FAKULTÄT FÜR !NFORMATIK

On Extending PostgreSQL with the Skyline Operator

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Ing. Bakk.techn. Hannes Eder
Matrikelnummer 9521554

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Univ.Ass. Dr.rer.nat. Fang Wei

Wien, 27.01.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Ing. Bakk.techn. Hannes Eder
Lerchenfelderstrasse 138/2/29
1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27.01.2009

(Unterschrift)

Zusammenfassung

Im Bereich der multikriteriellen Optimierung ist das Design einer geeigneten Zielfunktion aus Anwendersicht eine Herausforderung. Der *Skyline Operator* filtert aus einer potentiell großen Datenmenge *interessante* Tupel heraus. Ein Tupel gehört genau dann zur *Skyline*, wenn es nicht durch ein anderes *dominiert* wird, d.h. es gibt kein Tupel, das in allen Kriterien zumindest gleich gut ist und in zumindest einem besser ist. Unabhängig wie die Präferenzen innerhalb der Attribute gewählt werden, sind nur jene Tupel, die unter einer *monotonen Scoring-Funktion* am besten bewertet werden, Teil der Skyline. Mit anderen Worten, die Skyline schließt alle Tupel aus, die niemand als Favorit hat. Das Konzept der Skyline ist auch unter dem Namen *Pareto Optimalität* und ihre Berechnung als *Maximum Vektor Problem* bekannt.

Ziel dieser Diplomarbeit ist PostgreSQL, ein Open Source relationales Datenbankmanagementsystem (RDBMS), um den Skyline Operator zu erweitern und Skyline Algorithmen im RDBMS Kontext zu evaluieren. Das Endziel ist eine Skyline-Abfrage-Optimierung zu entwickeln, die automatisch gute Abfragepläne bezüglich I/O, Zeit und Speicherverbrauch erstellt. Diese Arbeit ebnet den Boden für weitere Forschung und verpflanzt den Skyline Operator von standalone Implementierungen in sein natürliches Habitat, das RDBMS.

Unsere Implementierung bietet verschiedene *physische Operatoren*, um die Skyline zu berechnen, allen voran: BNL, SFS und eine Variante von LESS. Zusätzlich erweitern wir die Standardsyntax, um Semantik und verschiedene operationale Aspekte zu beeinflussen. Als Nebenresultat unserer Arbeit haben wir einen Fehler in der Originalversion von BNL entdeckt und liefern dazu eine korrigierte Version. Wir schlagen ein neues Einsatzgebiet für den Elimination Filter (EF) vor und zwar: BNL+EF. Diese Kombination stellt eine erhebliche Verbesserung gegenüber BNL dar.

Es ist eine bekannte Tatsache, dass die Performance von Skyline Abfragen von einer Reihe von Parametern abhängt. Aus umfangreichen Experimenten mit unserer Implementierung haben wir mehrere bemerkenswert einfache und nützliche Regeln abgeleitet, die nur schwer theoretisch zu gewinnen sind. Unsere Resultate helfen Heuristiken für die Skyline-Abfrage-Optimierung zu entwickeln und liefern einen Beitrag zum tieferen Verständnis der Skyline-Abfrage-Charakteristik.

Alle Resultate, der Source-Code und ein Web-Interface zum Testen unserer Implementierung sind auf `http://skyline.dbai.tuwien.ac.at/` verfügbar.

Abstract

In the realm of multi-criteria optimization designing an appropriate objective function is a challenging task from a user's perspective. The *skyline operator* filters out the *interesting* tuples from a potentially large dataset. A tuple belongs to the *skyline* if it is not *dominated* by any other tuple, i.e. there is no tuple which is at least as good as in all and better in at least one criteria. No matter how we weight our preferences along the attributes, only those tuples which score best under a *monotone scoring function* are part of the skyline. In other words, the skyline does not contain tuples which are nobody's favorite. The notion of skyline is also called *Pareto optimal set* and its computation *maximum vector problem*.

In this thesis we aim at extending PostgreSQL, an open source relational database management system (RDBMS), with the skyline operator and the evaluation of skyline algorithms in the RDBMS context, with the ultimate goal of building a skyline query optimizer to automatically generate a good query plan w.r.t. I/O, time, and memory consumption. This effort lays the ground for future work in this area and moves the skyline operator from standalone implementations to the habitat it belongs to: an RDBMS.

Our implementation provides several physical operators for computing the skyline, including: BNL, SFS, and a variant of LESS. In addition, we extend the standard syntax to influence the semantics and various operational aspects. As a byproduct of our work, we discovered a flaw in the original version of BNL and give a corrected version. We propose a new use case for the elimination filter (EF), namely: BNL+EF. It turns out that BNL+EF is a substantial improvement to BNL.

It is well known that the performance of skyline queries is sensitive to a number of parameters. Extensive experiments on skyline implementations helped us to discover several remarkably simple and useful rules, which are hard to obtain from theoretical investigations. Our findings are beneficial for developing heuristics for the skyline query optimization, and in the meantime provide some insight for a deeper understanding of the skyline query characteristics.

All results, the source code, and a web-interface to test-drive the implementation are available at: `http://skyline.dbai.tuwien.ac.at/`.

Acknowledgements

First and foremost my special thanks go to Reinhard Pichler and Fang Wei for supervising and proof-reading my thesis, especially to Fang, as she continued to supervise my thesis even after she had left the department. Then I would like to thank Katrin Seyr, Toni Pisjak and Markus Pichlmair for their great assistance in bringing `http://skyline.dbai.tuwien.ac.at` to live. Furthermore thanks to all people from the Database and Artificial Intelligence Group of Vienna University of Technology for supporting me cordially in various issues. The author likes to thank Donald Kossmann for providing the source code for the dataset generator. I also like to acknowledge the discussions I had with Park Godfrey about the question if the English language is ready for a new verb "to skyline". Furthermore I would like to thank Albin Bucek and Jens Kober from Österreichisches Bundesdenkmalamt (BDA) for providing me seven computers to run the experiments on. My thanks further goes to my friends Emmanouil "Manos" Paissios and Chris Roschger, for all the lively discussions, the support, and for proof-reading all the material, and to Nadine Zollpriester for also proof-reading my thesis. Last but not least I am deeply grateful for all the support I got from my family, from Wiebke, and our son Paul Isidor, who was born while I was working on this thesis. I love you all.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 General Introduction

In everyday-life we are used to making decisions based on multiple criteria, e.g. price, quality, and delivery time for products. Frequently there is no clear winner over all criteria, i.e. a high-priced product with high quality and a cheap product with lower quality. Surely we are not interested in a product with a high price and poor quality, we will later refer to this situation as *this product is dominated by another one*, as there are others which fulfill our criteria better. The products which are not dominated are more or less equally good based on our given criteria, and we need to apply additional criteria to come to a decision. These products are called *Pareto optimal* based on the given criteria. For this notion Börzsönyi et al. [2001] coined the term *skyline*. The associated relational operator is called *skyline operator*.

Figure 1.1: Viennese skyline (nightshot with long exposure time, taken by Franz Pflügl / Vienna)

When thinking of the term *skyline*, images like Figure 1.1 might come to our mind. The popularity of such photographs is illustrated by following figures: *Google images* returns $\approx$ 3,580,000 hits, not counting the hits for the Nissan's skyline sports car, for the search term *skyline -nissan*, and istockphoto.com, a popular website with a collection of member-generated royalty-free images, has ≈ 25,000 photos in stock for the search term *skyline*, as of May 23 2008. The Merriam-Webster's online dictionary gives the following definition for *skyline*:

1. the apparent juncture of earth and sky: *horizon*
2. an outline (as of buildings or a mountain range) against the background of the sky

From a more abstract point of view a skyline is made up of objects (buildings, mountains and such) which do not have another object closer and taller to the eye of the beholder. If we are only interested in the objects that are "touching the sky", we consider only the tallest object in each direction of view. On the other hand if we care about visibility, an object is visible as long as there is no object closer and taller. We ask the alert reader to excuse the simplifications we made here, the authors are aware of the concept of perspective projection and of the fact that the earth is not a disk.

In the realm of database systems the *skyline operator* is important for supporting multi-criteria decision making applications. Given a set of d-dimensional data points, the *skyline* consists of those points, called *skyline points*, which are not *dominated* by any other data points. One data point r dominates another data point s if it is at least as good as s in all dimensions and better in at least one. Börzsönyi et al. [2001] proposed a *skyline* extension of the SQL syntax with this form:

```
SELECT ...  FROM ...  WHERE ...
GROUP BY ...  HAVING ...
SKYLINE OF [DISTINCT] a_1 [MIN|MAX|DIFF], ..., a_m [MIN|MAX|DIFF]
ORDER BY ...                                                      (1.1)
```

Computing the skyline is also known as the *maximum vector problem* [Kung et al., 1975; Preparata and Shamos, 1985]. The terms *skyline* and *skyline operator* in the realm of databases were coined by Börzsönyi et al. [2001]. The following examples give a motivation why *skyline operator* is a good name for this type of *preference queries*, they also illustrate the usage of the DIFF and DISTINCT keywords.

In Figure 1.2 the yellow arrow indicates the direction of view and the colored cuboids illustrate buildings. Table 1.1 shows the coordinates of the lower left edge of each building and gives them names $(a, b, \ldots)$. The same color indicates the same x coordinate. For y we have only two cases, *front* and *back*.

Example 1 (Which buildings are touching the sky?). *For sure building c (the green one in the background) and building f (the blue one in the foreground) are part of the answer and d and e are not. As a and b do have the same height, we could take them both or make an arbitrary choice among a and b. We will later see that this taking both or choosing an arbitrary one can be influenced by the keyword* DISTINCT. *So overall, possible answers are $\{a, c, f\}$, $\{b, c, f\}$, or $\{a, b, c, f\}$.*

Example 2 (Which buildings are visible?). *To answer this question we also have to take into account how close a building is, i.e. the y coordinate. Building a is covered by b and e by f, we will later call this property* dominance. *It is obvious that c and d are part of the answer. The complete answer is $\{b, c, d, f\}$.*

The question from Example 1 could be formalized in the following way:

```
SELECT * FROM building SKYLINE OF x DIFF, z MAX;
```

In the above query a and b will be part of the answer, whereas in the following one it is up to the database system to make a choice between a or b, as they are equally good:

```
SELECT * FROM building SKYLINE OF DISTINCT x DIFF, z MAX;
```

On the other hand the query for Example 2 looks like:

```
SELECT * FROM building SKYLINE OF x DIFF, y MIN, z MAX;
```

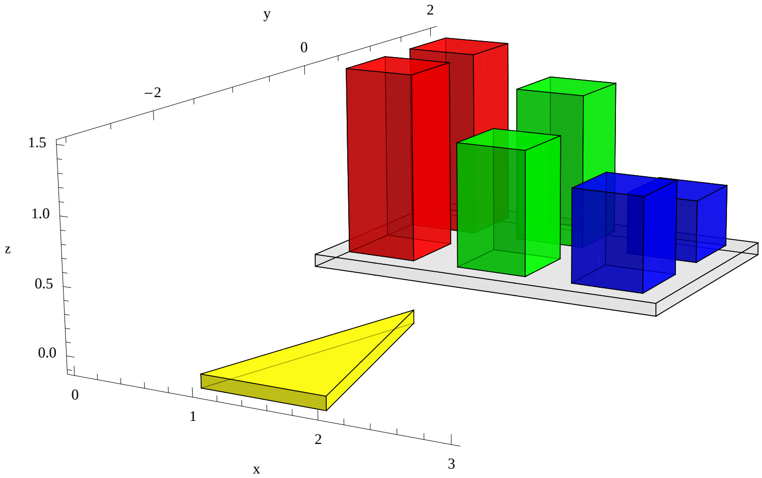


Figure 1.2: Skyline of buildings

id	x	y	z	color	row
a	0	1	1.5	red	back
b	0	0	1.5	red	front
c	1	1	1.25	green	back
d	1	0	1.0	green	front
e	2	1	0.5	blue	back
f	2	0	0.75	blue	front

Table 1.1: Data for Figure 1.2

1.2 Why extending SQL?

The *skyline operator* [Börzsönyi et al., 2001] is within the expressive power of SQL and in section 2.5 we show how to rewrite an SQL query with a `SKYLINE OF`-clause into standard SQL. However there are good reasons for including a *skyline* or *preference operator* into SQL and building it into an RDBMS:

- From a user's perspective, it is easier to specify and easier to grasp what's going on than a query with a correlated subquery or using SQL's `EXCEPT` (see Equation 2.18).

- It is faster to evaluate, as there are numerous algorithms for skyline queries. Furthermore when this preference query is expressed with a correlated subselect it boils down to a naïve "nested-loop". Grust et al. [1997]; Braumandl et al. [1998] show that such a query cannot be unnested.

- The study of preference queries on their own gave birth to a new set of optimization opportunities [Chomicki, 2003; Kießling and Hafenrichter, 2003].

- The skyline operator can be integrated with other relational operators, thus more sophisticated queries can be constructed. Moreover, no work on concurrency control and transaction management has to be addressed.
- The existing index structures such as R-trees in an RDBMS can be adapted to implement index-based skyline algorithms.
- If several skyline algorithms are available in an RDBMS, the system is able to select the most efficient one for the given datasets.

It is well known that the cost estimation of the skyline queries is a non-trivial task [Chaudhuri et al., 2006] since the performance of a skyline query is sensitive to a number of parameters [Godfrey et al., 2007]. Hence building a skyline query optimizer is a challenging task.

1.3 Goal and Main Results

In this thesis we aim at extending PostgreSQL with the skyline operator and the evaluation of skyline algorithms in the RDBMS context, with the ultimate goal of building a skyline query optimizer to automatically generate a good query plan w.r.t. I/O, time, and memory consumption.

We wish to acquaint the skyline operator to a broader audience and lay the ground for future work in this area by providing an open source implemenation in an RDBMS, which is the natural habitat for the skyline operator.

The scope of this thesis is restricted to non-index-based skyline algorithms. Dedicated algorithms have been proposed for the case where the whole relation fits into main memory [Preparata and Shamos, 1985], nevertheless we do not take them into account as memory is always a rare resource in a database scenario, either due to the number of concurrent users or limited total main memory, e.g. on a handheld device.

So far we have implemented the non-index-based algorithms *Block Nested Loop* (BNL), *Sort First Skyline* (SFS), and a variant of *Linear Elimination Sort for Skyline* (LESS) in PostgreSQL. We propose a new use case for the elimination filter (EF), namely: BNL+EF. It turns out that BNL+EF is a substantial improvement to BNL. In addition, we extend the standard syntax to specify the following aspects:

- The treatment of NULL values (`NULLS FIRST` and `NULLS LAST`)
- The usage of order relations other than < and > (`USING` *`Op`*)
- Operational aspects of skyline computation, such as
  - *method* (BNL, SFS)
  - *tuple window size* in terms of memory and/or number of slots
  - *tuple window policy* (append, prepend, entropy, random)
  - *usage of indexes* (`NOINDEX`)
  - *usage of elimination filter (EF)*

We prove that the original version of BNL does not terminate in all cases and give a corrected version.

To achieve the goal of building a skyline query optimizer, we conducted extensive experiments on the skyline operator implementations. The experimental environment we have designed and set up is noteworthy on its own. Our experiments confirm the result from

Börzsönyi et al. [2001], that specialized algorithms are faster than the same queries in standard SQL by orders of magnitude.

The comparisons of the skyline algorithms we have so far implemented reveal that there does not exist a clear winner in all aspects. The efficiency of the algorithms depends on the properties of the datasets such as dimension, distribution, cardinality and so on.

However we discovered several *hidden rules*, which are remarkably simple and useful, but hard to obtain from the theoretical investigation. We expect that our exposition of experimental results on skyline algorithms could serve as a guideline for developing heuristics of a skyline query optimizer, and in the meantime provide some insight for a deeper understanding of the skyline query characteristics.

- For instance, the elimination filter is effective only if the selectivity factor for the skyline query is not more than 0.1, and
- BNL is more efficient than SFS and even LESS, if the dimension of the data is less than 5 and the number of tuples is relatively small.

For the purpose of repeatability we have set up a website to present our experimental results. The source code for our implementation and the log-files from our experiments are online accessible at this website. Furthermore it is possible to test-drive our implementation through a web-interface. All this is available at: `http://skyline.dbai.tuwien.ac.at/`.

1.4 Further Organization

The thesis is organized as follows: In chapter 2 we introduce the notation and show mathematical properties of the skyline operator. In chapter 3 we briefly introduce the implemented skyline algorithms and speak about the closest related works. Chapter 4 gives the very details of our implementation. Chapter 5 describes the experimental setup and presents the extensive experiments over various data settings, followed by the analysis of the results. Directions of future work are discussed in chapter 6. Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

2.1 Typographical Conventions

For interactive input and output we will show our typed input in a **`bold font`**, the output `like this`, *and comments like this.*

```
$ psql                                              start PostgreSQL interactive terminal
db=# SELECT * FROM a2d1e5s0 SKYLINE OF d1 MIN, d2 MIN;          issue a skyline query
  id   |          d1           |          d2
-------+-----------------------+-----------------------
   417 |     0.660430708919594 |    0.0734207719527077
  1329 |    0.0486199019148477 |     0.663867116113964
[...]                                                                53 rows omitted
 98953 |     0.136247931652335 |     0.607638615081091
(56 rows)

db=# EXPLAIN ANALYZE SELECT * FROM a2d1e5s0 ↩    long lines are wrapped with ↩ and →
  → SKYLINE OF d1 MIN, d2 MIN;
[...]                                          omitted output is indicated with [...]

db=# \q                                                                     quit psql
$                                                          back at the shell prompt
```

The example above also illustrates two other concepts: "`$`" indicates a shell prompt and "`db=#`" indicates the prompt for PostgreSQL interactive terminal `psql`. Furthermore "`db-#`" is displayed when `psql` is waiting for more input. All of them are used throughout the text.

As long as it did not change the semantics of the input or output we took the freedom to add and remove whitespace in order to increase readability, to emphasize a certain point, or just to work around the 80 columns limitation. Where extra long lines are wrapped we inserted "↩" before the break and "→" after the break. Furthermore at certain points we even allowed ourselves to strip some output, we indicated such omissions by an ellipsis ("[...]").

2.1.1 Pseudo-code

In the real code for our implementation we do concern about issues of software engineering, like error handling, which we ignore in the pseudo-code to convey the essence of the algorithms more concisely.

For the presentation we adapt the convention in [Cormen et al., 2001, Page 19], in the very essence i.e.:

- No **begin**/**end** or curly brackets for blocks, we use *indentation* to express the *block structure*, like in Python or similar languages.
- We do explicitly **break** at the end of **case** blocks in a **switch**-statement, because in some rare situations we *fall through* to the next case. Nevertheless we omit the **break** if it is unreachable, e.g. because it is preceded by a **return**-statement.

Furthermore we took the freedom to simplify some function names and remove some nasty details which are not essential to grasp the algorithms.

2.2 Basic definitions

We are in the realm of the relational model of data. As we aim at an implementation of the skyline operator into a relational database management system (RDBMS) we restrict ourselves to finite database instances.

We assume two infinite[1] domains: N (numbers) and D (uninterpreted constants). The domain D will be used for attributes which are not subject to skyline computation, i.e. for the hotel example from literature we will not use the name of the hotel as a skyline criterion, therefore D will be used as domain for this attribute. For attributes or expressions which are subject to skyline computation we use the domain N. No distinction is made between different numeric domains, since it is not necessary for this thesis. For N we require that equality ($=$), inequality ($\neq$), and the binary relations $<$ (strictly less than), $\leq$ (less than or equal), $\geq$ (greater than or equal), and $>$ (strictly greater than) are defined with the usual properties.

As the skyline operator is a special case of the *winnow operator* [Chomicki, 2003], we define it in a very similar way.

Definition 1 (Preference Relation [Chomicki, 2003]). *Given the domains $U_i, 1 \leq i \leq n$, such that U_i is either equal to D or N and the n-ary schema $\mathcal{R}(a_1, a_2, \ldots, a_n)$, such that the domain of the attribute a_i is U_i, a relation $\rhd$ is a* preference relation *over $\mathcal{R}$ if it is a subset of $(U_1 \times U_2 \times \cdots \times U_n) \times (U_1 \times U_2 \times \cdots \times U_n)$.*

To give an intuition, $\rhd$ is a binary relation between pairs of tuples from the same database relation $\mathbf{R}$ with schema $\mathcal{R}$. We say r *dominates* s in $\mathcal{R}$ iff $(r, s) \in \rhd$ (or in infix notation $r \rhd s$). We use the term database relation in the usual sense, a (database) relation $\mathbf{R}$ is a finite instance of a schema $\mathcal{R}$.

We require the relation $\rhd$ to be a *strict partial order*, i.e. $\rhd$ is irreflexive, asymmetric and transitive. These properties are formalized as usual:

- *irreflexivity:* $\forall x : \neg(x \rhd x)$
- *asymmetry:* $\forall x, y : x \rhd y \Rightarrow \neg(y \rhd x)$
- *transitivity:* $\forall x, y, z : (x \rhd y \wedge y \rhd z) \Rightarrow x \rhd z$

The properties for the corresponding *partial order* $\unrhd$ are formalized as:

- *reflexivity:* $\forall x : x \unrhd x$
- *anti-symmetry:* $\forall x, y : x \unrhd y \wedge y \unrhd x \Rightarrow x \overset{\unrhd}{=} y$
- *transitivity:* $\forall x, y, z : (x \unrhd y \wedge y \unrhd z) \Rightarrow x \unrhd z$

[1] Although in a real computer everything is finite.

If a strict partial order also has the property of

- *totality:* $\forall x, y : x \rhd y \vee y \rhd x \vee x = y$,

then it is a *total order.*

Definition 2 (preference formula (pf) [Chomicki, 2003])**.** *A* preference formula (pf) $C(r, s)$ *is a first-order formula defining a preference relation* $\rhd$ *in the standard sense, namely*

$$r \rhd s \quad \text{iff} \quad C(r, s).$$

Because of this close relationship between the first-order formula C and the binary relation $\rhd$ we will later use skyline$_{\rhd}$ and skyline$_C$ interchangeably.

