task 2

In this task, we modify `Sort` such that tuples are sorted according to the function `CompareTupleWithTuplePref` based on the preference attributes. We start by cleaning all the files and system data that is running. Then, we create a tuple of appropriate size and set its header, followed by the creation of a new heap file. Now, we iterate through the list of preference attributes and set the tuple field to an integer type, followed by the insertion of this tuple into the heap file. Once this is done, we create an iterator for the tuples and run a file scan on the heap file. Now, we start sorting this heap file based on the preference list. The modification on this will be based on certain conditions. SortPref takes the total attributes, the attribute size of the tuple, fscan iterator, the tuple order, the preference list or the skyline attributes that we want to sort on, the length of the preference list, and the buffer pages that we can use to sort the data. The SortPref iterator sorts the data on the sum of skyline attributes then returns the sorted values when we call the get_next() function. The sortpref iterator is made using making changes to Sort iterator by changing the compare function which sorts on a single attribute to sort on multiple attributes by using the CompareTupleWithTuplePref. We also modified files like pnodePQ.java and pnodeSplayPQ.java. We also had to change the MIN_VAL function and MAX_VAL functions from sort iterator so that they accommodate the list of attributes in place of a single attribute. The TupleUtils.SetValue is also modified so that all the attributes are set to the correct values. SortPref works by dividing the input into several batches and sorting them using a buffer then combining them so that the entire file is modified. As sortpref is made by extending the iterator class so we also needed to override the get_next method.

Problems Faced:

1) Sort.java class in iterator is made to sort on a single attribute. We faced problems in making the sort work on multiple attributes.
2) SortPref is throwing some weird errors when run on large datasets because the number of buffers are not sufficient for the sort. Needed to increase the buffer to make it work.
3) SortPref is also used in SortFirstSky and BTreeSky iterators had difficulty in integrating those tasks as SortPref acts weird when the n_pages available are not sufficient and iterator heap file names must also be different as they may replace each other.

Task 3

In this task, we implement the skyline functions in three ways. The Nested Loop method, Block Nested Loop method, and the Sorted Skyline method.

To implement these methods, the parameters passed are the name of the heapfile containing the tuples, the iterator on this heapfile, the total number of columns of these tuples, the data type of all columns, the preference attributes, the number of preference attributes, and the total number of buffer pages provided to run the algorithm.

Nested Loop Method:

For the Nested Loop method, we do not use the buffer pages. We iterate through all the tuples in 2 loops, search for the skyline elements and add them to an output heapfile.

We start with opening the input heapfile and initializing a filescan to scan through the tuples. We start a loop using this scan which we will consider as the outer loop. Here, we initialize a new filescan and loop through it. This is the inner loop. We have an array of flags that is set to 0 initially, which we will use to determine if a tuple needs to be considered for checking dominance.

For a single iteration of the outer loop, we iterate through all the records in the inner loop. If the flag corresponding to the inner tuple is 0, the Dominate function is called on the outer and inner tuples, with all necessary parameters. This determines whether the outer tuple dominates the inner tuple. If it does, i.e the return value is 1, the flag of the inner tuple is set to 1. If it does not, the return value being 0, the flag remains 0. The inner loop then moves to the next tuple. After the inner loop is iterated through for a single outer tuple if its corresponding flag is 1, it is considered a skyline element is added to the output heapfile. Then outer loop moves to the next tuple whose flag is 0 and this process continues until the outer loop is done.

The output heapfile now contains all the skyline elements from the input heapfile. A getnext function is implemented on this output heapfile to provide a method to get the skyline elements one at a time.

Problems Faced:

1) The inner loop opens many fscan iterators as we instantiate them with the new operator; this works fine for small datasets but for large datasets, this throws exceptions we handle them meticulously by closing fscan iterators whenever we are done using those iterators.

Block-Nested Loop Method:

For the Block-Nested Loop method, we use a buffer page to store skyline candidates. We Iterate through the input heap file using filescan. We then check if each element in the filescan is dominated by the skyline candidates and if the skyline candidates dominate the heap element. If the filescan element is not dominated by any of the skyline candidates in the buffer, then the filescan element is added to the buffer (We implement this using a flag). However, if any skyline candidate gets dominated, it is removed from the buffer. At the end of the filescan, the elements left in the buffer are bonafide skyline elements.
We have also considered limitations on the buffer size. If the buffer pages are at capacity and a skyline candidate needs to be added, the element is added to a heapfile. After the filescan is completed all the elements in the buffer that were added before the creation of the heapfile are bonafide skylines and emitted through get_next. The skyline candidates added to the buffer after the creation of the heapfile remains in the buffer. The heap file will be processed as a fresh filescan and when there are no elements stored in the heapfile at the end of the filescan we have retrieved all bonafide skylines.

