Journal of King Saud University – Computer and Information Sciences (2016) 28, 447–456

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Skyline computation for frequent queries in update intensive environment

R.D. Kulkarni *, **B.F. Momin**

Department of Computer Science and Engineering, Walchand College of Engineering, Sangli, Maharashtra, India

Received 27 March 2014; revised 7 March 2015; accepted 14 April 2015
Available online 2 December 2015

KEYWORDS

Skyline queries;
Frequent queries;
Query Profiler

Abstract The skyline queries produce the tuples which are 'promising' on the dimensions of the user's interest. The popular datasets often get queried by the users where dimensions of the user queries often overlap. For such frequent, overlapping skyline queries repeating computations on large datasets result in unacceptable response time. In the scenarios where, there exists a little deviation in the query dimensions than those of the popular dimensions or when the dataset gets updated, the re-use of the previous results can help in either avoiding or reducing further computational costs.

In this paper we focus exactly on this problem and aim at optimizing the response time of frequent or near to frequent skyline queries raised against the static and the update intensive dataset. We propose two novel, simple yet efficient algorithms namely the QPSkyline and the QPUpdateSkyline algorithm which make use of the proposed data structure called as 'Query Profiler' which aims at preserving the metadata of the skyline queries. The QPSkyline algorithm works in static environment and the QPUpdateSkyline algorithm is applicable for the datasets which experience frequent updates. The experiments performed on the real life dataset demonstrate the effectiveness and scalability of the proposed algorithms.

1. Introduction

The computational geometry has highlighted 'maximum vector problem' which aims at finding the 'maximum like hood' of some of the dimensions of the data. The concept of skyline queries stands up on similar lines. The skyline query helps to identify the "best" objects in a multi-attribute dataset. To understand the concept of the skyline query consider an example of a person going on holiday to Goa and looking for a hotel that is both cheap and close to the beach. This means the person is interested in all such hotels that are not worse than any other hotel on both dimensions. This set of interesting hotels is the "Skyline". From the skyline, one can make the decision after weighing the personal or the imposed preferences. The concept of the skyline was proposed for the first

* Corresponding author.

Peer review under responsibility of King Saud University.

ELSEVIER Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.04.003

time by Borzsonyi et al. (2001). They defined the skyline as a set of points which are not dominated by any other points in the dataset. This definition uses the concept of 'dominance'. A point is said to dominate other points when it is better on at least one dimension than the other points. As an example, consider the following sample dataset of hotels shown in Fig. 1(a) which includes dimensions such as hotel-name prize and distance to the beach.

The tuple $h3$ dominates all other tuples on dimensions of distance and the tuple $h1$ dominates all other tuples on the dimension of prize. However due to the multiple preferences given in the query, both the tuples $h1$ and $h3$ formulate the skyline. Every other point is said to be 'out of the sky'! Fig. 1(b) demonstrates this concept.

A common observation is that, the dataset often gets queried on popular dimensions! The users may generate queries which are specific to certain dimensions only or a slight deviation from those of the popular dimensions is observed. In such scenarios, peculiar to the dataset there exists a set of frequent queries and a set of near to frequent queries. By near to frequent queries, we mean all those queries where dimensions the user queries often overlap. That is either new dimensions are involved or a few dimensions are skipped from the set of popular dimensions. In this scenario, the cost of recomputation of the skyline queries can either be saved or minimized by maintaining the profile of the frequent queries in some way. This can be done if we save the results of the frequent queries in some way and make use of them to optimize the response time of the frequent or near to frequent queries which follow in future. Also the dataset undergoes the updates like addition of new records, deletion of the records or updates on the field values. Upon such updates on the dataset, repeating the skyline computation for a previously processed skyline query is a useless task as the updates may not always affect the existing skyline. Hence the problem is to avoid or reduce the re computational efforts whenever the updates made to a dataset do not affect the existing skylines of various queries. This is the motivation behind our research. We aim at optimizing the response time of the frequent and near to frequent skyline queries in update intensive environment though means of preserved statistics of the queries. To serve this purpose we propose the novel data structure called "*Query Profiler*". We also propose the algorithms '*QPSkyline'* which aims at minimizing the skyline computation efforts of the frequent and near to frequent queries and the algorithm '*QPUpdateSkyline'* which has the similar aim of that of the *QPSkyline'* algorithm however it works in the update intensive environment.

Through this paper we contribute as follows:

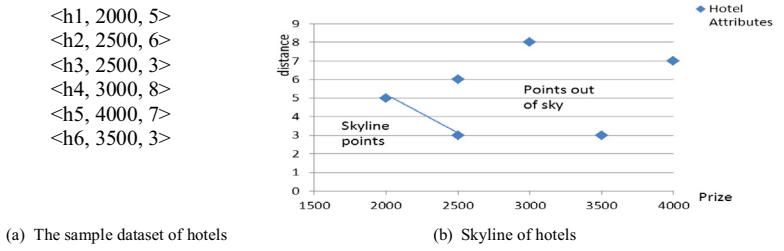
(1) We introduce the concept of *'Query Profiler'* for maintaining the metadata of the skyline queries.
(2) We propose a simple yet efficient algorithm *QPSkyline* which makes use of the *Query Profiler* to optimize the response time of frequent and near to frequent skyline queries.
(3) We also propose the *QPUpdateSkyline* algorithm which aims at optimizing the response time of such skyline queries in update intensive environment.
(4) We present the experimental results to assert the effectiveness of the proposed algorithms

The rest of the paper is organized as follows. Section 2 presents a brief review of the previous techniques related to the skyline computation. In Section 3, the proposed concept of *Query Profiler* (*QP*) has been discussed along with the *QPSkyline* algorithm, related experimentation and the analysis of the obtained results. Section 4 covers the *QPUpdateSkyline* algorithm along with the experimental results and related discussion. Section 5 concludes the paper.