In our case we only focus on *intrinsic* preference formulas, i.e. the value of $C(r, s)$ only depends on the attributes of r and s and not on other tuples in the same or other relations. If the preference formula depends on other tuples, like in *"we prefer tuples that are better than the average"*, which involves aggregation, it is called *extrinsic* preference formula. We now define a special form of preference formula.

Definition 3 (Skyline Preference Formula)**.** *Given the two tuples*

$$r = (\underbrace{r_1, \ldots, r_k}_{\texttt{MIN}}, \underbrace{r_{k+1}, \ldots, r_l}_{\texttt{MAX}}, \underbrace{r_{l+1}, \ldots, r_m}_{\texttt{DIFF}}, \underbrace{r_{m+1}, \ldots, r_n}_{extra})$$

and

$$s = (\underbrace{s_1, \ldots, s_k}_{\texttt{MIN}}, \underbrace{s_{k+1}, \ldots, s_l}_{\texttt{MAX}}, \underbrace{s_{l+1}, \ldots, s_m}_{\texttt{DIFF}}, \underbrace{s_{m+1}, \ldots, s_n}_{extra})$$

both elements of an n-ary relation **R***, then the* skyline preference formula *is of the form:*

$$\begin{aligned} C(r, s) = {} & \overbrace{\left(\bigwedge_{1 \le i \le k} r_i \le s_i \right)}^{\text{“\texttt{MIN} at least as good”}} \wedge \overbrace{\left(\bigwedge_{k+1 \le i \le l} r_i \ge s_i \right)}^{\text{“\texttt{MAX} at least as good”}} \wedge \overbrace{\left(\bigwedge_{l+1 \le i \le m} r_i = s_i \right)}^{\text{“\texttt{DIFF} equal”}} \wedge \\ & \wedge \left(\underbrace{\left(\bigvee_{1 \le i \le k} r_i < s_i \right)}_{\text{“\texttt{MIN} better than”}} \vee \underbrace{\left(\bigvee_{k+1 \le i \le l} r_i > s_i \right)}_{\text{“\texttt{MAX} better than”}} \right). \end{aligned} \tag{2.1}$$

To state (2.1) in an informal way, a tuple r dominates a tuple s if it is equal in all `DIFF` dimensions, at least as good in all `MIN`/`MAX` dimensions and better than in at least one of the `MIN`/`MAX` dimensions. The above formula corresponds to a skyline query with the following skyline clause:

$$\texttt{SKYLINE OF } a_1\,\texttt{MIN}, \ldots, a_k\,\texttt{MIN}, a_{k+1}\,\texttt{MAX}, \ldots, a_l\,\texttt{MAX}, a_{l+1}\,\texttt{DIFF}, \ldots, a_m\,\texttt{DIFF}$$

To simplify the definition of formula (2.1) and combine the "`MIN`- and `MAX`-at least as good" condition to a just "at least as good" condition we define:

$$\succeq_i := \begin{cases} \le & \text{if } 1 \le i \le k, \\ \ge & \text{if } k+1 \le i \le l, \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.2}$$

Same for the "better than" condition:

$$\succ_i := \begin{cases} < & \text{if } 1 \leq i \leq k, \\ > & \text{if } k+1 \leq i \leq l, \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.3}$$

Using the above two definitions formula (2.1) simplifies to:

$$r \rhd s := \underbrace{\left(\bigwedge_{l+1 \leq i \leq m} r_i = s_i \right)}_{\text{“}\mathtt{DIFF}\textit{ equal”}} \wedge \underbrace{\left(\bigwedge_{1 \leq i \leq l} r_i \succeq_i s_i \right)}_{\textit{“at least as good”}} \wedge \underbrace{\left(\bigvee_{1 \leq i \leq l} r_i \succ_i s_i \right)}_{\textit{“better than”}} \tag{2.4}$$

We now define the *skyline operator* which is a special case of the *winnow operator* [Chomicki, 2003], where the preference formula has exactly the form of formula (2.1).

Definition 4 (Skyline Operator)**.** *Let $\mathcal{R}$ be a relation schema and C a skyline preference formula defining a preference relation $\rhd$ over $\mathcal{R}$, then the* skyline operator *is denoted by* skyline$_\rhd(\mathcal{R})$*, and for every instance $\mathbf{R} \in \mathcal{R}$:*

$$\text{skyline}_\rhd(\mathbf{R}) := \{ r \in \mathbf{R} | \nexists s \in \mathbf{R} \colon s \rhd r \}$$

To give the intuition, the skyline operator returns all tuples which are *not dominated*, in other words all tuples that do *not* have a *witness*.

We now like to introduce some further concepts:

Definition 5 (Weak Preference)**.** *There is a weak preference between r and s if r is equal in all* `DIFF` *dimensions and at least as good in all other skyline dimensions compared to s, formally:*

$$r \unrhd s := \underbrace{\left(\bigwedge_{l+1 \leq i \leq m} r_i = s_i \right)}_{\text{“}\mathtt{DIFF}\textit{ equal”}} \wedge \underbrace{\left(\bigwedge_{1 \leq i \leq l} r_i \succeq_i s_i \right)}_{\textit{“at least as good”}} \tag{2.5}$$

Definition 6 (Non Distinct)**.** *If two tuples r and s are equal on all skyline dimensions, we say r and s are* non distinct*, denoted by:*

$$r \overset{\rhd}{=} s := \left(\bigwedge_{1 \leq i \leq m} r_i = s_i \right) \tag{2.6}$$

If $r \overset{\rhd}{=} s$ and in case of `SKYLINE OF DISTINCT`, then it is left up to the implementation to include either r or s in the skyline, otherwise r *and* s are included. Please note that in general $r = s$ is not equivalent to $r \overset{\rhd}{=} s$, only when $m = n$, i.e. all attributes are subject to skyline computation.

For convenience we define the following two *commutator* relations:

$$s \lhd r :\Leftrightarrow r \rhd s.$$

$$s \unlhd r :\Leftrightarrow r \unrhd s.$$

Without proof we give the following equivalences:

$$r \trianglerighteq s \Leftrightarrow r \triangleright s \vee r \overset{\triangleright}{=} s \tag{2.7}$$

$$r \triangleright s \Leftrightarrow r \trianglerighteq s \wedge \neg(r \overset{\triangleright}{=} s) \tag{2.8}$$

$$r \overset{\triangleright}{=} s \Leftrightarrow r \trianglerighteq s \wedge r \trianglelefteq s \tag{2.9}$$

Definition 7 (Incomparable)**.** *If neither r dominates s nor s dominates r then r and s are* incomparable*:*

$$r \parallel s := \neg(r \triangleright s) \wedge \neg(s \triangleright r)$$

In case of $r \parallel s$, r and s may both be part of the skyline, if none of them is dominated by another tuple $t \in \mathbf{R} \backslash \{r, s\}$. The commutativity of $\parallel$ is obvious from its definition, i.e.

$$r \parallel s = s \parallel r.$$

Definition 8 (Comparable)**.** *The tuples r and s are* comparable *iff r dominates s or s dominates r:*

$$r \perp s := r \triangleright s \vee s \triangleright r$$

It is clear from the definition that $\perp$ is commutative, i.e.

$$r \perp s = s \perp r.$$

2.2.1 `DIFF`-Induced Equivalence Relation

Please note that the following subformula (2.10) of formula (2.1) induces an equivalence relation on $\mathcal{R}$, i.e. $\mathcal{R}$ is partitioned into groups where the attributes $a_{l+1}, \ldots, a_m$ are equal:

$$\bigwedge_{l+1 \leq i \leq m} r_i = s_i. \tag{2.10}$$

As noted in [Chomicki et al., 2003], the `DIFF` directive works like a `GROUP BY` within the skyline, and the skyline for each group of `DIFF` attributes' values is found. This property can be exploited, if an ordered index on any subset of the attributes $a_{l+1}, \ldots, a_m$ exists, since the tuple window can be flushed each time the group is changed, see section 6.3.4

2.3 Properties of the Skyline Operator

2.3.1 Monotone vs. Linear Scoring Functions

One might be tempted to assume that the result of the skyline operator can also be produced with linear scoring functions, but that is not true, we take the example from [Chomicki et al., 2002] to show this. Before going into detail let us define the class of linear scoring functions:

Definition 9 (Linear Scoring Function)**.** *Let the w_i's be positive real constants, then we define a* positive linear scoring function *W over all the tuples $r \in \mathbf{R}$ as:*

$$W(r) = \sum_{1 \leq i \leq n} w_i r_i.$$

Let us consider the relation $\mathbf{R} = \{(4,1), (2,2), (1,4)\}$. It is clear that all tuples are part of the skyline, i.e. skyline($\mathbf{R}$) = $\mathbf{R}$. Finding a linear scoring function that prefers $(4,1)$ or $(1,4)$ is obvious. Now let us try to find a linear scoring function that gives $(2,2)$ the highest score. We require that:

1. $2w_1 + 2w_2 \geq 4w_1 + 1w_2$, which simplifies to $w_2 \geq 2w_1$, which in turn is equivalent to $2w_2 \geq 4w_1$, and
2. $2w_1 + 2w_2 \geq 1w_1 + 4w_2$, which simplifies to $w_1 \geq 2w_2$

Chaining the inequalities from (1) and (2) yields $w_1 \geq 2w_2 \geq 4w_1$, i.e. $w_1 \geq 4w_1$, which does not have a solution for $w_1 > 0$. Therefore there is no linear scoring function which gives $(2, 2)$ the best score.

It should be noted that it might be difficult to come up with appropriate weights for the different attributes from a user's perspective.

On the other hand the class of monotone scoring functions is defined as follows:

Definition 10 (Monotone Scoring Function)**.** *Let the f_i's be monotone increasing functions with $f_i \colon N \to \mathbb{R}$, where N is the domain of the attributes of schema $\mathcal{R}$, then a* monotone scoring function *S which ranks a tuple $r \in \mathbf{R}$, where $\mathbf{R}$ is an instance of $\mathcal{R}$, has the form:*

$$S(r) = \sum_{1 \leq i \leq n} f_i(r_i).$$

A nice property of the skyline operator is stated in the following theorem:

Theorem 1 ([Chomicki et al., 2002])**.** *The skyline contains all, and only tuples yielding maximum values of monotone scoring functions.*

One of the nice conclusions of Theorem 1 is, no matter how we weight our preferences along the attributes, our favorite is part of the skyline, as long as we weight each attribute with a monotone increasing function. Furthermore only tuples which score best under at least one scoring function are part of the skyline, in other words, the skyline does not contain tuples which are nobody's favorite.

We now give an example to emphasize our assertion, that it is possible to give the tuple $(2, 2)$ a higher score than the tuples $(4, 1)$ and $(1, 4)$. Using the monotone increasing functions

$$f_1(x) = f_2(x) = \begin{cases} 0 & \text{if } x < 2, \\ 1 & \text{otherwise} \end{cases}$$

we define the monotone scoring function

$$S(r) = f_1(r_1) + f_2(r_2),$$

which has the asserted property:

1. $S(\,(2, 2)\,) = f_1(2) + f_2(2) = 1 + 1 = 2 > 1 = 1 + 0 = f_1(4) + f_2(1) = S(\,(4, 1)\,)$, and
2. $S(\,(2, 2)\,) = f_1(2) + f_2(2) = 1 + 1 = 2 > 1 = 0 + 1 = f_1(1) + f_2(4) = S(\,(1, 4)\,)$.

2.3.2 Independence of Attribute Order

The following property is a well known fact about the skyline operator:

Property 1 (Independence of Attribute Order)**.** *The semantics of the skyline operator is independent from the order in which the attributes are specified in the query.*

Proof. This property immediately becomes clear, when looking at the definition of the *preference formula* (2.1), as logical and ($\wedge$) and logical or ($\vee$) are commutative. □

To give a simple example the following queries will yield the same result:

```
SELECT * FROM a15d1e5 SKYLINE OF d1 MIN, d2 MIN;
SELECT * FROM a15d1e5 SKYLINE OF d2 MIN, d1 MIN;
```

2.4 Further Notions

2.4.1 Dominance and Anti-dominance regions

In the context of the skyline operator the following two notions naturally arise:

Definition 11 (Dominance Region). *The Dominance Region $DR_{\triangleright}(r)$ of a tuple r of the relation $\mathbf{R}$ is the set of tuples in $\mathbf{R}$ that are dominated by r, formally:*

$$DR_{\triangleright}(r) := \{x \in \mathbf{R} | r \triangleright x\}$$

Definition 12 (Anti-Dominance Region). *The Anti-Dominance Region $ADR_{\triangleright}(r)$ of a tuple r of the relation $\mathbf{R}$ is the set of all tuples in the relation $\mathbf{R}$ that dominate r, formally:*

$$ADR_{\triangleright}(r) := \{x \in \mathbf{R} | x \triangleright r\}$$

The question if a point is a skyline point can be answered by checking if the anti-dominance region is empty. This can be done with a range query, or even facilitating an index.

2.4.2 Skyline Stratum and K-Skyband

In [Chomicki, 2003] the concept of skyline stratum is called *iterated preference*, and they speak of *ranking*, we prefer to use the term *stratum*, and we reserve the term ranking for scoring functions (see section 2.3.1). To the best of our knowledge the term *skyline stratum* first appeared in [Chan et al., 2005]. The n-th skyline stratum skyline_C^n is recursively defined as:

$$\text{skyline}_C^1(\mathbf{R}) := \text{skyline}_C(\mathbf{R}) \quad (2.11)$$

$$\text{skyline}_C^{n+1}(\mathbf{R}) := \text{skyline}_C(\mathbf{R} \setminus \bigcup_{1 \leq i \leq n} \text{skyline}_C^i(\mathbf{R})) \quad (2.12)$$

To give an example, the query $\text{skyline}_C^2(\mathbf{R})$ returns the set of "second-best" tuples. The strata are of interest especially if there is a single *killer tuple* or very few tuples which are dominating the entire dataset. In the context of the relation *Hotel(Price, Distance to beach)*, a barrack on the beach is cheaper and closer to the beach than any other accommodation, but might not meet our standards. Another example, if we are bored always having lunch at the same "best" restaurant, we might want to try the "second-best" and enjoy the little bit longer walk we would have to take in that case.

A *K-skyband* query, introduced by Papadias et al. [2005], is a concept similar to the skyline stratum. A K-skyband query reports all tuples which are dominated by *at most* K points. The skyband rank of a point p can be computed by counting the points in the anti-dominance region of p.

It might be tempting to see a direct relationship between *skyline stratum* and *K-skyband*, but as can be seen from the following example, there is no straightforward relationship between these two concepts:
Let $\mathbf{R} = \{r, s, t\}$ and $\triangleright = \{(r,t),(s,t)\}$, i.e. $r \triangleright t$ and $s \triangleright t$, then $\text{skyline}_{\triangleright}^1 = \{r, s\}$ and $\text{skyline}_{\triangleright}^2 = \{t\}$, but $\text{skyband}_{\triangleright}^0 = \{r, s\}$, $\text{skyband}_{\triangleright}^1 = \{r, s\}$ and $\text{skyband}_{\triangleright}^2 = \{r, s, t\}$.
Hence no easy general relation can be established between K-skyband and skyline stratum, which is intuitive as K-skyband involves counting while skyline stratum involves just an existential quantifier. So the following holds:

$$\text{skyband}^0_C(\mathbf{R}) = \text{skyline}^1_C(\mathbf{R}) \tag{2.13}$$

$$\text{skyband}^n_C(\mathbf{R}) \neq \bigcup_{1\leq i\leq n+1} \text{skyline}^i_C(\mathbf{R}) \quad \text{for } n \geq 0 \tag{2.14}$$

$$\text{skyline}^n_C(\mathbf{R}) \neq \text{skyband}^{n+1}_C(\mathbf{R}) \setminus \text{skyband}^n_C(\mathbf{R}) \quad \text{for } n \geq 1 \tag{2.15}$$

The branch-and-bound-skyline (BBS) algorithm [Papadias et al., 2005] supports the computation of K-skyband queries, as stated by Papadias et al. [2005], BNL and SFS can compute K-skybands, this can be easily done by maintaining the number of tuples that dominate a tuple in the tuple window. Increase the number by one each time a tuple in the tuple window is dominated by the current tuple and once this count is greater than K drop the tuple from the tuple window.

An interesting combination of `SKYLINE OF` and `TOP` k is given by the *top-k-skyline* operator [Brando et al., 2007; Goncalves and Vidal, 2005b,a], where as many skyline strata are computed until at least k tuples are found, otherwise the entire relation is returned. Nevertheless it should be noted that the selectivity of the top-k-skyline operator is even less than the skyline operator alone.

2.5 Rewrite Skyline Queries into Standard SQL

Let *rel* be a relation with the attributes $a_1, \ldots, a_n$, and a non empty target list $\emptyset \neq \mathit{target\_list} \subseteq \{a_1, \ldots, a_n\}$, then a skyline query with at least one criterion ($0 \leq k \leq l \leq m \leq n$ and $1 \leq m \leq n$) and the following form:

```
SELECT target_list FROM rel AS r WHERE condition
SKYLINE OF a_1 MIN,...,a_k MIN,a_{k+1} MAX,...,a_l MAX,a_{l+1} DIFF,...,a_m DIFF  (2.16)
```

can be rewritten, as exemplified in [Börzsönyi et al., 2001], into

```
SELECT o.target_list FROM rel AS o WHERE (condition{r ← o}) AND NOT EXISTS
  (SELECT * FROM rel AS i WHERE (condition{r ← i})
    AND i.a_1     <= o.a_1     AND ... AND i.a_k  <= o.a_k      -- "MIN" as good[2]
    AND i.a_{k+1} >= o.a_{k+1} AND ... AND i.a_l  >= o.a_l      -- "MAX" as good
    AND i.a_{l+1}  = o.a_m     AND ... AND i.a_m   = o.a_m      -- "DIFF" equal
    AND (
          i.a_1     < o.a_1     OR ... OR i.a_k  < o.a_k        -- "MIN" better
       OR i.a_{k+1} > o.a_{k+1} OR ... OR i.a_l  > o.a_l        -- "MAX" better
    )
  )                                                              (2.17)
```

Note that the attributes specified as criteria in the skyline query need not necessarily appear in the *target_list*. The correspondence with the basic definition of the skyline operator (cf. Definition 4) and with the definition of a binary preference relation (2.1) is obvious. The join is a θ-join and not an equi-join. A little notion we use here is $\mathit{condition}\{r \leftarrow o\}$ to express that any reference to the relation alias r must be replaced with the alias o. Another way to rewrite the query is given in [Godfrey, 2004, Page 3]:

```
SELECT target_list FROM rel WHERE (condition)
EXCEPT
SELECT o.target_list FROM rel AS i, rel AS o
WHERE (condition{r ← i}) AND (condition{r ← o})
```

[2] Note that "--" starts a single line comment in SQL.

```
AND $i.a_1$      <= $o.a_1$      AND ... AND $i.a_k$   <= $o.a_k$       -- "MIN" as good
AND $i.a_{k+1}$  >= $o.a_{k+1}$  AND ... AND $i.a_l$   >= $o.a_l$       -- "MAX" as good
AND $i.a_{l+1}$   = $o.a_m$      AND ... AND $i.a_m$    = $o.a_m$       -- "DIFF" equal
AND (
      $i.a_1$     < $o.a_1$      OR ... OR $i.a_k$   < $o.a_k$          -- "MIN" better
   OR $i.a_{k+1}$ > $o.a_{k+1}$  OR ... OR $i.a_l$   > $o.a_l$          -- "MAX" better
)
```
(2.18)

The principle used here is different, the skyline of a set is the set without those tuples that have a *witness* that they are dominated. This can be formally expressed as:

$$\text{skyline}_{\rhd}(\mathbf{R}) = \mathbf{R} \backslash \{r \in \mathbf{R} | \exists s \in \mathbf{R}\colon s \rhd r\} \tag{2.19}$$

Computing the skyline this way is expensive, as the intermediate results can get very large.

2.5.1 Rewrite queries with SKYLINE OF DISTINCT

In the presence of the optional DISTINCT modifier in the SKYLINE OF-clause the situation is changed, such a query has the following form:

```
SELECT $target\_list$ FROM $rel$ AS $r$ WHERE $condition$
SKYLINE OF **DISTINCT** $a_1$ MIN, ..., $a_k$ MIN, $a_{k+1}$ MAX, ..., $a_l$ MAX,
  $a_{l+1}$ DIFF, ..., $a_m$ DIFF
```
(2.20)

Before we describe how to rewrite such a query into standard SQL and where the limits of this transformation are, we shall repeat what the intended semantics of SKYLINE OF DISTINCT is from [Börzsönyi et al., 2001]. If the DISTINCT modifier is present, then duplicates are eliminated in the following way: if for two tuples r and s the condition

$$\bigwedge_{1 \leq i \leq m} r_i = s_i,$$

(i.e. they are equal on all skyline criteria, in other terms $r \overset{\unrhd}{=} s$ (cf. (2.6))) holds, then either r or s is retained, and the choice is left to the implementation. Note that the attributes $a_{m+1}, \ldots, a_n$ are not affected by this, they are still part of the retained tuple, i.e. no implicit projection is performed. This indicates some difficulties, standard SQL does not have any non-deterministic feature, so we somehow have to express whether a set of tuples coincides on the skyline criteria $\{a_1, \ldots, a_m\}$, i.e. if $r \overset{\unrhd}{=} s$:

1. if target list is *not* a subset of the skyline criteria $target\_list \not\subseteq \{a_1, \ldots, a_m\}$, then how to pick just one tuple, the first, the last, an arbitrary one, but just one, or

2. otherwise $target\_list \subseteq \{a_1, \ldots, a_m\}$, how to eliminate duplicates.

