QATAR UNIVERSITY

COLLEGE OF ENGINEERING

EFFICIENT SKYLINE SYSTEM DEVELOPMENT FOR NORMAL AND HIDDEN DATABASES: APPLICATION FOR GOOGLE FLIGHTS

BY

GEORGES J. ADAM

A Project Submitted to

the Faculty of the College of

Engineering

in Partial Fulfillment

of the Requirements

for the Degree of

Masters of Science in Computing

January 2018

COMMITTEE PAGE

The members of the Committee approve the Project of Georges J. Adam

defended on 21/12/2017.

Dr. Ali Mohamed Jaoua
Thesis/Dissertation Supervisor

Dr. Abdelkarim Erradi
Committee Member

Dr. Aiman Erbad
Committee Member

Dr. Khaled Md Khan
Committee Chair

ABSTRACT

ADAM, GEORGES, JOSEPH., Masters : January: 2018, Masters of Science in Computing

Title: Efficient Skyline System Development for Normal and Hidden Databases: Application for Google Flights

Supervisor of Project: Ali, Mohamed ,Jaoua.

Deep web databases provide strict search interface and limited web access with top-k results based on a pre-defined ranking function. However, top-k results may not be suitable for multi-criteria decision making because of the variety in preferences. To make the results more relevant to such a decision maker, skyline records were introduced, and as per definition these records are not dominated by any other record such that a record dominates another if it is better or as good as other for all attributes and better in at least one attribute.

In this report, we introduce an algorithm for discovering skyline records from hidden databases using different multi-objective attributes on a real-world database. We predicted a new lower bound for the minimum issued number of queries to extract the skyline. This was supported by our algorithm which accomplished the above task in an efficient manner including the worst-case scenario hence proving our theory via running rigorous experiments on a hidden database given the limitations on hand.

ACKNOWLEDGMENTS

To start, I would like to thank Allah, for this opportunity and gifting upon me the ability to finish this thesis and with it, complete my Master's Degree.

This project would not have been made possible, if not for the help of a few people and their significant contributions to my work.

First, I would like to express my deepest gratitude to Professor Ali Jaoua, my research supervisor, for his continued guidance, support, encouragement, and patience, throughout this process.

Second, I am eternally grateful to my family, for their unconditional love, support, encouragement, and patience throughout this exciting and challenging period in my life.

I would also like to take the opportunity to thank my friends, who supported me throughout this research project.

This contribution was made possible by NPRP grant #07- 794-1-145 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Real world databases are often queried by users interested in knowing the data and may be using it to find or reach their goal. It is also known that some users are not looking to search through the search results to find their answers. Querying such accurate query is almost difficult to the search engine to understand what the user truly wants. Such engines require pre-defined ranking functions in order to rank results in a way that the most relevant results to the user are at first followed by the rest of results. Skyline queries were proposed to give the user final and fast decision by allowing him to place his personal preference on his desired objectives. Such a system can provide more accurate results defined as either skylines or the best records in a database.

1.1 Problem Statement

Skyline discovery or extraction has never been a problem when dealing with normal databases that you can query at any time or have complete view over the database attributes and what values they contain. For example, one can simply use the “SKYLINE OF” clause in any SQL statement [1] and directly process the extraction of skyline objects. In contrast, hidden databases such as Amazon, Google Flight and Twitter, place sever limits on how the user can interact with it. Such databases may restrict the number of web accesses and other may minimize the number of returned results.

Here is a quick overview table on the main differences between normal and hidden database functions:

Table 1

Main Differences Between Normal and Hidden Database

Traditional Databases	**Hidden Databases**
Expose a ranked list of all tuples to a pre-known ranking function	Known ranking function and unknown to users.
Full SQL Power	Limited number of (Top-k) results
Easy to crawl	Limited number of queries per IP
Easy to compute skylines using traditional techniques(DC,NN,NBL)	Hard to compute skylines

As we can see from Table 1, the calculation of skylines could be difficult due to the limitations placed by providers on their databases. This is because they only allow user interaction through straightforward queries from search interfaces or by the provided Application-programming interface (API). In this project, we investigate and research this problem by answering the following research questions:

- How can we extract skyline records from hidden databases using the least number of sent queries?

Our main research problem in this investigation is to find a way to crawl or query as little as possible in the hidden database in order to overcome its limitations and find the maximum number of skylines.

- How are we calculating skyline points?

In this work, we also seek to find skyline points in an efficient manner without skipping or leaving any skyline records for normal and hidden databases.

- What type of query optimizations such a system may benefit from to reduce the number of web accesses or sent queries?
- How do we interact with APIs to define the values of the next sent query?

Our scope of work defines an algorithm to follow in order to crawl any database starting from sending conjunctive queries based on the values of the returned results. We explain the algorithm later in the following chapters.

1.2 Aim and Objective

This research aims to find a way to calculate skyline records for both hidden and normal databases but mainly focuses on hidden databases. The goal is to reduce the number of sending queries down to an efficient level.

Our aim is to find solutions for the various following objectives:

- Investigate an online hidden database such as Google Flights.
- Extract skyline flights from different trips and time for validation with our final results.
- Prepare an experimental case to research and solve.
- Implement an algorithm or framework that utilizes the API to help reduce the number of queries sent.
- Perform experiments on different scenarios and cases.
- Aggregate experiments result and conclude.

1.3 Proposed Solution

The project should deliver a fully working system designed to query an online hidden database containing flights with airfares on hundreds of airlines to help the traveler pick his best flight. The project will rely on Google Flights API in dealing with its database for query optimization and customization with the minimum number of sent queries.

The design of such a system should be clear to follow the logic behind the algorithm and check the returned result before sending the next query. One way is to log and trace the execution of skyline detection and comparing results with true skylines to validate.

1.4 Summary of Contributions

The main project contribution is to help the traveler pick his best flight by specifying only the flight itinerary for initializing the first query.

The backend system will calculate skylines based on the preferred scenario and provide the user with skyline flights. This might improve query cost and provide faster search results by bringing more relevant results as answers to the user for his specific preference.

In most of the cases, the results returned by the hidden databases are usually processed and calculated based on specific automated scenarios by the backend system to provide answers for the user's query.

In this project report, there will be three main contributions. First, is to focus on finding skyline records. Second, is to return the number of skylines found with respect to the number of submitted queries. Finally, the system was built off the concept of finding skyline flights using user-input for the number of returned results, origin and destination airports, and date of travel.

The report will be presented as follows: In chapter 2, we define the problem of top-k queries, introduce skyline queries, show methods and algorithms to calculate the skylines, and evaluate it on a real RDBMS (relational database management system). In chapter 3, we explore hidden databases models and approaches taken for crawling deep web databases since it will be our main focus. In chapter 4, we briefly look into related work for finding a skyline on a hidden database. Chapter 5 presents and defines our system. In chapter 6, we discuss the experimental setup and test our system by running specific algorithms to extract skyline points on a real world hidden database. Finally, in chapter 7, we demonstrate and conclude system efficiency while specifying future work.

CHAPTER 2

Background and Related Work

In this section, we have a look at an overview of skyline queries and why they are important and how they provide better feedback from users when compared to top-k queries. We also get to see related work such as the skyline operator and skyline-join operator. Furthermore, we look into different algorithms and approaches used now to compute skyline objects and evaluate them on a real RDBMS.

2.1 Background

A. Overview

In relational database management systems, we process and express queries usually using normal SQL by giving the system the chance to provide the best possible results it can retrieve for us by querying the database using normal SQL queries. For example, to find the nearest restaurant names to the city center where the distance is at most 2km, we should normally issue the query in Figure 1.

```
SELECT restaurants.name
FROM restaurants
WHERE restaurants.distance <= 2
```

Figure 1 SQL Query

We specified exactly what we wanted by imposing a condition on the distance to be less than 2. However, in most cases, it is much better to let the database system give the most probable best answer by making the user help the system by providing multiple criteria. This can be very helpful because it can help the system build a ranking function in order to get the best answer for the user. Such an example could be, combining two criteria in order to produce the ranking function. The user could ask the system for the lowest or highest values and get his answer. Top-k queries can be defined as queries that return top score tuples of any given database based on a scoring function and the number of expected results k. Moreover, it is always required for the user to define the number of returned objects in the answer and the scoring function can be made by many attributes in the database. An example showing a top-k query can be seen in figure 2. Consider a database restaurant with DcityCenter is the distance to the city center and *DBeach* is the distance to the beach.

```
SELECT  restaurants.name
FROM restaurants
ORDER BY (restaurants. DcityCenter + restaurants.DBeach)
STOP AT 5
```

Figure 2 TOP-5 SQL Query

Top-K queries had many drawbacks including difficulty defining a meaningful ranking function when the features had different semantics. To avoid this, Skyline queries were proposed.

B. Introduction to Skyline

A Skyline query does not require a ranking function and the integer k. A Skyline is defined as those points which are not dominated by any other point. By definition for domination in Skylines, "*A point dominates another point if it is as good or better in all dimensions and better in at least one dimension*"[1]. The interesting thing about Skyline queries is that it doesn't matter how you choose the weights of the attributes you are looking at, since the skylines are guaranteed to discover your restaurant or hotel of choice. It is more interesting in both cases to allow the user the ability of minimizing attribute values and/or maximizing them in functions that are monotone on all attributes and these objects cannot get dominated by any other object[1].

C. Skyline Examples

The definition behind domination of tuples underlies tuples dominating other tuples is that a relation R (A1, A2, A3, A4…An) in a relational database D where A_i are the attributes which can be ranked. Assuming that the less the values of the attributes the better. A record or object t dominates another object t' if and only if:

$$\boxed{\forall j = 1, \ldots, m : t[Aj] \leq t'[Aj] \land \exists j : t[Aj] < [t'Aj]}$$

Figure 3 Definition of Skyline

Some examples on finding the skyline tuples from a table:

Table 2

Minimization of Attributes

	A1	A2	A3
t1	5	1	9
t2	4	4	8
t3	1	3	7
t4	3	2	3

In table 2, we can see a table of 4 tuples (t1, t2, t3 and t4) with 3 ranking attributes (A1, A2, A3). We prefer "Minimum" over all the attributes so the less the value the better.

We can conclude that t2 is dominated by t3 and t4. t1, t3 and t4 are not dominated by any other tuple so they are considered as skylines.

The skyline queries have been used over a period of time in making several multi-criteria decision support applications. Due to the dominance in the relationship dataset the query returns the objects that cannot be dominated by other objects.

Consider a person that cannot choose a restaurant over different restaurant because of the variety of service, price, décor and the quality of food. The user would like high service, food and décor but low price. Table 3 shows different restaurants with its corresponding values (1=bad, 30=good).

Table 3

Restaurants Databases

	Service	Food	Décor	Price
Zakopane	24	20	21	56
Yamanote	22	22	17	51.5
Summer moon	21	25	19	47.5
Fenton & Pickle	16	14	10	17.5
Brearton Grill	15	18	20	62
Briar Patch BBQ	14	13	3	22.5

Restaurants, which are dominated in table 3, are the 2 restaurants "Brearton Grill" and "Briar Patch BBQ".

Thus, several algorithms have been proposed to work on the skyline query processing deeply like the window-based, progressive, distributed, geometric based, index-based, divide and conquer, and dynamic programming. Application-specific problems like the k-dominant skylines, top-k dominating queries, spatial skyline queries and others several variations were proposed[2]. An example of skyline query can be seen in Figure 1 when the data objects are two dimensional points (2D) in the Euclidean plane while we search for the minimal dimension on each point. Figure 4 plots 16 points with coordinates:

a (1; 12), b (2; 7), c (4; 22), d (5; 14), e (6; 5), f (8; 19), g (9; 9), h (10; 4), I (12; 13), j (15; 15), k (15; 22), l (16; 6), m (17; 10) n (17; 20), o (21; 3), p (22; 14).

A.

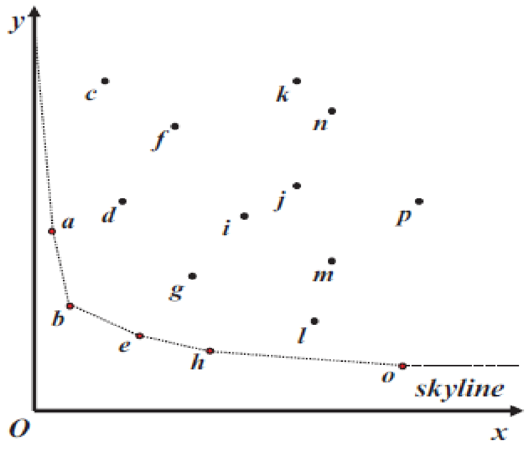


Figure 4 Skyline Example

The skyline query returned the points/objects {a; b; e; h; o} which appear on the skyline for users expecting the minimal x and y and in our example of restaurants could be the minimal distance to city center and minimal distance to the beach[2].

