PeaQock: A PostgreSQL Extension with Evaluation Algorithms for Skyline and Top-k Skyline Queries

Carmen Brando, Marlene Goncalves and Vanessa González
Universidad Simón Bolívar, Dpto. de Computación y T.I.
Caracas, Venezuela, 89000
{carmen,mgoncalves,vgonzalez}@ldc.usb.ve

Abstract
Today, many information systems might benefit from preference based languages to get the best answers according to user-specified criteria. Skyline and Top-k are two languages that allow users to express their criteria. On the one hand, Skyline queries identify the best answers based on a multi-criteria function. On the other hand, Top-k queries determine the top $k$ answers based on a score function. Additionally, a third language that integrates the prior has been proposed, Top-k Skyline. Unfortunately, traditional relational DBMSs don’t include the notion of preferences. However, some solutions for Skyline and Top-k queries have been implemented as part of the core query engine, showing performance improvements when compared to solutions implemented as an external application (middleware). To the best of our knowledge, there are no relational DBMSs that support Top-k Skyline queries. In this paper, we extend PostgreSQL with evaluation algorithms for Skyline and Top-k Skyline queries, and present experimental results to show their properties.

Keywords: Databases, Preference based Queries, Skyline, Top-k, Multi-criteria function, Score function.

Resumen
Actualmente, muchos sistemas de información pueden beneficiarse de lenguajes de consultas basadas en preferencias con el fin de obtener las mejores respuestas de acuerdo a criterios especificados por el usuario. Skyline y Top-k son dos lenguajes que permiten la expresión de criterios de preferencias de usuarios. Por un lado, las consultas Skyline identifican las mejores respuestas en términos de una función multicriterio mientras que las consultas Top-k determinan las $k$ mejores respuestas en base a una función de score. Adicionalmente, se ha propuesto un tercer lenguaje llamado Top-k Skyline que integra ambos. Desafortunadamente, los sistemas relacionales tradicionales no incluyen la noción de preferencias. Sin embargo, algunas soluciones para las consultas Skyline y Top-k han sido incorporadas como parte del motor relacional mostrando beneficios de rendimiento con respecto a aquellas soluciones que son implementadas como una aplicación externa al manejador. Hasta el momento, no se conocen trabajos sobre manejadores de bases de datos que den soporte a consultas Top-k Skyline. En este trabajo, se realiza la extensión de PostgreSQL con algoritmos de evaluación de consultas Skyline y Top-k Skyline, así como se presentan resultados experimentales que muestran las propiedades de los mismos.

Palabras clave: Bases de datos, Consultas basadas en Preferencias, Skyline, Top-k, Función multicriterio, Función de score.
1 Introduction

Many decisions taken every day are guided by personal preferences. Such preferences, as wishes, are free and not guaranteed to be fulfilled. In general, when people can’t get exactly what they want, they are willing to compromise or negotiate between the perfect match for one wish and a worst one for another simultaneous wish. Thus, life is not about getting exactly what one wants, but about finding the best option, the one closest to our desire. In other words, preferences introduce the notion of soft constraints.

Relational database management systems were conceived to handle hard constraints. These constraints either satisfy the request completely or they don’t. Contrary to the more flexible soft constraints, that don’t try to make exact matches but rather find the tuples within the data set that best fulfill user expectations [15].

Given the need to specify and evaluate user preference criteria inside relational DBMS, two SQL extensions have been defined: Top-k and Skyline. The evaluation of Top-k queries produces the best \( k \) tuples based on a user-defined score function that induces a total order over the tuples. On the other hand, Skyline queries identify non-dominated tuples based on a multi-criteria function that maximizes or minimizes some attributes, inducing a partial order over the tuples. Some tuple \( a \) is said to dominate another tuple \( b \), if \( a \) is as good or better than \( b \) for all attributes and strictly better than \( b \) in, at least, one. Hence, if criteria are conflictive, various optimal answers are retrieved.