2. Background and related work

Given a n dimensional relation R, a tuple t_i $(v_{i1}, v_{i2}, \ldots, v_{in})$ dominates other tuple t_j $(v_{j1}, v_{j2}, \ldots, v_{jn})$ if on all the dimensions t_i is as good or better than t_j and on at least dimension d, v_{id} is better than v_{jd}. The dominance between two tuples is denoted by $t_i > t_j$. The preferences of the dimensions are specified in the skyline query. (for example minimum prize, minimum distance). A tuple t is said to be in skyline of the relation, if there does not exist any other tuple in R, which dominates t.

Borzsonyi et al. proposed the concept of skyline operator (Borzsonyi et al., 2001). They proposed two basic algorithms namely 'Block Nested Loop' (*BNL*) and 'Divide and Conquer' (*D&C*) which compute the skyline by reading and processing all the input tuples. *BNL* (used by QPSkyline and QPUpdateSkyline) maintains a *window* of incomparable tuples in the main memory. When a tuple p is read from the input, it is compared with all the tuples in the window and if it is found to be a dominated tuple, it is eliminated from the consideration. In the other case (when it is found to be a dominating



(a) The sample dataset of hotels (b) Skyline of hotels

Figure 1 Sample dataset and skyline of it.

tuple) it either replaces a tuple in the window (the one it dominates) or being incomparable to all other tuples in the window, gets added in the window. If the window gets full during this process, p is written into a temporary file which gets processed in next iterations. The other algorithm *D&C* divides the data space iteratively computing the sub-skylines for each data space until the data space is very small. Merging and elimination of the sub skylines happens to produce the final skyline. The 'Sort Filter Skyline' (*SFS* Chomicki et al., 2003), is also a multi pass algorithm as *BNL*; however the input data are sorted first topologically on the dimensions compatible with the skyline criteria, before the algorithm proceeds. This helps to reduce the number of dominance tests and initial response time of the skyline queries. The 'Linear Elimination Sort of Skyline' (*LESS* Godfrey et al., 2005) and 'Sort and Limit Skyline (*SaLSa* Bartolini et al., 2006) evolved after SFS. These two algorithms also do the pre-sorting of the input data like in *SFS* and also preserve all its strengths like limiting the number of input tuples to be read, returning the results progressively, simplified management of the window and optimal number of passes of the filter phase. In Kossmann et al. (2002), an altogether different approach is used, called *Bitmap*. In the *Bitmap* algorithm, each record is mapped to an *m-bit vector* where m is sum of the distinct attribute values over all dimensions. Because bitwise operations are fast, a series of simple operations such as 'and, or' are performed on the bit vector and the result reflects in bits the 'dominance strength' of a record. For all sorts of datasets and higher dimensional queries, the techniques which index the dataset were required. In Sheng and Tao (2012), the research efforts focus on improvement of worst case time complexity of skyline computations. With index based techniques reading of the whole dataset is avoided as only a portion of an index is scanned to filter for the possible skyline candidate tuples. This is the beauty and the ultimate goal behind using the indexing structures. The researchers' community has used various indexing structures to improve the computational efficiency. Some of the popular indexing structures are B tree, B+ tree, R tree, R^* tree etc. The most simple algorithm BBS (Papadias et al., 2005) uses the R tree to index the data and a nearest neighbur (NN) search to compute the skyline. Because of effective utilization of the R tree index, *BBS* outperformed all the previous algorithms. Another algorithm *SUBSKY* (Tao et al., 2006) has used a single B tree index for the multi dimensional dataset by transforming them to a 1D value and it carries out pruning by heuristic techniques. Other algorithms which use indexing structures are *ZSearch* (Lee et al., 2007), SSPL (Han et al., 2013). Emergence of the modern hardware technologies encouraged the researchers' community to exploit the machine peculiarities like cache, cores etc. for the computation intensive algorithms. Caching of the results of the user queries has always been helpful in improving the response time and I/O performance of the data centric algorithms. This has been studied extensively in Dar et al. (1996), Ren and Kumar (2001), Bhattacharya et al. (2011). In Bhattacharya et al. (2011), the authors have proposed the idea of caching *semantic segments* for the user queries along with a cache replacement strategy. Some intelligent structures like *SkyCube* (Xia and Zhang, 2005; Yuan et al., 2005; Zhang et al., 2014), skyline graphs (Zheng et al., 2014) have also been proposed. They exploit the correlated computational dependencies among the queries however with the limited cache size; the techniques may not be able to work well with all the *SkyCube* sizes. The concept of clustering has been used in Ruan et al. (2013) where the skyline query based pre clustering has been applied on the dataset for parallel computation of the queries. The research work in this area peculiar to MANET, probabilistic data, fuzzy data, parallel and distributed computing (Papapetrou and Garofalakis, 2014) and other techniques do also exist. However we do not discuss it here for being out of scope of current research work.

In this paper, we utilize the concept of reusing the skyline results computed for earlier queries for serving subsequent, frequent skyline queries and near to frequent skyline queries with the help of properly managed and indexed structure: 'Query Profiler' (QP). Similar work on this line can be found in Ren and Kumar (2001), Bhattacharya et al. (2011), Xia et al. (2012), Siddique and Morimoto (2010), Siddique et al. (2012). Our research efforts differ from those of Bhattacharya et al. (2011), where the authors have not considered the situations where the dataset is frequently updated which is the obvious scenario. Also with the help of our proposed concept of QP, the metadata about the skyline queries is preserved and hence can be used later every time, whenever the queries are generated against the dataset in the future. This is the benefit over the cache memory used in Bhattacharya et al. (2011). The approach used in Xia et al. (2012), uses the concept of Compressed SkyCube (CSC) to return the skyline of any subspace without consulting the base table. However the benefits of our approach are its space efficacy and simplicity as against the memory requirements of the proposed cube structure and related computational complexity. The research work in Siddique and Morimoto (2010), Siddique et al. (2012) deals with efficient maintenance of all k-dominant skyline query results. The authors have used the properties of the dataset to process the skyline queries which is a tedious job for the update intensive and large datasets. As against this, we focus on exploiting the metadata of the queries which is more efficient when the queries repeat or overlap.