Consider the example of hotels but changing attributes to price, distance to the beach and the number of stars such as the user would like higher number of stars (table 3).

Table 4

Hotel Database

Hotel	#Stars	Distance	Price
Aga	2	0.7	1175
Fol	1	1.2	1237
Kaz	1	1.2	750
Neo	3	0.2	2250
Tor	3	0.5	2250
Uma	2	0.5	980

From table 4, the skyline hotels highlighted satisfy all users and are not dominated by any other hotel.

D. Skyline Operator

In investigating the optimization problem that is used for filtering database results to keep desirable results, Borzsonyi, Kossmann and Stocker [1] proposed a solution for expanding database systems. Skyline operator, accordingly, helps filter out database results from a possibly large dataset. Skyline can be represented graphically as in Figure 3, thus the reason for using it. While illustrating about the application of skyline operation in a travel agency where the customers are interested in finding hotels that are close to the beach and have a low price, the authors emphasized on decision support mechanisms. According to the authors, the study focused on how skyline operation is extensively used for database visualization in the form of graphical representations. Other points cannot dominate the points of interest. In the investigation, the authors demonstrated that there are ways to extend SQL to present skyline queries, assess other algorithms for skyline operator implementation and illustrate the way of integrating skyline to operate with other existing database operations such as top N and join. In this context, the argument that the authors talked about is that it is possible to get a nested query after transforming the skyline query. The study [1] also illustrates the implementation and demonstration of how skyline queries are important on relational database systems. During the implementation process, there are no modifications on the existing database system, whether object oriented or relational. In the proposed method, skyline queries can be implemented to extend the current database systems (object-relational, object oriented or relational) using the logical operator or clause known as skyline operator.

In the experiment carried out, the authors used uncorrelated and correlated databases to assess algorithm performance. In their findings, they emphasized the significance of skyline in making the final decision. In examining skyline operator implementations, the authors focused on distinct known approaches. The main approaches were divide and conquer, two-dimensional skylines and block-nested loop algorithm. To extend the derivations arrived at and theoretical analysis of divide and conquer algorithm previously done, [1] they extended the same algorithm so that it worked better within the context of the database. In addition, they proposed that the other alternative algorithm, block nested loops, was better than previously examined algorithms.

Borzsonyi, Kossmann and Stocker study [1] extends other previous works focusing on skyline operator. While previous research had suggested that points can collectively fit into memory, this study illustrated how to integrate skyline operator into the existing database system. To facilitate skyline queries specification, they described the potential SQL extensions. To show that the initially proposed algorithm, in terms of database performance, had poor performance, the authors presented and analyzed alternative algorithms for skyline operator computations.

To analyze skyline queries extensively, the study discussed the exploitation of standard index structures like R-trees and B-trees but to calculate the nearest neighbors (NN) to the skyline, R-trees were used. The main reason of computing the nearest neighbors is to determine the appropriate rows and prune the branches the rows that dominate. B-trees, in contrast, are used to determine the superset for the skyline.

Borzsonyi, Kossmann and Stocker, in general, showed several ways to extend a database system in order to calculate several important points by using the skyline operator by expanding it with more operators such as MIN MAX and DIFF.

```
SELECT *
FROM Hotels
Skyline OF Price min, Distance min
```

Figure 5 SQL Example Using “SKYLINE OF” Operator

To extend the SELECT statement used in SQL, the authors proposed the clause, *SKYLINE OF*. They used experimentation data to assess other algorithms used in computing the skyline, discussed the use of indices in supporting skyline operation as well as interaction between skyline operator and other existing database operators. The result from the experiment performed led to the conclusion that divide and conquer algorithm is useful for tough case implementation while block nested loops algorithm is applicable to good cases.

```
SELECT *
FROM Hotels h
WHERE h.city = ‘Hawaii' AND NOT EXISTS(SELECT *
FROM Hotels h1
WHERE h1.city = ‘Hawaii'
AND h1.distance ≤h.distance
AND h1.price ≤h.price
AND (h1.distance＜h.distance OR h1.price＜h.price));
```

Figure 6 SQL Example Without The “SKYLINE OF” Operator

2.2 Skyline Computation in Traditional Database Systems

Similar to study [1], Kossmann, Ramsak and Rost [3] , in analyzing skyline queries algorithms, argued that skyline queries, with a suitable online algorithm, helped select the best points. The suggested algorithm is suitable for interactive environments. They proposed, in their investigation, an online algorithm which calculates the skyline. Dissimilar to other algorithms which calculate the skyline using batch processing, the algorithm suggested returning immediately the first results, continuously providing more results and helping users provide user preferences during runtime. As such, the user can control the type of the results yielded by allowing him to choose cheaper hotels or better distance to the beach. In the work, the authors evaluate NN algorithm in comparison to others, majorly Bitmap algorithm and B-trees, which were found to be limited.

In the investigation, Kossmann, Ramsak and Rost found that while an online algorithm is likely to take more processing time compared to batch-based algorithms, they quickly produce skyline subsets. Batch algorithms were also found to have long running time. As such, the authors had to define appropriate assessment properties. In assessing online skyline computation, the authors suggested 6 properties that characterize the algorithm. The properties include: a) instantaneous return of the first result; b) production of a full skyline; c) not return points that are good points and later replace them with other better points; d) fair in terms of not only favoring points that are good in 1 dimension; e) proper control by using a GUI where the user can click on the screen to let the algorithm bring back the next skyline points that are near the skyline he clicked on and f) universality for database data sets (indexes) and skyline queries.

The study by Kossmann, Ramsak and Rost proposed the algorithm for 2-D skyline queries. The online algorithm was also generalized to the other higher dimensional (e.g. 3 D) skyline queries. They, however, presented three assumptions in presenting the algorithm, namely: a) all values were positive real numbers; b) the data set did not have any duplicates and c) the algorithm tends to determine the interesting points close to the initial one. Under these assumptions, it is still possible to use the proposed algorithm in duplicate queries or data sets, find maximal value and all real numbers (positive as well as negative).

In assessing skyline queries, the study compares alternative algorithms and variants. Unlike the low emphasis on dimensions as investigated by [1], the research by Kossmann, Ramsak and Rost compared several algorithm variations, focusing on dimensions. They examined performance tradeoffs for four main variants, namely propagate, laisser-faire, hybrid and merge. The experimental results helped the authors confirm that hybrid and propagate variants outperform laisser-faire and merge variants. They found that laisser-faire and merge variants, overall, spend more time on duplicates, thus lowering performance. To use to do list indices, the authors found that indexing structures like B-trees were not very useful. To improve on the performance of one variant, hybrid, the authors proposed tuning as the applicable solution.

To support the findings by [3], Kossmann, Ramsak and Rost study and illustrate the significance of skyline queries in applications such as decision support, data visualization and customer information systems. The analysis, in relation to other related algorithms, demonstrated distinctive virtues. The proposed algorithm, NN, serves as the single algorithm which gave users free control to provide preferences thus directing the output. With respect to B-tree algorithm, the authors revealed that the algorithm, in one dimension, points well but lately returns other good points in multiple dimensions. The study, in addition, concluded that the bitmap algorithm usually scans the whole database system and used bitmaps to determine the points which form part of the proposed skyline but doesn't allow the user to interact with the system because he has no control where the clustering of the database is responsible for returning the skyline objects returned first.

In similar research done by Papadias et al [4], they studied and proposed an advanced algorithm suitable for skyline queries, similar to [1] and [3] studies. In the study, the authors described a skyline as D-dimension points which have some points not controlled by other points. The investigation focused on skyline computation with respect to databases, in particular for progressive algorithms which return the first points of the skyline quickly since it does not need to read all data file contents. In the assessment, the authors acknowledged nearest neighbor as the most efficient algorithm using R-trees and being implemented using divide and conquer model as illustrated in the study by [3].

Despite desirable features exhibited by NN algorithm (applicable to random dimensions or data and high speed in returning the points), there is need to avoid duplication, large space overhead and multiple access problems. So, they devised a branch-and-bound skyline (BBS) algorithm that operates taking into account nearest neighbors with optimum use of input and output. The algorithm proposed carries out one access to the R-tree nodes likely to contain the required skyline points. Moreover, the suggested algorithm has smaller space overhead and does not in any way retrieve duplicates compared to NN algorithm.

The authors concluded the suggested algorithm, BBS, can be implemented easily and applied effectively to several skyline queries. Using experimental and analytical comparisons, the study eventually demonstrated that in all problem cases, taking into account order of magnitude, BBS outperforms the NN algorithm proposed in [3]. Overall, BBS algorithm used numerous data partitioning techniques. For simplicity, the authors only used R-trees.

In the study, Papadias et al illustrated the significance of skyline operator in multiple criteria decisions. The research done by [1] , in this context, suggested an enhanced SQL syntax while using the skyline operator. For a query regarding price and distance of the beach for several hotels in Figure 4, the syntax is given, Select*, From Hotels, Skyline of Price min, Distance min. In the presented syntax, min suggests minimization of distance and price attributes. In addition, it shows distinct conditions including group by and joins. In the investigation, these authors assumed that the computation of the skyline is anchored on min conditions for all dimensions considered. Nonetheless, all approaches analyzed in the investigation are applicable in any combination of conditions.

To build on the existing work, Papadias et al discussed several approaches such as bitmap technique, index approach, Block Nested Loop (BNL) and divide and conquer method. For BNL, the user selects one point, say p, and compare others to this point to calculate the skyline, and argument supported by [1]. Data in the data file is scanned completely to compute the skyline in this method. For divide and conquer, the authors showed that a dataset is divided into partitions and the main memory algorithm used to calculate the skyline partially. In the approach, there is a need to read the whole data set, thus not pertinent to online processing. The bitmap method encodes information essential for deciding on the skyline point using bitmaps. The approach, according to [1], does not allow for user preferences, a significant feature of most online algorithms, making it limited. In the index approach, dimensional points are usually organized into lists, later assigned to show the skyline. The approach was found limited since it does not support user preferences and returns fixed skyline points.

A. Algorithms to Extract Skylines

There exist many proposed algorithms for the processing of skyline queries and can be described in two categories: Index based and non-index based algorithms. This comes after the datasets are getting bigger and require having them on disks.

For two-dimensional, skyline is computed by sorting data of the first attribute (from better to worse) and then iterating through the tuples and by eliminating the dominated ones. Whatever is left from the tuples, they are considered skylines because they are not dominated by any other points.

For more than 2 dimensions, sorting does not work. It requires better algorithms for extracting the skylines. Many skyline query processing algorithms and techniques were proposed such as nested block loops (NBL), divide and conquer (DC) and nearest-neighbor based (NN) [2].

- Block-Nested-Loop (BNL)

The algorithm uses a window in main memory that has the best tuple and a temp file to write to if the window has no space. BNL compares every record with every other record of the dataset through a nested loop. The quadratic complexity O (N^2) makes the algorithm inefficient (N is the total number of objects that are contained in the datasets) [2]. The idea is to use a window, which is a memory block with limited space to hold a number of limited data objects. The objects are checked to confirm if they dominate each other. If any of the objects are dominated they are then eliminated and the dominant object is inserted into the window. Thus, is the case if they are incomparable with all the objects in the window. If the window is full, a temp disk file is used to contain the candidate objects. The BNL works effectively when the skyline is effectively small with the worst case being $O(N^2)$ when a much better I/O behavior is put in place[2].

There exists another type of BNL called the sort-filter skyline algorithm (SFS) which is defined by the "topological sort with respect to the skyline dominance partial relation". The SFS makes the query process efficient and behaves better in a relational setting.

The SFS is further proposed and extended[5] in a SaLSa algorithm (known as the Sort and Limit Skyline algorithm) where the algorithm reduces the number of dominant checks.

- Divide and Conquer (DC)

This method computes the median value and divides the space in two partitions P1 and P2. The first partition P1 contains the better tuples (e.g. objects that are less than objects in P2). We then compute the skylines of each partition S1 and S2 respectively where again P1 and P2 are again partitioned to P12, P22, P11 and P21. The portioning stops when a partition has only 1 or a few points. At last, the algorithm merges the results of S1 and S2 thereby eliminating the dominated points.