Top-k and Skyline are two different preference based querying languages. Top-k requires the specification of a number \( k \) of desired answers and a user-defined score function such that all of its arguments are numeric attributes. However, it might not be natural for all users to express their preferences using a score function. On the other hand, Skyline retrieves tuples for which all criteria defined by the user are equally important. Nevertheless, the user might require exactly \( k \) tuples in the result and, for Skyline, it is not possible to discriminate among the answers because they are all optimal. Besides, \( k \) could be greater than the cardinality of the Skyline, yielding an incomplete result. Considering the aforementioned limitations, Top-k Skyline was proposed as a language that integrates Skyline and Top-k in order to retrieve exactly the best \( k \) tuples from the set of necessary strata. Here, two definitions stand out: stratum and necessary stratum. The \( i \)-th stratum consists of a data partition that includes tuples that are not dominated by each other and are dominated only by the previous \( (i-1) \)-th stratum; except, of course, for the first stratum that corresponds to the Skyline. A stratum is necessary, if and only if the cardinality of the union of previous strata is strictly smaller than \( k \).

But then, solutions to preference based queries could be classified in two groups: those integrated into a DBMS and those implemented as a middleware, being better the former, because the integration with the relational engine avoids the performance degradation caused by accessing data per tuple, instead of disk block, and the additional cost of the two-layer processing.

Considering that these solutions have proven to be successful, we propose to extend PostgreSQL with evaluation mechanisms for Skyline and Top-k Skyline queries. To the best of our knowledge, there are no open-source DBMSs that support Skyline and Top-k Skyline queries.

This paper comprises six sections. In Section 2, we introduce related works and background for preference based queries. In Section 3, we briefly describe the architecture of PostgreSQL. In Section 4, we present evaluation mechanisms for Skyline and Top-k Skyline queries and the development of PostgreSQL extension. In Section 5, we present an experimental study that shows the properties of the evaluation algorithms implemented. Last, in Section 6, we point out concluding remarks and future works.

2 Preference based Queries

Different approaches have been proposed for the problem of specifying and evaluating preference based queries: Skyline, Top-k and Top-k Skyline. Skyline [3], also known as Pareto Curve [21] or Maximal Vector Problem [2] [16] [22], consists in finding a set of incomparable or non-dominated tuples. A tuple dominates another if it is better or equal than the other in every dimension and better in at least one. Dimensions correspond to attributes whose values are maximized, minimized or grouped, as specified by the multi-criteria function [3].

Formalizing the definition for dominance: given two tuples \( t_i \) and \( t_j \) from relation \( R \), and a multi-criteria function; \( R \) has \( p \) attributes. So, \( t_i \) and \( t_j \) are such that

\[
\begin{align*}
t_i &= (a_1, ..., a_k, a_{k+1}, ..., a_l, a_{l+1}, ..., a_m, a_{m+1}, ..., a_p) \\
t_j &= (b_1, ..., b_k, b_{k+1}, ..., b_l, b_{l+1}, ..., b_m, b_{m+1}, ..., b_p)
\end{align*}
\]
\[ t_j = (b_1, b_k, b_{k+1}, ..., b_l, b_{l+1}, ..., b_n); \] where \( a_i \) and \( b_i \) are their values for the \( i \)-th attribute in \( R \), for every \( 1 \leq i \leq p \).

Additionally, the multi-criteria function indicates the minimization of the first \( k \) attributes, the maximization of attributes \( k+1 \) to \( l \) and the grouping of attributes \( l+1 \) to \( m \).

Hence, \( t_i \) dominates \( t_j \), if and only if
- \( a_i \leq b_i \) for every \( i = 1, ..., k \),
- \( a_i \geq b_i \) for every \( i = (k+1), ..., l \) and
- \( a_i = b_i \) for every \( i = (l+1), ..., m \).

Moreover, if \( a_i = b_i \) holds for every \( i = 1, ..., m \), then \( t_i \) and \( t_j \) are incomparable or non-dominated. Consequently, both belong to the Skyline.

Skyline queries can be specified using the \texttt{SKYLINE OF} clause that extends SQL \texttt{SELECT} statement, as presented in Figure 1. Here, \( d_1, ..., d_m \) denote Skyline dimensions and \texttt{MIN}, \texttt{MAX} and \texttt{DIFF} represent the directives that indicate whether their corresponding attributes should be minimized, maximized or grouped. The set of dimension-directive pairs comprises the multi-criteria function that induces a partial order over the data.

\begin{verbatim}
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF d1 [MIN | MAX | DIFF], ..., dm [MIN | MAX | DIFF]
\end{verbatim}

Figure 1: SQL Query for Skyline problems.