3. The Query Profiler

In this section we discuss the terms related to the proposed algorithms and also comment on their efficiency. Without loss of generality we assume that all skyline queries are raised against a single relation. Also in the coming discussion a term 'query' refers to 'skyline query'.

3.1. Processing the subsequent skyline queries

Let us first discuss how the various subsequent queries can be processed. For that let us revisit the query categorization as discussed in Bhattacharya et al. (2011). A subsequent query generated on relation R can be like one of these: (1) it carries dimensions exactly the same as some previous query (an exact query), (2) it carries all those dimensions which happen to be a subset of dimensions of some previous query (a subset query), (3) It carries some of the dimensions which happen to be a sub-

set of few previous queries and also may carry additional dimensions (a partial query) or (4) a query whose dimensions do not match with dimensions of any of the previous queries, (a novel query). Every query fired by the user can be categorized in the above order and manner before it is considered for computation. Now let us discuss how the subsequent query can be processed. To simplify the discussion consider the sample dataset below and assume user preferences will be on finding *minimum* of the available dimensions.

3.1.1. Exact queries

For each novel query, the skyline is computed by using any of the previous algorithms and the results are saved. If the subsequent query happens to be an exact query then recomputation of the skyline can be avoided by returning those results immediately which were saved previously. For example in Fig. 2(b) above, query with Qid = 6 is an exact query and its skyline is nothing but the skyline saved for query with Qid = 1.

3.1.2. Subset queries

If a subsequent query happens to be a subset query, then its skyline is completely contained in the skyline of the previous query to which it happens to be a subset query. For example in Fig. 2(b) above, query with Qid = 4 is a subset query of query with Qid = 2. However note that, for query with Qid = 2, there exist tuples which are not in skyline of query with Qid = 4. Also note that, query with Qid = 4 is also a subset query of query with Qid = 3. In such cases, the intersection of the skylines of the queries with Qid = 2 and Qid = 3 can serve as the initial set for computation of the skyline of query with Qid = 4. Only the tuples in the initial set need to be examined.

Thus the skyline for the subsequent query which gets categorized either an exact or a subset query can be served without referring to the dataset again.

3.1.3. Partial queries

If a subsequent query happens to be a partial query, then its skyline can be computed as follows. The skyline of the query to which the current query happens to be partial, serves as the initial set. For example a query with Qid = 5 is partial to a query with Qid = 3. However it is possible that the skyline for a partial query may contain tuples that are not part of the skyline of any of the previous queries to which it is partial. (The skyline of Qid = 5 contains tuple t0, which is not part of skyline of query with Qid = 3). Note that a query may be partial to multiple previous queries. (query with Qid = 5 is also partial to queries with Qid = 1 and 2). In such cases, union of the skylines of all such previous queries to which the current query appears to be partial can serve as the initial set for carrying out the dominance tests. So to cater for the possibility mentioned above and the additional possibility that a partial query may contain a dimension or set of the dimensions, which is not contained in the dimensions of any of the previous queries to which it happens to a partial query, the dataset has to be referred. However the initial set as computed in above manner is always helpful as it serves as the first window that contains the filtering tuples for carrying out the further dominance tests.

3.1.4. Novel queries

If a subsequent query happens to be a novel query, then its skyline has never been computed earlier. Its skyline is computed in a traditional manner that is by scanning the entire dataset and using any of the previous skyline computation algorithms.

From this discussion it is clear that if the skylines are preserved for the user queries then from the reusability point of view, the preserved results serve best for the exact and the subset queries. And for the partial queries, they can assist in speeding up of the computation. Also with the continuous queries generated against the same dataset, the queries which were partial or novel, may become exact or subset queries and their skyline can be served relatively faster. So we conclude that, in the scenario where frequent and overlapping queries are fired by the users, a structure that keeps logs of various statistics of every query is needed to optimize the response time of the skyline computation. We call this structure as the 'Query Profiler'. Let us discuss this term in detail.

3.2. Structure of the Query Profiler

For each skyline query raised by the user, we intend to save various statistics (or metadata) of the query so that they can be used to either avoid or reduce the re-computational efforts of the results. Such statistics should be preserved in some structured manner for efficient retrieval of the information. We call these data structures as the Query Profiler (QP). Each Query Profiler contains the following fields.

- Query id (*QId*): is the unique identification number for each query.
- Attributes (*Att*): is the list of the dimensions involved in the user query. It is required for categorizing the queries as mentioned earlier in 3.1.
- Skyline (*S*): is the skyline of the user query.
- Subset (*Sb*): is the list of query ids, indicating to which different queries, the current query appears to be a subset.
- Partial (*Pr*): is the list of query ids, indicating to which different queries, the current query appears to be partial.
- Query Frequency (*Qf*): is a positive number indicating the frequency of the query raised.

Thus a query profiler can be represented as QP = $\{QId, Att, S, Sb, Pr, Qf\}$. Now refer the Fig. 2(b) to understand the query profiler data saved for the query with Qid = 5. It is represented as $QP_5 = \{5, \{1,4\}, \{0,1,2\}, \text{nil}, \{1,2,3\}, 1\}$ and interpreted as follows: The query id 5 queries the dimensions 1,4 and its skyline is {0,1,2}. It happens to be a subset of neither of the previous queries. (Here we assert again that the checking as per the given query categorization order, is important). It happens to partial query to queries with Qid 1, 2 and 3. Hence the union of the skyline of these three queries is used as the initial set for the computation of the skyline of this query with Qid = 5. As it occurred for the first time its frequency is set to 1. Now let us discuss how the implementation of the QP can be done.