Sorted Skyline:

For the Sorted Skyline method, we sort the tuples before determining the skyline elements and then add them to an output heapfile.

We start by opening the input heapfile and initializing a filescan on it. We then call the SortPref method on the tuples to sort them in descending order. We sorted the tuples in descending order by modifying the `keyCompare` method of the B+ Tree implementation in minibase. Since we are using a buffer in this methodology, we initialize n_pages worth of buffer memory.

Sorted skyline works by first sorting the elements using sort pref on the sum of all the attributes. Then this sorted file acts as an outer loop, now we insert the first element of the sorted file directly into the buffer as it must definitely be the skyline. Then when we are inserting from the second element we compare it with all elements in the buffer and if it's not getting dominated by any other element in the buffer we insert it into the buffer as it belongs to the final skyline elements. If the buffer gets exceeded we insert the exceeded elements into the heap. Now the elements from the buffer are removed and inserted into the final skyline heap file. The exceeded heap file will act as the new outer loop as its elements are not vetted against each other. Now the buffer is empty so now again we fill this buffer and repeat the entire process again and again until the buffer is not full. To accommodate this process we are using 2 heap files and swapping them whenever we want them to act as the outer loop or as the overflow heap.

When n_pages = 5, in the testing we do not get any skyline because the n_pages are not sufficient to sort. The same holds true for n_pages = 10. Both the test files have approximately 7300 tuples. So, when the PAGE_SIZE is 1024, each page can accommodate 64 tuples. We require approximately 115 buffer pages for `SortPrefSky` to function (7300/64 ~ 115).

Similarly, when the PAGE_SIZE is 128, each page can accommodate 8 tuples. So, we require approximately 915 tuples for this to function.

When enough buffer pages are available, the disk reads and writes are 0.

Task 4

In this task, the aim is to create n B+ Trees where n is the number of the preference attributes, using `BTreeSky` iterator that returns skyline tuples. This is a Data Structure that prunes the tuples that are being dominated by the other tuples, one of the starting steps towards computing the skyline for the given data table, on the set of preference attributes.

In this algorithm, we first form a B+ Tree for each attribute in descending order. We compare the first tuple in the first attribute's B+ Tree with all the tuples in the remaining B+ Trees using the `CompareTupleWithTuple`. When the same tuple is encountered, we break the loop because the tuples after this will have values lesser than this tuple. This is because they are in a sorted manner. [1]

An unclustered B+ Tree consists of a collection of records of the form (key, rid) where Rid is the ID of the record being indexed and the key is the value for the search key. The leaf values of the B+ Tree store the key and rid pairs while the internal nodes store the key and the pageids of their children. We need to access the leaf nodes to obtain the sorted data.

Step Wise Algorithm

Input: A Dataset D (B+-tree)
Output: The Set of skyline points of dataset D.

1) S = ϕ //list of the skyline points
2) D //dataset D of all the points
3) Create a heapfile f named as 'unsorted file' (this will store all the tuples)
4) Create a heapfile hf named as 'skyline candidate' (this will prune the tuples and store them)
5) Insert all the tuples in the heap file f
6) Initialize a B+ TreeFile btf for each attribute and sort the B+ Tree based on that attribute.
7) Set *temp_tuple* as 1st tuple of the B+ Tree sorted on the 1st pref_attr
8) for i = 1 to len(pref_attr)
   Start Index scan
   Compare *iter_tuple* to *temp_tuple* (obtained by get_next())
   if *iter_tuple = temp_tuple* then

break the loop

else

if heapfile ‘skyline candidate’ does not contain iter_tuple then

skyline candidate.insertRecord(iter_tuple)

9) Pass the heapfile to SortFirstSky and compute the skyline to store in S

In our dataset,

Movie Dataset

This is a randomly created dataset of movies. Here, we will use this as an example to explain the next task.

Movie Name	Viewer's Ratings	Views in 1000s (k)
T1	3.5	90
T2	4.0	150
T3	1.7	20
T4	2.8	50
T5	4.8	160
T6	2.0	90
T7	3.0	95
T8	4.9	155
T9	3.8	110
T10	5.0	140

Tuples t5, t8, t2, t10 are the tuples stored in a heap file that is fed to SortFirstSky and then skyline is computed.