Godfrey et al. [11] proved that the Skyline problem has asymptotic runtime complexity of \( O(n) \) for the average-case scenario and \( O(n^2) \) for the worst-case; where \( n \) is the cardinality of the input set. This complexities might produce unacceptable execution times for very large databases. For this reason, many efficient algorithms have been proposed to solve the Skyline problem in a database context. The first Skyline algorithm was proposed by Kung et al. [16] and remained a study problem during the 80’s and the 90’s, when algorithms for high dimensional Skylines [18] and parallel Skyline algorithms [23] were introduced. However, after Börzönyi et al. published [3], Skyline algorithms started to be designed for relational settings, introducing a SQL extension that would easily allow the expression of user preferences in queries. Furthermore, this work proposed a new algorithm known as Block-Nested-Loops (BNL). BNL scans the input table while it maintains a window of non-dominated tuples in main memory; tuples in the window can be replaced by any succeeding tuple. Later, Chomicki et al. presented another algorithm called Sort-Filter-Skyline (SFS) [10], a BNL variant that requires a previous topological sort compatible with the Skyline criteria and, unlike BNL, doesn’t replace window tuples thanks to the initial sort phase. Finally, Godfrey et al. [11] introduced Linear Elimination Sort for Skyline (LESS) that initially sorts the input like SFS does, but provides two improvements over it: the first sort phase uses an elimination-filter window to quickly discard dominated tuples and combines the last sort phase with the first Skyline filter to eliminate remaining dominated tuples. Nevertheless, none of these algorithms are implemented in a web available DBMS.

Skyline was formalized using partial order semantics, because tuples in the answer are incomparable. On the contrary, Top-k relies on a total ordering of the tuples. This approach identifies the best answers based on the evaluation of a monotone score function defined by the user. Query results are sorted on the function value and only the first \( k \) answers are returned [20]. The Top-k problem states that a tuple \( t_i \) belongs to the result set if and only if there are not at most \( k \) tuples with a higher score.

SQL notation for Top-k queries was introduced by Carey and Kossmann [6] [7] and is presented in Figure 2. SQL \texttt{SELECT} statement is extended with the \texttt{STOP AFTER} clause for the specification of Top-k queries, and its associated score function is indicated using the \texttt{ORDER BY} clause.

\begin{verbatim}
SELECT * FROM T1, T2, ..., Tm WHERE join_condition(T1, T2, ..., Tm) ORDER BY f(T1.score, T2.score, ..., Tm.score) STOP AFTER k;
\end{verbatim}

Figure 2: SQL query for Top-k problems. Source: [14]
Generally, the Top-k problem has asymptotic runtime complexity of $O(1)$ and, for the worst-case scenario, $O(n)$. As for Skyline, it might be unacceptable for users to wait such an evaluation time when data cardinality is large. For relational databases, there are many algorithms that solve the Top-k problem. Proposed algorithms seek to return the best $k$ tuples stopping before the score function has been evaluated for all of the input tuples. In [8], the authors presented the advantages and disadvantages of processing a Top-k queries by translating them into regular SQL. The strategy consists in obtaining a subset of the data that possibly holds the $k$ answers and, using histograms, a threshold that limits the subset. Then, the score function is evaluated on this subset and the best $k$ tuples are chosen. However, the subset might contain less than $k$ tuples, because the threshold is statistically estimated. If this happens, a smaller threshold is chosen and the strategy is restarted until $k$ answers are found. Bruno et al. [5] later suggested an improvement for this strategy, redefining the score function as a distance function that assigns each tuple a value for its proximity to the objective.