Id	a1	a2	a3	a4
t0	1	10	20	17
t1	2	10	2	15
t2	3	7	18	6
t3	4	13	21	8
t4	5	4	16	10
t5	6	10	1	21
t6	7	11	22	7

(a) Sample Dataset

Qid	Query dimensions	Skyline	Query type
1	{a1,a2}	{t0,t2,t4}	Novel
2	{a1,a2,a3}	{t0,t1,t2,t4,t5}	Novel
3	{a2,a3,a4}	{t1,t2,t4,t5}	Novel
4	{a2,a3}	{t4,t5}	Subset
5	{a1,a4}	{t0,t1,t2}	Partial
6	{a1,a2}	{t0,t2,t4}	Exact
7	{a3,a4}	{t1,t2,t4,t5}	Subset
8	{a1,a3}	{t0,t1,t5}	Subset

(b) Queries, their skyline and their categorization

Figure 2 Sample dataset and query categorization.

3.2.1. Implementation of Query Profiler

The QP can be implemented in this way.

(i) For the first time, it is maintained in the main memory and for efficient access it is indexed by using a hashed index.
(ii) For the subsequent usage of the dataset, it can be saved on a disk. However while resolving the queries on the dataset again at run time; it gets managed in the main memory.

Whenever a user fires a query, the QP is searched for categorizing the query as either exact, subset, partial or novel query. In QP, each exact and the unique subset and partial queries get stored only once. If the same query is repeated by the user, the related *Qf* field of QP is incremented by one each time. For each novel query, a separate entry is done in QP. During the process of resolving the user queries, when the subset and partial queries are repeated, they eventually become the exact queries.

We think of the following three strategies for making the searches on QP more efficient.

(i) Managing QP with efficient indexing.
(ii) Keeping the QP size minimum.
(iii) Keeping the QP sorted on two aspects: query frequency and number of dimensions used in the queries.

These strategies have been discussed below:

The QP access is made efficient using a hashed index. For a d dimensional dataset, a d-bit vector is used. The dimensions being queried are mapped on this vector to give the location for the related QP. Those dataset dimensions involved in the query are set to 1 in the bit vector, setting remaining bits for the unused dimensions to 0. For example if the dataset under consideration is a 3 dimensional dataset and it user fires a query which involves dimensions 1 and 3 then bit vector would be 101 and $H(101) \rightarrow \text{QP}(i)$, where i is nothing but the location of the related entry in QP.

Now another aspect of having efficient searches on QP is that, its size should be kept minimum. The QP size can be managed in this way:

(i) When a sub sequent query comes, the *Att* fields of previous entries in QP get searched for categorizing the query. If query gets categorized as an exact query then the existing entry in QP is used and the related *Qf* field of the query is incremented by one. If the query gets categorized as a subset, partial or novel query then separate entry in QP is made and the related *Qf* field of the query is set to one.
(ii) For anti correlated dataset, chances are high that two different queries will have the same skyline. (Refer Fig. 2(b): Qid = 3 and Qid = 7 have different dimensions but same skylines.) In such cases the QP entry of the older query which has the same skyline as that of the new query, is reused. The *Att* field of the older entry is modified to contain attributes of the new query. The related *Qf* field is incremented by one.

The last strategy mentioned above can be used when no indexing is employed. The goal is to make the searches on QP more efficient by keeping it always sorted. We achieve this by sorting QP entries on the two fields: *Qf* and *Att*. This means that the topmost entry of QP is always for the highest frequent query. The *Qf* field which indicates the query frequency is incremented in two cases: first when a query repeats and second when two different queries have the same skylines. So if such entries with higher *Qf* are kept on the top in QP, they will be visited earlier during the searches. Also if a query Q' has a higher number of dimensions it is a better candidate to serve the other queries which may appear either as a subset or partial to Q'. So it is a good idea to keep the queries with higher dimensions on the higher side in QP.

We conclude this discussion, giving the summary of the QP management procedure below

The user input query Q is received and the bit vector of Q's dimensions is hashed to know the location *loc* of Q in the QP. Then Q is categorized. For each unique Q, separate entry is maintained in QP and if Q is not unique (repeated query) then the entry with location *loc* is referred to assist the skyline computation of Q. Each repeat occurrence of Q results in increment of related *Qf* field by one.

3.3. The QPSkyline algorithm

The QP management strategies explained above have been taken care by the *QPSkyline* algorithm. The pseudo code for the algorithm is given below.

Algorithm 1

Input: Skyline Query Q involving d dimensions
Output: Skyline of Q.

```
1.  Begin QPSkyline
2.  Initialize Q. (QId = 0, Att ← d, S ← φ, Sb ← φ, Pr ← φ,
Qf = 0)
3.  loc = hash(Q.Att)
4.  c = categorize(Q)
5.  if (c = exact)
6.     copy_profile(Q, loc)
7.     return(Q.S)
8.  end if
9.  if (c = subset)
10.    get_other_parents_subset(Q)
11.    Q.S ←intersection(skyline_of_parents_subset(Q))
12.    Store_QP(loc)
13.    return(Q.S)
14.  end if
15.  If (c = partial)
16.    get_other_parents_partial(Q)
17.    initial_set←union(skyline_of_parents_partial(Q))
18.    Q.S ← compute_skyline(initial_set)
19.    store_QP(loc)
20.    return(Q.S)
21.  end if
22.  If (c = novel)
23.    Q.S compute_skyline_novel((Q)
24.  store_QP(loc)
25.    return(Q.S)
26.  end if
27.  match_skylines(Q)
28.  sort(QP)
29.  end QPSKyline
```