The basic logic here is,

1) t10 is definitely a part of the skyline
2) any tuple that has not yet been inspected is not a part of the skyline candidates
3) All the tuples before t10 may or may not be part of the skyline.

Problems faced:

1) Initially we decided to implement another algorithm similar to Fagin's Algorithm, which required to form HashMap inside the buffer. Later we realized that, this would cause issues in buffer for larger datasets.

Task 5

In this task, we have to implement the `BTreeSortedSky` which computes the skyline based on the sum of preference attributes using B+ Trees. Basically, we start with a heap file that contains our data. Once that file is created, we create a scan on the file and make an index file on the sum of preference attributes. To create the index, we first create a new index file and keep calling the get_next method on the data to get the next tuples. The tuples in the leaf nodes will be sorted in descending order of the sum of preference attributes.

Now, the first tuple (say M1) in the sorted B+ Tree will be a part of our skyline. Hence, we send it to the buffer. Now, we start iterating over other tuples but there's a condition that any tuple that comes later (say M2 to Mn) from the B+ Tree will not dominate other tuples in the buffer (because they are sorted on the monotonic function).

Now, M1 may or may not dominate M2. If M1 dominates M2, then M2 cannot be in the skyline and the outer loop moves forward to the next tuple. However, if M1 does not dominate M2, we insert M2 into the buffer. Assuming that M2 was sent to the buffer, we again compare the next element M3 with both M1 and M2. Now, as the B+ Tree is sorted in descending order, M3 cannot dominate M1 and M2. If M1 and M2 both do not dominate M3, it is a skyline member, however, if even one member from the buffer dominates a member coming in from the B+ Tree, then that tuple cannot be a skyline member. In the same manner, the entire B+ Tree is iterated and we keep filling the buffer.

On completion of the entire outer loop (B+ Tree), the tuples in the buffer are skyline members, plus we have a temporary heap file to store the tuples when the buffer gets full while the iteration is still going on in the outer loop. Suppose that the next tuple to come from the outer loop after M3 is M4, and now the buffer is full. So we store M4 into the heap file. Next tuple M5 will iterate through the buffer even though M4 is in the heap. Until the buffer is still full, we do not check the remaining tuples with the tuples in the heap file. Thus, once the entire iteration is over, we empty the buffer which contains all the skyline attributes. There will still be tuples in the heap file, so that becomes the outer loop and again the buffer will be empty as we removed all the skyline attributes from it. Hence, the same process is repeated on the heap file and the buffer. The first tuple in the heap file will obviously be a skyline attribute. [3] We switch between two heap files `temp` and `tempone` alternatively so that they can act as the outer loop and file to accommodate tuples when the buffer gets full.

For task4 and task5, we had noticed that the B+ Trees accommodated values of attribute types String and Integer but not float. So key creation, key comparisons and other B+tree operations did not work on Float variables. We went ahead and added the functionality for the B+Tree to accept float values and operate on them.
This algorithm involves a lot of records fetching which proves to be very costly. Hence the number of disk reads is close to a million. Hence, this is cost-intensive. When the PAGE_SIZE is 128, the disk reads are close to 20 million.
But, in this algorithm, the advantage is that any tuple in the buffer is surely a skyline tuple and it is not necessary to wait for an entire iteration of the outer loop to be sure whether a tuple in the buffer is a skyline tuple. This is a problem that arises in `BlockNestedSkyline`.

Problems faced:
The IoBuf.java and OBuf.java buffers are a part of the Minibase implementation but these buffers functioned in a way that were not entirely fulfilling the purpose of the `SortFirstSky` and `BTreeSortedSky` algorithms. Hence, a new buffer had to be created.
The new buffer “Put” the new tuple in a heap file if the buffer was full instead of flushing the entire buffer into the heapfile like it did previously. The `BufCompareForDomination` method took the outer heapfile tuple as a parameter and ran a loop over the entire inner loop of buffer tuples. It inserted the tuple in the heapfile if the buffer was full. This new “skBuf.java” buffer was used for finding the skyline tuples using `SortFirstSky` and `BTreeSortedSky` algorithms.