The latter is easy, as we just have to eliminate duplicates, which can be done with the DISTINCT keyword in the SELECT-clause. As a result, the rewritten query of (2.20) is the same as in (2.17), except for an additional DISTINCT:

```
SELECT **DISTINCT** $o.target\_list$ FROM $rel$ AS $o$
  WHERE ($condition\{r \leftarrow o\}$) AND NOT EXISTS
    (SELECT * FROM $rel$ AS $i$ WHERE ($condition\{r \leftarrow i\}$) [...]
```
(2.21)

The same effect can be achieved by aggregating over all attributes in the target list.

Things get more involved in the first case. If we can establish an order on the tuples, then we are able to pick the first or the last[3]. In the presence of a unique and ordered key[4] this is obvious, in the other case as long as on each attribute a total order exists, we can construct a lexicographic order. Duplicate elimination with `SELECT DISTINCT` might still be necessary.

To exemplify this, let the target list be $\{a_{j_1}, \ldots, a_{j_h}\}$, $\{j_1, \ldots, j_h\} = J$, with $J \subseteq \{1, \ldots, n\}$ and $J \cap \{m+1, \ldots, n\} \neq \emptyset$, i.e., the target list contains at least one attribute that is not a part of the skyline criteria. Then the query (2.20) can be rewritten into:

```
SELECT DISTINCT $o.a_{j_1}, \ldots, o.a_{j_h}$
FROM $rel$ AS $o$ WHERE ($condition\{r \leftarrow o\}$) AND NOT EXISTS
  (SELECT * FROM $rel$ AS $i$ WHERE ($condition\{r \leftarrow i\}$)
    AND (
      (      $i.a_1$     <= $o.a_1$     AND ... AND $i.a_k$  <= $o.a_k$      -- "MIN" as good
        AND $i.a_{k+1}$ >= $o.a_{k+1}$ AND ... AND $i.a_l$  >= $o.a_l$      -- "MAX" as good
        AND $i.a_{l+1}$  = $o.a_{l+1}$ AND ... AND $i.a_m$   = $o.a_m$      -- "DIFF" equal
        AND (
              $i.a_1$     < $o.a_1$     OR ... OR $i.a_k$  < $o.a_k$        -- "MIN" better
          OR $i.a_{k+1}$  > $o.a_{k+1}$ OR ... OR $i.a_l$  > $o.a_l$        -- "MAX" better
        )
      )                                                                    -- i dominates o
      OR (
             $i.a_1$      = $o.a_1$     AND ... AND $i.a_m$   = $o.a_m$     -- non distinct
        AND ($i.a_{j_1}$ < $o.a_{j_1}$
          OR ($i.a_{j_1}$ = $o.a_{j_1}$ AND $i.a_{j_2}$ < $o.a_{j_2}$
            OR ($i.a_{j_2}$ = $o.a_{j_2}$ AND $i.a_{j_3}$ < $o.a_{j_3}$
              ⋱
                OR ($i.a_{j_{h-1}}$ = $o.a_{j_{h-1}}$ AND $i.a_{j_h}$ < $o.a_{j_h}$)...) -- order lexical.
      )
    )
  )                                                                   (2.22)
```

In the above query we retain the first tuple. In case such an order cannot be established, e.g. an order operator is not defined for the used data type, then in Microsoft SQL we can work around this with the special aggregate functions `FIRST` and `LAST`, which results in a query like this:

```
SELECT FIRST($t.a_{j_1}$), ..., FIRST($t.a_{j_h}$)
FROM (
  SELECT $o.a_{j_1}, \ldots, o.a_{j_h}$ FROM $rel$ AS $o$ WHERE ($condition\{r \leftarrow o\}$) AND NOT EXISTS
    (SELECT * FROM $rel$ AS $i$ WHERE ($condition\{r \leftarrow i\}$)
      AND $i.a_1$   <= $o.a_1$   AND ... AND $i.a_k$  <= $o.a_k$      -- "MIN" as good
  [...]  (same as in (2.17))
) AS $t$
GROUP BY $t.a_{j_1}, \ldots, t.a_{j_h}$                                (2.23)
```

In PostgreSQL subselects which return at most one value can be used as expressions, by means of that duplicates can be eliminated in a way we just like to sketch[5]:

[3] Of course we can also express to pick the second, the third and so forth, but all we know is that within a group we have at least one tuple.

[4] On some RDBMS special pseudo-columns for such purposes do exist.

[5] We thank Chris Roschger. He came up with this idea.

```
SELECT
  (SELECT $a_{j_1}$ FROM $rel$ WHERE $a_1$ = $o.a_1$ AND ...AND $a_m$ = $o.a_m$ LIMIT 1),
  (SELECT $a_{j_2}$ FROM $rel$ WHERE $a_1$ = $o.a_1$ AND ...AND $a_m$ = $o.a_m$ LIMIT 1),
  ⋮
FROM
  (SELECT DISTINCT $a_1, \ldots, a_m$ FROM $rel$ WHERE NOT EXISTS
    [...] condition for skyline query
  ) as $o$                                                                (2.24)
```

This returns the correct result only if all the subselects will return their attributes from the same tuple, which can be expected as the same filter condition on the same relation is used, then almost for sure the same access path will be used. The `LIMIT`-clause is used, which is not part of the SQL 2003 standard [Melton, 2003]. We conjecture that such a query cannot be expressed with standard SQL.

For the corner case of one dimensional skyline queries, i.e. $m = 1$, other methods do exist, we show them in section 4.8.2 and 4.8.3.

2.6 Classification of Skyline Algorithms

Kossmann et al. [2002] suggested a set of criteria for evaluating skyline algorithms that we like to cite here:

1. *Progressiveness*: the first results should be reported to the user almost instantly and the output size should gradually increase.

2. *Absence of false misses*: given enough time, the algorithm should generate the entire skyline.

3. *Absence of false hits*: the algorithm should not discover temporary skyline points that will be later replaced.

4. *Fairness*: the algorithm should not favor points that are particularly good in one dimension.

5. *Incorporation of preferences*: the users should be able to determine the order according to which skyline points are reported.

6. *Universality*: the algorithm should be applicable to any dataset distribution and dimensionality, using some standard index structure.

2.7 How our implementation extends the mathematical model

Our implementation of the `SKYLINE OF` clause is actually a bit more flexible than the mathematical definition given above. With our implementation the skyline operator is not restricted to attributes with a numerical domain, it can be applied to any attribute, as long as a *sort function* is defined for the domain in question, i.e. any expression valid in a SQL `ORDER BY` clause is valid as an expression in a `SKYLINE OF` clause.

This gives the opportunity to include expressions of almost any data type in a skyline query, even user-defined ones, since it is possible for the user to define full-fledged data types in PostgreSQL. For more information on user-defined data types see PostgreSQL documentation on *User-Defined Types*[6] and *Operator Classes and Operator Families*[7]

Although viable, it is questionable whether it is of practical use to include a e.g. `VARCHAR` column in a skyline query. Nevertheless this might be a good use case for a user-defined ordering operator.

Furthermore our implementation allows arbitrary expressions instead of a single attribute, e.g.

```
SELECT * FROM nba.players
SKYLINE OF (h_feet * 12 + h_inches) MAX NULLS LAST,
  weight MAX NULLS LAST                                          (2.25)
```

2.8 Related Problems

Related to computing the skyline of a dataset are the following problems: convex hull, *top-K* queries, and (especially in the context of spacial skylines like e.g. "give me the next good restaurant").

In particular, as noted in [Papadias et al., 2005], the convex hull contains the subset of skyline points that may be optimal only for linear preference functions (as opposed to any monotone function).

[6] http://www.postgresql.org/docs/8.3/static/xtypes.html

[7] http://www.postgresql.org/docs/8.3/static/xindex.html#XINDEX-OPFAMILY

Chapter 3

Existing Skyline Algorithms and Related Works

Since the introduction of the skyline operator [Börzsönyi et al., 2001], a number of secondary-memory algorithms have been developed for efficient skyline computation. These algorithms can be classified into two categories. The first one involves solutions that do not require any preprocessing on the underlying dataset. Algorithms such as *Block Nested Loops* (BNL) [Börzsönyi et al., 2001], *Divide and Conquer* (D&C) [Börzsönyi et al., 2001], *Sort First Skyline* (SFS) [Chomicki et al., 2003], and *Linear Elimination Sort for Skyline* (LESS) [Godfrey et al., 2005] belong to this category. The algorithms in the second category utilize different index structures such as sorted lists and R-trees to reduce the query costs. Well-known algorithms in this category include *Bitmap* [Tan et al., 2001], *Index* [Tan et al., 2001], *Nearest Neighbor* (NN) [Kossmann et al., 2002], and *Branch and Bound* (BBS) [Papadias et al., 2003, 2005].

The advantage of index-based algorithms is that they need to access only a portion of the dataset to compute the skyline, while non-index-based algorithms have to visit the whole dataset at least once. However, index-based algorithms have to incur additional time and space costs for building and maintaining the indexes. Comparisons of these methods are presented in several works [Tan et al., 2001; Kossmann et al., 2002; Chomicki et al., 2003].

Furthermore from a relational algebraic point of view, index-based skyline algorithms can only be placed at the leaves of an algebraic expression, whereas the non-index-based skyline algorithms are first order citizens and can be placed anywhere in the relational algebra expression.

There is a whole class of other skyline algorithms, referred to as *continuous skyline computation*. The task here is to initially compute the skyline and then update the skyline when tuples are added or removed from the dataset, e.g. in a streaming scenario. Such algorithms are presented in [Lin et al., 2005; Tao and Papadias, 2006]. In this work we will not consider this scenario further.

The skyline algorithms we have implemented are: BNL, SFS, BNL+EF and SFS+EF. EF stands for *Elimination Filter*, which was introduced in [Godfrey et al., 2005] as an essential routine for the LESS algorithm. We give a brief description of them in the following sections. Furthermore we derive properties of BNL and SFS concerning the (non)-preservation of relative tuple order. We prove that BNL as described in the original paper is non-terminating under certain conditions and we show how to fix this.

3.1 Block Nested Loops (BNL)

BNL [Börzsönyi et al., 2001] works as follows. A *tuple window* is maintained in the main memory for storing the potential skyline tuples. Once the window becomes full, an overflow file (temp file) has to be generated. At the end of the original input BNL switches to the temp file for reading to process these tuples in a second pass. If the window becomes full again, further overflow files are generated and BNL will make another pass. The tuples in the tuple window get a timestamp to be able to decide which tuples in the window have been compared against all other tuples and therefore can be written to the output and removed from the tuple window. See Listing 3.1 for details, Table 3.1 summarizes the symbols used in the pseudo-code.

Symbol	Description
$\mathcal{I}$	input for skyline computation (*type*: set of d-dimensional points)
$\mathcal{O}$	output of skyline computation (*type*: set of d-dimensional points)
$\mathcal{T}$	temporary file (*type*: set of d-dimensional points)
$\mathbb{W}$	tuple window in main memory (*type*: set of d-dimensional points with a timestamp value for each point and if the window policies *entropy* or *random* are used, then each point has an associated *rank*)
$p \rhd q$	point p dominates point q

Table 3.1: Symbols used throughout the pseudo-code

3.1.1 Non-termination of Block-Nested-Loops (BNL) Algorithm

During the implementation of BNL into PostgreSQL we discovered a flaw in the algorithm from the original paper [Börzsönyi et al., 2001]. We are using a little bit different notation to present to algorithm. In our pseudo code (see Listing 3.1) we only add a tuple to the tuple window if there is enough space. In the original version a tuple is added in any case and later removed again if space was exhausted. Besides from that the original and our version in Listing 3.1 coincide.

Our finding is that under certain conditions the code from Listing 3.1 does not terminate. To show this property, we construct a dataset as follows:

$$\mathcal{I} = \{a_1, \dots, a_n, b_1, \dots, b_m, c_1, \dots, c_n\} \tag{3.1}$$

where n is the number of tuples that fit into the tuple window $\mathbb{W}$ and $m \geq 1$, with the following dominance structure:

$$\forall i \in \{1, \dots, n\} \colon c_i \rhd a_i \tag{3.2}$$

$$\forall i, j \in \{1, \dots, n\} \colon i \neq j \Rightarrow a_i \parallel a_j \wedge c_i \parallel c_j \wedge a_i \parallel c_j \tag{3.3}$$

$$\forall i \in \{1, \dots, n\}, j \in \{1, \dots, m\} \colon a_i \parallel b_j \wedge c_i \parallel b_j. \tag{3.4}$$

In terms of a binary relation the dominance can be expressed as:

$$\rhd = \{(c_i, a_i) | i \in \{1, \dots, n\}\} \subset \mathcal{I} \times \mathcal{I}, \tag{3.5}$$

In essence this means every a_i is dominated by c_i and all other tuples are incomparable. To construct such a dataset, generate $a_1, \dots, a_n, b_1, \dots b_m$ anti-correlated and let the c_i's just

Listing 3.1: Original Block-Nested-Loops (BNL) Algorithm, which does not terminate in all cases

```
function SkylineBNL($\mathcal{I}$)
    // initialization
    $\mathcal{O} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$; $\mathbb{W} \leftarrow \emptyset$; $timestampIn \leftarrow 0$; $timestampOut \leftarrow 0$;

    while $\neg$EOF($\mathcal{I}$)
        // propagate points that have been compared to all
        foreach $q \in \mathbb{W}$
            if $q_{timestamp}$=$timestampIn$
                append($\mathcal{O}$, $q$);
                release($\mathbb{W}$, $q$);

        // fetch the next point
        $p \leftarrow$ next($\mathcal{I}$);
        $p_{timestamp} \leftarrow timestampOut$;
        $timestampIn \leftarrow timestampIn + 1$;
        $isCandidate \leftarrow true$;

        // compare p to all points in the tuple window $\mathbb{W}$
        foreach $q \in \mathbb{W}$
            if $q \rhd p$
                free($p$);
                $isCandidate \leftarrow false$;
                break;
            else if $p \rhd q$
                release($\mathbb{W}$, $q$);

        if $isCandidate$
            if hasfreespace($\mathbb{W}$)
                // add p to the tuple window $\mathbb{W}$
                add($\mathbb{W}$, $p$);
            else
                // write p to the tempfile $\mathcal{T}$
                append($\mathcal{T}$, $p$);
                free($p$);
                $timestampOut \leftarrow timestampOut + 1$;

        // continue with next pass if necessary
        if EOF($\mathcal{I}$)
            // switch to tempfile $\mathcal{T}$
            $\mathcal{I} \leftarrow \mathcal{T}$; $\mathcal{T} \leftarrow \emptyset$;
            $timestampIn \leftarrow 0$; $timestampOut \leftarrow 0$;

    // flushing the tuple window
    foreach $q \in \mathbb{W}$
        append($\mathcal{O}$, $q$);
        release($\mathbb{W}$, $q$);

    return ($\mathcal{O}$);
```

Listing 3.2: Fixed Block-Nested-Loops (BNL) Algorithm

```
function SkylineBNL($\mathcal{I}$)
    // initialization
    $\mathcal{O} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$; $\mathbb{W} \leftarrow \emptyset$; $timestampIn \leftarrow 0$; $timestampOut \leftarrow 0$;

    forever
        // propagate points that have been compared to all
        foreach $q \in \mathbb{W}$
            if $q_{timestamp}$=$timestampIn$
                append($\mathcal{O}$, $q$);
                release($\mathbb{W}$, $q$);

        // continue with next pass if necessary
        if EOF($\mathcal{I}$)
            if EOF($\mathcal{T}$)
                break; // we are done

            // switch to tempfile $\mathcal{T}$
            $\mathcal{I} \leftarrow \mathcal{T}$; $\mathcal{T} \leftarrow \emptyset$;
            $timestampIn \leftarrow 0$; $timestampOut \leftarrow 0$;
            continue;

        // fetch the next point
        $p \leftarrow$ next($\mathcal{I}$);
        $p_{timestamp} \leftarrow timestampOut$;
        $timestampIn \leftarrow timestampIn + 1$;
        $isCandidate \leftarrow true$;

        // compare p to all points in the tuple window $\mathbb{W}$
        foreach $q \in \mathbb{W}$
            if $q \rhd p$
                free($p$);
                $isCandidate \leftarrow false$;
                break;
            else if $p \rhd q$
                release($\mathbb{W}$, $q$);

        if $isCandidate$
            if hasfreespace($\mathbb{W}$)
                // add p to the tuple window $\mathbb{W}$
                add($\mathbb{W}$, $p$);
            else
                // write p to the tempfile $\mathcal{T}$
                append($\mathcal{T}$, $p$);
                free($p$);
                $timestampOut \leftarrow timestampOut + 1$;

    // flushing the tuple window
    foreach $q \in \mathbb{W}$
        append($\mathcal{O}$, $q$);
        release($\mathbb{W}$, $q$);

    return ($\mathcal{O}$);
```

be a little better in each dimension than the a_i's. It is easy to see that following identities hold:

$$\begin{aligned}
\text{skyline}_{\rhd}(\{a_1,\ldots,a_n,b_1,\ldots,b_m\}) &= \{a_1,\ldots,a_n,b_1,\ldots,b_m\} && (3.6)\\
\text{skyline}_{\rhd}(\{a_1,\ldots,a_n,c_1,\ldots,c_n\}) &= \{c_1,\ldots,c_n\} && (3.7)\\
\text{skyline}_{\rhd}(\mathcal{I}) &= \text{skyline}_{\rhd}(\{b_1,\ldots,b_m,c_1,\ldots,b_n\}) && (3.8)\\
&= \{b_1,\ldots,b_m,c_1,\ldots,c_n\} && (3.9)
\end{aligned}$$

In the following we outline the cause of non-termination of the original BNL for the dataset $\mathcal{I}$ and dominance relation $\rhd$. Note that a relation or a set does not have the concept of *nextness*, but we treat it here like a physical table, where the tuples do have a sequence.

1. the tuples $\{a_1,\ldots,a_n\}$ are processed, this entirely fills up the tuple window of size n slots with these tuples as they are all incomparable

2. the tuples $\{b_1,\ldots,b_m\}$ are processed, as they are all incomparable to the a_i's they all end up in the temp file $\mathcal{T}$; at this point $\textit{timestampOut} = m$.

3. the tuples $\{c_1,\ldots,c_n\}$ are processed, as each c_i dominates a_i, all a_i are removed from the tuple window and replaced by the c_i; the timestamp for the c_i's is m and the tuple window is completely filled with the c_i's tuples

4. end of input $\mathcal{I}$ is reached, so switch to temp file $\mathcal{T}$ containing $\{b_1,\ldots,b_m\}$, *timestampIn* is reset to 0

5. the tuples $\{b_1,\ldots,b_m\}$ are processed, they will all end up in the temp file again, as they are all incomparable to the c_i's; at this point $\textit{timestampIn} = m$ but at the same time the end of input is reached, resulting in a switch to the temp file and resetting *timestampIn* and *timestampOut* to zero. No tuple c_i from the tuple window will be propagated to the output as they have a timestamp of m but the propagate code will not be reached with $\textit{timestampIn} = m$; *the result is that this last step will be repeated over and over again*

The simplest case to reconstruct this non-termination behavior is with $n = m = 1$. A possible fix is to duplicate the lines 7–10 between 38 and 39 in Listing 3.1, so the propagation is done once EOF is reached. Nevertheless we fixed it with a slight modification in the control flow, see Listing 3.2. And of course, our implementation in PostgreSQL is based on our fixed version.

3.1.2 Non-preservation of relative Tuple Order

It is not completely obvious that BNL does not preserve the relative order of tuples from the input in the output, even when the tuple window policy *append* is used. To exemplify this property we use a dataset as in the previous section with $n = m = 1$, i.e.

$$\mathcal{I} = \{a_1, b_1, c_1\}, \quad \text{with } a_1 \parallel b_1,\ b_1 \parallel c_1,\ c_1 \rhd a_1.$$

After the first pass the tuple window contains c_1 and b_1 is in the temp file, a_1 was eliminated by c_1. In the next pass b_1 and c_1 are compared against each other, but as they are incomparable b_1 ends up in the temp file again, but this time c_1 can be propagated to the output as it was compared against all other tuples. In the final pass b_1 is propagated to the output. Hence the output is

$$\mathcal{O} = \{c_1, b_1\},$$

where b_1 and c_1 have changed their relative order. It is easy to imagine that this can happen in other configurations as well and therefore BNL does not preserve the relative order of tuples. This physical property of BNL is important during the query planning, as the relative order of tuples is an important aspect here. Very offen, a prior operator in the query plan needs to establish a certain order or preserve such an order to save an explicit sorting operation, in order to execute further physical operators such as aggregation or natural join.

3.2 Sort First Skyline (SFS)

SFS [Chomicki et al., 2003] differs from BNL in that the data is topologically sorted at start-up time. It is shown in the original paper that this condition is sufficient for the following property. In SFS once a tuple is incomparable to all tuples in the tuple window it is known to be a skyline tuple as well, and it can be written to the output directly. Furthermore no timestamps have to be maintained and at the end of each pass the tuple window can be completely purged. The pseudo-code is given in Listing 3.3.

3.2.1 Preservation of relative Tuple Order

Please note that in the output of SFS the relative order of tuples from $\mathcal{I}_{presorted}$ is preserved. SFS has this physical property regardless of the tuple window placement policies used. This is because if a tuple survives the dominance check it is appended to the output.

3.3 Linear Elimination Sort for Skyline (LESS)

Godfrey et al. [2005] proposed the LESS algorithm, an improvement over SFS [Chomicki et al., 2003]. LESS is generally considered as the fastest known non-index-based skyline algorithm in literature. The essential differences between SFS and LESS are:

1. it uses a so called *elimination filter* in pass zero of the external sort routine to eliminate tuples early, hence less tuples left to sort; and

2. the skyline-filter is interweaved in the final pass of the external sort routine.

In the following we describe the elimination filter (EF) as the second difference is merely an implementation detail.