The algorithm is explained below:

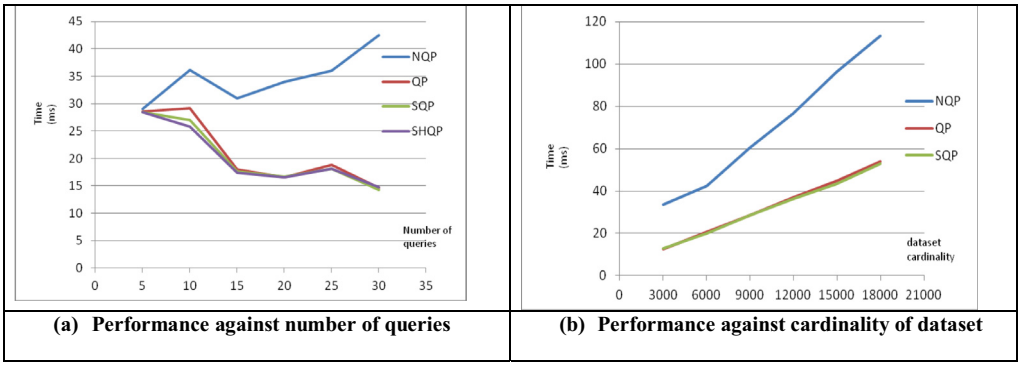
When the user query arrives, the algorithm first uses the hash search to see if it pre exists. The ***hash*** function accepts the dimensions of the user query and gives the location *loc* if the query gets searched in QP otherwise generates a new location for storing the QP of the user query. The query categorization is done by the ***categorize*** function. The exact queries get served by QP at location *loc* and skyline can be immediately returned. The ***copy_profile*** function increases the Qf by one for the exact query. For serving the subset query the various queries to which the query occurs as the subset query is found by the ***get_other_parents_subset*** function, the related $Q.Sb$ is modified and the intersection of their related skylines is computed by the ***intersection*** function and the computed skyline in this way is retuned. The QP for the user query is recorded by ***store_QP*** function. For the partial queries, function ***get_other_parents_partial*** finds the various queries, to which the user query appears as the partial query, updates the $Q.Pr$ field and the ***union*** function computes the union of skylines of the queries returned by the function ***get_other_parents_partial*** as the initial set. With the help of this initial set and the dataset the skyline of the user query needs to be computed which is done by ***compute_skyline*** function. This function uses the BNL (Borzsonyi et al., 2001) algorithm to compute the skylines. Again the QP for the user query is recorded by ***store_QP*** function. The user query which gets categorized as novel query gets served by the function the ***compute_skyline_novel*** which computes the skyline of the user query by scanning the whole dataset and by using the traditional BNL algorithm. The skyline computation for the novel query is not assisted by the initial set as explained earlier. To help reduce the size of QP the ***match_skyline*** function works. This function aims at matching of the previous skylines with that of skyline of the user's query and on positive findings updates the $Q.Att$ (as explained in Section 3.2.1) and increments the $Q.Qf$ field by one for the queries where skylines matched. Again to help reduce the searching time on QP, the QP should be managed in sorted order on query frequencies, keeping the popular queries on the top. The ***sort*** function at the end sorts the QP on two attributes: query frequencies and number of query attributes used in a query.

Thus the algorithm *QPSkyline* computes the skylines for frequent and near to frequent skyline queries. Through this algorithm, we have focused on reusing the earlier results which are efficiently managed by QP and have aimed at keeping the QP size to the minimum for allowing the efficient searches. The experimental results assert the effectiveness of our algorithm.

3.4. Experimental results: the QPSkyline algorithm

For our work, we have used a real life dataset of *players* available at www.basketball.com. The experiments have been carried out on a machine with peculiarities as: Intel Core i-3 2100 CPU, 3.10 GHz, 2 GB RAM having window 7 environment. The experiments involved performance comparison of skyline computation methods where the skylines were computed using four different techniques: (1) *NQP*: the traditional method which used BNL, (2) *QP*: the method which used concept of QP (without sorting and hashing), (3) *SQP*: the method which used ***QPSkyline*** with sorting, without enabling hashing and (4) *SHQP*: the method which used ***QPSkyline*** (with sorting and hashing enabled). The experiments evaluate the response times against the variance of cardinality of dataset, dimensionality of dataset and number of user queries. The results have been discussed below.

For the first experiment, we have set $n = 3925$ and $d = 5$. As shown in Fig. 3(a), when the number of queries is increased, the *QPSkyline* algorithm outperforms the traditional method of skyline computing. This is obvious because as the number of user queries grow, the queries eventually get categorized either as exact or a subset or partial and skylines of the previously computed queries serve for the subsequent computations and the response time for subsequent queries improves. For the second experiment, we have a set $q = 25$ and $d = 5$. As the dataset cardinality grows, the number of dominance tests to be carried out by the traditional method, obviously grows. In the same scenario, the *QPSkyline* algorithm either cuts off these tests (for exact queries) or reduced them greatly (for subset and partial queries) and results with optimized response time. (As the concept of hashing is applied for getting the proper location for storage of QP of the user query, the related experimentation is relevant only in scenario (a) hence the *SHQP* comparison has been skipped in experiment (b).)



(a) Performance against number of queries **(b) Performance against cardinality of dataset**

Figure 3 Effect of (a) number of queries, (b) cardinality of dataset.

We may also carry out the same experiments for larger dimensions, larger cardinalities and a large number of user queries. However it is obvious that the re-usability factor will grow and the same has been exploited by the *QPSkyline* algorithm Hence it is bound to give better results in those scenarios too.