Figure 7 below shows the divide and conquer algorithm running on the graph of Figure 4.

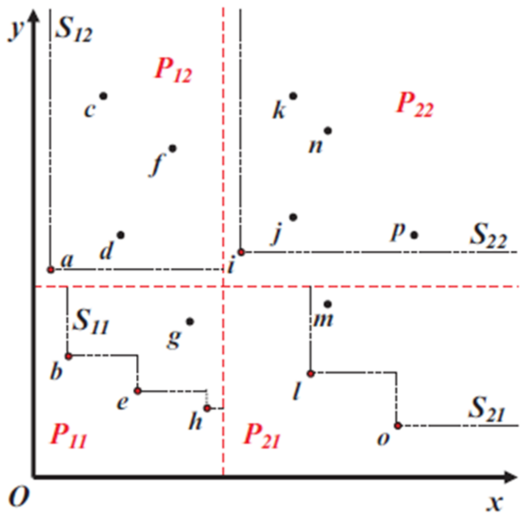


Figure 7 Divide and Conquer Algorithm Example

The authors of [2] explained how the algorithm runs and how they managed to collect all skyline points. The figure contains 4 partitions P11, P12, P21, and P22 while the partial Skylines are S11 (b, e, h), S12 (a), S21 (l, o) and S22 (i) respectively. In order to obtain all the skylines S, the dominated points are removed by some points in other partitions. All points of the skyline of P11 must appear in the final skyline and those in P22 are discarded because they are dominated by any point in P11. The skyline points in P12 are compared with points in P11 because there is not any point in P22 or P21 that can dominate those in P12. This example shows point (a) is not dominated by b, e, h, therefore it is included in the final skyline which is S. Also, the skyline P21 is also compared to the points in P11 which results in the removal of point "L" which ensures that point "O" remains. Thus the algorithm terminates the skyline set S={a, b, e, h, o}[2]. This algorithm has a variation, an optimal algorithm named (DCSkyline) which helps in computing skylines in 2-D spaces. It is similar to the BBS [4] only that it contains pruning mechanisms.

B. Skyline with Presorting and Skyline-Join

Skyline with Pre-sorting

A 2003 study by Chomicki [6] suggested a skyline algorithm, called sort filter skyline (SFS). The algorithm takes into account presorting on skyline queries, and is very efficient in the relational database environment. The skyline algorithm suggested was found to have good performance and works very well in relational database systems. Through presorting, it is argued that the block nested loop algorithm can be revised to generate SFS, a general skyline relational database algorithm. The authors illustrate how to handle preference queries through the extension of relational systems.

Among the existing skyline algorithms, it is clear that index structures are used in majority of the algorithms. The preciseness in these queries, however, makes the algorithms applicable to only specific cases. Generalizations are thus not possible. The study reveals a comparison involving three algorithms, namely SFS, divide and conquer algorithm and block nested loop algorithm. The authors argue that divide and conquer is only suitable for queries with more than five dimensions. The algorithm has some limits because it cannot work properly on large volumes of data as well as small buffer pools. For mid-range dimensions, the study showed that BNL algorithm is suitable. So, BNL also seems a good option that requires revisions in relational settings.

Among the algorithms introduced by [2], there are proposals on how to improve BNL algorithm. According to [6], tuples require presorting before BNL is applied. The presorting practice is a guarantee that the tuples not read are not considered as skyline points upon termination of algorithm processing. An assessment of several presorting functions shows that best performance can only be achieved through maximum coordinate pre-sort.

In their findings, [6] demonstrated that SFS serves as a practical algorithm that can be extensively implemented in relational database skyline queries. In addition, the study showed that several improvements are possible on the algorithm when pursuing improved skyline algorithms. The existing sort merger join algorithm; accordingly, help integrate sorting and filter skyline stages to guarantee an optimized SFS algorithm. So, performance increases because of decline in the number of passes. In the study, several advantages of SFS algorithms were revealed; well behaved in the relational database environment, offers ordering capability, is output pipelineable and allows for optimizations.

Skyline-Join

The study by Sun et al [7] examines the skyline operation in the distributed database settings. The authors address the main problem about skyline queries processing using numerous tables within the distributed environment. The proposed skyline operator, called skyline join, stems from two operators, skyline operation and join operation. The authors also suggest two methods for processing the skyline join queries. The methods considerably minimize processing time and processing costs.

Skyline query processing in the distributed environment, based on the article [6], has been gaining attention recently. Existing algorithms, in this case, are extended to minimize data transfer and support skyline queries in several database settings. In the algorithm suggested, partitioning and routing information help forward skyline queries. Tuples are categorized into three main classes; a) general skyline points, local skyline points and others. In the first two categories, joining the tuples help produce final results. In the third category, other tuples dominate others, leading to discursion of tuples. Overall, study [7] proposes more search space pruning method.

After proposing two methods, innovative iterative algorithm and existing skyline algorithm (SaLSa) [5] used two synthetic datasets to assess their performance. In the first synthetic dataset, the findings show that both algorithms cannot efficiently prune the existing search space. In addition, both introduce extra central processing unit (CPU) cost because of the running processes which are very complex. In the experiment performed, the authors found that to process the anti-correlated datasets, the modified skyline algorithm should be invoked to produce a temporal table and then the algorithm applied in determining the final results. In the comparison, key elements that reveal algorithm disparities include effect of dimensionality join selectivity impacts, iterative algorithm effect and network costs.

In the second synthetic dataset, TPC-DS datasets were generated (decision support standard measure for benchmark). Table sizes are modified because it is impossible to tune the TPC-DS tables. The experiments, taking into account two algorithms, were evaluated in terms of iterative algorithm effectiveness, dimensionality and network costs. The paper [7] revealed similar results to those found using Synthetic dataset 1. The study demonstrated that iterative algorithm calculates the (local) skyline points for each temporary table and generate partial results leading to the pruning of the resultant skyline join. AP-domination as well as outsiders eventually prunes the remaining tuples. Bloom filter accuracy, data correlation, skyline attributes and data distribution determine the algorithm performance.

C. Evaluation of Skyline Algorithms in PostgreSQL RDBMS

Eder and Wei [8] study assessed the performance of skyline algorithms within the PostgreSQL settings. The algorithms included Linear Elimination Sort for Skyline (LESS), Divide and Conquer (DC), SFS, Branch and Bound (BBS), Nearest Neighbor, index, bitmap and BNL. The algorithms are categorized into two groups to assess efficient computations. The first class comprise of solutions provided without dataset processing. The second class includes algorithms that employ sorted lists, R-trees and index structures to minimize query costs. The number of skyline parameters, in this case, determines skyline queries performance.

PostgreSQL setting is useful in skyline algorithm assessments. In the relational database management systems (RDMS), there are several advantages linked to skyline query operation. To construct additional complex queries, it is easier to integrate skyline with other existing relational operators. Similarly, index-based skyline algorithms can be implemented using current indexing structures. In addition, the RDBMS system can choose the most efficient algorithm in the given dataset while comparing several algorithms.

In the previous skyline operation comparisons [6] and [7], [10] argue that an algorithm that is best taking into account all evaluation aspects has not been found. Overall, the common metrics in comparisons subsume cardinality, dimensionality, network costs and distribution [7]. These properties, accordingly, give rise to algorithm efficiency. Considering all data distributions, SFS+EF algorithm was found to have best time performance. Compared to BNL, SFS+EF algorithm performs better. LESS was also found to outperform SFS algorithm. Considering time dimensionality performance, SFS performs better than the remaining algorithms, specifically for high dimension values.

Using skyline implementation experiments, the study revealed simple and useful rules that cannot be found in the existing theoretical explanations. Two main findings (not in theoretical explanations) were found: a) for the dataset with about 500 tuples and comparatively small dimension, the performance of BNL in all properties is the best and b) with selectivity factor less than or equal to 0.1, there is an effective elimination filter. The study findings are significant because they examine skyline query optimization in database environment and provide insights into skyline query attributes.

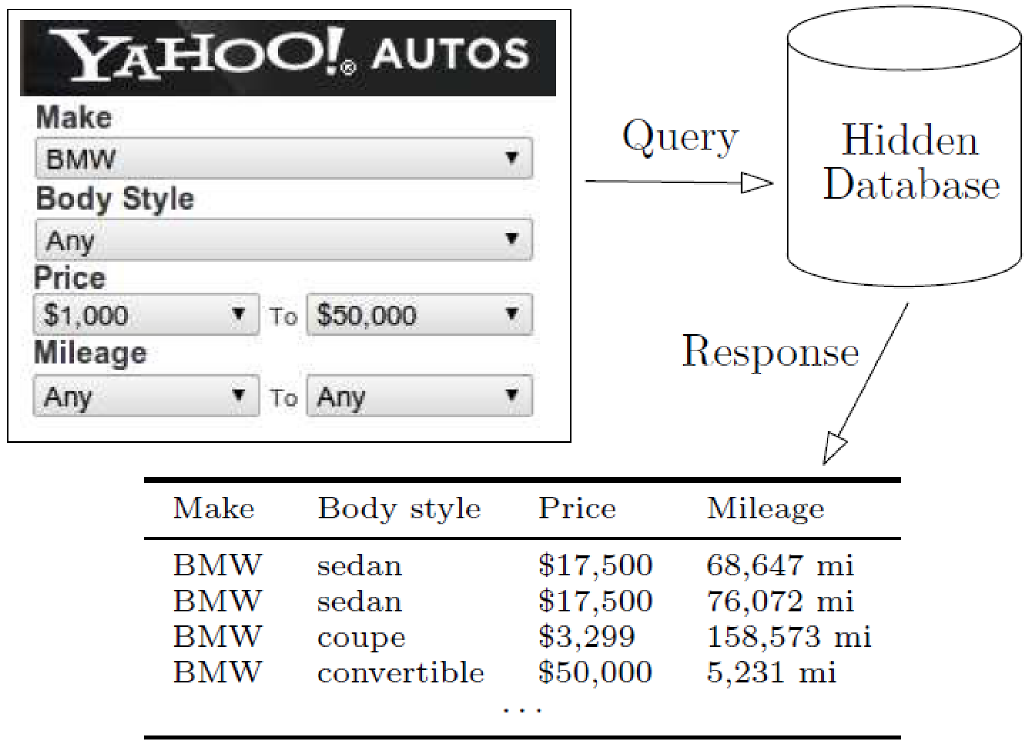
CHAPTER 3

Hidden Web Databases Crawling

Deep web databases, also known as hidden databases, cannot be crawled directly because of the strict interfaces given by the provider to the user. In this chapter, we look at different approaches by different authors and see how they interacted with such obstacles.

3.1 Approaches for crawling a hidden database

In the related literature dealing with skylines, scholars have examined extraction, integration and evaluation of data from hidden databases. Sheng et al [9] study proposed algorithms applicable to web-based hidden databases. For hidden databases, users use a web search interface to present queries. Users who follow static links, thus, do not access data. Rather, users query the web interface and read the result page to obtain data. During run time, users can interact with the database interface provided to access the available data. Figure 8 shows how a user can interact with the back-end database through a restricted web interface on Yahoo autos to search for a car to buy. Search engine, based on this investigation, crawls hidden databases using optimal algorithms. The proposed algorithm extracts all rows from the hidden database.



Make	Body style	Price	Mileage
BMW	sedan	$17,500	68,647 mi
BMW	sedan	$17,500	76,072 mi
BMW	coupe	$3,299	158,573 mi
BMW	convertible	$50,000	5,231 mi
		...	

Figure 8 Yahoo GUI for Crawling Of Hidden Database [9]

The algorithms proposed are efficient; in work case scenarios, they efficiently perform few queries. With respect to the problem defined in terms of hidden databases, the authors sought to prove reduction in the number of database queries. In the theoretical results provided, Sheng et al demonstrated asymptotically optimal algorithms. In other words, using constant factors does not guarantee improvements in terms of efficiency. The authors used lower and upper boundaries to present experimental results that show problem properties. Previous experiments also confirmed the findings presented in the investigation.

Sheng et al investigation [9], clearly, systematically presents a detailed analysis about hidden database crawling where there is need to retrieve data from a hidden server using few database queries. The study has high level contribution since it examines algorithms found efficient on real data and fast when applied in worst case scenarios. Using lower bound results, the study establishes the hardness linked to the defined problem statement. Based on the results, there are several underlying factors that impact hardness. The lower bound results, in addition, confirm the asymptotic nature of the algorithms suggested.