Task 6

In this task, we take input data and make the function calls to compute skylines on this data.
We start by reading the data from a .txt file and writing it into a heapfile. The skyline computation algorithms based on heap files are then run on this.
For the algorithms that are based on B+Trees, an index file is created on the heapfile before running them. The function written in Task 4 builds the index on B+trees on its individual attributes while the function written in Task 5 builds the index on B+trees on the sum of preference attributes.
Problems faced while implementation:

- While reading the large datasets and storing them in the heapfile, an Out-of-Space exception occurred. This was resolved by increasing the number of pages in the DB parameter in the SystemDefs constructor call.
- While performing the tests we were unable to delete the heap file, that is the reason we are getting multiple copies of the same skyline tuples. The heapfile.delete() function is not removing the tuples that are in the file i.e the heap count remains the same before and after deletion.

Task 7

This task does the basic computation of the number of reads and writes taking place on the disk. The `pcounter.java` increments or decrements the disk reads and writes whenever a disk page is written to or read from.

Interface specifications

In the codebase, to test the skyline functions against various input files, test scripts are written in the “skyline” folder with names in the form *task<i>file<j>.java*, where i=task number and j=data file number.

Test	Test Script	Typescript
Nested Loop Skyline	• task31.java - testing on data2.txt and data3.txt • NestedLoopsSky.java - iterator In the first file, change the GLOBAL CONSTANTS for changing the PAGE_SIZE and change the name of the	typescript

	filename variable to use both data2.txt and data3.txt	
Block Nested Loop Skyline	BlockNestedLoopSky.java This is the iterator	
Sort First Skyline	• SortFirstSky.java - iterator • sortFirstSkyTest.java- test on dummy data • skBuf.java - buffer • Task33file2or3.java In the last file, change the GLOBAL CONSTANTS for changing the PAGE_SIZE and change the name of the filename variable to use both data2.txt and data3.txt	• t33f2(1) • t33f3(1) • t33f2(2) • t33f3(2).txt In this naming convention, (1) signifies page size 1024 and (2) signifies 128 page size. f2 signifies data2.txt and f3 signifies data3.txt
BTreeSkyline	• BTreeSky.java- iterartor • BTreeSkyTest.java- Testing on dummy data • Task4.java	
BTreeSortedSkyline	• skBuf.java- buffer • BTreeSortedSky.java -iterator • BTreeSortedSkyTest.java- Testing on dummy data • task5file2.java tests data2.txt • task5file3.java tests data3.txt In the last two files the GLOBAL CONSTANTS have to be modified manually to test for 128 and 1024 page size.	t5f2(1).txt, t5f3(1).txt, t5f2(2), t5f3(2) In this naming convention, (1) signifies page size 1024 and (2) signifies 128 page size. f2 signifies data2.txt and f3 signifies data3.txt

System Requirements and execution instructions

The code additions as a part of this project were done in Linux based Machines. We used Visual Studio Code as the IDE to debug and run.
To test the methods implemented, the files mentioned under the Test Script heading can be run.
To change the minibase size, the MINIBASE_PAGESIZE, MINIBASE_BUFFER_POOL_SIZE and MAX_SPACE values are changed.

Conclusion

After completing the phase 2 of this project, we collectively learnt about the various methods of implementing the skyline operator. It was observed that the Nested Loop Skyline operator functioned better than the Naive Based Skyline operator. However, the Block Nested Loop Skyline operator worked better than the Nested Loop Skyline operator because it makes use of the Buffer which results in lesser disk accesses, and also scans the outer loop only once. In SortFirst Skyline operator, we take the tuples then sort them using sort pref and then using buffers we are able to obtain the skylines.

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.

Appendix

Contributution by members:
Reading papers: All team members read the papers related to this project thoroughly
Task 1: Nithya Vardhan worked on writing the Dominates and CompareTupleWithTuplePref algorithms.
Task 2: Dheeraj worked on sortpref implementation.
Task 3: Nithya Vardhan started the NestedSkyline method adding the tuple iteration part, Dheeraj took over and added the main comparison logic of the Nested loop skyline and finished it. Karthik worked on implementing the BlockNested skyline. Dheeraj and Riya worked on Sort First Skyline and completed its implementation.
Task 4: Rutva and Dheeraj implemented task 4.
Task 5: Haard and Riya implemented the task 5
Task 6: Nithya Vardhan implemented the 6th task by reading the input files and writing to heapfiles. Later Haard, Riya, Rutva and Dheeraj collaborated on this to integrate their implementations and generate typescripts.
Task 7: Riya implemented the Task 7 counter method.

Report: Haard compiled most of the report, where other team members added details of their implementations in the respective tasks.