4. The use of Query Profiler in update intensive environments

In this section we extend the concept of the *QP* to make it work in update intensive environment. In practice the database undergoes the update operations like addition of new tuples, deletion of tuples and update of the tuple values. If a dataset gets updated then the older skyline for a repeated skyline query may no longer remain valid. Let us elaborate this point in detail. Recall the hotel dataset discussed in Section 1 for which the skyline has been computed as {$h1$, $h3$}. We have covered the major three cases of the update operations.

Case 1: addition of a tuple: suppose a new tuple is added as $\langle h0, 1000, 2\rangle$. In this case the older skyline will become invalid and the new skyline will be {h0}. However the addition of the tuple $\langle h0, 8000, 9\rangle$ will not affect the skyline.

Case 2: deletion of a tuple: suppose the first tuple $\langle h1, 2000, 5\rangle$ is deleted. In this case, since the tuple was part of the older skyline, the new skyline will be {h3}. However the deletion of the tuple $\langle h5, 4000, 7\rangle$ will not affect the skyline. It is important to note here that, the deletion of a tuple which contributed to the skyline can make other existing tuples in the dataset to contribute to the newer skyline.

Case 3: update of a tuple value: suppose the first tuple $\langle h1, 2000, 5\rangle$ gets updated and becomes $\langle h1, 1000, 2\rangle$. In this case, the new skyline will be {h0}. However if the tuple $\langle h6, 3500, 3\rangle$ is updated to become like $\langle h6, 8000, 3\rangle$, this update will not affect the skyline. It is important to note here that, the update of a tuple which contributed to the skyline can make other existing tuples in the dataset to contribute to the newer skyline.

From the above examples it is clear that re computation of the skyline becomes necessary whenever an update operation on the dataset affects the existing skyline of the query Q. In all cases discussed, the skyline results saved in the related *Query Profiler* seem to be of no use. However if we modify the structure of QP, then we can still reduce the re computation efforts in some of the update cases. We now add the following field to the QP.

- Min[n] (*Min-n*): an *n*-dimensional list which stores the minimum of all values for each of the n- dimensions of the dataset.

This field gets its values when the whole dataset is scanned for computing the skyline for the first novel query and it is referred for computing the new skyline whenever the dataset undergoes any of the update operations enlisted above. In each of the updates, the field get updated if the newer minimum value is generated for any dimension after the update. It proves helpful in the decision of whether the re computation of the skyline should be done or not. In the cases where there computation efforts are saved, the skyline is immediately returned to the user and the tuple under update gets recorded for carrying the 'lazy update' later on. This is optimization of the response time. The next discussion covers the point in details.

4.1. Handling the updates

To help understand the skyline computations upon updates, we present the following lemmas. Recall our assumption that all the skyline queries are raised against a single relation R and without loss of generality we have assumed that the skyline criteria are finding the minimum of the concerned dimensions. In the next discussion whenever we refer to the values of the dimensions, we have assumed that those dimensions are involved in the skyline query raised against the relation. Now we present guidelines about handling of the updates and the related skyline computation with the help of following lemmas.

Lemma 1. *When a tuple* $t\ \langle v_1, v_2, \ldots v_n\rangle$ *to be added to a dataset holds* $\forall\ i \subseteq n$: $v_i > min_i$, *then the addition of the tuple does not affect the skyline S of the query Q raised against an n-dimensional dataset.*

Proof. This means that the addition of the tuple does not affect the skyline if all values of the tuple happen to be greater than their related minimum values. In the proof we present two points:

(1) Suppose that a tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ is added to a n-dimensional dataset. If the values are of the kind $v_1 > \min_1, v_2 > \min_2, \ldots v_n > \min_n$ then they will not

contribute to S as the minimum values $v_{min1}, v_{min2}, \ldots v_{minn}$ of all the contributing dimensions to Q are already in S.

(2) Even when a single of the values of $t\ \langle v_1, v_2, \ldots v_n\rangle$ is either equal or lesser than the related minimum value of the dimension, then the tuple needs to be examined for its eligibility to contribute to the newer skyline. In the later case ($v_i < \min_i$) the tuple will be obviously become part of S as it holds for some dimension i, the minimum of the values. In the case when $v_i = \min_i$ the tuple may contribute to the skyline depending upon the other values v_j, where $j \neq i$ and $j \subseteq n$.

In the first case discussed above, the re computation efforts of the skyline are completely saved. In the cases where $v_i < \min_i$, the field min[i] of the related dimension i of the dataset is updated to reflect the newer minimum value.

To understand this better re consider the sample dataset of hotels where the $\min_1 = 2000$ and $\min_2 = 3$. In the cases where the added tuples have values greater than $\langle 2000, 3\rangle$, the re computation efforts of the skyline are completely saved. The $\min_1$ and $\min_2$ values saved in the QP for the related skyline query help in avoiding the re computation as the tuple having the values greater than the minimum values for each of the dimensions, cannot become part of the skyline. However in the cases where any added tuple has values which are either equal to or lesser than $\langle 2000, 3\rangle$, the older skyline may get affected as explained in point 2 above and the re computation can be invoked. □

Lemma 2. *When a tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ gets deleted from the dataset, the skyline S of the query Q raised against an n-dimensional dataset is not affected if t holds $\forall\ i \subseteq n$: $v_i \neq min_i$.*

Proof. When any tuple t gets deleted from the dataset two cases are possible. In the first case, the tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ holds $\forall\ i \subseteq n$: $v_i > \min_i$. And hence it cannot be part of the skyline. So its deletion does not affect the skyline. In the second case, $\exists\ i \subseteq n$: $v_i = min_i$. In such cases, the tuple being deleted has to be part of the skyline and as discussed above its deletion can make other existing tuples in the dataset to contribute to the skyline and re computation of the skyline has to be invoked in such deletions.