Slice-cover algorithm showed the worst performance, which is confirmed in the theoretical assessments provided since there is reflection of optimality on hardest dataset. Compared to depth first search (DFS), a suboptimal solution, the slice-cover algorithm does not provide improved efficiency. The authors discussed the effectiveness and significance of heuristic, which is confirmed through significant improvements exhibited by lazy-slice-cover algorithm. To illustrate the extensions to existing algorithms, the authors propose a hybrid algorithm that integrates rank-shrink and lazy-slice-cover algorithms.

While examining numeric attributes in the data set provided, the study focused on the performance of two algorithms, namely rank-shrink and binary-shrink. To examine categorical algorithms, the authors focused on how efficient the three algorithms are, namely lazy-slice-cover, DFS and slice-cover. To examine these algorithms, real data sets from three sources were used; NFS with 47,816 tuples, adult with 45,222 tuples and Yahoo with 69,768 tuples. The experimental evaluations carried out showed that the algorithms proposed demand for fewer queries compared to alternative solutions.

Search engines, as per now, cannot index the hidden databases effectively. So, they cannot pose queries to appropriate data repositories. Hidden data has increased, and the problem has restricted the amount of information ordinary internet users can access. So, the investigation by Sheng et al addressed the concern evident in reduction of queries while accessing hidden database servers. The goal of the study [9] was to determine the method useful in crawling hidden databases at minimum costs. The authors developed algorithms that help solve the problem where the dataset has categorical attributes, numeric attributes or a combination of the two.

3.2 Querying the Hidden Web

Crawlers are useful in retrieval of indexed information. An investigation by Raghavan and Garcia-Molin [10] on hidden web crawling ought to address the problem of designing crawlers that extract content from hidden Web pages that can be accessed via hypertext links. The authors proposed a general operational framework for a crawler that accesses the hidden web. In addition, they provided comprehensive descriptions about realizing the framework through the Stanford crawler prototype called Hidden Web Exposer. The study explored a Layout-based Information Extraction Technique (LITE), describing how the method extracts semantic information from the response pages and search forms. The technique was later tested and validated using empirical findings.

Hidden Web crawler design is a very challenging task [10]. The crawler needs to parse, interact and process human readable interfaces. In addition, users have to provide search queries to the web crawlers, begging the question of equipping crawlers with appropriate input during search engine construction. These two challenges are addressed in the study through the task-specific and human assisted method proposed. To ensure task specificity, the authors suggest that the crawler selects hidden web portions, and use specified tasks to extract the content required. Such an archive is built through resource discovery and content extraction procedures. To make sure that the information presented suits specified tasks, human assistance is essential.

The operational model proposed by [10] comprises of four components, namely task specific database, internal form representation, matching function and response analysis. The crawler builds an internal form of representation having received a form page. The task specific database has information essential in formulation of search engine queries linked to specified tasks. The matching function is an algorithm that produces final output from database contents, input and internal form representation. The response analysis module receives form submission responses and stores requested pages. It then distinguishes the pages that include search results and those with error messages. The matching function uses the feedback to display appropriate responses based on searches performed.

The hidden Web crawler prototype was built based on the described operational model [10]. The HiWe crawler seeks to extract descriptive labels or information for the corresponding elements. In the crawler, the database (task specific database), is arranged bearing in mind finite number of classes and concepts. Each concept has a corresponding label. To calculate candidate value assignments (in sets), the matching function matches the match form labels against the task specific database. The matching function in this case matches the labels first and then ranks the value assignments discovered.

The proposed HiWE prototype help address design issues including information to be collected by the crawler, useful metadata information related to matching functions, match algorithm that maximizes submission efficiency, organization, updating and accessing the task specific database and using feedback provided by the response analysis component to fine-tune the matches [10]. URL List is used as the primary data structure. The list includes all URLs already discovered by the crawler. The prototype has a Crawl manager responsible for controlling all crawling processes. The prototype has a parser that adds hyperlink texts into the URL list after extracting them from the web pages already crawled. The Form Processor, the Response Analyzer and the form analyzer are responsible for submission operations and form processing. The task-specific database, in general, is implemented using the LVS table.

3.3 Google Hidden Web Crawl

Some researchers have examined the web with respect to search engine coverage. For instance, Madhavan et al [11] study, similar to [10] investigation, focusing on deep web crawling, taking into account Google. The study describes a Deep-Web content surfacing method. For every HTML form, submissions are pre-calculated and the final HTML pages added onto the search engine index. The study included the surfacing results into Google search engine which drives numerous content while dealing with content from the deep web.

There are several problems originating from deep web surfacing. As such, [11] sought to index HTML form content which extents numerous domains and languages. Attaining such an objective requires an automatic, very efficient and highly scalable method. Since majority of the HTML forms usually have text inputs, users have to submit valid values. As such, the authors suggest an algorithm that helps identify inputs that only accept specified value types (e.g. integers or characters). The third problem associated with deep web surfacing is the presence of more input values on HTML forms; an immature approach of computing the whole Cartesian product that includes all potential input values generating numerous URLs. To address this problem, [11] proposes an algorithm which navigates the whole search space efficiently (accounts for all combined possible inputs) and determines the values that produce URLs appropriate for web search index inclusion.

In the experimental results, [11] described several experiments which support the hypothesis that the proposed algorithm effectively predicts the necessary templates for a given HTML form. Forms with select menus were considered while focusing on the templates instead of value selection in the inputs for texts. The authors manually inspected all HTML forms and found that select menus having less than five options were largely presentation inputs. The researchers compared ISIT algorithm with others and showed that for all algorithms, numerous URLs cannot be generated. It is easier to traverse the entire search space efficiently, based on this research finding. Practically, the authors found that the underlying database can be covered properly using the existing algorithm.

To choose query templates, the study revealed two primary challenges. First, there is need to choose the template that lacks binding presentation inputs given that query templates retrieve similar database records without presentation input. Nonetheless, it is impossible to know beforehand, whether or not the input is actually presentation input. The second consideration is that templates with appropriate dimensions should be selected. One method of ensuring proper dimensions is to select the one with numerous binding inputs (largest potential dimensions). Selecting an appropriate template, according to [11], helps generating all probable queries guaranteeing maximum coverage. The approach, nonetheless, increases crawling traffic eventually generating empty result sets.

[11] Successfully provided a comprehensive coverage and description of technical innovations that underlie Deep-Web surfacing system performed on a large-scale level. Numerous users enjoy the surfacing presently, which covers numerous domains (about 700), many forms and numerous languages. The search traffic performed significantly exemplifies the role of value gained from deep web surfacing. The study, in general, demonstrates three main principles likely to inform future research. Deep web indexing methods can be explored using the informativeness tests undertaken in relation to form input. Making best web crawling efforts, based on the experimental study results, can help the prototype [10] crawl deep web sites to maximize site traffic, eliminate complete site crawling and reduce crawler burden.

CHAPTER 4

Skyline Extraction over Hidden Databases

In Chapter 3, we had an overview on different approaches to crawl/query hidden databases. For such a purpose, we are interested in applying a research project that will initiate an application on the problem concerning the discovery of skylines over top-k hidden web databases. Extracting Skylines from a hidden web database will also enable a variety of innovative third-party applications.

4.1 Challenges and Limitations

In relational databases, we can crawl the entire database and apply the traditional methods for skyline computation (such as the algorithms discussed in chapter 2) over a local copy of the database. These databases have full SQL power and can have a ranking function that is already known to display all records according to it.

The challenge here in, hidden databases, is that we can only query the database through a top-K search interface to compute the skyline tuples. Hidden databases are different than traditional databases because the databases providers of the real-world databases place sever limits on how the user can interact with the database. The top-k search interface provides K results according to a ranking function the user never knows about. The data access model of web databases is completely different from the traditional databases access model [12].

A user looking for skylines can query the entire database using the different algorithms in Chapter 3 and apply algorithms for skyline computation over this database locally. However, it’s not always the case. Hidden databases also limit the number of web accesses (queries sent) per IP address or API key limit. A good example is Google Flights, which I did my project on, limit the number per API key for 50 queries per day. Solution to this for skyline computation over hidden databases is to send as many queries as necessary and stay under the limit through the restricted search interface [12].

4.2 Query Search Interface

An example of Google Flights search interface is shown in figure 9. Calculating the skyline point over a restricted search interface with different types remains a challenge [12].

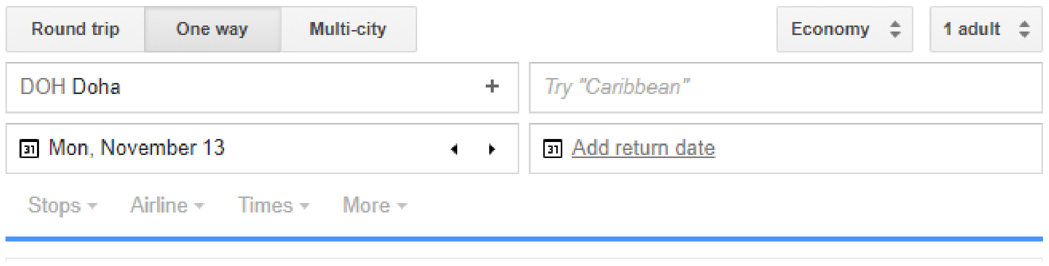


Figure 9 Google Flights Search Interface

A. *One Ended Range Predicates (SQ)*

In this category, you only specify the upper bound. An example of that is the mileage of a car (Mileage <= 31068 miles). A predicate on any attribute Ai can be $A < v, A \leq v$ or $A = v$.

B. *Two Ended Range Predicates (RQ)*

A two-ended range predicate gives the choice to choose the lower and upper bounds. $Ai < (or \leq) v, Ai = v \text{ or } Ai > (or \geq) v$

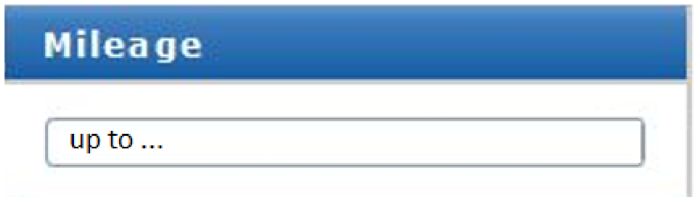


Figure 10 Single Ended Range Query

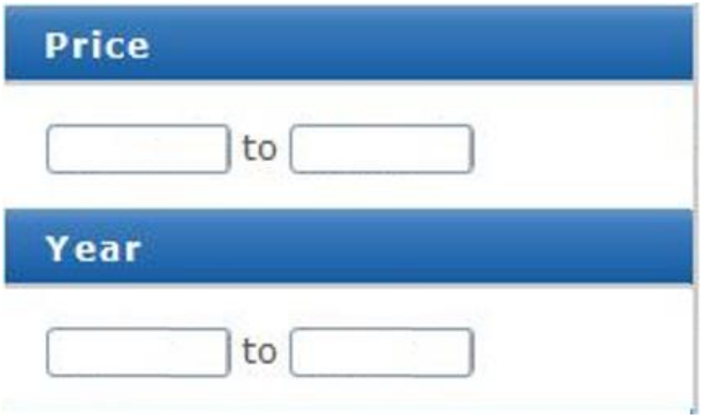


Figure 11 Two Examples of Range Query Predicates

C. *Point Predicates (PQ)*

For the third category of search interface categories, predicates can only be in form of equality where the predicate on attribute can only be of the form Ai = v.

D. *Mixed Predicates*

From SQ, RQ and PQ interfaces, we can have a mixture of them called Mixed Query.

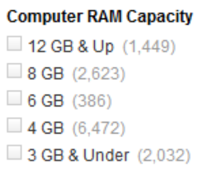


Figure 12 MQ Search Interface

4.3 Performance Measure

As mentioned earlier, we would require minimizing the number of queries sent to the databases because of the limit that the databases providers impose on the user. So, to calculate the efficiency and improve it, we need to calculate the number of queries sent and minimize it. In traditional databases, when computing the skylines, they usually calculate the computation time and/or the number of I/O.

4.4 Skyline Computation Algorithms

We study an algorithm at a time for each search interface category mentioned in 4.2.

A. SQ-DB-SKY

The SQ-DB-SKY algorithm was proposed in [12] which is a divide-and- conquer skyline algorithm that issues broad queries with queries that contain few predicates. It determines the queries that are issued next based on the records that are received.