3.3.1 Elimination Filter (EF)

The concept of the elimination filter (EF) is simple but effective. It tries to eliminate as many non-skyline tuples at an early stage. To do so it compares every incoming tuple p against a tuple window $\mathbb{W}$, if p is dominated by any tuple in the tuple window, then p is dropped and the next tuple is read, otherwise p is piped out and the next tuple is read. Of course the elimination filter does not guarantee to filter out all non-skyline tuples, but a skyline candidate is guaranteed to survive the elimination filter. Another notable difference to BNL and SFS is that EF does not produce any temporary files once the tuple window gets full. This makes EF suffer in the same way as BNL from a bad distribution of tuples in the input, e.g. the tuple window gets filled up by skyline tuples with a low dominance factor.

Godfrey et al. [2005] did not give pseudo-code for the elimination filter in their paper. In Listing 3.4 we give a sketch for the code as we used it in our implementation. Note that we give the algorithm in a form that fits into a pipelined architecture that is used in the

Listing 3.3: Sort First Skyline (SFS) Algorithm

```
function SkylineSFS($\mathcal{I}$) begin
    // presort input
    $\mathcal{I}_{presorted} \leftarrow$ sort($\mathcal{I}$);

    // initialization
    $\mathcal{O} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$; $\mathbb{W} \leftarrow \emptyset$;

    while $\neg$EOF($\mathcal{I}_{presorted}$)
        // loop invariant: $\mathcal{I}_{presorted}$ and $\mathcal{T}$ are sorted

        // fetch the next point
        $p \leftarrow$ next($\mathcal{I}_{presorted}$);
        $isSkyline \leftarrow true$;

        // compare p to all points in the tuple window $\mathbb{W}$
        foreach $q \in \mathbb{W}$
            if $q \rhd p$
                free($p$);
                $isSkyline \leftarrow false$;
                break;
            else if $p \rhd q$
                release($\mathbb{W}$, $q$);

        if $isSkyline$
            if hasfreespace($\mathbb{W}$)
                // add p to the tuple window $\mathbb{W}$
                add($\mathbb{W}$, $p$);
            else
                // write p to the tempfile $\mathcal{T}$
                append($\mathcal{T}$, $p$);
                free($p$);

            // add tuple to the output
            append($\mathcal{O}$, $p$);

        // continue with next pass if necessary
        if EOF($\mathcal{I}_{presorted}$)
            // switch to tempfile $\mathcal{T}$
            $\mathcal{I}_{presorted} \leftarrow \mathcal{T}$; $\mathcal{T} \leftarrow \emptyset$;

            // clean tuple window
            $\mathbb{W} \leftarrow \emptyset$;

    return ($\mathcal{O}$);
```

PostgreSQL query execution engine. The function `EliminationFilter` has to be called for each output tuple, until `NULL` is returned. While doing so `EliminationFilter` consumes the entire input $\mathcal{I}$. Please also note that the variable `state` has to be preserved over calls to `EliminationFilter` as it keeps the internal state of the function, later we will call such a function an iterator.

Tunable parameters for the elimination filter are the tuple window size and the tuple window placement policy, in the same way as these parameters are tunable for BNL and SFS.

In [Godfrey et al., 2005] the elimination filter is only used in conjunction with an SFS style algorithm, in our implementation we demonstrate that this concept is also very effective when used with a BNL style algorithm, we call it BNL+EF, cf. section 5.4

Furthermore it should be noted that the elimination filter preserves the relative order of tuples from the input in the output, i.e. those tuples that do survive the elimination filter are in the same relative order as in the input.

Listing 3.4: Elimination Filter (EF) (in iterator style)

```
function EliminationFilter($\mathcal{I}$, var $state$) begin
    forever
        switch $state$
        case INIT:
            $\mathbb{W} \leftarrow \emptyset$;
            $state \leftarrow$ PROCESS;
            break;

        case PROCESS:
            if EOF($\mathcal{I}$)
                $state \leftarrow$ DONE;
                break;

            $p \leftarrow$ next($\mathcal{I}$);
            $isCandidate \leftarrow true$;

            // compare p to all points in the tuple window $\mathbb{W}$
            foreach $q \in \mathbb{W}$
                if $q \rhd p$
                    free($p$);
                    $isCandidate \leftarrow false$;
                    break;
                else if $p \rhd q$
                    release($\mathbb{W}$, $q$);

            if $isCandidate$
                if hasfreespace($\mathbb{W}$)
                    // add p to the tuple window $\mathbb{W}$
                    add($\mathbb{W}$, $p$);

                return $p$;

            break;

        case DONE:
            return NULL;
```

3.4 Related Works

The closest related works to this thesis are [Chaudhuri et al., 2006] and [Goncalves and Vidal, 2005a,b]. The authors of both papers did an implementation of the skyline operator into an RDBMS as a full-blown relational operator.

Chaudhuri et al. [2006] at Microsoft Research extended a Beta Version of Microsoft SQL Server 2005. To the best of our knowledge neither source nor binary versions of this implementation are available. The skyline operator is fully integrated into the cost-based optimizer. The syntax they are using is Preference SQL [Kießling and Köstler, 2002].

[Goncalves and Vidal, 2005a] extended PostgreSQL with a hybrid approach between *top-k* and *skyline operator* called top-k-skyline. Skyline strata (see section 2.4.2) are computed until at least k records are returned. With this approach all the second and higher order best records do get a chance to end up in the result, nevertheless the selectivity of this approach is even lower than skyline operator alone.

Chapter 4

Implementation

4.1 SQL Extension

Börzsönyi et al. [2001] proposed the following extension to the SQL's `SELECT` statement with an optional `SKYLINE OF`-clause:

`SELECT ...  FROM ...  WHERE ...`

`GROUP BY ...  HAVING ...`

`SKYLINE OF [DISTINCT]` a_1 `[MIN|MAX|DIFF], ...,` a_m `[MIN|MAX|DIFF]`

`ORDER BY ...` (4.1)

In addition, we extended the standard syntax to specify:

- The treatment of NULL values (`NULLS FIRST` and `NULLS LAST`)
- The usage of order relations other than < and > (`USING` *`Op`*)
- Operational aspects of skyline computation, such as
  - *method* (`BNL`, `SFS`, `MNL`, `PRESORT` (2 dim only))
  - *tuple window size* in terms of memory and/or number of slots
  - *tuple window policy* (append, prepend, ranked by entropy, ranked by a random value)
  - *usage of indexes* (`NOINDEX`)
  - *usage of elimination filter (`EF`), with the possibility to influence the EF window in the same way as for the BNL or SFS node*

In the following sections we describe in detail the extension to the SQL syntax we have implemented.

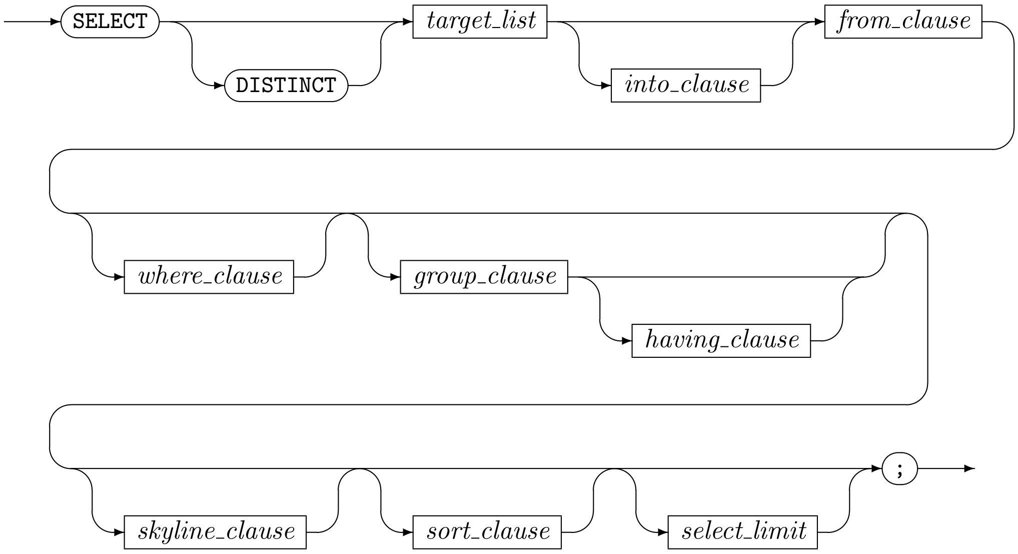
4.1.1 Syntax (Railroad Diagrams)

To describe the formal grammar of the `SKYLINE OF`-clause extension to the SQL query syntax we use syntax diagrams (or *railroad diagrams*). Railroad diagrams are a graphical alternative to Backus-Naur form or EBNF and they have been made popular by books such as the "Pascal User Manual" written by Niklaus Wirth. Nowadays railroad diagrams are used in user manuals such as the "Oracle Database SQL Reference" (see [Oracle, 2005]). We believe the general concept of railroad diagrams is easy to grasp and does not need a further explanation at this point, so we can directly focus on our `SKYLINE OF` extension.

The following diagram depicts how the `SKYLINE OF`-clause (denoted by *skyline_clause* in the diagram) fits into an SQL select query. It is right between the optional `GROUP`

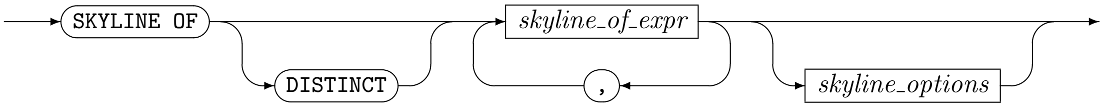
BY/HAVING-clause and the optional ORDER BY-clause, denoted by *group_clause* / *having_clause* and *sort_clause*. We omit some details here but it should be clear from the context that *target_list*, *into_clause*, *from_clause*, *where_clause*, *group_clause*, *having_clause*, *sort_clause*, and *select_limit* have their usual meaning. We refer the readers to the corresponding SQL standard and especially to the PostgreSQL documentation on SQL *SELECT*[1]. For the non-terminals that we omitted here we selected the same name as in the implementation (see `src/backend/parser/gram.y`), in order to easily find the exact definition.

select_clause



The *skyline_clause* itself is preluded by the keywords SKYLINE OF, followed by an optional DISTINCT. The essential part is the comma separated list of *skyline_of_expr*, and it is closed by the optional *skyline_options* part. For the semantics of DISTINCT see section 2.5. For the entire skyline clause we introduced just a single new reserved keyword (SKYLINE), see section 4.1.2.

skyline_clause



A *skyline_of_expr* specifies how to treat a single dimension of a skyline query. This includes the specification of the column or even an expression (*c_expr*) and the specification of MIN, MAX, DIFF, and USING *qualOp*. We call the last specification *skyline direction* or just *direction* for short, this is because a similar concept (ASC/DESC) in the ORDER BY-clause is also called direction. Furthermore the treatment of null values can be specified using NULLS FIRST and NULLS LAST. We have the same concept for the ORDER BY-clause. This special treatment of null values was introduced in PostgreSQL 8.3.0 (see *SQL SELECT/ORDER BY*[2]). For the semantics of this see section 4.2.2.

The introduction of USING *qualOp* in skyline queries is our idea, but as for the treatment of null values it was inspired by a PostgreSQL extension to the ORDER BY-clause. See section 4.2.1 for more details.

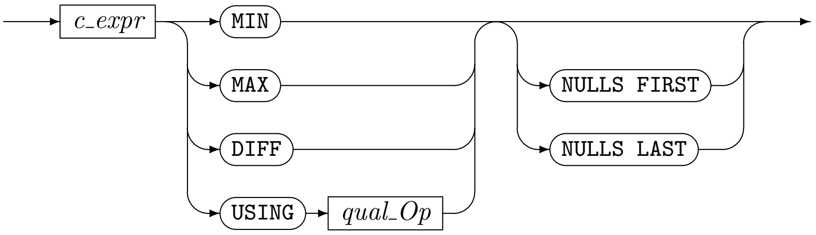
[1] http://www.postgresql.org/docs/8.3/static/sql-select.html

[2] http://www.postgresql.org/docs/8.3/static/sql-select.html#SQL-ORDERBY

In the *target_list* of the *select_clause* and in many other places *a_expr* is used. This is the concept of a general expression and the heart of the expression syntax. In certain places *b_expr* is used, which is a restricted subset to avoid shift/reduce conflicts.

c_expr contains all the productions that are common to *a_expr* and *b_expr*. Hence *c_expr* is factored out to eliminate redundant coding, but in our case we use it directly to avoid shift/reduce conflicts. Please note that `'('` *a_expr* `')'` is a *c_expr*, so an unrestricted expression can always be used by surrounding it with parenthesis. Furthermore *c_expr* contains the most common case, a column reference (*columnref*), anyway and also function calls can be used without an extra pair of parenthesis.

skyline_of_expr



The syntax we have described so far deals only with specifying the skyline query itself and the ultimate goal of our work is building a skyline query optimizer to automatically generate a good query plan w.r.t. I/O, time, and memory consumption and decide all operational aspects of the query evaluation. It is well known that the cost estimation of the skyline queries is a non-trivial task [Chaudhuri et al., 2006] since the performance of a skyline query is sensitive to a number of parameters [Godfrey et al., 2007].

To be able to study different query plans, different physical operators, and various options for the physical operators we introduced a set of options. This options list is preluded by the keyword `WITH`.

Except for the order of the tuples in the output all of the following options do not have an impact on the result of the skyline computation. Using either BNL or SFS will change the order of tuples in the output, as SFS does a sort prior to skyline computation. Furthermore the size of the tuple window and the tuple window policy have an impact on the order of tuples. And even the elimination filter and its tuple window size have an (indirect) impact on the output order, as a different amount of potential skyline tuples reaches the final skyline computation node, although the elimination filter does not change the relative order. See detailed discussion on relative tuple order preserving property in section 4.7.2. To enforce a specific order, if desired, sort the query result by means of an appropriate `ORDER BY`-clause. We did so in regression testing, we ordered on the unique `ID` column, no matter what method and what options used, the results are the same. If not this indicates a flaw in the implementation.

The options fall into three groups: elimination filter (`EF`), physical operator (`BNL`, `SFS`, etc.), and access path selection (`NOINDEX`) and associated options if applicable.

When the skyline option `EF` is present the query planner will include an elimination filter (EF) in the query plan. If the *efwindowoptions* are omitted, the query planner uses the defaults and plans the elimination filter with an 8 KB tuple window and the tuple window policy `APPEND`. The 8 KB tuple window size stems from PostgreSQL page size `BLCKSZ`, which is 8 KB on most platforms.

The next group of options is used to specify the main algorithm used for skyline computation. Currently the work horses of our skyline operator implementation are the BNL and the SFS algorithm. BNL is the default in the query optimizer. If a suitable index access path (i.e. index) is available SFS is selected. This default procedure can be overruled by specifying `BNL` or `SFS` as an option. Furthermore the size and the tuple placement policy for the tuple window used by BNL or SFS can be specified with the *windowoptions*, presented below. In fact the options for the tuple window could be specified, while leaving the decision which physical operator to take to the query optimizer, but nevertheless these *windowoptions* only have an effect on BNL and SFS. See section 4.7.3 for more details on how the physical operator is selected.

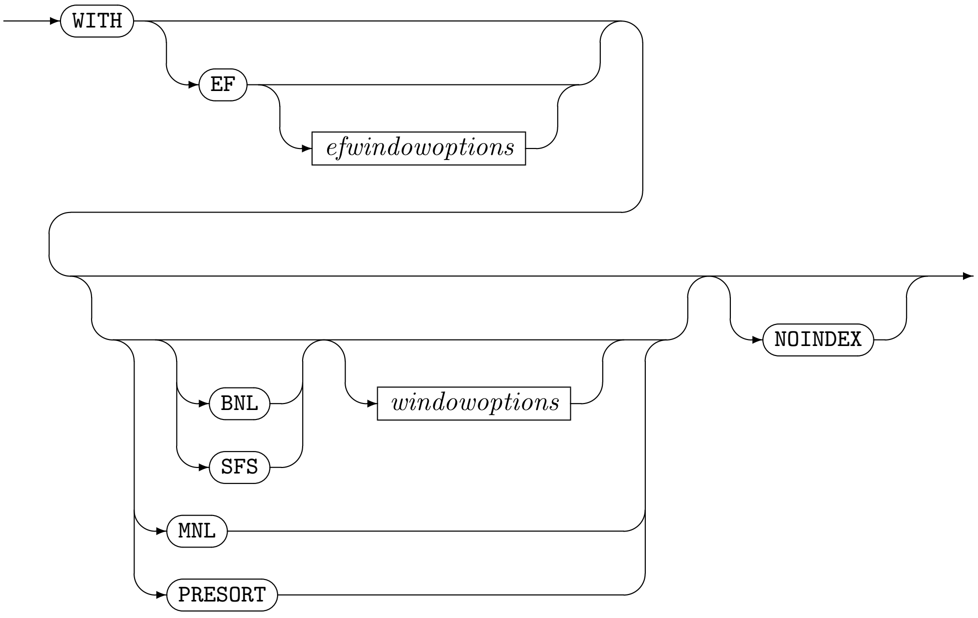
The PRESORT method is more or less a special case of SFS, which only works in the two dimensional case, it was proposed by Börzsönyi et al. [2001]. In the two dimensional case once the data are ordered according to the two skyline expressions a single tuple is enough to compute the skyline. See section 4.8.4.

MNL stands for *materialized nested loop*, which is our own naïve skyline computation algorithm. Nevertheless it is fully generic, i.e. skylines of any dimensionality, distinct and non-distinct with the same type of expressions and directions (MIN, MAX, ...) can be computed. The approach is very simple: materialize all tuples from the outer query plan node in a tuple store[3] and compare one tuple after the other against all other tuples in the tuple store. It is obvious that this algorithm has $O(n^2)$ runtime complexity and very poor I/O behavior. Nevertheless it was easy to implement and gave us something to play with while implementing other parts of the system and of course for regression testing, because this method was easy to get right and later on to use it to verify the results from other algorithms. See section 4.8.5.

Two other special methods do exist, namely 1DIM and 1DIMDISTINCT, as the names suggest they are only for the 1 dimensional non-distinct and distinct case. These methods are directly selected by the query planner and no options can be set for them. The tuple store used by the method 1DIM uses the default work_mem setting. See section 4.8.2 and section 4.8.3.

We modified the query planner/optimizer to decide in a cost based way to use an IndexScan instead of a SeqScan, in case a suitable index is present. With the NOINDEX option we force the query planner not to use an IndexScan, this is of interest especially when profiling SFS, as SFS does a sort prior to skyline computation.

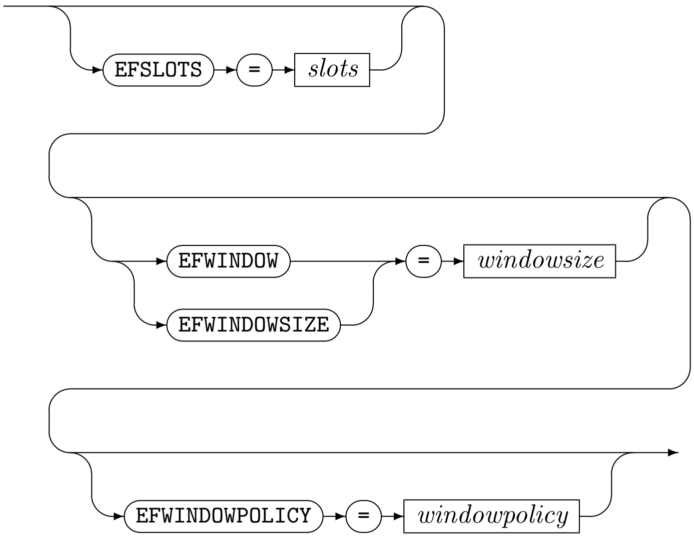
skyline_options



[3] Tuple store is PostgreSQL concept for temporary files to store tuples.