In the first case discussed above, the re computation efforts of the skyline are completely saved. We have not discussed above the case where the tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ holds $\forall\ i \subseteq n$: $v_i < \min_i$ as it is not possible. □

Lemma 3. *When a tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ gets updated on the values, the skyline S of the query Q raised against an n-dimensional dataset is not affected if t holds $\forall\ i \subseteq n$: $v_i > min_i$.*

Proof. When any tuple t gets updated on its value(s) three cases are possible. In the first case, the tuple $t\ \langle v_1, v_2, \ldots v_n\rangle$ holds $\forall\ i \subseteq n$: $v_i > \min_i$. And hence it cannot be part of the skyline and its update does not affect the skyline. In the second case (extreme case), $\forall\ i \subseteq n$: $v_i < \min_i$. This is the simplest case from skyline computation point of view as this tuple will then be the only tuple in he skyline as it holds minimum values of all the concerned dimensions. In the third case, the tuple t holds $\exists\ i \subseteq n$: $v_i = \min_i$. Such a tuple may contribute to the skyline depending upon the other values v_j, where $j \neq i$ and $j \subseteq n$.

In the first two cases discussed above, the re computation efforts of the skyline are completely saved. In the second and third situations, the related min[i] field gets updated to reflect the newer minimum values.

In the next discussion, we present the *QPUpdateSkyline* algorithm, which makes use of the modified *QP* to optimize the response time of frequent skyline queries in update intensive environment. □

4.2. The QPUpdateSkyline algorithm

The extended QP and related usage strategies explained above have been taken care by the *QPUpdateSkyline* algorithm. The pseudo code for the algorithm is given below.

Algorithm 2

```
Input Q: set of skyline Queries, QP: Query Profiler, F: set of
update operations
Output: S: Skyline of Q.
1. Begin QPUpdateSkyIine
2. oper ← read_operation_to_be_peformed(F)
3. t ← read_tuple_to_be_operated(F)
4. if (oper = 'insert')
5.   oper_status = check_ins_tuple(t,QP)
6.   if (oper_status = will_affect_sky)
7.     insert(t)
8.     S ← compute_sky(Q)
9.   else
10.    record(t)
11.    S ← return_old_sky(Q)
12.  end if
13. end if
14. if (oper = "delete')
15.   oper_status = check_del_tuple(t, ,QP)
16.   if (oper_status = will_affect_sky)
17.     delete(t)
18.     S ← compute_sky(Q)
19.   else
20.     record(t)
21.     S ← return_old_sky()
22.   end if
23. end if
24. if (oper = 'update')
25.   oper_status = check_new_tuple (t,QP)
26.   if (oper_status = will_affect_sky)
27.     update(t)
28.     S ← compute_sky(Q)
29.     else
30.     record(t)
31.     S ← return_old_sky(Q)
32.   end if
33. end if
34. lazy_update_recodered_operation(t)
35. End QPUpdateSkyIine
```

The algorithm is explained below:

The algorithm takes three things as input. The frequent skyline queries, their related *QP*s (as first novel queries get processed, *QP*s are built as explained earlier) and set of update operations on the dataset. The tuple to be added or deleted or updated is read from the list of operations and depending on the case either of the methods *check_ins_tuple( ), check_del_tuple( )* or *check_new_tuple( )* is invoked respectively. Each of these methods performs the check on tuple values under operation and the related QP's min[*n*] field. Whenever the tuples values are found not to affect the older skyline, the skyline is immediately returned to the user from *QP*. This saves the re computation efforts and the tuple is recorded for the lazy propagation of the update later on. When the methods find either of the updates can affect the skyline (we do not repeat the explanation of the situations when this will happen as it is already covered in Section 4.1 above.), the update operation takes place first. This is because such an update can cause other existing tuples in the dataset to contribute to the newer skyline. In the occurrence of such a case, the min[*i*] field of the related *QP* is updated to reflect the newer min[*i*] values.

The next section discusses the experiments carried out the obtained results

4.3. Experimental results: the QPUpdateSkyline algorithm

For these experiments as well, we have used the experimental environment and the dataset as described in 3.4. The experiments involved performance comparison of the *NQP* and *QP* methods. The *NQP* method computes the skyline of the queries in the traditional way (we have used the BNL algorithm) after each of the updates done on the dataset. The QP method uses the *QPUpdateSkyline* algorithm and the modified version of the QP structure. The update operations that were carried on the dataset included all the three update operations (insert, delete and update of a tuple) intermixed carefully to cover them all. The percentage of the updates was varied from 2% to 10% of the total dataset size. The following figure describes the obtained results.

Fig. 4 is a right indicative of the applicability of the proposed concept of the *Query Profiler* in the skyline computation of the popular (frequently queried) and update intensive datasets. The figure shows that the use of *QP* method optimizes the response time of the queries in the update intensive environment in a better manner. This is because of the preserved metadata in the form of the *QP* structure. Its effective use helps in saving the re computation efforts of the skylines in case of those update operations which do not affect the previous skylines, The *NQP* method does not have the benefits of the assistance of the *QP* and computes the skyline for each query after each of the updates. It does not consider whether the update operation will affect the existing skyline or not and hence exhibits the comparatively higher response time.

It is obvious that the cost of the skyline computation for the dataset increases with the dimensionality of the dataset and the size of the dataset. However irrespective of the dimensionality and the size of the dataset, as long as the correct skyline can be returned to the user even when the dataset undergoes the updates, it will always optimize the response time of the skyline computation as compared to the re-computation of the skyline after the update operations on the dataset. We have done exactly this by means of the proposed concept of *QP*. Hence our experimentation involved a single parameter viz. variance of the percentage of the updates.