In the below figure, the pseudo code for SQDBSKY is illustrated.

```
1: QueryQ = {SELECT * FROM D};        $S = \{\}$
2: while QueryQ is not empty
3:     $q =$ QueryQ.deque();        $T =$ Top-$k(q)$
4:     if $T$ is not empty
5:         Append the none-dominated tuples in $T$ to $S$
6:     if $T$ contains $k$ tuples
7:         Construct $m$ queries $q_1, \ldots, q_m$ where query $q_i$ appends

8:             predicate "$A_i < T_0[A_i]$" to $q$
9:         Append $q_1, \ldots q_m$ to QueryQ
```

Figure 13 SQDBSKY Algorithm [12]

For table 1 of database D (3 attributes A1, A2 and A3), we would like to calculate the number of required queries. The proposed algorithm starts with issuing the first query q1: Select * from D. Suppose that the top-1 returned tuple is t1. Now for each attribute we send 3 queries based on the values of t3.

q2: SELECT * FROM D WHERE A1 < t1[A1]

q3: SELECT * FROM D WHERE A2 < t1[A2]

q4: SELECT * FROM D WHERE A3 < t1[A3]

By doing this we are looking for skylines that are not dominated by t3 so every tuple returned must satisfy at least one of q2, q3, q4 or it would have been dominated. Suppose that q2 returned t2, so we keep sending 3 queries based on the number of attributes for each sub tree.

q5: WHERE A1 < t2[A1]

q6: WHERE A1 < t1[A1] AND A2 < t2[A2]

q7: WHERE A1 < t1[A1] AND A3 < t2[A3]

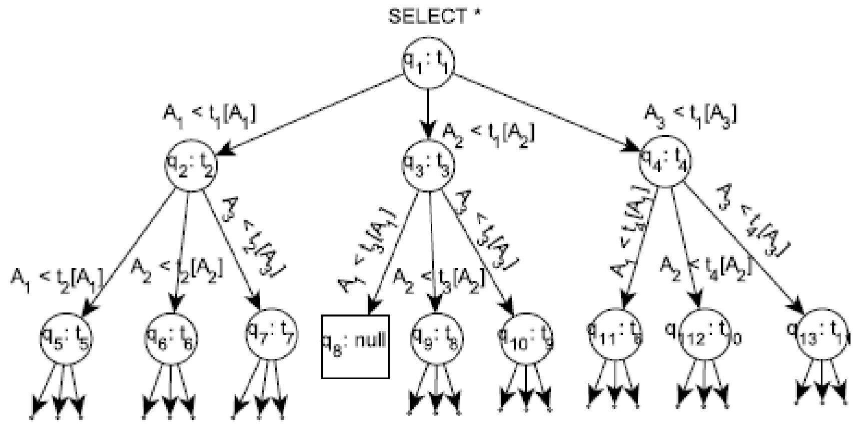


Figure 14 SQDBSKY Tree Flow

This algorithm is guaranteed to find all the skyline tuples and the proof of this is proved in the paper of Abolfazl [12].

SQDBSKY algorithm also guarantees that every tuple returned by the query is a skyline because it cannot be dominated by any other tuples not matching the query[12]. So the highest number of search queries is $O(|S|^m)$. There is also something bad regarding this algorithm where a tuple can be returned by many nodes matching the query and this could lead to a higher query cost and the worst case grows exponentially with the number of attributes m where $O(m. |S|^{m+1})$.

In the average-case, S(q): the set of skyline tuples matching q which is a randomly chosen skyline tuple from S(q). To understand why this is, we start from the simplest case of |S| = 1. The SELECT * query returns the single skyline tuple, while the m branches of it all return empty, finishing the algorithm execution. In other words, the query cost is always C1 = m + 1 (where the subscript 1 stands for |S| = 1). So, query cost for 1 skyline tuple returned for 3 attributes is C1=1+3=4 queries.

Let m0 be the number of empty branches, when |S| > 2,

$C_{|S|} = 1 + m_0 + m_1 .C_1 + \ldots + m_{s-1}. Cs\text{-}1$

In the process of finding skylines for hidden databases and learning the outcome of SQDBSKY algorithm we developed a small python code that is able to give similar results to the work of [12]. The code can be found in Appendix A and B.

B. RQ-DB-SKY

We consider now the approach taken for range query predicates where we specify the lower and upper bounds. The algorithm is similar to SQDBSKY but the authors of [12] revised it by changing the three queries of q2 to q4 and making them mutually exclusive instead of the overlapping queries.

q2: WHERE A1 < t1[A1]

q3: WHERE A1 >= t1[A1] & A2 < t1[A2]

q4: WHERE A1 >= t1[A1] & A2 >=t1[A2] & A3 < t1[A3]

This also implies that the total skyline discovery is not affected as for example t1 still satisfies the conditions of q2 to q4.

```
$S = \{\}$;    Seen = $\{\}$
traverse the SQ-DB-SKY tree in depth first preorder and at each $q$ in the tree
  if $\nexists\, t \in$Seen that matches $q$
    $T = \text{Top-}k(q)$
    if $T$ contains $k$ tuples
      generate the children of $q$ based on $T_0$
  else
    $T = \text{Top-}k(R(q))$
    if $T$ contains $k$ tuples
      if $\exists t' \in S$ that dominates $T_0$
        generate the children of $q$ based on $t'$
      else, generate the children of $q$ based on $T_0$
  Update $S$ by $T$;    Seen=Seen $\cup\, T$
```

Figure 15 RQDBSKY Algorithm

The query cost of this algorithm is to count the nodes of the tree $O(m.\min(|S|^{m+1}, n))$. It can be explained by finding the internal node, that at least must satisfy one skyline record or it could have returned null and became a leaf. Then if the internal node is the not first node to return a skyline record then the R(q) must give at least one tuple that is unique and not given before because it would have been an empty node and would have become a leaf. In this case, the upper bound of the number of interior nodes is $\min(|S|^{m+1}, n)$.

C. PQ-DB-SKY

For point query predicates, [12] stated the key differences between having a 2D space or high dimensional cases. The paper discusses and explains the method used for 2D databases by finding an algorithm called PQ-2D-SKY for this kind of space. The algorithm works by issuing a first query “select *” which always return a skyline. After that we partition the search space into rectangles R1 and R2. We loop the search space until it is fully explored by taking a rectangle and preparing the second query to issue. If a tuple is returned we append the results to skyline points and keep on pruning the search space based on the point. This algorithm has proved instance optimality because it is guaranteed to discover all skyline points.

In the case of having higher dimensionality, the authors agree that the PQ Skyline discovery doesn’t guarantee instance optimality but they suggest high-level greedy heuristic algorithm for dealing with such cases.

D. *MXED-DB-SKY*

For the mixed query predicates, the authors suggested to combine the ideas of all three algorithms discussed above to produce the ultimate algorithm they call MQ-DB-SKY. In this algorithm, all search interfaces were supported from the ranged query predicates to the point query predicates.

CHAPTER 5

Methodology and System Development

In Chapter 4, we discussed different approaches taken by the authors of paper [12] regarding the variety of search interfaces and the existence of an algorithm for each interface. In this Chapter, we define the main concepts of our system by looking at the different aspects of the algorithm, system structure and main goal. We also integrate and develop our system over three phases on a real-world database.

Dealing with hidden databases required special attention to the queries as we send them because of the limitations that are imposed by the provider. The goal is to find a method that can satisfy the database rules by staying under the limit of query quota and using the provided API if any. The user should have a goal to find such as finding the best hotels that are near the beach and cheap or booking the cheapest flight ticket with minimum number of transit time. For this purpose, we introduce an algorithm that takes the user preference into consideration for finding skyline points from a hidden database.

5.1 Skyline Computation over Normal Database

For the purpose of extracting skylines, we develop a function capable of returning skyline points over any given database. This function is a general algorithm for skyline extraction. This part will explain the logic taken in computing skyline records.

As we might know and as explained in chapter 2, there exist many algorithms for calculating skyline points. These algorithms work with known attributes from a database without giving importance to user preference or objective. For this manner, we provide the user with an application capable of dealing with multi objective attributes. In multi objective attributes, the user can specify his mixed preference over his interesting attributes.

Let’s take the hotel example where the user is looking for a hotel considering the distance to the beach and the price. The user might be interested in closer distance to the beach so that is minimum distance beach and cheaper price so that is minimum price. This problem is called minimization of attributes.

A second example could be a traveler looking for a flight considering the connection duration and number of stops. This user could be interested in high connection time for transit and less number of stops so in this case we will have minimum stops and maximum connection time. This problem is called multi objective optimization which is an area of multi-criteria decision making. In our algorithm, we deal with the maximum attributes as minimum attributes. This is done by converting all returned values to minimal through the process of finding the highest value in the column and subtracting it with other values in the column then multiplying by -1. This step is repeated every time new records are discovered. The figure below shows an example of converting maximal values to minimal.

Maximal	Minimal
10	1
3	8
11	0

Figure 16 Converting Values to Minimal

For our system, we will use the multi objective optimization concept since we are willing to find the best records for the user upon his mixed preference and goal setting.

After choosing the preference over attributes, the database is given to the system to extract skylines by iterating through records while comparing each record to all other records and eliminating the dominated records. At the end of the execution, only skyline points are left because they are not dominated by any other record. From here, we can provide the user with answers to his query for him to pick the best record. We also validate later the skyline number and similarity we found in our experiments with the results of other research work we call literature.

5.2 Skyline Extraction over Hidden Database

Due to the limitation of search interfaces and number of queries and not knowing enough information on the database, we introduce a method in scanning the hidden database in order to extract skylines. The logic followed is to send an initial query to the system without any effort and return a small number of results. These results will be used to prepare the second query and send it over again for more results. This process continues until we find no more interesting results such as no more data found or getting same data. The challenge with this process is the ability to send a reasonable number of queries without exceeding the quota and finding the maximum number of skylines. The allowed number of sent queries is defined by the database provider and our goal is to send lower number of queries.

Consider a database with three interesting attributes A, B and C. We are interested in sending queries over and over again until we are satisfied. We start by issuing a select * query to fetch initial data and then we will experiment the effect of different scenarios. A select * query can be defined as an initial query that the system should send to get information on the database. The purpose of the scenarios is to allow us to find different data than the initial data in order to calculate the skylines. An example of a scenario or expression can be “A AND B OR C”. In this scenario, once we have found our initial data by the first query, we find the best values on each of A, B and C. As an example, for finding the best value, if the user chose minimum B then the smallest value in column B is the best value and vice versa. From the new result values, we send another query specifying a more detailed query with what we are looking for. After receiving new data, we start calculating the skylines using the method described in skyline computation for normal databases.

Many scenarios can be tested in order to find new data every time. The goal of doing different scenarios is to test the effect and find more results that could lead to more partial skylines or more true skylines. What we mean by true skylines are skylines that are found from traditional methods on normal databases and are non-dominated records over the entire database. Partial skylines could also still be dominated by other records that are not yet found in the process.

This approach can be used on any database (normal or hidden) and any number of attributes. Knowing the correct number of interesting attributes can lead into combined Boolean expressions.

For A, B and C (3attributes database), we could have different combination to study the effect such as the following example of combined Boolean expressions:

- A AND B AND C
- A AND B OR C

- A OR B AND C
- A OR B OR C

Sending queries based on the values of the previous queries could lead for better results and better skyline extraction. Issuing queries to the database can be through the API (explained in the next chapter) or through top-k search interfaces. Our method is a general method that requires only simple knowledge of the database for query formulation and a way of database interaction.

After submitting the initiative query (select *) and getting first results, consider the following combined Boolean expression for 3 attributes “A AND B OR C”. The logic is to send a query built by the corresponding above combined Boolean expression. The best values are extracted from the initiative query and sent by the new query. For example, if the best values of A, B and C are 4, 3 and 12 then the query that should be sent is:

“SELECT A, B, C from D where A <= 4 AND B <=3 OR C <=12”

This keeps going until no more data found or same returned data. We also experiment with combining all of the combinations of scenarios in order to deliver more results to the system at each stage of querying. This could lead to higher query cost but could also provide more accurate skylines. However, experiments will show if we are still under the quota.

CHAPTER 6

Experimental Evaluation

6.1 Experimental Setup

After defining our methodology to follow, we define the system components, architecture and design in order to achieve our goal. We also prepare for experiments with different scenarios and define a real-world database as our testing environment.