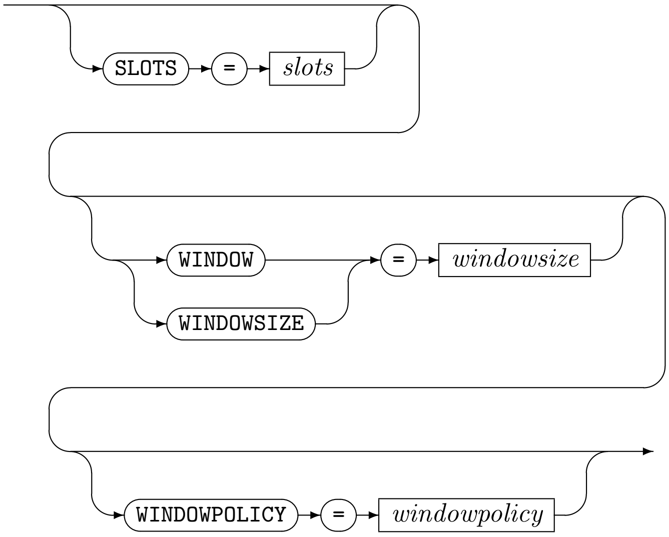
The options for the elimination filter (EF) tuple window *efwindowoptions* are modulo renaming the same as for the tuple window use by BNL and SFS cf. *windowoptions* below.

efwindowoptions



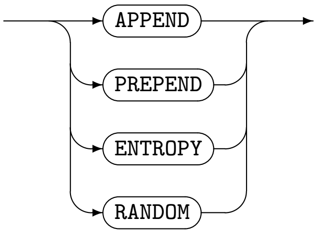
BNL and SFS are using a tuple window to do their job. Our implementation allows to specify the size and the tuple placement policy. The size can be defined in two different terms, either in the number of slots or in the amount of RAM (in KB). When placing a tuple into the tuple window it takes up exactly one slot and as much RAM as required for the chunk of memory allocated with PostgreSQL's `palloc`. If both are given, i.e. `SLOTS` and `WINDOWSIZE` the tuple window is only limited by the number of available slots and the RAM constraints are ignored. When the `WINDOWSIZE` option is used an according amount of RAM is used for the tuple window. `WINDOW` is just an alias for `WINDOWSIZE`. If neither `SLOTS` nor `WINDOWSIZE` are given the configuration variable `work_mem` is used, which is 1 MB per default. The other property which can be influenced is the order in which tuples are placed in the tuple window, where the default is `APPEND`.

windowoptions



As indicated above a tuple window is used by the elimination filter (EF) and the BNL and SFS algorithms. For each of them the order in which tuples are placed into the window is set by this option, where the default is `APPEND`. See section 4.8.6 for details.

windowpolicy



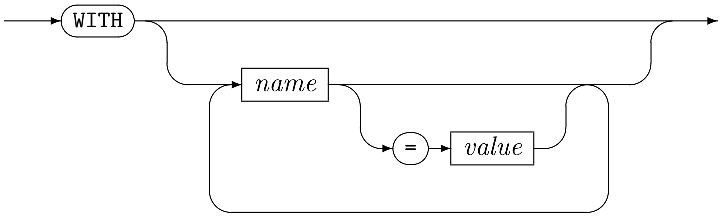
At this point we give a little example how the options can be used (we slightly reformatted the output to make it fit onto this page):

```
db=# EXPLAIN ANALYZE                              see the query plan and execution statistics
db-# SELECT * FROM a15d1e5s0idx             note: on relation ad1e50idx an index is defined
db-# SKYLINE OF d1 min, d2 min, d3 min                           we do a 3 dim skyline query
db-# WITH                                                       options for SKYLINE OF follow
db-#   EF                                                       use an elimination filter (EF)
db-#     EFWINDOWSIZE=16                          use 16 KB RAM and policy ENTROPY for EF
db-#     EFWINDOWPOLICY=ENTROPY
db-#   SFS                                                                            use SFS
db-#     WINDOWSIZE=512                          use 515 KB RAM and policy PREPEND for SFS
db-#     WINDOWPOLICY=PREPEND
db-#   NOINDEX;                                        do not use an available index access path

                                  QUERY PLAN
--------------------------------------------------------------------------------
 Skyline  (cost=25106.90..25107.06 rows=9 width=124) ↩                      estimated
  → (actual time=329.052..342.313 rows=196 loops=1)                          actual
   Skyline Attr: d1, d2, d3
   Skyline Method: sfs 3 dim                                          3 dim and SFS
   Skyline Stats: passes=1 rows=1303                          some stats on the SFS node
   Skyline Window: size=512k policy=prepend                            as requested
   Skyline Cmps: tuples=25953 fields=62229                         # of comparisons
   ->  Sort  (cost=25104.70..25104.86 rows=66 width=124) ↩
         → (actual time=329.046..329.564 rows=1303 loops=1)
         Sort Key: d1, d2, d3
         Sort Method:  quicksort  Memory: 368kB
         ->  Elimination Filter  (cost=24852.69..25102.69 rows=66 width=124) ↩
               → (actual time=0.088..325.329 rows=1303 loops=1)
               Elim Filter Attr: d1, d2, d3
               Elim Filter Method: elimfilter 3 dim
               Elim Filter Stats: passes=1 rows=
               Elim Filter Window: size=16k policy=entropy             as requested
               Elim Filter Cmps: tuples=313895 fields=828555
               ->  Seq Scan on a15d1e5s0  (cost=0.00..2924.00 rows=100000 ↩
                     → width=124) (actual time=0.043..71.488 rows=100000 loops=1)
                           the effect of NOINDEX can be seen here as no IndexScan is used
 Total runtime: 343.200 ms
(17 rows)
```

The alert reader might notice that the grammar for *skyline_options* contains a rather big amount of ε-productions and that this grammar would be easy to implement without any shift/reduce conflicts using yacc/bison. In fact we defined the grammar for *skyline_options*, just as simple as that:

skyline_options



While the definition given above describes exactly the intended semantics, this definition gives us great flexibility during the development. Using the simple definition we were able to introduce new options as we liked on the fly. The options are passed around from the parser, through query planning to the execution engine as a simple *name/value* list. So whenever we needed a new option we just had to query this list at the desired point in the code. If the equal sign and the value are omitted, this has the semantics of assigning the integer constant 1 to the name.

Note that the grammar for *select_clause* is defined a little bit differently[4] but only in the aspect of the associativity of the entire `SKYLINE OF` clause, this is because the SQL:1992 Standard [ISO/ANSI, 1991] requires the following statement:

`SELECT foo UNION SELECT bar ORDER BY baz` (4.2)

to be parsed as

`(SELECT foo UNION SELECT bar) ORDER BY baz` (4.3)

and not as

`SELECT foo UNION (SELECT bar ORDER BY baz)` (4.4)

For the `SKYLINE OF` clause we decided that the it should be left associative, i.e.

`SELECT * FROM foo UNION SELECT * FROM bar SKYLINE OF baz` (4.5)

will be parsed as

`SELECT * FROM foo UNION (SELECT * FROM bar SKYLINE OF baz)` (4.6)

We did so, because we believe the `SKYLINE OF` clause is closer related to the `GROUP BY` clause than to the `ORDER BY` clause, therefore we parse it in the same way.

4.1.2 Reserved Keywords

On the lexical level we introduced one new *reserved keyword*: “`SKYLINE`”[5]. Introducing a reserved keyword has the usual consequences: it cannot be used as normal identifier, like illustrated in the following example:

```
db=# CREATE DATABASE SKYLINE;
ERROR:  syntax error at or near "SKYLINE"
LINE 1: CREATE DATABASE SKYLINE;
                        ^
```

[4] see `src/backend/parser/gram.y` for details

[5] see `src/backend/parser/keywords.c`

Nevertheless the solution is easy and the same for every reserved word, just quote it:

```
db=# CREATE DATABASE "SKYLINE";
```

We did not define `MIN`, `MAX`, and `DIFF` as reserved keywords, because the SQL aggregate functions `MIN` and `MAX` are implemented as functions and are looked up at runtime and not directly hardcoded into the grammar. Defining these keywords as reserved keywords would mess up the implementation of SQL aggregate functions.

4.2 Semantics

In this section we describe only these aspects of the skyline operator semantics which are not clear through the basic definition. The semantics of the options for the `SKYLINE OF`-clause is either already given in the description of the syntax in section 4.1.1 or in the appropriate subsection of section 4.8. The semantics of `SKYLINE OF DISTINCT` vs. non-`SKYLINE OF DISTINCT` is described in section 2.5.

4.2.1 `USING` *qualOp*

PostgreSQL has the feature of applying user-defined ordering operators for the `ORDER BY`-clause, where `ASC` is equivalent to `USING <` and `DESC` is equivalent to `USING >`, but the creator of a user-defined ordering operator can give it any name, see PostgreSQL documentation on *ORDER BY-clause*[6]. We borrowed this concept and integrated it into the SQL extension for the `SKYLINE OF`-clause, where `MIN` is equivalent to `USING <` and `MAX` is equivalent to `USING >`, e.g.

```
SELECT * FROM a15d1e5s0 SKYLINE OF d1 USING <, d2 USING >;          (4.7)
```

4.2.2 Skyline in the presence of NULL values

The issue of dealing with null values is usually not treated in the literature, but a real implementation in a database system of course must be able to cope with null values. Furthermore we have a syntax to specify how to treat null values when ordering values. Once again this is a feature inspired by a feature of PostgreSQL's `ORDER BY`-clause, where the keywords `NULLS FIRST` and `NULLS LAST` allows the user the specify how to treat null values. If sorting ascending (`ORDER BY` *expr* `ASC`), then the default is to sort null values *after* non-null values. The default for `ORDER BY` *expr* `DESC`, i.e. sort descending, is to sort null values *before* non-null values. For more details see the PostgreSQL documentation on ORDER BY-clause.

If `NULLS FIRST` or `NULLS LAST` is omitted, the default is to sort null values after all the non-null values for `SKYLINE OF MIN` and for `SKYLINE OF MAX` to have the null values before the non-null values. Specifying `NULLS FIRST/LAST` overrides this default. The question may arise if `NULLS LAST` should be the default for `SKYLINE OF MAX`, but as this is not the case for the `ORDER BY DESC` we stick with `NULLS FIRST` in order to be consistent.

4.2.3 **`SKYLINE OF`** and **`GROUP BY`**

It is well known that there are some dependencies between what one can specify in the *target_list* of a `SELECT`-clause and the relvars[7] specified in the *from_clause*. A violation of this rule is reported as an error, e.g.:

[6] `http://www.postgresql.org/docs/8.3/static/sql-select.html#SQL-ORDERBY`

[7] The term *relvar* was coined by C. J. Date as an abbreviation for "relation variable". This usually refers to a *table*, but could also be a sub-select or such.

```
db=# SELECT foo FROM a15d1e5s0;
ERROR:  column "foo" does not exist
LINE 1: select foo from a15d1e5s0;
               ^
```

Clearly such limitations also apply to the expressions in the SKYLINE OF-clause. In the presence of a GROUP BY-clause more limitations apply, which immediately becomes clear when we look at the operator tree (cf. Figure 4.1). Only those columns or expressions that appear in the GROUP BY-clause or are part of an aggregate function can be used as a SKYLINE OF expression. The same applies to the ORDER BY-clause.

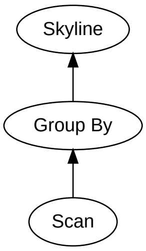


Figure 4.1: Query plan for query with aggregation and skyline operator

Again a violation of this limitation is reported with an appropriate error message, e.g.:

```
db=# SELECT id, COUNT(*) FROM a15d1e5s0 GROUP BY id SKYLINE OF d1 MIN;
ERROR:  column "a15d1e5s0.d1" must appear in the GROUP BY clause or be used
in an aggregate function
```

4.3 PostgreSQL Architecture and Concepts

Before we go into the details we would like to give a brief overview of the PostgreSQL architecture. According to [Conway and Sherry, 2006] PostgreSQL has five main components:

1. *parser / analyzer:* parse the query string
2. *rewriter:* apply rewrite rules. SQL views work by means of this mechanism.
3. *optimizer:* determine an efficient query plan
4. *executor:* execute a query plan
5. *utility processor:* process DDL statements like CREATE TABLE

How these components work together to execute a query is illustrated in Figure 4.2.

In the following subsections we describe some concepts of PostgreSQL which are essential to understand our implementation.

4.3.1 Pathkeys

In a sole set oriented algebra there is no concept of *nextness* or *order of tuples*, but in an RDBMS the relative order of tuples is an important *physical property* of a tuple stream. The knowledge about this is encoded in PostgreSQL by means of so called *pathkeys*, they arise naturally in conjunction with an access path, e.g. when scanning a tuple via a clustered index or using an index scan. *Pathkeys* are essentially a list of which each member is a *pathkey*, which encodes the sort order on a single column. Actually this can be an arbitrary

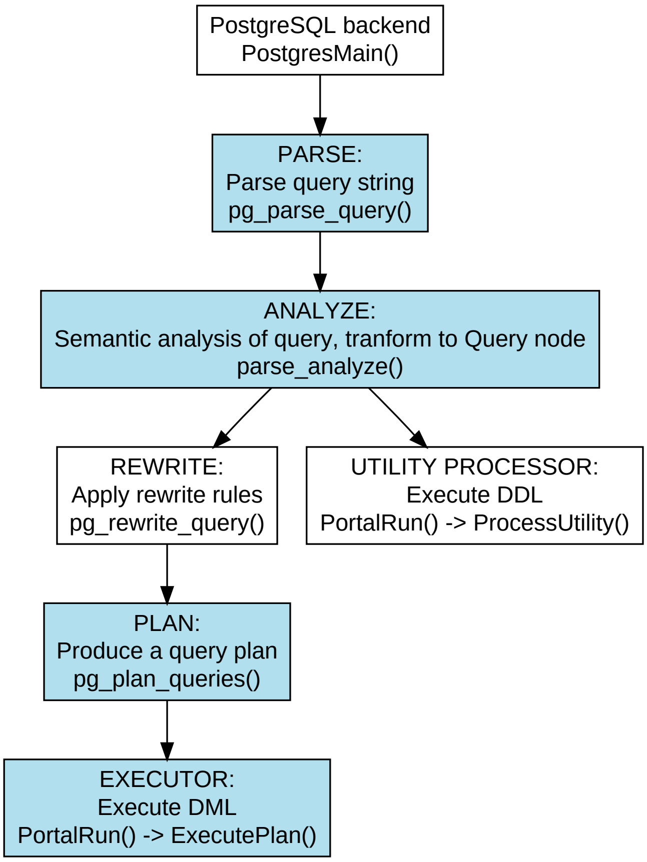


Figure 4.2: Architecture Diagram. Light blue background indicates the components we modified to support skyline queries.

expression and a pathkey contains a reference to the according target list entry, the sort operator, and a flag on how to treat NULL values (cf. NULLS FIRST/NULLS LAST). The information if the sort order is ascending (ASC) or descending (DESC) is encoded in the sort operator used, as sort operators always appear in pairs, an operator and its commutator, i.e. the same order just reversed. This information provided by pathkeys plays an important role throughout the query plan. A node within the query plan might create, can make use of, or destroy certain pathkeys. If created or preserved the nodes higher up in the query plan can make use of it as well. This information is considered by the query planner when deciding among different physical operators, e.g. *stream aggregate operator* vs. *hash aggregate operator*, or different forms of the *join operator*, for *duplicate elimination* and so forth. *Selection*, *projection* and their like typically preserve pathkeys, i.e. the relative tuple order remains the same. Others like *index scan* and *sort operator* establish a certain relative order.

4.3.2 Pipelined execution

PostgreSQL's query execution engine has a top down driven pipelined architecture, i.e. each node consumes the tuples from its children and produces tuples and in general no intermediate results are materialized, see `src/backend/executor/README`. A simple suitable interface for this iterator would just expose these methods: `open()`, `tuple next()` and `close()`. Nevertheless PostgreSQL's iterator interface is a bit richer, the dispatch functions are in `src/backend/executor/execProcnode.c`. In order to specialize just add an appropriate case in the these dispatch functions and call your own code:

```
ExecCountSlotsNode -- count tuple slots needed by plan tree
ExecInitNode       -- initialize a plan node and its subplans
ExecProcNode       -- get a tuple by executing the plan node
ExecEndNode        -- shut down a plan node and its subplans
```

Dispatch functions for less frequently used methods are in `src/backend/executor/exexAmi.c`:

```
ExecReScan                -- Reset a plan state node so that its
                          -- output can be re-scanned
ExecMarkPos               -- Marks the current scan position
ExecRestrPos              -- restores the scan position previously
                          -- saved with ExecMarkPos
ExecSupportsBackwardScan -- does a plan type support backwards
                          -- scanning?
```

Each execution node is coded in the fashion of a state machine. Do something in each state and then if necessary proceed to another state. The whole state is stored in the node's execution state information and is preserved across calls.

4.3.3 Simple Object System: Nodes

As described in [Conway and Sherry, 2006, Page 12-14] PostgreSQL has a very simple object system with support for single inheritance. The C structure `Node` is the root of the class hierarchy.

Listing 4.1: PostgreSQL's simple object system

```
typedef struct
{
    NodeTag type;
} Node;

```

```c
typedef struct
{
    NodeTag type;
    int     baseclass_field;
} Baseclass;

typedef struct
{
    Baseclass   baseclass;
    int         subclass_field;
} Subclass;
```

Since the memory layout for `Baseclass` and the first member of `Subclass` (`baseclass`) is the same, a pointer `Subclass *` can be treated like `Baseclass *`. The first field of any class in this simple object system is `NodeTag`, which is used to determine a `Node`'s specific type at runtime.

There are a couple of support functions to work with the object system: `makeNode` (create a new `Node`), `IsA` (runtime type testing), `equal` (deep equality test), `copyObject` (deep object copy), `nodeToString`/`stringToNode` (serialize/deserialize to/from text). When introducing new classes make sure to update these files:
`src/backend/nodes/{equalfuncs,copyfuncs,outfuncs}.c`.

4.4 Parsing the SKYLINE OF-clause

In the parser component the textual representation of a query, namely SQL, is parsed into a data structure. This is done as described in the PostgreSQL documentation for the *parser stage*[8]. The data structure uses the PostgreSQL simple object system, as described in the previous section. We extended this stage to handle the `SKYLINE OF`-clause as defined in section 4.1, see `src/backend/parser/gram.y` for the definition of `skyline_clause`. The result of this stage is called *parse tree*.

4.5 Query Analyzing

In this phase, which is also described in the PostgreSQL documentation on the *parser stage*[9], the *parse tree* from the previous stage is semantically analyzed, the outcome is called *query tree*. Look-ups into the system catalog are done in this stage to resolve references to tables, columns, functions, and operators, see `transformSkylineClause`[10].

4.6 Query Rewriting

For our implementation it was not necessary to modify the *query rewriting* stage. For details see documentation on the *PostgreSQL Rule System*[11].

4.7 Query Planning/Optimizing

The *query tree* from the previous stage is transformed into a *plan tree* during this stage. Usually there are different possible plans for a given query. The goal is to create an optimal

[8] `http://www.postgresql.org/docs/8.3/static/parser-stage.html`

[9] `http://www.postgresql.org/docs/8.3/static/parser-stage.html`

[10] in `src/backend/parser/parse_clause.c`

[11] `http://www.postgresql.org/docs/8.3/static/rule-system.html`

plan. Although it is computational feasible to examine each possible plan and selecting the best, often this exhaustive enumeration is too expensive. Various trade-offs are made, for more details see PostgreSQL documentation on *Planner/Optimizer*[12].

In the following subsections we describe how we modified the query planner/optimizer:

4.7.1 Selecting Indexes

It is a well known fact that the skyline operator is insensitive to the order of attributes, i.e. the queries

```
SELECT * FROM a15d1e4s0 SKYLINE OF d1 MIN, d2 MIN;
```

and

```
SELECT * FROM a15d1e4s0 SKYLINE OF d2 MIN, d1 MIN;
```

produce the same result. Clearly the same is not true in general for

```
SELECT * FROM a15d1e4s0 ORDER BY d1, d2;
```

and

```
SELECT * FROM a15d1e4s0 ORDER BY d2, d1;
```

Yet from this example alone one gets the intuition that the `SKYLINE OF`-clause can benefit from a richer set of indexes (access paths) than the `ORDER BY`-clause. To be precise we must speak of *pathkeys*, i.e. both of the above skyline queries can make use of the following index, but only the first query with a `ORDER BY`-clause:

```
CREATE INDEX ix_a15d1e4s0_d1 ON a15d1e4s0 (d1, d2);
```

A proof for this property is given in section 2.3.2. We modified the query planner accordingly, c.f. function `query_planner`[13] and `grouping_planner`[14]. The matching of the pathkeys is done in `skyline_pathkeys_contained_in` [15].

This modification has an impact on how the join trees are formed and which access paths are considered.

Pull up subqueries

If a subquery contains a `SKYLINE OF`-clause the query must not be flattened out, i.e. do not pull up the subquery. The function `is_simple_subquery`[16] is modified accordingly.

4.7.2 Preserving relative tuple order

The relative order of tuples is an important physical property of a tuple stream, knowledge about this is encoded by means of *pathkeys* as described in section 4.3.1. As the skyline operator is a full-blown operator we have to provide this information for it as well. In section 3.1.2, 3.2.1, and 3.3.1 respectively, we bring forward arguments why BNL does *not* preserve the relative tuple order, independent from the tuple window policy used, and that SFS and EF *do* preserve the relative tuple order. As the two dimensional algorithm with presort is a special case of SFS it shares the same properties. For the one dimensional case and for our naïve algorithms it is easy to see that they preserve the relative tuple order. The properties are summarized in Table 4.1. Note that for the *established pathkeys* property we cheat a bit here, as the order is actually established by an extra planned sort node and this

[12] `http://www.postgresql.org/docs/8.3/static/planner-optimizer.html`

[13] in `src/backend/optimizer/plan/planmain.c`

[14] in `src/backend/optimizer/plan/planner.c`

[15] in `src/backend/optimizer/path/pathkeys.c`

[16] in `src/backend/optimizer/prep/prepjointree.c`

is only done when the required order is not established beforehand, for instance by an index scan.

This knowledge is encoded in the function `skyline_method_preserves_tuple_order`[17] and `grouping_planner`[18].

Physical Operator	*Preserves Pathkeys*	*Establishes Pathkeys*
1DIM	yes	no
1DIM_DISTINCT	yes	no
MNL	yes	no
2DIM_PRESORT	yes	yes
BNL	no	no
SFS	yes	yes
EF	yes	no

Table 4.1: Tuple Order (Pathkeys) preserving / establishing Property

4.7.3 Physical operator selection

The physical operator selection is rather straightforward. We call the *physical operators* als *methods*. If a method is specified in the skyline options, this method is taken, only some sanity checks are performed.

In the one dimensional case either 1DIM or 1DIM_DISTINCT is selected depending on the usage of `SKYLINE OF DISTINCT`. This happens also if a suitable access path is present, as we do not make use of such an access path yet. We justify this limitation as we believe one dimensional skyline is of limited use anyway (see section 6.3.2).

In the presence of a suitable access path, for two dimensions 2DIM_PRESORT and for higher dimensions SFS is selected.

BNL is the *catch all* default. The physical operator selection is implemented in the function `skyline_choose_method`[19]. We acknowledge that this behavior is hardcoded and it should be more based on cardinality and cost estimation, here space for improvements is left, see section 4.7.5.

4.7.4 Using relation statistics

When using the tuple window placement policy *entropy* a so called *entropy value* for each tuple is computed, see section 4.8.6. In (4.16) it is assumed that the domain for each attribute is $[0, 1]$. For a real world implementation this is too restrictive. To adhere to this restriction we normalize the range (domain) of each attribute to $[0, 1]$. We use PostgreSQL statistics on the base relations to get the information about the range of each attribute. If at least one of them is missing, we fall back to the placement policy *append*. In future work we might engage *sampling* here, see section 6.2.3.