Finally, a third solution for preference based query evaluation poses the integration of the previous query languages [13]. On the one hand, Skyline answers are all optimal, so it is not possible to return only $k$ tuples, as Top-k does, because Skyline does not discriminate among results. On the other hand, it is not possible to define a score function when the user is not interested in assigning weighs to the criteria, but rather considers them equally important, as for the Skyline case. Consequently, solutions have recently been proposed for situations where both paradigms must be combined [1] [12] [17]. These proposed solutions are limited to calculate the first stratum or Skyline with some post-processing. None of them identifies the $k$ best answers when $k$ is greater than the cardinality of the Skyline. To this end, Goncalves and Vidal [12] introduced the Top-k Skyline language, which allows users to obtain exactly the $k$ best answers from a set of necessary strata.

Our main interest relies in Top-k Skyline queries, because fewer evaluation algorithms have been proposed for them. We do not implement algorithms for Top-k queries, given that for the Top-k Skyline problem data is partitioned according to Skyline criteria. Therefore, we implement evaluation algorithms for Skyline and Top-k Skyline queries, which are integrated inside a free software DBMS. The chosen DBMS is PostgreSQL, given that this system provides a complete and stable foundation for development that stimulates the generation of projects to expand it. Moreover, its readily availability in the web and its free software status promotes massive usage of this system.

### 3 PostgreSQL

PostgreSQL is an object-relational database management system (ORDBMS), developed at the Computer Science Department from the University of California at Berkeley. From its beginnings, many developers united in the task of strengthen it, without concerning about the complexity of the source code, that consisted of approximately 250,000 written lines. Later, in 1996, its name changed to PostgreSQL and it became popular among users and programmers. At the moment, PostgreSQL is the most advanced open source database available in the Web [19].

In [9], the authors propose a general model to represent the architecture of PostgreSQL, dividing it in three subsystems: client, server and storage manager. As presented in Figure 3, the client consists in an application able to connect itself to the server, e.g. a Java application using JDBC. The client also needs the Libpq component, which is responsible of handling the communication processes with the server and to send queries specified by the client.

The server comprises two subsystems: the postmaster and the backend. The former takes connection requests from clients, handles authentication and user control, and, if nothing fails, invokes the postmaster for a connection; then, the connection between the client and the backend is established. Under this philosophy, PostgreSQL follows a per client process model, meaning that each client connects to exactly one server process (backend).

Additionally, the storage manager is responsible for general storage management and resource control of the backend, including shared buffer management, file management, consistency control and lock manager.

Going further in the backend architecture, all of its components interact with a client component called traffic cop. Basically, the traffic cop is in charge of yielding operational control to the backend components, namely the parser, rewriter, optimizer and executor.

PostgreSQL postmaster listens for requests in TCP/IP port 1521. When the client receives a query, it requests a connection to the postmaster, which creates a new server process or backend, following the
client-server model. The postmaster creates a new backend for every request received. Once the client is connected to a backend, it sends the query. The backend delivers the received query to the parser as a string.

Then, the parser verifies the syntax and semantics of the query, and generates the corresponding parse tree. The rewriter receives this parse tree, and does the appropriate type checking against the data dictionary. And, additionally, creates the rewritten tree for the query.

The Optimizer plans the execution of the query considering the rewritten tree. Once the query is planned, its corresponding plan tree is generated. The plan tree contains the query plan that indicates the executor which algorithms to use for every node in the tree. Plan tree generation consists of three steps. First, several possible plans that describe different execution strategies are studied. Second, execution costs for the generated plans are evaluated. And, third, the least expensive plan is chosen as the plan tree to be delivered to the executor.

Finally, query evaluation occurs in the executor. The executor operates in three phases: initialization, execution and ending. During initialization phase, the executor recursively scans the plan tree in order to create the execution tree, which reproduces the structure of the former. In the second phase, it executes each operator following the order stated in the plan tree, until the result is achieved. The last phase simply finalizes the operators and clears shared memory pools.

Additionally, two complimentary components interact with the ones already described. These are: commands and utilities. The first one processes queries that only require simple handling, like administration tasks; the latter supports utility routines used by other backend components.

When the evaluation of a query is completed, the executor returns the result to the client and notifies the traffic cop of its commitment. Traffic cop handles concurrent operations to the database.