A. Testing Environment

For the purpose of testing our methodology logic and to compare our results with different related work, we decided to test on Google Flights online hidden database. The first reason was that the work of [12] used this database in their experimental evaluation. The second reason was that Google flight provides an API for the interaction between the user and the database. The third reason was to give the user a meaningful conclusion behind skyline records and their importance in multi decision making. We believe that such a system could be important in the market as well. The fourth important reason was that this database restricts the number of web accesses per day (queries sent) to 50 queries only in which we are trying to find a solution.

Google flights makes online search for flights easier by allowing the user to look at different prices from different third-party providers. It was launched on 13 September 2011 by Google. Google flights provided a search interface which allowed the user to interact with the database.

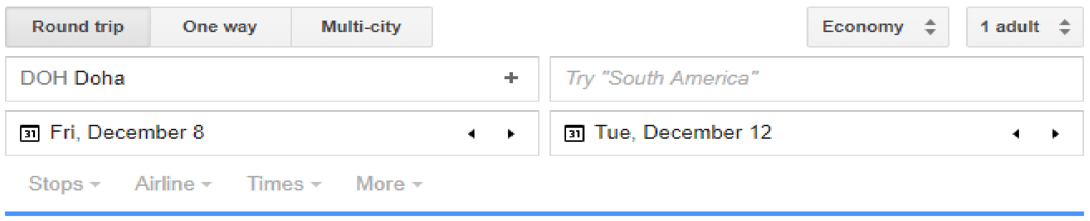


Figure 17 Google Flights Search Interface

The search interface required some inputs from the user such as origin, destination, departure date and arrival date. Google Flights provided an API called QPX Express API for providing our system with real time flights information as JavaScript Object Notation (JSON) interface.

We chose to code our system using Python version 2.7 because of the interesting libraries such programming language has. It also allows easier coding and minimum lines of code to express clear concepts. The system depends on many libraries provided in python to facilitate our work.

B. System Components

- QPX Express API

The API provided by Google allowed our system to interact with the database. The api provided an online document for how to use the API. It can be found in the following link "https://developers.google.com/qpx-express/v1/trips/search". The link also provides description over each and every parameter sent from our system and received from the request. The database was queried using the mentioned API from a python code while specifying the needed parameters. Before the system was built, the API was studied extensively in order to know what parameters were important and the JSON response for finding ways of interpreting it. Testing such API required a lot of effort to better understand the schema and extract important parameters.

The following table includes important parameters used by the system to generate queries based on the documentation provided by Google.

Table 5

Parameters Used in Sending Query

Parameter	**Description**
origin	Airport designator of the origin
destination	Airport designator of the destination
date	Departure date in YYYY-MM-DD format.
maxStops	The maximum number of stops the passenger(s) is willing to accept in this slice.
maxConnectionDuration	The longest connection between two legs, in minutes or time in transit.
maxPrice	Return flights that don’t have more price than the specified.
earliestTime	The earliest time of day in HH:MM format.
latestTime	The latest time of day in HH:MM format.
solutions	The number of solutions to return.

After submitting the query using the parameters mentioned above, API returns a JSON file containing flights in a JSON file. The next following table explains the JSON data items that we extracted from JSON files to build our own table containing the flights itineraries.

Table 6

Used Data Objects from Returned JSON File

Object Data	**Description**
Flight ID	Flight ID of the whole trip. A trip may contain many flights.
Price	Total ticket price.
C time	Total connection time or transit time of the whole trip.
Stops	Total number of stops or transits.
D time	Departure Time.

- Requests and JSON Library

The module called "Requests" was used to send some type of HTTP request to Google by passing specified headers to the following URL: "https://www.googleapis.com/qpxExpress/v1/trips/search?key={API}". After receiving data using requests, the data was saved and dumped using python JSON module.

- Pandas Library

Pandas module was used to better deal with the received data because the system required many steps in processing flights and saving data for future use. The library provided data frames for faster data manipulation using the built-in indexes. It also provided many tools to drop duplicates and eliminate certain records and compare records through iteration.

- Matplotlib library

This module helped in plotting 2D or 3D spaces. After extracting the skyline records, an example of 2D and 3D plotting

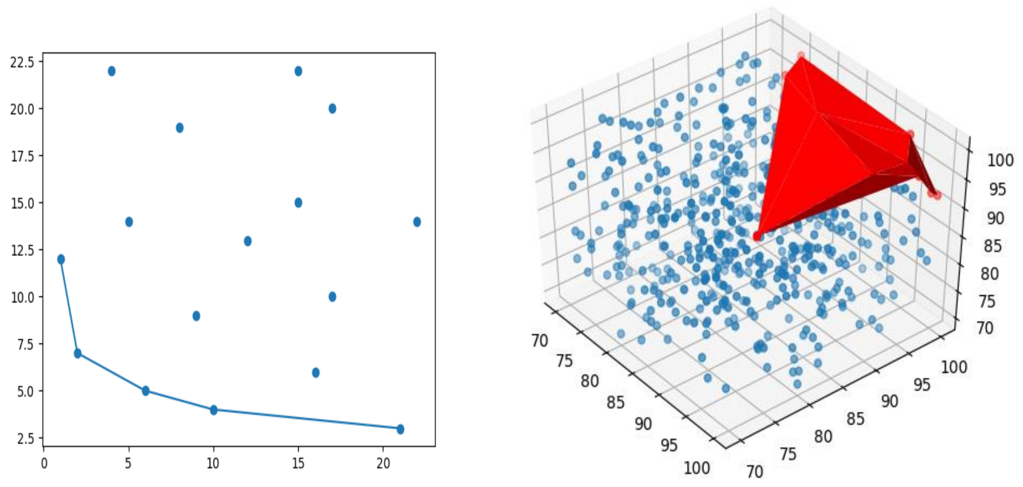


Figure 18 Skyline Examples of 2D and 3D Plotting

C. System Design

The system designed is able to extract skylines from Google flights by sending queries based on the results of the previous query using the above-mentioned methodology and concluding with query cost.

In this subject, the work was divided into three phases of development. First phase is to develop the extraction of skylines as discussed previously and testing it on Google Flights (considered as a normal database for this phase). Second phase is to implement the logic explained in the methodology section in extracting data from hidden databases and calculating skylines as it goes. In the last phase, a simulator was designed for different scenarios on different airports and prepares the testing environment.

Considering the work done by Abolfazl in [12], the same story is taken into consideration, that of a traveler looking for a flight for vacation in the next few days where the price, connection time, number of stops are low and the highest latest departure time. From here, it is concluded that the traveler is looking for the following attributes in his query: <u>minimum</u> price, <u>minimum</u> connection time, <u>minimum</u> number of stops and <u>maximum</u> departure time.

- Phase 1

In this phase, the method used on Google Flights is discussed on how the algorithm was implemented for extracting skyline flights.

Google Flights allowed 500 returned results per query only. So as a way of verification and to validate the results of the next phases, it required finding routes between two airports where the number of returned results is less than 500 flights and calculating the skyline flights on the returned result of the query. By doing this, completeness of skyline discovery was achieved (finding all skylines with a single query because the system returned all the flights for this trip < 500). The following below algorithm explains the steps followed for skyline discovery on a normal database.

Algorithm 1: Find_Skylines(data)

```
skylines = [ ]          #Fill it with non-dominated records
FOREACH field in data.FIELDS :
IF BEST_VALUE_IS_MAX(field) :  # If value is max, convert value to minimal
    max = MAX(field.VALUES)  # find maximal value
        FOREACH value in field.VALUES :
            value = -1*(value - max) # convert to minimal
FOREACH row_a IN data.ROWS : # Loop through all records
    FOREACH row_b in data.ROWS: #compare each record to the other
        dominated = TRUE
        IF row_a == row_b : #if equal skip record
            BREAK
        FOR filed in row_b.FIELDS : #if value of record is lower
            IF row_a.FIELDS[field] .VALUE < row_b[field].VALUE :
                dominated = FALSE #set record to non-dominated
        IF dominated : # if not set to false record is dominated so skip
            BREAK
    IF NOT dominated :   #add non-dominated record to skyline records
        Skylines.ADD(row_a)
```

Figure 19 Skyline Discovery: Algorithm 1 on a Normal Database

- Phase 2

After finding skyline flights, the flights are saved into a temporary result table for comparison later on. For this phase, the methodology was implemented for extracting data from hidden database such as Google flights in order to avoid limitations such as the strict number of queries per day (50 web accesses per day for Google flights).

In this phase, the algorithm is designed to use the function in phase 1 to compare with results in this phase. The purpose is to validate the answers, so if the same number of results is met then that means all skyline records were reached. It should also be kept in mind that having different number of skylines means that skyline flights found in phase 2 should not necessarily match skyline flights from phase 1. Why? Because skylines calculated in phase 2 were calculated on different data and some of the flights were not yet discovered by the queries so that means that there exist some flights that are dominant in true skylines but were not detected yet. The following algorithm 2 shows how skyline flights are calculated based on specific scenarios and to be discussed afterwards.

```
Algorithm 2: find_skylines_scenario(scenarios)

n = K #number of returned results of every query
data = {SELECT * FROM DATABASE_TABLE} #initial query to explore database
skylines = Find_Skylines(data)  #apply algorithm 1 to compare skylines (optional)
result = [ ]
FOREACH scenario IN scenarios:  loop in the scenario
    data = {SELECT * FROM DATABASE_TABLE LIMIT n} #query database
    partial_skylines = Find_Skylines(data) #calculate skylines on returned results
    exit_reason = ‘’
    previous_partial_skylines = [ ]
    WHILE TRUE:
        If previous_partial_skylines == partial_skylines :
            exit_reason = ‘Same data’ # same data is returned (loop)
            BREAK
        If data.LENGTH == 0:
            exit_reason = ‘No more data’ #no data found
            BREAK
        parameters = GET_PARAMETERS (scenario,data)
        #send new query with the values
        data = {SELECT * FROM DATABASE_TABLE WHERE parameters}
        #add to skyline table then calculate skylines again for this table
        partial_skylines = partial_skylines + Find_Skylines(data)
    result.ADD([partial_ skylines,exit_reason])
        #Keep sending queries until the scenario is done by adding more skylines
```

Figure 20 Skyline Discovery: Algorithm 2 on Hidden Database

Scenario: is an expression of attributes related by operators (AND, OR). Some examples on scenarios focusing the four attributes price, connection time (C time), number of stops(stops), and departure time (D time) are:

Price OR C time OR Stops OR D time

Price AND C time OR Stops OR D time

A user must write his scenarios in a specific format and keywords with no parentheses. If it has more complex formulas, user must simplify it to be as follows:

A1 AND B1 OR A2 AND B2 OR A3 AND B3…

Examples:

- EX 1: Price AND C time OR Stops AND D time

  Priority is given to the AND operator, this will result in the union of two requests

- EX 2: Price OR C time OR Stops AND D time

  This will result in the union of three requests (Price, C time, Stops AND D time)

- EX 3: (Price OR C time) AND (Stops OR D time)

This must be written as: Price AND Stops OR Price AND D time OR Stops AND C time OR D time AND C time

The user is able to test any date, origin airport, destination airport, number of returned results and choose a scenario to run in order to experiment the query cost and skyline flights at the same time.

Algorithm 2 was also enhanced into merging all possible scenarios for finding skylines in one run for experimentation. In this case, after finding true skylines and returning “n” initial data, all scenarios are run at the same time by querying with all possible scenarios then merge results of all queries for skyline discovery and keep going with the same step again and again until it gets the same data back then stop.

- Phase 3

An experimental environment is prepared by modifying the code to run as a simulation of airports on different number of solutions because there is interest in finding the efficiency of the algorithm with query costs and number of true skylines (found in phase

1). Also combined were all possible scenarios to an algorithm called algorithm 3.