On the downside we noticed that PostgreSQL 8.3.5 does not do a good job with statistics and subqueries, i.e. PostgreSQL does not derive the statistics for the subselect in the query below, even when omitting the `LIMIT`-clause:

```
SELECT *
FROM
    (SELECT * FROM i15d1e6s0 LIMIT 1000) AS a
SKYLINE OF d1 MIN, d2 MIN WITH WINDOWPOLICY=ENTROPY;
```

[17] in `src/backend/utils/skyline/skyline.c`

[18] in `src/backend/optimizer/plan/planner.c`

[19] in `src/backend/utils/skyline/skyline.c`

4.7.5 Cardinality and Cost Estimation

For a full-blown relational operator it is important to compute or at least estimate the *cardinality* of the result, i.e., answer the question how many tuples will be in the output, based on the number of tuples in the input and possible other properties of the input. A related question is what is the computational cost of computing the result. It is sufficient to answer the last question relative to other operators.

Unfortunately neither question is easy to answer. This is especially true for the non-trivial skyline algorithms like BNL and SFS. Of course the question about the cardinality is independent from the algorithms selected.

The first bounds for the number of comparisons required have been derived by Kung et al. [1975], where the number of comparisons for n vectors in d-dimensional space is denoted by $C_d(n)$. Clearly $C_1(n) = n - 1$ and $C_d(n) \leq O(n^2)$ for $d \geq 2$. In their paper they showed:

$$\begin{aligned} C_d(n) &\leq O(n \log n) \quad \text{for} \quad d = 2, 3, && (4.8)\\ C_d(n) &\leq O(n(\log n)^{d-2}) \quad \text{for} \quad d \geq 4, \quad \text{and} && (4.9)\\ C_d(n) &\geq \lceil \log n! \rceil \quad \text{for} \quad d \geq 2 && (4.10) \end{aligned}$$

The first result for the cardinality was obtained by Bentley et al. [1978]. The result they derived is for a set of n vectors in a d-dimensional space under the assumption that all $(n!)^d$ relative orderings are equally probable, the average number of maxima is:

$$O((\ln n)^{d-1}) \tag{4.11}$$

Buchta [1989] refined this result to:

$$\Theta\left(\frac{(\ln n)^{d-1}}{(d-1)!}\right) \tag{4.12}$$

The work of Godfrey [2002] elaborates these results more in the context of skyline queries, but still under strong assumptions, like uniform distribution and unique real-valued attributes. The most recent work we are aware of about cardinality and cost estimation for the skyline operator is the paper of Chaudhuri et al. [2006]. A theorem for estimating the cardinality of a skyline dataset drawn from an arbitrary distribution is derived. This theorem is relaxed to estimate categorical attributes as well. Cost estimation for BNL and SFS is addressed.

In our implementation we took the result from Buchta [1989] for cardinality estimation, see `estimate_skyline_cardinality`[20].

In PostgreSQL the costs are expressed as a linear combination of the following values: `seq_page_cost`, `random_page_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost`, and `cpu_operator_cost` [21].

Our planner/optimizer is not yet cost-based, see section 6.2.1, therefore we took little effort to estimate the costs for the different physical skyline operators, especially the results from Chaudhuri et al. [2006] have not yet been integrated. Nevertheless a skeleton with some flesh is there, see `cost_skyline`[22].

Tuple window size

One could be tempted to set the tuple window size accordingly to the outcome of the cardinality estimation, but the assumption that at no time more than `output_tuples` will be in the window does not hold. To see this imagine the BNL algorithm running on a uniformly

[20] in `src/backend/optimizer/plan/createplan.c`

[21] see `src/backend/optimizer/path/costsize.c`

[22] in `src/backend/optimizer/path/costsize.c`

distributed dataset, which implies the expected result set size is $\Theta(\log(n)^{d-1}/(d-1)!)$ but the order of the tuple stream is such that before seeing the actual skyline tuples a lot of anti-correlated tuples show up in the tuple stream. The tuple window will fill up with non skyline tuples.

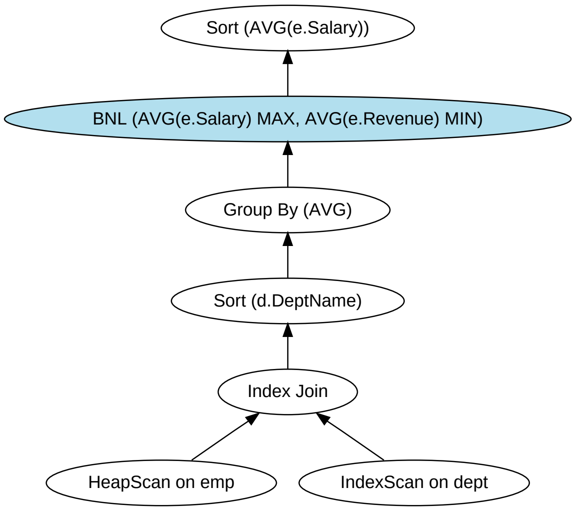
We would like also to point out, that larger window and and less passes do not always mean better runtime, as with larger windows the number of comparisons per tuple could increase and therefore more CPU time is required, so there is a break even point.

`LIMIT / TOP` k

The `LIMIT/TOP` k-clause is used to limit the number of tuples in the result. In the implementation no special measures had to be taken, as the `LIMIT/TOP` k-node will stop fetching tuples as soon as enough tuples have been seen. If a `LIMIT/TOP` k is present we use this information in the cost and cardinality estimation. Also see `skyline_method_can_use_limit`[23].

4.8 Query Execution

The original System R prototype compiled query plans into machine code, whereas other systems like INGRES, the grandfather of PostgreSQL, generated an interpretable query plan. In the 1980s query interpretation was considered a "mistake" by the INGRES authors. Because of the increasing CPU speed (Moore's law) and enabling cross-platform portability, every system now compiles queries into some kind of interpretable data structure; the only difference across systems these days is the level of abstraction [Hellerstein and Stonebraker, 2005]. We stick to the representation as a tree, as it is used by PostgreSQL (see Figure 4.3).



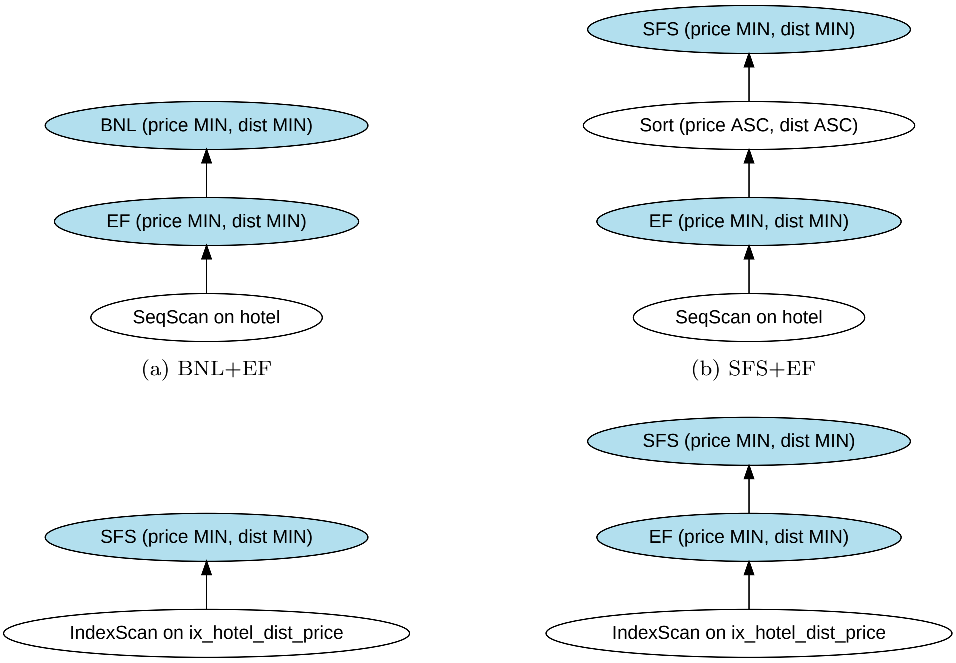
(a) Query Plan

```
SELECT d.DeptName, AVG(e.Salary), AVG(e.Revenue)
FROM emp e JOIN dept d on (e.Dno = d.DeptId)
GROUP BY d.DeptName
SKYLINE OF AVG(e.Salary) MAX, AVG(e.Revenue) MIN
ORDER BY AVG(e.Salary) DESC
```

(b) Query

Figure 4.3: Example query plan for BNL

[23] in `src/backend/utils/skyline/skyline.c`


BNL (price MIN, dist MIN)
EF (price MIN, dist MIN)
SeqScan on hotel
SFS (price MIN, dist MIN)
Sort (price ASC, dist ASC)
EF (price MIN, dist MIN)
SeqScan on hotel
SFS (price MIN, dist MIN)
IndexScan on ix_hotel_dist_price
SFS (price MIN, dist MIN)
EF (price MIN, dist MIN)
IndexScan on ix_hotel_dist_price

(a) BNL+EF

(b) SFS+EF

(c) SFS *w/ index*

(d) SFS+EF *w/ index*

Figure 4.4: Query plans for: `SELECT * FROM hotel SKYLINE OF price MIN, dist MIN;`

4.8.1 Comments on the Pseudo-code

All the pseudo code presented in the following subsections is in a form of state machines, as required for the PostgreSQL query execution engine. One concept that we would like to mention at this point is the *tuple table*. During query execution tuples are stored in a tuple table when passed around between the nodes of the execution plan. A tuple table is made up of a set of *tuple table slots* or *slots* for short. This concept is used throughout the code.

4.8.2 Special case: one dimensional

Although this case is of limited use, the general case algorithms like BNL and SFS can handle this case. Moreover it is expressible in SQL in a very natural way, thus we have a special implementation for this case. A skyline query of this type typically looks like:

```
SELECT * FROM a15d1e4s0 SKYLINE OF d1 MIN;
```

which rewrites to the following standard SQL query:

```
SELECT * FROM a15d1e4s0 WHERE d1 = (SELECT MIN(d1) FROM a15d1e4s0);
```

As can be seen from the query plan below the standard SQL query takes two passes over the input, first finding the minimum by means of aggregation and then filtering the input with the found minimum.

```
                               QUERY PLAN
---------------------------------------------------------------------------------
 Seq Scan on a15d1e4s0  (cost=318.01..636.01 rows=1 width=124)
   Filter: (d1 = $0)
   InitPlan
     ->  Aggregate  (cost=318.00..318.01 rows=1 width=8)
           ->  Seq Scan on a15d1e4s0  (cost=0.00..293.00 rows=10000 width=8)
```

Another way to express this query in standard SQL is:

```
SELECT * FROM a15d1e4s0 o WHERE NOT EXISTS
    (SELECT * FROM a15d1e4s0 i WHERE i.d1 < o.d1);
```

but this is even worse and has a runtime complexity of $O(n^2)$. This approach corresponds to a naïve nested-loop, the query plan is accordingly:

```
                                   QUERY PLAN
---------------------------------------------------------------------------------
 Seq Scan on a15d1e4s0 o  (cost=0.00..1247.10 rows=5000 width=4)
   Filter: (NOT (subplan))
   SubPlan
     ->  Seq Scan on a15d1e4s0 i  (cost=0.00..318.00 rows=3333 width=124)
           Filter: (d1 < $0)
```

We took a different approach, while scanning over the input we collect all minimal tuples in a tuple store. For comparison we need to keep just one tuple. If a tuple is equal to this one, i.e. another potential minimal/maximal tuple we add it to the tuple store. On the other hand, if this tuple is dominated we clean the tuple store, and keep the dominating tuple. After the initial scan over the input we pipe out the tuple store, cf. Listing 4.2. For the sake of a clearer pseudo-code we gave ourselves the freedom to present the tuple store interface (`tuplestore_*`) a little differently to the actual implementation. A functionality we added to the tuple store is to remove all tuples from the tuple store, which is cheaper than recreating it. In the pseudo-code this function is called `tuplestore_clearall`, while in the real implementation its name is `tuplestore_catchup`, which fits better as we are just modifying read and write pointers.

Listing 4.2: Special case: 1 dimensional (in iterator style)

```
function ExecSkyline_1Dim(var state)
    switch state.status
    case INIT:
        tuplestore_init(state.tuplestore);

        resultSlot ← ExecProcNode(outerPlanState(state));

        if resultSlot ≠ NULL
            // input contains at least one tuple
            slot ← ExecProcNode(outerPlanState(state));

            // while not end of input reached
            while slot ≠ NULL
                if slot $\overset{\triangleright}{=}$ resultSlot
                    // we found another potential minimal tuple
                    tuplestore_put(state.tuplestore, slot);
                else if slot $\triangleright$ resultSlot
                    // we found a new lower bound
                    resultSlot ← slot;
                    tuplestore_clearall(state.tuplestore);
                    tuplestore_put(state.tuplestore, slot);

                slot ← ExecProcNode(outerPlanState(state));

        state.status ← PIPEOUT;

        // fall through

    case PIPEOUT:
        if ¬tuplestore_eof(state.tuplestore)
            return tuplestore_get(state.tuplestore);
        else
            tuplestore_done(state.tuplestore);
            state.status ← DONE;
            return NULL;

    case DONE:
        return NULL;
```

4.8.3 Special case: one dimensional with distinct

Again, although this case is of limited use and all general skyline algorithms can deal with it, we implemented a specialized algorithm for this case. It is even simpler than the non-distinct case and it easy to see that a single scan over to input while keeping the current minimal/maximal tuple suffices to compute the skyline in this case, one just has to take care of border cases, such as no tuples in the input, see Listing 4.3. A skyline query of this type looks like:

```
SELECT * FROM a15d1e4s0 SKYLINE OF DISTINCT d1 MIN;
```

and will return at most one tuple, if more than one minimal tuple exists our implementation returns the first.

The SQL 2003 standard [Melton, 2003] does not provide a way to limit the size of a result set. Nevertheless PostgreSQL and other RDBMS do provide syntax for this and consider it during query planning. The query and the query plan for the above query without the skyline operator would look like:

```
SELECT * FROM a15d1e4s0 WHERE d1 = (SELECT MIN(d1) FROM a15d1e4s0) LIMIT 1;
output omitted

EXPLAIN SELECT * FROM a15d1e4s0 WHERE d1 = [...]

                                 QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=318.01..636.01 rows=1 width=124)
   InitPlan
     ->  Aggregate  (cost=318.00..318.01 rows=1 width=8)
           ->  Seq Scan on a15d1e4s0  (cost=0.00..293.00 rows=10000 width=8)
   ->  Seq Scan on a15d1e4s0  (cost=0.00..318.00 rows=1 width=124)
         Filter: (d1 = $0)
```

Please note that again two scans over the input are done, although the second one might stop earlier. For Microsoft SQL Server the same query looks like this:

```
SELECT TOP 1 * FROM a15d1e4s0 WHERE d1 = (SELECT MIN(d1) FROM a15d1e4s0);
```

An ad hoc experiment that we conducted showed that Microsoft SQL Server 2005 produces a very similar execution plan.

In Oracle the pseudo-column ROWNUM can be used to limit the number of tuples returned by a query.

If the relation in question contains a unique identifier with a total order, let us call it id, then the following solution in standard SQL can be found:

```
SELECT * FROM a15d1e4s0 o WHERE NOT EXISTS
    (SELECT * FROM a15d1e4s0 i WHERE i.d1 < o.d1 OR (i.d1 = o.d1 AND i.id < o.id));
```

Note that the condition `i.d1 = o.d1 AND i.id < o.id` is used to select the first one of the minimal tuples, according to the order induced by `id`. The runtime complexity of this query is also in $O(n^2)$.

Listing 4.3: Special case: 1 dimensional distinct (in iterator style)

```
function ExecSkyline_1DimDistinct(var $state$)
    switch $state.status$
    case INIT:
        $resultSlot \leftarrow$ ExecProcNode(outerPlanState($state$));

        if $resultSlot \neq$ NULL
            $slot \leftarrow$ ExecProcNode(outerPlanState($state$));

            // while not end of input reached
            while $slot \neq$ NULL
                if $slot \rhd resultSlot$
                    $resultSlot \leftarrow slot$;

                $slot \leftarrow$ ExecProcNode(outerPlanState($state$));

        $state.status \leftarrow$ DONE;
        return $resultSlot$;

    case DONE:
        return NULL;
```

4.8.4 Special case: two dimensional with presort

The two dimensional case of skyline computation is usually the smallest case considered in literature and of course BNL and SFS can deal with it. Nevertheless as pointed out in [Börzsönyi et al., 2001] skylines for the two dimensional case can be computed in a simple way if the input relation is ordered according to the skyline criteria, then the skyline can be computed with a single scan over the input, while keeping just the most recently found skyline tuple, see Listing 4.4. Of course in the presence of a suitable access path the input needs not to be sorted, performing an index scan suffices. This method is simple to implement, and we believe the two dimensional case is already a quite useful one, from a user's perspective.

An interesting aspect of this algorithm is how it deals with the `DISTINCT` modifier for the `SKYLINE OF`-clause. For the semantics of the `DISTINCT`-modifier see section 2.5.1. In the pseudo-code in Listing 4.4 `DISTINCT` is a Boolean variable, which is *true* if the `DISTINCT` modifier is present, and *false* otherwise. Let p be the current skyline tuple, then a tuple q becomes the next skyline if the following condition holds:

$$q \rhd p \vee (q \overset{\rhd}{=} p \wedge \texttt{DISTINCT}). \tag{4.13}$$

If `DISTINCT` is *true*, then (4.13) simplifies to:

$$\begin{aligned} q \rhd p \vee (q \overset{\rhd}{=} p \wedge \top) &= q \rhd p \vee q \overset{\rhd}{=} p \\ &\overset{\text{by (2.7)}}{=} q \trianglerighteq p \end{aligned} \tag{4.14}$$

Otherwise if `DISTINCT` is *false*:

$$q \rhd p \vee (q \overset{\rhd}{=} p \wedge \bot) = q \rhd p \tag{4.15}$$

Listing 4.4: Special case: 2 dimensional with presort (in iterator style)

```
function ExecSkyline_2DimPresort(var $state$)
    switch $state.status$
        case INIT:
        $resultSlot \leftarrow$ ExecProcNode(outerPlanState($state$));

        if $resultSlot$ = NULL
            // input does not contain a single tuple
            $state.status \leftarrow$ DONE;
            return NULL;

        $state.status \leftarrow$ PROCESS
        // the first tuple is already a skyline tuple
        return $resultSlot$;

    case PROCESS:
        forever
            $slot \leftarrow$ ExecProcNode(outerPlanState($state$));

            if $slot$ = NULL
                // end of input reached
                $state.status \leftarrow$ DONE;
                return NULL;

            if $slot \rhd returnSlot \lor (slot \overset{\rhd}{=} returnSlot \land$ DISTINCT$)$
                // found another skyline tuple
                $returnSlot \leftarrow slot$;
                return $returnSlot$;

    case DONE:
        return NULL;
```

4.8.5 Naïve method: *MNL: materialized nested loop*

This method is the first general method which can deal with any dimensionality and treat the DISTINCT modifier. It directly corresponds to the naïve $O(n^2)$ nested loop approach, compare every tuple to all others as long as no witness is found, which dominates the current tuple. Why is it called *materialized* nested loop? This *materialized* stems from the fact that we materialize the tuples from the outer plan, to be able to read them over and over again, this is because the skyline operator can sit at any non-leaf node in the query plan and not only on top of a table or index scan. To handle the DISTINCT case, we just use the counters *state.pos* and *innerPos* and favor the first tuple.

Listing 4.5: Materialized Nested Loop (MNL) (in iterator style)

```
function ExecSkyline_MaterializedNestedLoop(var $state$)
    switch $state.status$
    case INIT:
        ExecMarkPos(outerPlanState($state$));
        $state.pos \leftarrow 0$;
        $state.status \leftarrow$ PROCESS

        // fall through

    case PROCESS:
        forever
            // get next current tuple
            ExecRestrPos(outerPlanState($state$));
            $slot \leftarrow$ ExecProcNode(outerPlanState($state$));
            $state.pos \leftarrow state.pos + 1$;
            ExecMarkPos(outerPlanState($state$));

            if $slot$ = NULL
                // end of input reached
                $state.status \leftarrow$ DONE;
                return NULL;

            // dominance check
            ExecReScan(outerPlanState($state$));

            $innerPos \leftarrow 0$;

            forever
                $innerSlot \leftarrow$ ExecProcNode(outerPlanState($state$));
                $innerPos \leftarrow innerPos + 1$;

                if $innerSlot$ = NULL
                    // no witness found
                    return $slot$;

                if $innerSlot \triangleright slot$
                    break;
                else if $innerSlot \overset{\triangleright}{\equiv} slot \wedge$ DISTINCT
                    // favor the first
                    if $innerPos < state.pos$
                        break;
        break;

    case DONE:
        return NULL;
```

4.8.6 Tuple Window

A tuple window is a data structure used by BNL, SFS and EF to store skyline (candidate) points. As internal representation we used a double linked list with a sentinel (cf. [Cormen et al., 2001, Page 204–209]). Using a sentinel simplifies the code to manipulate a double linked list a lot.

The external interface is a simple iterator interface with a few more functions[24]:

- `TupleWindowState *tuplewindow_begin(**int** maxKBytes, **int** maxSlots, TupleWindowPolicy policy)` creates a tuple window. The maximum size of the tuple window is limited to `maxKBytes` KB or `maxSlots` slots. The number of slots is the stronger limit. For policy see the next subsubsection. The returned value must be used in all subsequent calls to the tuple window interface functions.
- `**void** tuplewindow_end(TupleWindowState *state)` destroys a tuple window and frees all resources.
- `**void** tuplewindow_clean(TupleWindowState *state)` removes all tuples from the tuple window.