4 PeaQock

We have extended PostgreSQL to support preference based queries, chiefly due to the absence of relational DBMS that evaluate such queries. Despite the fact that there are some DBMSs that implement solutions for Skyline and Top-k queries, these are not readily available. Besides, to the best of our knowledge, no available DBMS evaluates Top-k Skyline queries.

PeaQock is the name of our PostgreSQL extension. It is a word game derived from peacock, where $P$ and $Q$ stand for Preference Queries. This decision followed the current standard in open source development of using animal symbols to identify projects.

PeaQock is expected to provide performance benefits for preference based queries in comparison to any solution existent outside the core query engine, for several reasons. First, because PostgreSQL is implemented in C language, whose efficiency is widely known. Second, because implementing an application on top of the DBMS might cause additional processing costs and data access will be performed by the tuple, instead of per block. Besides, given that PostgreSQL is a free software project, it provides stable support for users and developers, as encourages its widespread and use.

To develop PeaQock, several evaluation algorithms were studied. For the case of Skyline queries in relational DBMS, evaluation algorithms can be divided into two groups. The first group refers to index-
based algorithms, while the second group includes algorithms that sequentially scan the input table. In this paper, we do not consider index-based algorithms for two main reasons. First, it would require the creation of indexes on every attribute that the user might refer to in a query. Second, if a tuple that is going to be inserted dominates some of the tuples in the previously calculated index, it would make the entire index invalid, because the index will no longer hold the Skyline.

Block-Nested-Loops (BNL) [3], Sort-Filter-Skyline (SFS) [10] and Linear Elimination Sort for Skyline (LESS) [11] are three relevant algorithms for Skyline evaluation in relational settings. All of them use sequential scan to access the tuples. We extended BNL to evaluate Top-k Skyline queries in a relational DBMS. This extension is called Extended Block-Nested-Loops (EBNL). EBNL does not discard dominated tuples, moreover it stores them in a temporary file in case another stratum is needed. So, before generating a new stratum the algorithm checks whether there already are \(k\) or more tuples. We also extended SFS to evaluate relational Top-k Skyline queries, our SFS extension is called Extended Sort-Filter-Skyline (ESFS). For further details pertaining this algorithms, please refer to [4]. For our Top-k Skyline problem would not be advantageous to extend LESS, because it will only benefit from the Skyline ordering in the first stratum, while intermediate results for the succeeding strata will need to be sorted again. Note that intermediate results are stored as materialized tuples in temporary files with no particular order. Hence, multiple sorting phases will be required for LESS, in detriment of the its performance [2] and, in every sort phase for each necessary stratum, the algorithm could be sorting a huge amount of dominated tuples.

4.1 Design

For better understanding of the extension performed on PostgreSQL, we have elaborated some UML diagrams pertaining its executor, since most of the changes required for the implementation of the algorithms took place there. The Class Diagram for PostgreSQL preference based query executor is shown in Figure 4. Among the classes presented there, the main one is NodeProcessor, which accesses two data structures that rule query execution: the PlanTree and the ExecutionTree. The first consists of a hierarchy of PlanNodes. While, the second is composed of ExecutionStateNodes. Each ExecutionStateNode contains the data structures necessary for the execution of its corresponding physical operator. The ExecutionStateNode subclasses that are of interest for preference based queries are SkylineState and TopKSkylineState, given they correspond to Skyline and Top-k Skyline operators, respectively. These ExecutionStateNodes are used -as indicated by dashed arrows in the diagram- by SkylineProcessor and TopKSkylineProcessor. These classes are children of NodeProcessor that execute the aforementioned preference operators.

![Figure 4: PostgreSQL preference based query executor Class Diagram.](image)

On the other hand, Sequence Diagrams depict the interaction between class objects and facilitates the recognition of each class methods. To this end, we have designed a Sequence Diagram relative to the second phase of PostgreSQL preference based query executor, presented in Figure 5. The second phase executes the query until the result is achieved.