Algorithm 3: find_skylines_all_scenarios(scenarios)

```
n = K #number of returned results of every query
data = {SELECT * FROM DATABASE_TABLE}
#calculate skyline flights to compare skylines (optional)
skylines = Find_Skylines(data)
#send first initial query as in algorithm 2 to explore database for n results
data = {SELECT * FROM DATABASE_TABLE LIMIT n }
partial_skylines = Find_Skylines(data)
exit_reason = ‘’
previous_partial_skylines = []
WHILE TRUE :
    IF previous_partial_skylines == partial_skylines :
        exit_reason = ‘Same data’ #quit when same is returned
        BREAK
    IF data.LENGTH == 0:
        exit_reason = ‘No more data’ # quit when no more returned data
        BREAK
    #prepare for next query ,getting new values
    parameters = GET_PARAMETERS(scenario,data)
    #Loop through all the scenarios and get more data and new values
    FOREACH scenario IN scenarios :
        data = data + {SELECT * FROM DATABASE_TABLE
            WHERE parameters }
    partial_skylines = partial_skylines + Find_Skylines(data)
```

Figure 21 Skyline Discovery: Algorithm 3 on Hidden Database

Algorithm 3 is responsible of combining the results of iteration from the different scenarios. It has the same logic of algorithm 2 but instead of continuing the iteration on the scenario data itself, it combines the scenario data of other scenarios at the same iteration level then extracts skyline flights on the total scenario data.

The figure below shows the system architecture design.

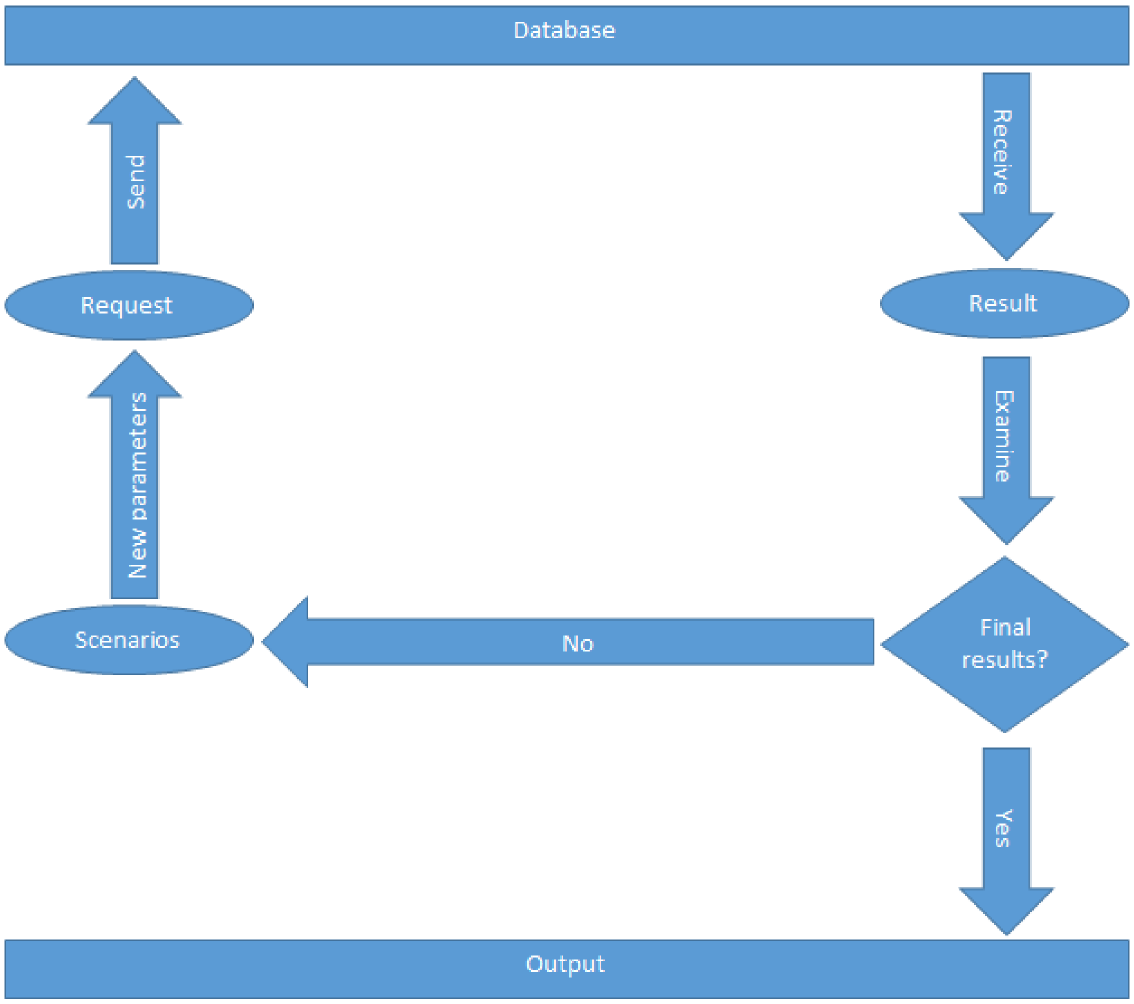


Figure 22 System Activity Diagram

The system is designed to query the database with the initial query "SELECT *" to retrieve K results where K is defined by the user. This is done just to have information on the database and prepare the next query based on the best values of each attribute. Each query sent to database must follow the proper scenario assigned to it such as following the order and the expression. The database again returns K results and the system examines the results. If new records were found, another query must be sent to the database to find new records. The system stops execution in only two conditions:

- Same data: queries return the same data over and over again and stop only after one time of extra queries.
- No data found: queries return no data to examine (empty results).

The different scenarios were evaluated for skyline extraction while varying the number of solutions (returned result on each query). The results of our experiments on Google flights real world database are also discussed.

6.2 Experimental Validation of Algorithm 1

The purpose of this experiment was to evaluate the algorithm chosen for skyline extraction from a known database without implying any scenario.

The experiment included testing skyline extraction to find the true number of skylines on different trips and dates so 6 airports were chosen randomly (‘ATL', 'DXB’, ‘PEK’, ‘LAX’,’DEN,’SAN’ and ’LGA’).

Table 7 shows the results of a simple experiment to find skyline flights from the total number of flights of different trips.

Table 7

Results of Experiment 1

Flight	**Total Number of Flights**	**Number of Skyline Flights**
ATL -> BEY	246	10
ATL -> DXB	119	4
ATL -> LAX	227	4
ATL -> PEK	301	16
ATL -> DEN	106	8
ATL -> SAN	279	6
ATL -> LGA	128	11

It can be seen that the discovery of skyline flights doesn’t depend on the total number of flights but depends on the data itself because flights that are not dominated by any other flights are considered as skyline flights.

6.3 Experimental Validation of Algorithm 2

In this experiment, algorithm 2 is followed with the logic of a scenario and sending many queries to the database with different values based on the result of the query before, in order to calculate skyline flights for hidden database. A scenario will be an expression of attributes related by operators (AND, OR) to send next queries based on the result of the returned values (scenarios can be seen in appendix C). Every possible scenario is run on different trips to evaluate the query cost (number of queries sent) and the number of skylines. The number of returned results "K" was also modified for each scenario in order to study the impact. Every scenario is noted as a number from one to eight (details in appendix C).

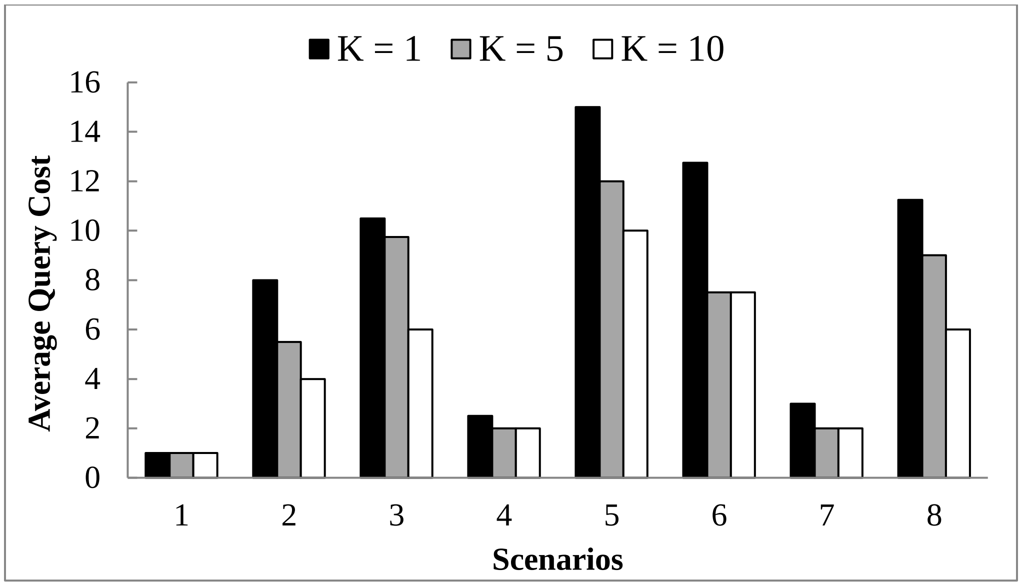


Figure 23 Average Query Cost on Each Scenario

From the results of figure 23, it is noted that scenario 5 has the highest query cost for K=1, 5 and 10 because it is queried in every iteration four times and then union the results to calculate skyline flights on the given data. The lowest query cost among the experiments was scenario 1 because this is just 1 query sent using the API of Google specifying all the best values for all attributes in 1 query. Other scenarios results varied upon the scenario and the data it got from Google. Also concluded from this experiment was that K= 1 had the highest query cost in most of the scenarios because t returned just 1 flight after each query which required sending more queries to get more data.

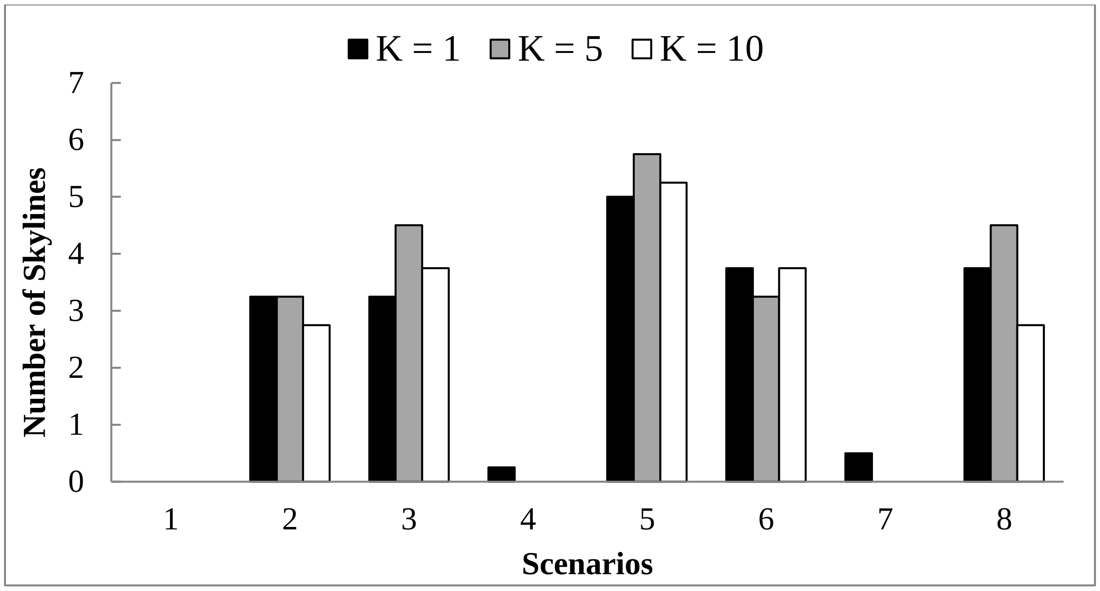


Figure 24 Number of Skylines Found on Each Scenario for Different K

In Figure 24, the first scenario didn't return any skyline flights because it's a very strict scenario and all conditions must be satisfied. It can also be seen that scenario 4 and 7 didn't return high number of skylines for K = 1 and no skylines for K=5 and 10 because of the different values returned from the queries that didn't meet the conditions for the next query. The highest number of skyline flights found was in scenario 5 because this scenario allowed more data to be crawled and therefore better skyline detection.