[24] see `src/include/utils/tuplewindow.h`

- `bool tuplewindow_has_freespace(TupleWindowState *state)` returns true if the tuple window can hold at least one more tuple. Note that we ignore the fact that the tuple that is going to be inserted might consume a little more space than is actually left, but this eases the implementation and an existing PostgreSQL interface, the `tuplestore` interface, is designed in the same way.

- `void tuplewindow_setinsertrank(TupleWindowState *state, double rank)` when policy *entropy* or *ranked* is used, the rank of the new tuple has to be set with this function before starting to scan over the tuple window, so ensure to call `tuplewindow_rewind` prior to this call.

- `void tuplewindow_puttupleslot(TupleWindowState *state, TupleTableSlot *slot, int64 timestamp, bool forced)` inserts the tuple in the tuple table slot `slot` into the tuple window with the given timestamp. For a ranked window policy `forced` can be set to true, i.e. if the new tuple has a higher rank than the lowest ranked tuple in the window then this lowest ranked tuple is removed — if the space for the higher ranked tuple is needed. We use this especially for the implementation of the elimination filter. We speak of a *forced insert*. Currently the `timestamp` argument is only used by BNL and in all other algorithms it is set to zero.

- `void tuplewindow_rewind(TupleWindowState *state)` sets the *cursor* to the first tuple.

- `bool tuplewindow_ateof(TupleWindowState *state)` returns true if the cursor is at the end of the tuple window.

- `void tuplewindow_movenext(TupleWindowState *state)` moves the cursor to the next tuple.

- `void tuplewindow_removecurrent(TupleWindowState *state)` removes the current tuple. Note that this modifies the cursor. So when calling `tuplewindow_removecurrent`, calling `tuplewindow_movenext` afterwards skips over a tuple.

- `bool tuplewindow_gettupleslot(TupleWindowState *state, TupleTableSlot *slot, bool removeit)` gets the tuple at the current position of the cursor. When called with `removeit` true, the returned tuple is removed from the tuple window. This is not the same as calling `tuplewindow_gettupleslot` and `tuplewindow_removecurrent`, as `tuplewindow_removecurrent` also frees the tuple itself and not only removes it from the tuple window.

- `int64 tuplewindow_timestampcurrent(TupleWindowState *state)` returns the timestamp of the tuple at cursor position. Currently only used by BNL.

Placement Policies

With tuple window placement policy we want to express how tuples are organized in the window and in which order they are compared against each other [Godfrey et al., 2007]. The implementation supports four different placement policies:

1. *append:* a new tuple is always inserted at the *end* of the list

2. *prepend:* instead of the end, new tuples are inserted at the *beginning* of the list

3. *entropy:* tuples are kept in *rank descending order*. The insertion position is found in $O(n)$, this is done by computing the rank before starting to scan over the window and the pointer is only incremented as long as the rank of the tuple to be inserted is smaller or equal to the rank of the current tuple. We choose this approach because the tuple window is scanned anyway. With this policy and a *forced insert*, in case the

tuple window is full, the tuple with the lowest ranking is removed from the window to make space for a higher ranked tuple.

Let r be a tuple and $r_1, \ldots, r_m$ the attributes the skyline is computed on with $\forall i \in \{1, \ldots, m\}\colon r_i \in [0, 1]$, then the entropy value $E(r)$ for the tuple r is defined as (see [Chomicki et al., 2003, 2005]):

$$E(t) = \sum_{i=1}^{m} \ln(r_m + 1) \tag{4.16}$$

4. *random:* behaves like *entropy* but instead of a computed entropy value for a tuple a random value is used.

4.8.7 BNL

The implementation of the BNL algorithm is based on our fixed version of [Börzsönyi et al., 2001], see Listing 3.2. In the following we give a pseudo-code in terms of a state machine as required by the PostgreSQL query execution engine, see Listing 4.6. This implementation is fully general, it can deal with any dimension ≥ 1 and can treat the `DISTINCT` modifier. To ensure BNL is used for query execution specify the `BNL` keyword as an option to the skyline clause. All other options that can be specified effect the tuple window.

Listing 4.6: Block Nested Loop (BNL) (in iterator style)

```
function ExecSkyline_BlockNestedLoop(var state)
    forever
        switch state.status
        case INIT:
            state.source ← OUTER;
            tuplestore_init(state.tempOut);
            tuplewindow_init(state.window);
            state.tsIn ← 0;
            state.tsOut ← 0;
            state.status ← PROCESS;
            break;

        case PROCESS:
            if state.source = OUTER
                slot ← ExecProcNode(outerPlanState(state));
            else
                slot ← tuplestore_get(state.tempIn);

            if slot = NULL
                if state.source = TEMP
                    tuplestore_done(state.tempIn);

                if state.tsOut = 0
                    // nothing written to temp -> done
                    tuplestore_done(state.tempOut);
                    tuplewindow_rewind(state.window);
                    state.status ← FINALPIPEOUT;
                    break;

                state.source ← TEMP;
                state.tempIn ← state.tempOut;
                tuplestore_init(state.tempOut);
                state.tsIn ← 0;
                $state.tsOut \leftarrow 0$;
                tuplewindow_rewind($state.window$);
                $state.status \leftarrow$ PIPEOUT;
                break;

            $state.tsIn \leftarrow state.tsIn + 1$;

            tuplewindow_rewind($state.window$);
            if $state.ranked$
                tuplewindow_setinsertrank($state.window$, ExecSkylineRank($state$, $slot$);

            forever
                if tuplewindow_ateof($state.window$)
                    if tuplewindow_hasfreespace($state.window$)
                        tuplewindow_put($state.window$, $slot$);
                    else
                        tuplestore_put($state.tempOut$, $slot$);
                        $state.tsOut \leftarrow state.tsOut$;

                    break;

                $innerSlot \leftarrow$ tuplewindow_current($state.window$);

                if $innerSlot \triangleright slot \vee (innerSlot \overset{\triangleright}{=} slot \wedge \texttt{DISTINCT})$
                    break;
                else if $slot \triangleright innerSlot$
                    tuplewindow_removecurrent($state.window$);
                else
                    tuplewindow_movenext($state.window$);

            tuplewindow_rewind($state.window$);
            $state.status \leftarrow$ PIPEOUT;
            break;

        case PIPEOUT:
            while $\neg$tuplewindow_ateof($state.window$)
                if tuplewindow_timestampcurrent($state.window$) = $state.tsIn$
                    return tuplewindow_get($state.window$);

                tuplewindow_movenext($state.window$);

            $state.status \leftarrow$ PROCESS;
            break;

        case FINALPIPEOUT:
            if $\neg$tuplewindow_ateof($state.window$)
                return tuplewindow_get($state.window$);

            tuplewindow_done($state.window$);
            $state.status \leftarrow$ DONE;
            return NULL;

        case DONE:
            return NULL;
```

4.8.8 SFS

Listing 3.3 translates straightforward into a state machine, the way it fits into the query execution engine, see Listing 4.7.

Listing 4.7: Sort First Skyline (SFS) (in iterator style)

```
function ExecSkyline_SortFilterSkyline(var state)
    forever
        switch state.status
        case INIT:
            state.source ← OUTER;
            tuplestore_init(state.tempOut);
            tuplewindow_init(state.window);
            state.neednextrun ← false;
            state.status ← PROCESS;
            break;

        case PROCESS:
            if state.source = OUTER
                slot ← ExecProcNode(outerPlanState(state));
            else
                slot ← tuplestore_get(state.tempIn);

            if slot = NULL
                if state.source = TEMP
                    tuplestore_done(state.tempIn);

                if ¬state.neednextrun
                    // nothing written to temp -> done
                    tuplestore_done(state.tempOut);
                    tuplewindow_rewind(state.window);
                    state.status ← DONE;
                    return NULL;

                state.source ← TEMP;
                state.tempIn ← state.tempOut;
                tuplestore_init(state.tempOut);
                tuplewindow_clean(state.window);
                state.neednextrun ← false;
                break;

            tuplewindow_rewind(state.window);
            if state.ranked
                tuplewindow_setinsertrank(state.window,  ↩
                    → ExecSkylineRank(state, slot);

            forever
                if tuplewindow_ateof(state.window)
                    if tuplewindow_hasfreespace(state.window)
                        tuplewindow_put(state.window, slot);
                        return slot;
                    else
                        tuplestore_put(state.tempOut, slot);
                        state.neednextrun ← true;

                    break;

                innerSlot ← tuplewindow_current(state.window);

                if $innerSlot \rhd slot \lor (innerSlot \overset{\trianglerighteq}{=} slot \land$ DISTINCT)
                    break;

                tuplewindow_movenext($state.window$);

            break;

        case DONE:
            return NULL;
```

4.8.9 Elimination Filter (EF)

The pseudo-code is already given in section 3.3.1, cf. Listing 3.4. We have used the elimination filter with SFS and BNL, see the following sections.

4.8.10 SFS+EF (a LESS Variant)

In [Godfrey et al., 2005] Godfrey et al. proposed the LESS algorithm, which is an improvement of SFS due to the introduction of the concept of *elimination filter* (EF). With the original LESS algorithm the elimination filtering is carried out in the pass zero of the external sort routine to eliminate records quickly.

In our case, in order to integrate the elimination filter into the SFS algorithm, while at the same time preserving the utilization of the physical sort operator, we implemented the elimination filter *before* the sort routine. Speaking in terms of the iterator tree, the EF node is a child of the sort node, which in turn is a child of the SFS node (see 4.4b). Furthermore, we do not integrate the skyline computation into the final pass of the merge sort phase, as described in [Godfrey et al., 2005]. Therefore, technically speaking, SFS+EF is not an equivalent implementation of LESS. However, the elimination filter is preserved in SFS+EF, which is essentially the gist of LESS.

The implementation of EF is similar to BNL. An elimination filter window is maintained in the main memory. We use 8 KB as default window size. The difference between EF and BNL is that EF does not write tuples to a temporary file if the tuple window is full, and that the relative order of tuples going through the EF is preserved.

4.8.11 BNL+EF

Inspired from the idea of the elimination filter in LESS, we experimented with a new combination of BNL and EF. That is, the elimination filter is executed before the BNL algorithm. Since our implementation of the elimination filter is independent from the external sort routine, the coding of this variant is straightforward. It turns out that the algorithm BNL+EF is a substantial improvement to BNL.

4.9 Development

In this section we share some of the experiences we made during development.

4.9.1 Source Code and Version Control

We started out our implementation effort of the skyline operator, when `CVS HEAD` of the PostgreSQL source repository was `8.3-devel`, i.e. the version 8.3 of PostgreSQL was currently under development. PostgreSQL's source is kept in a Concurrent Versions System

(CVS) repository. We have used the very handy Tortoise CVS to checkout and update our copy of the PostgreSQL source. We keep our source in a SubVersion (SVN) repository, and use TortoiseSVN to interact with our repository.

For each CVS tag we are interested in, namely `CVS HEAD` and `REL8_3_STABLE`, we keep the entire source tree as a branch in our SVN repository, including the CVS metadata directories `CVS`. By means of that we continuously merged `CVS HEAD` into our main development branch, called `trunk` in the SubVersion terminology. In order to have a more stable and reproducible setting we decided to base this thesis on the branch `REL8_3_STABLE`, i.e. all experiments in this thesis are based on Version 8.3.0 of PostgreSQL with our patch for the skyline operator applied.

Having the entire CVS source tree in the SVN repository is handy, it was quite easy to merge in the changes from `CVS HEAD` into `trunk`. To support this task we wrote two scripts, as CVS detects changes based on the file modification date/time, where SVN really looks at the file content.

- `cvs-entries-normalize.sh:` sorts the content of the `CVS/Entries*` files, to have a normalized form, and minimizes the changes that are committed to the SVN repository.

- `resetcvsts.pl:` sets the file date/time according to the values in `CVS/Entries`, this allows to easily move around the working copy across different machines.

Another benefit of using SVN was the smooth integration with an issue/bug-tracker called Trac. Trac has a wiki, milestones, all kinds of reports and the like, and proved itself as very handy for this size and type of project.

4.9.2 Debugging PostgreSQL

PostgreSQL creates one back-end process, which corresponds to an operating system process, per client connect. This property makes debugging a bit unpleasant. Nevertheless there is a good work around for this, just operate the PostgreSQL in *single user mode*. This gives one a prompt, to enter queries directly and the query is evaluated in the same process. So debugging or, with Microsoft Visual C++, even *edit & continue*[25] was possible. For further details see the PostgreSQL documentation on *postgres*[26]

4.9.3 Regression Testing

To facilitate regression testing we also integrated a test case in the PostgreSQL regression test suite. This test case tests just a few very basic scenarios, as a thorough test would take quite some time, and this would not be appreciated as part of a checked build. For the basic regression test see `src/test/regress/sql/skyline_base.sql` and `src/test/regress/expected/skyline_base.out`.

For a more extensive test see the Perl script `slregress.pl`, here a larger set of test cases is produced and the result is always checked against the query expressed in standard SQL. Instead of a table the random dataset returning function as described in section 5.1.2 is used.

We ran both, the PostgreSQL regression test suite and our Perl script `slregress.pl`, regularly during development to ensure that we did not break other parts of PostgreSQL and an already implemented feature of the skyline operator, while we were integrating new parts or optimizing and/or refactoring other parts.

[25] Edit & continue is the ability of modifying the source code while debugging, applying the changes and continuing to debug. Kind of deluxe for a C compiler.

[26] `http://www.postgresql.org/docs/8.3/static/app-postgres.html`

Chapter 5

Results

5.1 Experimental Setup

We ran our experiments on Dell OptiPlex 755 computers, with an Intel Pentium Dual-Core CPU E2160 (1.80GHz, 1MB L2 cache), with 1 GB RAM, and with a Seagate Barracuda 160 GB HDD (7200 rpm, SATA 3.0Gb/s, 8 MB cache) with a single NTFS partition, running Microsoft Windows XP (SP2). Our implementation was based on PostgreSQL 8.3.0 and was compiled using Microsoft Visual Studio 2005 (SP1) using the `RELEASE` configuration with assertions disabled. PostgreSQL was configured to use 200 MB of RAM for shared buffers, with auto vacuuming disabled, and all other settings were left as default.

If not otherwise mentioned, a tuple window size of 1024 KB (equal to the PostgreSQL `work_mem` setting), and an EF tuple window size of 8 KB (equal to the PostgreSQL block size) were used.

The tables for the test runs have been generated using the extended version [Eder, 2007] of the dataset generator presented in [Börzsönyi et al., 2001], which allows to generate tables with different initial seeds for the random generator. All experiments were carried out on three different sets with varying initial seed, for each distribution type (independent, correlated, and anti-correlated), and with 100, 500, 1k, 5k, 10k, 50k, 100k tuples.

Each tuple consists of a unique 4 byte integer *id* and 15 randomly generated 8 byte floats $d_1, \ldots, d_{15}$. With a 23 byte tuple header and the alignment it sums up to a tuple length of 152 bytes. The memory chunk used by the PostgreSQL memory allocation function (`palloc`) was 264 bytes long.

For the experiments using an index, a duplicate of the same set of tables was used, including six indexes, where index k is an index on $d_1, \ldots, d_k$. This all sums up to 126 tables and a database of approximately 1 GB. For most of our experiments we did not consider tables with more than 100k tuples, as the runtime for a 15 dimensional skyline query on 100k tuples went up to 30 minutes. Consult section 5.2.3 for the total computational workload done in the experiments.

All experiments were carried out with a *hot disk cache*, i.e. due to appropriate queries all pages were in the shared buffers. See section 5.2 for more details.

5.1.1 Lesson learned

A lesson we have painfully learned during our experiments is to minimize all possible influences on the runs to avoid skewed results. This especially includes: screen savers, power saving or standby options and any form of automatic software update such as Microsoft Windows Update or Google Pack software updater.

5.1.2 Random Dataset Generator

For our experiments we used a modified version [Eder, 2007] of the popular dataset generator from [Börzsönyi et al., 2001]. Our modified version can be used as command line utility or as a PostgreSQL module. The PostgreSQL module provides a set returning function and creates datasets on the fly. The details are described in the following sections. But first of all we describe what types of datasets are generated by the dataset generator.

Independent, Correlated, and Anti-Correlated Datasets

The main parameters one can vary for a generated dataset are *cardinality*, i.e. number of tuples, *dimensionality* and the *distribution type* and types are as of [Börzsönyi et al., 2001]: *independent*, *correlated*, and *anti-correlated*.

Before we give a detailed description of these distribution types, we show some of the basic building blocks used for generating them, as we believe the source code of the implementation is the most precise way to describe it:

```
static double
random_equal(double min, double max)
{
    double x = (double) rand() / RAND_MAX;
    return x * (max - min) + min;
}
```

As expected `random_equal()` returns a random value $x \in [\mathrm{min}, \mathrm{max}]$.

```
static double
random_peak(double min, double max, int dim)
{
    int     d;
    double  sum = 0.0;

    for (d = 0; d < dim; d++)
        sum += random_equal(0, 1);
    sum /= dim;
    return sum * (max - min) + min;
}
```

The function `random_peak()` returns a random value $x \in [\mathrm{min}, \mathrm{max}]$ as sum of `dim` equally distributed random values.

```
static double
random_normal(double med, double var)
{
    return random_peak(med - var, med + var, 12);
}
```

The function `random_normal()` is our way to generate a normally distributed value $x \in (\mathrm{med} - \mathrm{var}, \mathrm{med} + \mathrm{var})$ with expected value $E[x] = \mathrm{med}$. This implementation is motivated through the central limit theorem and based on the observation that a 12-fold sum of $[0, 1]$ uniformly distributed random value yields a sufficient good normally distributed value. A pre-requirement for this to work, is that the 12 uniformly distributed values are independent. While this cannot be guaranteed for `random_equal()` we verified that data generated by `random_normal()` are sufficiently normally distributed with the Shapiro-Wilk test.

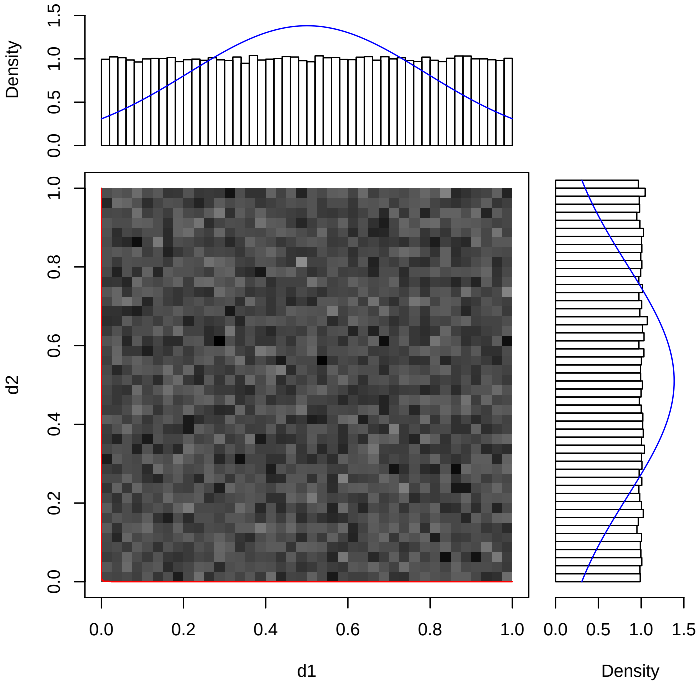
Now we focus on the different distribution types and how they are generated and we show some of their properties:

- *indep*: for this type of dataset, all attribute values are generated independently using a uniform distribution. Figure 5.1 shows a density plot and statistics of such an independent dataset with 100k tuples and $d = 2$. The skyline tuples of this dataset are the lower left corners of the red line, where the red line is the “skyline”. The density of points within a specific subregion is indicated by grayscale values, the darker the more points are in the subregion. Above and to the right of the grayscale density plot is the border distribution for each dimension, in this plots the blue line indicates the density of a normal distribution with the same mean and standard deviation as the border distribution.

  The implementation is as straightforward as expected:

```
int      d;

for (d = 0; d < dim; d++)
     x[d] = random_equal(0, 1);
```



(a) Density Plot

	d1	d2
mean	0.500	0.500
var	0.083	0.083
cor	d1	d2
d1	1.000	0.000
d2	0.000	1.000

(b) Statistics

Figure 5.1: Independent dataset (100k tuples)

- *corr*: a correlated dataset represents an environment in which points which are good in one dimension are also good in the other dimensions.

  A vector x with the dimension *dim* is generated in the following way:

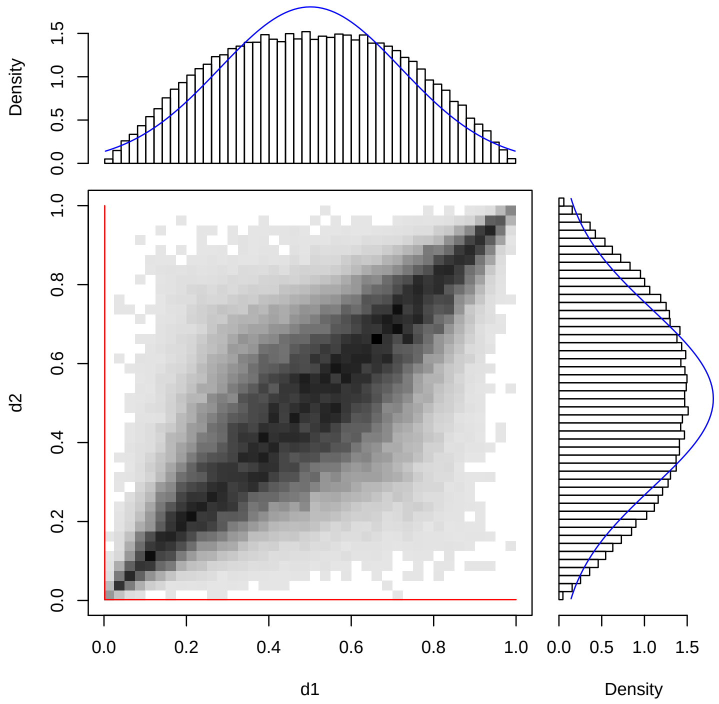
```
do
{
    int     d;
    double  v = random_peak(0, 1, dim);
    double  l = v <= 0.5 ? v : 1.0 - v;

    for (d = 0; d < dim; d++)
        x[d] = v;

    for (d = 0; d < dim; d++)
    {
        double h = random_normal(0, l);
        x[d] += h;
        x[(d + 1) % dim] -= h;
    }
} while (!is_vector_ok(dim, x));
```

  Due to the way it is computed any $x[d]$ could get out of the bound, `is_vector_ok()` verifies that all coordinates of x are within the interval $[0, 1]$.