During the second phase of execution, the operators involved in the query are evaluated on-demand, this situation is represented by the loop and alt frames. The loop frame enables the iteration over PlanNodes that determines the execution sequence for the operators. While the alt frame verifies the type of each PlanNode in order to invoke the appropriate operator (processor) for it. Such operators make use of the information held within ExecutionStateNodes.
By the end of this phase, the result of the query has been calculated. If an Skyline PlanNode is encountered, the NodeProcessor invokes the executeSkyline() method from SkylineProcessor class. The execution of that method requires frequent calls to updateSkylineState() method, which allows the reading and updating of data structures within the SkylineStateNode. This happens analogously for all of the other PlanNodes in the PlanTree.

4.2 Physical Operators

The fundamental achievement of this paper is PostgreSQL extension to support preference based queries, in particular, Skyline and Top-k Skyline queries. To this purpose, two logical operators where incorporated to the DBMS, one for Skyline and one for Top-k Skyline. Each of these operators have two different physical implementations. Skyline is evaluated using BNL and SFS, as Top-k Skyline is evaluated by their extensions, namely EBNL and ESFS. According to Skyline and Top-k Skyline query semantics [3] [13], these preference operators are located at the root of the plan tree. Thus, they are evaluated after every other operator involved in the query.

PostgreSQL parser is distributed among three files: scan.c, gram.c and parse_clause.c, corresponding to the lexical, syntactical and semantic analyzer, respectively. For PostgreSQL to acknowledge SKYLINE OF and STOP AFTER clauses, it was necessary to include them in keywords.c, the file that contains all the keywords recognized by the grammar. This file constitutes the input for Flex, the lexical analyzer generator. The changes required for the validation of the syntax of our new clauses took place in gram.y. This file holds the definition of the grammar and is input for Bison, the syntactical analyzer generator. At the semantic analyzer, we added routines that perform type checking for the multi-criteria and score functions of the new clauses. The file analyze.c runs the creation of the parse tree by invoking appropriate semantic analyzer routines.
PostgreSQL rewriter was not altered because we did not need to make special considerations for views
or rules associated to Skyline and Top-k Skyline queries.

PostgreSQL optimizer resides in planner.c, createplan.c and setrefs.c. The first file controls the planning
process. The second generates interesting plans. And, the third places operators in valid locations within
the plan hierarchy. In planner.c, we added the calls for the routines that generate plans for Skyline and
Top-k Skyline queries. Such routines are implemented in createplan.c. Finally, in setrefs.c we made sure
that preference operators where located at the root of the plans and, therefrom, of the plan tree.

PostgreSQL preference based query executor consists of three files. The first is execProcnode.c, that
rules the evaluation of queries and corresponds to class NodeProcessor. The other two are nodeSkyline.c
and nodeHybrid.c, that provide interface routines for initializing, executing and ending Skyline and Top-k
Skyline operators, respectively. These two files represent SkylineProcessor and TopKSkylineProcessor
classes. So, the aforementioned interface routines are: initiateSkyline(), executeSkyline() and endSkyline()
for the former, and initiateTopKSkyline(), executeTopKSkyline() and endTopKSkyline() for the latter, according
to the Class Diagram in Figure 4.

The Utilities package complements the executor with tupleskyline.c and tuplehybrid.c. These files contain
the implementation for Skyline BNL and SFS algorithms, as well as for, Top-k Skyline EBNL and ESFS.

Finally, the Nodes package comprises the files: parsenodes.h, plannodes.h and execnodes.h. These files
hold the definition of ParseNodes, PlanNodes and ExecutionStateNodes that form the different tree structures
used by the backend to evaluate queries.

5 Experimental Study

We have studied the performance of Skyline and Top-k Skyline queries on a relational setting. Our exper-
imental study was performed on PostgreSQL 8.1.4 and comprised experiments running over tables with
75,000 tuples. The tables contain an identifier and six real number columns that represent the scores; values
of numeric columns vary from 0 to 1. The execution environment consisted on a Sun Fire V240 with 2
UltraSPARC IIIi processors of 1503MHz, 2 GB of memory and an Ultra160 SCSI disk of 146 GB, running
SunOS 5.10.

Besides, the queries were generated randomly, characterized by the following properties: (a) there is only
one table in the FROM clause; (b) the attributes in the multi-criteria function and score function are chosen
randomly from the attributes of the table using a uniform distribution; (c) the directives for each attribute
of the multi-criteria function is selected randomly considering MIN and MAX; (d) the number of attributes
of the multi-criteria function is two, four and six; and (e) \( k \) corresponds to 3% of the data size.