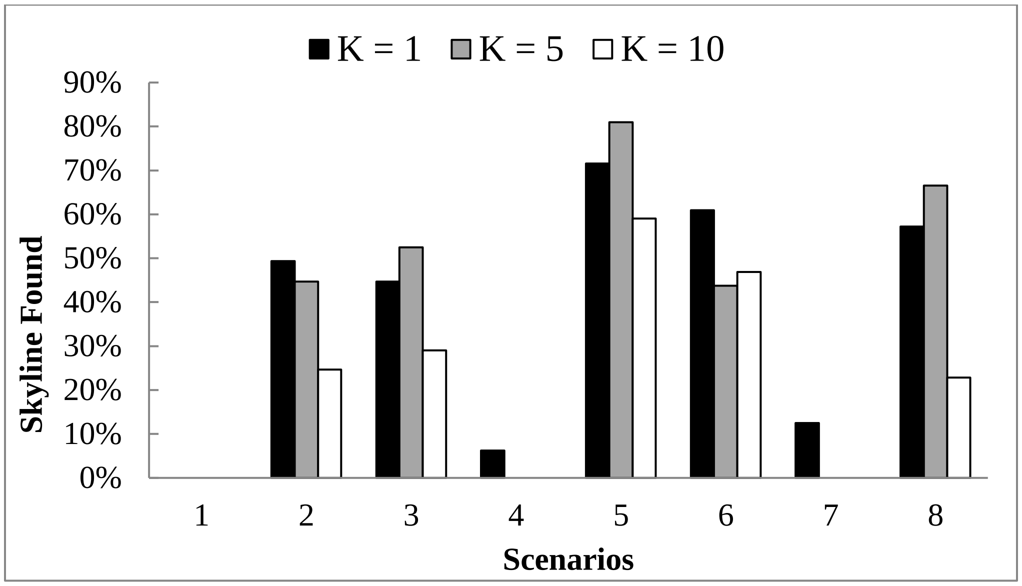


Figure 25 Percentage Found of True Skylines for Different K

In the experiments, true skyline flights are the flights that can never be dominated from any other flight though it was necessary to calculate the percentage found of true skylines because the skyline flights found can still be dominated by flights that were not discovered yet. Figure 25 illustrates results of percentage found of skylines for each K while scenario 5 achieved the highest percentage found of skyline flights.

6.4 Experimental Validation of Algorithm 3

First in this experiment, all possible scenarios were combined into one big scenario to calculate the skyline flights. 8 possible scenarios were combined to query and fetch data then combine these results to calculate skyline on it. The goal is to get as much new data as possible before calculating skyline flights. In this method, results were compared to the results of paper [12] because both goals aimed to find the number of skyline flights and the query cost. Experiments and testing environments conducted were the same as the work of Abufazl[12] on different dates and the dates chosen were between 23 December 2017 and 30 January 2018. Also randomly chosen, were origin and destination airports from the top 25 busiest airports in the United States of America. The same story of a traveler looking for a getaway for his holiday and searching for the lowest price, transit duration, number of stops and latest departure time was also considered.

It was noticed that the results of query cost did not fall within plausible outcomes as different scenarios yielded empty queries, as seen in the experimental results of algorithm 2, which increased the query costs significantly. Hence, an alternative methodology was proposed; the scenarios that only generated queries with high quality skyline flights were combined instead of all. Appendix D shows details on the chosen scenarios.

Table 8

Average Query cost and Percentage Found of Skylines

	Literature	K = 1	K = 5	K = 10
Avg Query Cost	44	46	31	26
Percentage Found	100	90	70	53

It was noticed from the results of table 8, that the methodology for K = 1 and choosing only the scenarios 2, 5, 6 and 8 achieved very similar results to the literature of Abufazl. The methodology for K=1 achieved an average of 90% of true skylines while Abufazl achieved 100%. The average query cost in the methodology was also very similar ranging between 44 and 46 queries.

In the results of table 8, it was also noticed that increasing the number of K would result in less discovery of skylines and less query cost on average. This comes because of the different best values of the returned results which varied the values to be sent in the next queries.

CHAPTER 7

Conclusion and Future Work

This chapter outlines the main points obtained during the research with a brief summary regarding results of the conducted experiments. It also proposes future work to improve our methodology and achieve hopefully better results.

7.1 Conclusions

In conclusion, suggesting a high-quality method to extract skyline records from any hidden database while able to maintain a reasonable query cost was the primary research objective in this project. The method to combine different scenarios with multi-objective optimization goal proved to be efficient in terms of number of skylines found and query cost.

The project research discussed the following main points regarding skyline extraction:

- Converting maximal preference attributes into minimal using a specific technique used in multi-objective optimization problems.
- Extracting skyline records for normal databases.
- Developed an efficient algorithm for skyline extraction based on scenarios and conducted experiments over a top-k interface for a real world hidden database which proved a low average query cost below the quota.

The methodology followed for extracting skyline records can be adapted for any hidden web database such that choosing the preferences for attributes and number of returned results “K” can allow getting more data as input for the system to calculate the non-dominated tuples.

7.2 Future Work

Many different approaches and experiments have been left in this work for many reasons. The main reason was the lack of time because the system was designed to deal with a real world hidden database which restricted the number of queries sent per day and made it difficult to conduct more experiments. There are also some proposed ideas in the system to enhance the algorithms to be as smart as possible by not sending the same queries again since it had already found the results. This report has mainly focused on testing a new approach when it comes to dealing with hidden databases and crawling with minimal query cost. The following ideas could be added in future:

- The way of interacting while querying the database could be improved by not repeating the same queries especially for conjunctive queries that required more than one query to find the result of the complete scenario.
- The system should be modified to bypass errors like error 503 “temporary overload. Wait before retrying”. This can be done by finding the query cost of having such an error and sleeping for a specific amount of time before querying next.
- Introducing a tree browser, which allows the system to adapt the next query based on the results of the previous query by following specific routes in the tree. This could lead to lower query cost and a more accurate number of skylines.

REFERENCES

[1] S. Borzsony, D. Kossmann, and K. Stocker, “The Skyline operator,” *Proc. 17th Int. Conf. Data Eng.*, pp. 1–20, 2001.

[2] E. Tiakas, A. N. Papadopoulos, and Y. Manolopoulos, “Skyline queries: An introduction,” *IISA 2015 - 6th Int. Conf. Information, Intell. Syst. Appl.*, 2016.

[3] D. Kossmann, F. Ramsak, and S. Rost, “Shooting Stars in the Sky: An Online Algorithm for Skyline Queries,” *Proc. 28th Int. Conf. Very Large Data Bases*, pp. 275–286, 2002.

[4] D. Papadias, Y. Tao, G. Fu, and B. Seeger, “An optimal and progressive algorithm for skyline queries,” *Proc. 2003 ACM SIGMOD Int. Conf. Manag. data - SIGMOD ’03*, p. 467, 2003.

[5] I. Bartolini, P. Ciaccia, and M. Patella, “SaLSa: computing the skyline without scanning the whole sky,” *Proc. 15th ACM Int. Conf. Inf. Knowl. Manag.*, pp. 405–414, 2006.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, “Skyline with presorting,” *Proc. IEEE Int. Conf. Data Eng.*, pp. 17–19, 2003.

[7] D. Sun, S. Wu, J. Li, and A. K. H. Tung, “Skyline-Join in Distributed Databases,” in *IEEE 24th International Conference*, 2008, pp. 176–181.

[8] H. Eder and F. Wei, “Evaluation of skyline algorithms in PostgreSQL,” *Proc. 2009 Int. Database Eng. Appl. Symp. - IDEAS ’09*, p. 334, 2009.

[9] C. Sheng, N. Zhang, Y. Tao, and X. Jin, “Optimal Algorithms for Crawling a Hidden Database in the Web,” *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1112–1123,

2012.

[10] S. Raghavan and H. Garcia-molina, “Crawling the Hidden Web,” *27th VLDB Conf. - Roma, Italy*, pp. 1–10, 2001.

[11] J. Madhavan, D. Ko, \Lucja Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, “Google’s Deep Web crawl,” *Proc. VLDB Endow. Arch.*, vol. 1, no. 2, pp. 1241–1252, 2008.

[12] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das, “Discovering the Skyline of Web Databases,” vol. 9, no. 7, pp. 600–611, 2015.

APPENDICES

Appendix A: SQ-DB-SKY (1)

The following python function generates the skyline points of any given database as CSV format while concluding with total number of skylines and query cost.

```
'''
This is main fucntion that calculate the the SQ_DB_SKY algorithm. The input passed to
this function is the tuple, and list of values to be compared. This function is
iterated for each record, and returns the next row to be checked.
'''


def SQ_DB_SKY (startTuple,q, compareValue, AlreadyDoneTuple,colNumber):
    ## next Occurring values that will be returned by this function
    smallerValues = []
    # comparing the select tuple with each respective
    for index, row in enumerate(compareValue):
        print ("Comparing Value: "+str(row+1))
        for i in range (0,colNumber):
            valueToCompare=A[i][row]

            tmpLargesValue=-1
            rowNumToAppend=-1;

            for id, value in enumerate(A[i]):
                if int(value) < int(valueToCompare) and id!=row and
int(value)>tmpLargesValue:
                    rowNumToAppend=id;

            if rowNumToAppend!=-1 and rowNumToAppend not in AlreadyDoneTuple :
                if q>4 and rowNumToAppend not in smallerValues:
                    smallerValues.append(rowNumToAppend)
                elif q<=3:
                    smallerValues.append(rowNumToAppend)
            # if T contains k Tuples    Construct m queries ....

            q+=1;
    for vl in compareValue:
        if vl not in AlreadyDoneTuple:
            AlreadyDoneTuple.append(vl)
    return q, smallerValues, AlreadyDoneTuple;
```

Appendix B: SQ-DB-SKY (2)

The following is the algorithm main function responsible in calculating skyline points and building the tree explained in Chapter 4.

```
def AlgorithmMain():

    global A;
    A = []
    SkyLines = []
    inputCSVFileName = 'test_data.csv'

    k= int(input("Enter the Number of Lines that you want to process"))
    inputRow=k+1
    inputColumNames= input ("Input the names of column as comma separated")
    colNames=inputColumNames.split(",")
    NumberOfColumns=len(colNames);


    incldueFirstColumn =False;    #setting this value to False will return

    #subtracting the value from the index of the columns
    if incldueFirstColumn==False:
        colNames[:] = [int(x) - 1 for x in colNames]



    #this function reads from the csv file
    #** QUERYQ= {SELECT * FROM D }; S ={}
    readCSVFile(inputCSVFileName, inputRow,NumberOfColumns, colNames, incldueFirstColumn)


    start_time = time.time()
    if incldueFirstColumn==True:
        # startTuple = random.randrange(1,inputRow)
        startTuple=0
        print("Tuple to start " + str(startTuple))

    else:
        # startTuple = random.randrange(0, inputRow-1)
        startTuple = 0
        print ("Tuple to start "+ str(int(startTuple)+1))
    #A.append(A1); A.append(A2); A.append(A3)
    q=1;

    if incldueFirstColumn==True:
        compareValue = [startTuple-1]
    else:
        compareValue = [startTuple]

    AlreadyDone=[]

    #* While QueryQ is not emmpty
    while len(compareValue)>0:
        #print ("Main", end='')
        #q= QueryQ.deque(); T= Top-k(q)
        q, compareValue,AlreadyDone = SQ_DB_SKY(startTuple, q, compareValue, AlreadyDone,
NumberOfColumns)

    #if T is not emppty
    #    Append the non-denominated Tuple in To to S
    for i in AlreadyDone:
        SkyLines.append(int(i)+1)

    # predict
    # Append  q1..... to QueryQ
```

Appendix C: System Configuration for Experiments

- Airports:

  "ATL","LAX","ORD","DFW","JFK","LAS","MSP","DTW","PHL","BOS","LGA","FLL","BWI","IAD","SLC","MDW""PHX","IAH","CLT","MIA","MCO","EWR","SEA","DEN","SFO".

https://www.tripsavvy.com/busiest-airports-in-the-usa-3301020

- SCENARIOS = [
  1- 'Price AND C time AND Stops AND D time',
  2- 'Price AND C time AND Stops OR D time',
  3- 'Price AND C time OR Stops OR D time',
  4- 'Price AND C time OR Stops AND D time',
  5- 'Price OR C time OR Stops OR D time',
  6- 'Price OR C time OR Stops AND D time',
  7- 'Price OR C time AND Stops AND D time',
  8- 'Price OR C time AND Stops OR D time'
  ]

- SOLUTIONS = number of returned trips (can be changed to any number less than 500)

- COLUMNS_DICT = {

```
'Price'      : DOMINATE_MIN,
'C time'     : DOMINATE_MIN,
'Stops'      : DOMINATE_MIN,
'D time'     : DOMINATE_MAX,
}
```

Appendix D: Scenarios used in experiment

The chosen scenarios for the experiment in section 6.3 for chapter 6 are:

- 'Price AND C time AND Stops OR D time'
- 'Price OR C time OR Stops OR D time'
- 'Price OR C time OR Stops AND D time'
- 'Price OR C time AND Stops OR D time'

We chose them from the total combined scenarios in Appendix C.