The next section discusses the conclusions

5. Conclusion and future scope

In this paper, we proposed the concept of the '*Query Profiler'*, (*QP*) the structure which preserves the metadata of the skyline queries. We then proposed the *QPSkyline* algorithm which aims at optimizing the response time of the frequent or near to frequent skyline queries by making use of *QP*. Then we modified the structure of *QP* to make it useful in situations where the datasets are frequently updated. The *QPUpdateSkyline* algorithm makes use of the modified *QP* to cut the re processing time of those queries where update operations do not affect the existing skylines preserved in *QP*. The experiments have been performed on the real life dataset. They demonstrate the efficiency and scalability of the proposed algorithms.

In future we aim at extending this work aiming at more efficient management of *QP* by exploiting the benefits of the cache memory. Also, in the current research work we have considered that the updates on the dataset take place sequentially, one after another. However in practice these updates may take place in parallel fashion. Hence we also aim at application of the proposed algorithms in parallel computing environments.

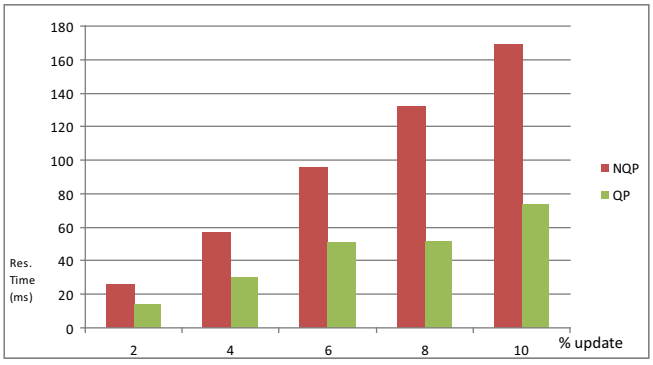


Figure 4 Performance comparison of the NQP and QP methods.

References

Bartolini, I., Ciaccia, P., Patella, M., 2006. SaLSa: computing the skyline without scanning the whole sky. In: Proc. ACM Int'l Conf. on Information and Knowledge Management, pp. 405–411.

Bhattacharya, A., Teja, P., Dutta, S., 2011. Caching stars in the sky: a semantic caching approach to accelerate skyline queries. In: Proc. Int'l Conf. on Database and Expert systems Applications, pp. 493–501.

Borzsonyi, S., Kossmann, D., Stocker, K., 2001. The skyline operator. In: Proc. IEEE Int'l Conf. on Data Engineering, pp. 421–430

Chomicki, J., Godfrey, P., Gryz, J., Liang, D., 2003. Skyline with presorting. In: Proc. IEEE Int'l Conf. on Data Engineering, pp. 717–719.

Dar, S., Franklin, M., Jonsson, B., Srivastava, D., Tan, M., 1996. Semantic data caching and replacement. In: Proc. IEEE Int'l Conf. on Very Large Databases, pp. 330–341.

Godfrey, P., Shipley, R., Gryz, J., 2005. Maximal vector computation in large data sets. In: Proc. IEEE Int'l Conf. on Very Large Databases, pp. 229–240.

Han, X., Li, J., Yang, D., Wang, J., 2013. Efficient skyline computation on big data. IEEE Trans. Knowl. Data Eng. 25 (11), 2521–2535.

Kossmann, D., Ramsak, F., Rost, S., 2002. Shooting stars in the sky: an online algorithm for skyline queries. In: Proc. IEEE Int'l Conf. on Very Large Databases, pp. 275–286.

Lee, K., Zheng, B., Li, H., Lee, W., 2007. Approaching the skyline in Z order. In: Proc. IEEE Int'l Conf. on Very Large Databases, pp. 279–290.

Papadias, D., Tao, Y., Fu, G., Seeger, B., 2005. Progressive skyline computation in database systems. ACM Trans. Database Syst. 30 (1), 41–82.

Papapetrou, O., Garofalakis, M., 2014. Continuous fragmented skylines over distributed streams. In: Proc. IEEE Int'l Conf. on Data Engineering, pp. 124–135.

Ren, Q., Kumar, V., 2001. Semantic caching and query processing. IEEE Trans. Knowl. Data Eng. 15 (1), 192–210.

Ruan, P., Xu, C., Huang, J., Qing, L., Ji, C., 2013. A distributed algorithm for skyline query based on pre-clustering. Period. Adv. Mater. Res. 756–759, 3982–3986.

Sheng, C., Tao, Y., 2012. Worst-case I/O-efficient skyline algorithms. ACM Trans. Database Syst. 37 (4) 26.

Siddique, M., Morimoto, Y., 2010. Efficient maintenance of all k-dominant skyline query results for frequently updated database. In: Proc. IEEE Int'l Conf. on Advances in Databases, Knowledge and Data Applications, pp. 107–110.

Siddique, M., Zaman, A., Islam, M., Morimoto, Y., 2012. Multicore based spatial k-dominant skyline computation. In: Proc. IEEE Int'l Conf. on Networking and Computing, pp. 188–194.

Tao, Y., Xiao, X., Pei, J., 2006. SUBSKY: efficient computation of skylines in subspaces. In: IEEE Int'l Conf. on Data Engineering, pp. 65–74.

Xia, T., Zhang, D., 2005. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In: Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp.493–501.

Xia, T., Zhang, D., Fang, Z., Chen, C., Wang, J., 2012. Online subspace skyline query processing using the compressed skycube. ACM Trans. Database Syst. 37 (2) 15.

Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J.X., Zhang, Q., 2005. Efficient computation of the skyline cube. In: Proc. IEEE Int'l Conf. on Very Large Databases, pp. 241–252.

Zhang, N., Li, C., Hassan, N., Rajasekaran, S., Das, G., 2014. On skyline groups. IEEE Trans. Knowl. Data Eng. 26 (4), 942–956.

Zheng, W., Zou, L., Lian, X., Hong, L., Zhao, D., 2014. Efficient subgraph skyline search over large graphs. In: ACM Int'l Conf. on Conference on Information and Knowledge Management, pp. 1529–1538.