5.1 Experiment 1: Physical Operator vs. Middleware and SQL Translated Query

To demonstrate the performance improvement gained by the Skyline algorithms implemented as part of the
query engine, we introduce Experiment 1.

Our first experiment aimed to measure the average runtime for 5 queries. These queries used 4-
dimensional multi-criteria functions. The purpose was to compare the performance of an Skyline algorithm
implemented as a middleware application\(^1\) against the the same algorithm implemented as a physical op-
erator of the core query engine, and, also, against the SQL translation for the same queries\(^2\). For this
experiment, the SFS evaluation algorithm was used. The queries were ran over a uniformly-distributed
75,000-tuple table.

Figure 6 presents the results drawn by this experiment, these are measured in \( 10^{-3} \) miliseconds. It can
be noted that runtimes for the queries when evaluated by the physical operator were much smaller than
the ones obtained when evaluated by the middleware or the SQL translation. Runtimes for the latter were
consistently high given these queries involve expensive operators as MINUS, theta-join and a self-join [3]. On
the other hand, middleware runtimes are affected, mainly, by connection time for every tuple requested from
the DBMS and the additional processing implied by the two software layers.

This results confirm that, for our study case, solutions integrated into the DBMS show better performance
than those external to it.

\(^1\)The algorithm was implemented using Java Language.
\(^2\)A Skyline query could be translated into a SQL query, for further details refer to [3].
5.2 Experiment 2: EBNL vs. ESFS

We introduce Experiment 2 to demonstrate performance properties of the Top-k Skyline algorithms, namely EBNL and ESFS. These algorithms were implemented as part of the query engine. We measured execution times for ten Top-k Skyline queries with 2-, 4- and 6-dimensional multi-criteria functions, over a table of 75,000 correlated tuples. The attribute values are correlated pair-wise with a 0.5 correlation coefficient. Results for this experiment are presented in Figure 7.

Note that ESFS shows significantly better performance than EBNL. This might be ascribed to the fact that EBNL could be inserting many dominated tuples in the window\(^3\), incrementing the number of comparisons and iterations necessary to calculate a stratum. Besides, EBNL replaces tuples in the window, confirming the influence these metrics hold over execution time.

Additionally, when the number of dimensions in the multi-criteria function increases, the growth rate for the execution time increases for both algorithms, given that Skyline size increases consistently. This

\(^3\)BNL might hold dominated tuples in the window, which are eventually replaced [3]. Since EBNL is a BNL extension, it inherits this behavior.
affects EBNL the most, because it performs more multi-criteria function comparisons due to the presence of dominated tuples in the window.

A special case stands out for 2-dimensional queries, where EBNL outperforms ESFS. EBNL benefits from the fact that the Skyline is small and sorting 75,000 tuples could impair ESFS.

Generally, ESFS is a good solution for Top-k Skyline queries.

Finally, we don’t present results for the performance comparison of BNL and SFS because such a study has already been published by Chomicki et al. in [10].

6 Conclusions

We have extended PostgreSQL with the possibility of specifying and evaluating preference based queries, in particular, Skyline and Top-k Skyline queries. For this work, it was necessary to modify some components of PostgreSQL backend. At the parser level, SKYLINE OF and STOP AFTER clauses were added to the grammar. At the optimizer, some changes were done to incorporate Skyline and Top-k Skyline operators at the root of the plan tree. Most of the modifications took place at the executor, where BNL and SFS algorithms, as well as their extensions, EBNL and ESFS, were implemented in order to evaluate Skyline and Top-k Skyline operators, respectively. This four algorithms were integrated into PostgreSQL as an attempt to improve performance for these queries. Our experimental study revealed that solutions integrated into a DBMS outperform those implemented as a middleware application. Also, ESFS showed acceptable execution times for Top-k Skyline queries. Finally, we propose as future work the extension of PostgreSQL optimizer to improve execution times for Skyline and Top-k Skyline queries.

References