  Figure 5.2 shows a correlated dataset with 100k tuples for $d = 2$.



(a) Density plot

	d1	d2
mean	0.500	0.500
var	0.049	0.049
cor	d1	d2
d1	1.000	0.717
d2	0.717	1.000

(b) Statistics

Figure 5.2: Correlated dataset (100k tuples)

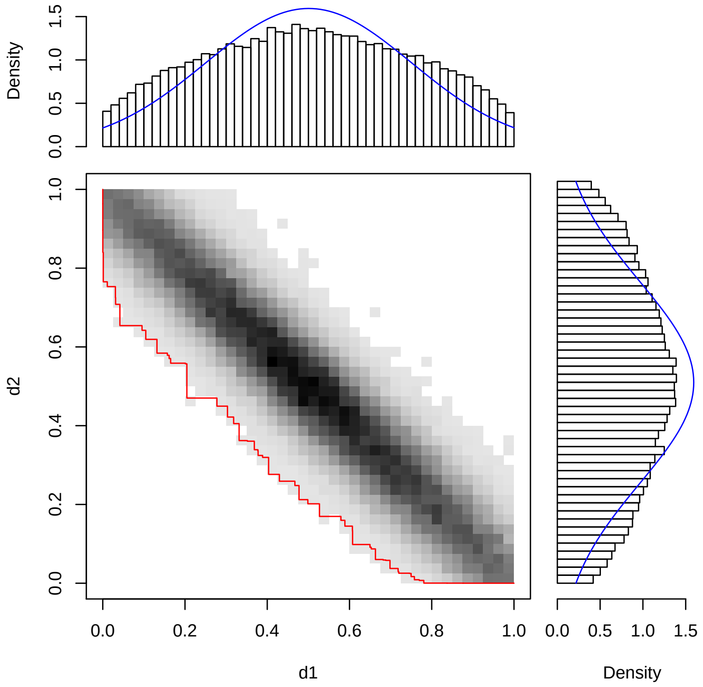
- *anti*: an anti-correlated dataset represents an environment in which points which are good in one dimension are bad in one or all of the other dimensions. The standard example used in the skyline literature falls into this category: hotels are either cheap and far away from the beach or expensive and close to the beach.

```
do
{
    int     d;
    double  v = random_normal(0.5, 0.25);
    double  l = v <= 0.5 ? v : 1.0 - v;

    for (d = 0; d < dim; d++)
        x[d] = v;

    for (d = 0; d < dim; d++)
    {
        double h = random_equal(-l, l);
        x[d] += h;
        x[(d + 1) % dim] -= h;
    }
} while (!is_vector_ok(dim, x));
```

Figure 5.3 shows an anti-correlated dataset with 100k tuples for $d = 2$.



(a) Density Plot

	d1	d2
mean	0.500	0.500
var	0.063	0.063
cor	d1	d2
d1	1.000	-0.944
d2	-0.9444	1.000

(b) Statistics

Figure 5.3: Anti-correlated dataset (100k tuples)

For the very details of the random dataset generator, please have a look at the implementation which is available from [Eder, 2007].

As command line utility

The most beneficial use case for the command line version of the random dataset generator is to pipe its output directly into the database. The usage message explains how to use the command line utility:

```
$ randdataset -?
Test Data Generator for Skyline Operator Evaluation
Usage: randdataset (-i|-c|-a) -d DIM -n COUNT [-s SEED] [-p] [-S] [-h|-?]

Options:
       -i       independent (dim >= 1)
       -c       correlated (dim >= 2)
       -a       anti-correlated (dim >= 2)

       -d DIM   dimensions >=1
       -n COUNT number of vectors
       -I       unique id for every vector
       -p PAD   add a padding field, PAD characters long

       -C       generate SQL COPY statement
       -R       generate SQL CREATE TABLE statement
       -T NAME  use NAME instead of default table name

       -s SEED  set random generator seed to SEED

       -S       output stats to stderr

       -h -?    display this help message and exit

Examples:
       randdataset -i -d 3 -n 10 -I -R
       randdataset -a -d 2 -n 100 -S
```

An invocation of `randdataset` suitable to pipe it directly into `psql`, i.e. load it into database, could look like this:

```
$ randdataset -i -d 3 -n 10 -I -R
DROP TABLE IF EXISTS "i3d10";
CREATE TABLE "i3d10" (id int, d1 float, d2 float, d3 float);
COPY "i3d10" (id, d1, d2, d3) FROM STDIN DELIMITERS ’,’ CSV QUOTE ’’’’;
1,0.000000000000000e+00,6.900010321708401e-01,5.054183972559023e-01
2,5.914905399975788e-01,5.547849133400177e-01,3.784288188342139e-01
[...]                                                     7 rows omitted
10,8.585111004572880e-01,9.898449857671955e-02,8.770675234855467e-01
\.
```

PostgreSQL Random Dataset Generator Function

To be able to generate test datasets on the fly, i.e. without explicitly creating tables and filling them up, we created a PostgreSQL module which implements the functionality of `randdataset` as a set returning function. To create, backup and restore a test database with an appropriate set of test tables can be very time consuming. Having such a function at hand is useful in many situations, all it takes is to run PostgreSQL's `initdb` and installing the set returning function. We used it many times after merging the PostgreSQL 8.3-devel branch into our branch, which often made it necessary to re-`initdb` the database. Especially for regression tests and during debugging this function was very handy. PostgreSQL's concept of *set returning functions*[1] allows to issue queries as such:

```
db=# SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)
```

This concept of set returning functions is quite flexible and allows also to return complex data types such as records. To test drive this PostgreSQL 'contrib' module (see (contrib/randdataset)), without installing it, even without installing PostgreSQL visit our web-interface at `http://skyline.dbai.tuwien.ac.at/`. To install it onto your PostgreSQL installation follow the instructions at `contrib/randdataset/README.randdataset`. Once installed the function can be used as follows:

```
db=# SELECT rds.id, rds.d1, rds.d2 FROM                 columns are referred by name
db-# pg_rand_dataset('indep', 2, 10, 0) AS     2 dim independent dataset with 10 tuples
db-# rds(id int, d1 float, d2 float);                          define names and types
 id |         d1         |         d2
----+--------------------+--------------------
  1 |  0.170828036112165 |  0.749901980510867
  2 | 0.0963716553972902 |  0.870465227342427
[...]                                                                 7 rows omitted
 10 |  0.715538595204958 | 0.0830042460388524
(10 rows)

db=# SELECT rds.* FROM
db-# pg_rand_dataset('corr', 3, 10, 0) AS                 3 dim, correlated, 10 tuples
db-# rds(id int, d1 float, d2 float, d3 float);
 id |        d1         |        d2         |        d3
----+-------------------+-------------------+-------------------
  1 | 0.482522432069401 | 0.327330244422693 | 0.207248995528228
  2 | 0.448836206689892 | 0.512027726159258 | 0.600054676092416
[...]                                                                 7 rows omitted
 10 | 0.319189108559988 | 0.407713612758059 |  0.30165467053249
(10 rows)

db=# SELECT rds.* FROM
db-# pg_rand_dataset('anti', 3, 20, 1) AS      3 dim, anti-correlated, 20 tuples, seed 1
db-# rds(id int, d1 float, d2 float, d3 float);
```

output omitted, as it is similar to above, except another distribution, more tuples, and another initial seed is used

The general signature for `pg_rand_dataset` is:

```
FUNCTION pg_rand_dataset(disttype text, dim int, rows int, seed int)
RETURNS setof record
```

Where the arguments have the following meaning:

- `disttype` specifies the distribution type (see section 5.1.2), allowed values are `'indep'`, `'corr'`, and `'anti'`
- `dim` specifies the number of dimensions, allowed values are 1 up to 20
- `rows` specifies the number of tuples that should be returned, and

[1] `http://www.postgresql.org/docs/8.3/static/functions-srf.html`

- seed is the seed for the random generator

We ensured that calling pg_rand_dataset with the same arguments will always return the same result set. In the PostgreSQL terminology this property of a function is called IMMUTABLE (see the PostgreSQL documentation on *Function Volatility Categories*[2]).

To make the usage more comfortable we defined a set of wrapper functions rds$k$d for the dimensions $k \in \{1, \ldots, 20\}$, they allowed us to issue the same queries as above in the following way (see contrib/randdataset/randdataset.sql.in):

```
db=# SELECT rds.* FROM rds2d('indep', 10, 0) AS rds;
[...]                                   output completely omitted, as it is the same as above

db=# SELECT rds.* FROM rds3d('corr', 10, 0) AS rds;
[...]                                                                           same here

db=# SELECT rds.* FROM rds3d('anti', 20, 0) AS rds;
[...]                                                                           same here
```

Buffer for Set returning function

In the PostgreSQL execution engine a function scan node is responsible for retrieving tuples from a set returning function. The current implementation reads all the tuples from the set returning function and stores them in a tuplestore before returning the first tuple to the caller. A tuplestore maintains an in memory buffer; once it runs out of memory the remaining tuples are written to a tempfile. The size of the in memory buffer is controlled through the configuration variable work_mem. The default value is 1024 KB. This configuration variable is also used to determine the amount of RAM used for sort operations, for merge joins, hash joins and many more. Hence the total amount of RAM used for a single query can be multiple times the value of work_mem. In our setting we use the configuration variable work_mem for the sort operation in SFS and as the default tuple window size for skyline algorithms that require a tuple window. In the following query PostgreSQL calls generate_series a million times and will discard all tuples except one due to the LIMIT 1-clause.

```
db=# SELECT * FROM generate_series(1,1000000) LIMIT 1;
 generate_series
-----------------
               1
(1 row)
```

This behavior is not desirable, at least we want that the tuples do not get swapped out to disk as tuplestore's in memory buffer gets full, since for parts of our experiments we were not willing to pay I/O penalty as we were studying just the CPU bounded behavior of the skyline algorithms.

On the other hand we do not want to increase work_mem, as this would, as mentioned above, influence the sort operation and the size of the tuple window for the skyline algorithms. Therefore we decided to introduce a new configuration variable function_scan_work_mem, solely for the tuplestore in the function scan node. The following utility query sets this configuration variable to 100 MB;

```
db=# SET function_scan_work_mem  = 102400;
```

A patch for this feature can be found at patches/function_scan_work_mem.diff.

[2] http://www.postgresql.org/docs/8.3/static/xfunc-volatility.html

5.2 Experimental Environment

We believe the systems environment in which we carried out our experiments is interesting on its own and therefore we sketch it in this section. First of all we give some definitions of terms we use to describe the experimental environment and how they relate to each other. To distinguish these terms from the general terms with the same name we start them with an upper case letter, i.e. Job vs. job.

Definition 13 (Query)**.** *A Query corresponds to a single SQL query with meta data embedded in comments.*

To embed meta data we are using the fact that "`--`" starts a single line comment in SQL, by means of that we can safely pass any extra information through SQL processor `psql` to the post processing steps. In our case to the programs that parse and analyze the log files. We use only the following types of meta data (see Figure 5.4, 5.5 and 5.6):

- `--<pid pid=`*pid*`/>`; *pid* identifies the machine the Job ran on.
- `--<comment>` and `--</comment>`; a Setup Query, see next definition. Queries between these tags are of course executed, but in a later stage just commented out.
- `--<query runid=`*runid*`>` and `--</query>`; A Run cf. definition below. Please note that the corresponding Setup Queries are not nested within the Run, they just precede the Run.
- `--<queryplan>` and `--</queryplan>`; is used in a later stage to preserve the query plan in the comment, but it is not used otherwise.

In our experimental studies our focus is on the execution plan[3] and the actual runtime and not on the output of a query. The output of a query was of interest for us, when we did regression testing to ensure our implementation is correct. To accomplish this we prefixed our queries with `EXPLAIN ANALYZE`. `EXPLAIN` alone just shows the execution plan of a statement without actually executing it. Whereas `EXPLAIN ANALYZE` carries out the command and shows the actual runtime. See the PostgreSQL documentation on *EXPLAIN*[4] for more details.

Definition 14 (Setup Query)**.** *A Setup Query is a query who's sole purpose is to setup things up for the query that is going to be profiled.*

We issued these types of query to prim the cache. Setup queries typically look like this:

```
EXPLAIN ANALYZE SELECT * FROM a15d1e5s0;
```

or with an index

```
EXPLAIN ANALYZE SELECT * FROM a15d1e5s0idx ORDER BY d1, d2;
```

and given enough shared buffers the entire relation and the used index will be in the disk cache afterwards. Since we set the shared buffer size way larger than our biggest relation with the associated indexes this was always the case.

Definition 15 (Run)**.** *A Run comprises zero, one or more Setup Queries and the Query that is going to be profiled.*

Each Run is identified by its unique `runid` and besides this general definition in our case a Run had zero or one Setup Query and we often grouped Runs in such a way, that two Runs shared a single Setup Query (cf. Figure 5.4).

[3] We use execution plan and query plan interchangeable.

[4] `http://www.postgresql.org/docs/8.3/static/sql-explain.html`

Definition 16 (Job)**.** *A Job comprises of one or more Runs batched together in a file.*

A Job is the smallest unit subject to workload distribution (see section 5.2.2) and is stored in a `.sql` file and typically looks like displayed in Figure 5.4.

```
--<comment>                                                    the setup query
explain analyze select * from a15d1e5s0;
--</comment>
--<query runid=bnl.ef.entropy.entropy.a15d1e5s0.5>                   first run
explain analyze select * from a15d1e5s0 ↩
 → skyline of d1 min, d2 min, d3 min, d4 min, d5 min ↩
 → with bnl windowpolicy=entropy ef efwindowpolicy=entropy;
--</query>
--<query runid=sfs.ef.entropy.entropy.a15d1e5s0.5>                  second run
explain analyze select * from a15d1e5s0 ↩
 → skyline of d1 min, d2 min, d3 min, d4 min, d5 min ↩
 → with sfs windowpolicy=entropy ef efwindowpolicy=entropy;
--</query>
```

Figure 5.4: A snipped from a Job file, containing a Setup Query and two Runs

5.2.1 Machine and Network Configuration

In our experimental environment we had one machine (SKY00) to control and monitor the experiments and seven machines (SKY01, . . ., SKY07) to perform the experiments. This "seven" is just because we did not have more. To have exactly the same configuration, we cloned SKY02, . . ., SKY07 from SKY01 using a fine piece of free software namely PING (Partimage Is Not Ghost)[5]. We also used Windows XP's `sysprep` during the clone process.

For interprocess communication we used file sharing and we used file locking for synchronization. The machine SKY00 shared a folder, which was mounted by SKY01, . . ., SKY07 with a directory layout as described in Table 5.1.

5.2.2 Life Cycle of a Run

A new Job was born by a script called `gj.pl` (see section 5.2.2) in the `prep/` directory. We distributed the Jobs to several directories below `prep/`, namely to `s0`, `s1`, and `s2` and their sub-directories. Using `xcopy /m` made this task easy, as only files with set archive bit were copied. On SKY00 a script called `feed.pl` was running, which moved the Jobs of these directories to the `jobs/` directory once this had no more Jobs. The Jobs in `jobs/` where picked up by a script called `js.pl` which moved the next Job into a directory with the same name as the machine which is running `js.pl`, SKY01 to SKY07 in our case. The job was then executed and finally moved to the directory `done/`. See subsubsection *Job Scheduling* below for more details. As a result of the Job being executed a raw log file (cf. Figure 5.5) was generated in the (log/) directory. We parsed this raw log files to CSV format with comment lines (cf. Figure 5.6) with a script called `qp2log`, see subsubsection *To CSV Format* below. The last step before aggregating and analyzing the data (see section 5.2.2) was to pivot and project the CSV files to a more compact representation (cf. Figure 5.7) by script called `pivot_log`, see subsubsection *Pivoting the CSF File* below.

Generating the Jobs

The creation of the Jobs is done by an ordinary Perl script called `gj.pl`. The only argument to this script is the seed (`s0`, `s1`, . . .) for which the Jobs should be generated. The only but

[5] available at: `http://ping.windowsdream.com/`

```
--<pid pid=SKY07/>                          the pid identifies the machine the job ran on
--<comment>
explain analyze select * from a15d1e5s0;
                                    QUERY PLAN
---------------------------------------------------------------------------------
 Seq Scan on a15d1e5s0  (cost=0.00..2924.00 rows=100000 width=124) ↩
  → (actual time=0.010..48.112 rows=100000 loops=1)
 Total runtime: 79.093 ms
(2 rows)

--</comment>
--<query runid=bnl.ef.entropy.entropy.a15d1e5s0.5>
explain analyze select * from a15d1e5s0 ↩
 → skyline of d1 min, d2 min, d3 min, d4 min, d5 min
 → with bnl windowpolicy=entropy ef efwindowpolicy=entropy;
                                    QUERY PLAN
---------------------------------------------------------------------------------
 Skyline  (cost=457884.24..457886.07 rows=79 width=124) ↩
  → (actual time=10618.204..11076.301 rows=4550 loops=1)
...                     we skip this query plan here as the one below is more representative
--</query>
--<query runid=sfs.ef.entropy.entropy.a15d1e5s0.5>
explain analyze select * from a15d1e5s0 ↩
 → skyline of d1 min, d2 min, d3 min, d4 min, d5 min ↩
 → with sfs windowpolicy=entropy ef efwindowpolicy=entropy;
                                    QUERY PLAN
---------------------------------------------------------------------------------
 Skyline  (cost=458169.07..458170.90 rows=79 width=124) ↩
  → (actual time=672.662..5747.821 rows=4550 loops=1)
   Skyline Attr: d1, d2, d3, d4, d5
   Skyline Method: sfs 5 dim
   Skyline Stats: passes=2 rows=27223, 1599
   Skyline Window: size=1024k policy=entropy
   Skyline Cmps: tuples=15109830 fields=37876691
   ->  Sort  (cost=457807.96..457809.79 rows=732 width=124) ↩
        → (actual time=672.653..712.584 rows=27223 loops=1)
         Sort Key: d1, d2, d3, d4, d5
         Sort Method:  external merge  Disk: 3816kB
         ->  Elimination Filter  (cost=457523.13..457773.13 rows=732 width=124) ↩
              → (actual time=0.014..597.068 rows=27223 loops=1)
               Elim Filter Attr: d1, d2, d3, d4, d5
               Elim Filter Method: elimfilter 5 dim
               Elim Filter Stats: passes=1 rows=
               Elim Filter Window: size=8k policy=entropy
               Elim Filter Cmps: tuples=1263179 fields=3533862
               ->  Seq Scan on a15d1e5s0  (cost=0.00..2924.00 rows=100000 ↩
                    → width=124) (actual time=0.009..43.352 rows=100000 loops=1)
 Total runtime: 5750.614 ms
(17 rows)

--</query>
```

Figure 5.5: An example for a raw log file

All input is preserved as comment, where comment lines start with "#".

```
#--<pid pid=SKY07/>                         the pid identifies the machine the job ran on
#--<comment>
#explain analyze select * from a15d1e5s0;
# [...]                                              the query plan is as in Figure 5.5
#--</comment>
#--<query runid=bnl.ef.entropy.entropy.a15d1e5s0.5>
```

For the query sections all relevant information from the query plan is parsed into CSV format with the columns `runid`*,* `name` *and* `value`*. See Table 5.2 for a description of the name/-value pairs.*

```
"bnl.ef.entropy.entropy.a15d1e5s0.5","method","bnl.ef.entropy.entropy"
[...]
"bnl.ef.entropy.entropy.a15d1e5s0.5","skyline.rows","4550"
"bnl.ef.entropy.entropy.a15d1e5s0.5","skyline.passes","2"
"bnl.ef.entropy.entropy.a15d1e5s0.5","skyline.passes.rows","27223, 1396"
[...]
```

Please note that the query and query plan are preserved below the data in CSV format.

```
#<queryplan>
#explain analyze select * from a15d1e5s0 skyline of d1 min, d2 min, [...]
#                                    QUERY PLAN
#--------------------------------------------------------------------------------
# Skyline  (cost=457884.24..457886.07 rows=79 width=124)
# [...]                                              the query plan is as in Figure 5.5
#</queryplan>
#--</query>
#--<query runid=sfs.ef.entropy.entropy.a15d1e5s0.5>
"sfs.ef.entropy.entropy.a15d1e5s0.5","method","sfs.ef.entropy.entropy"
"sfs.ef.entropy.entropy.a15d1e5s0.5","skyline.est.cost","458169.07..458170.90"
"sfs.ef.entropy.entropy.a15d1e5s0.5","skyline.est.cost.start","458169.07"
[...]                                                       similar for the second run
#<queryplan>
# [...]                              as above, original query plan is preserved as comment
#</queryplan>
#--</query>
```

Figure 5.6: A log file parsed with `qp2log`. For the name and description of the extracted fields see Table 5.2.

jobs/	
js.pl	job scheduler, see section 5.2.2
feed.pl	batch feeder, moves prepared jobs into the jobs/ directory once this gets empty, see section 5.2.2
js.lock	lock file used by js.pl and feed.pl to synchronize access to the jobs/ directory
*.sql	jobs waiting to be scheduled by js.pl
prep/	
gj.pl	job generator, see section 5.2.2
*.sql	prepared jobs
s0/	To this and to the directories below we copied the prepared jobs. We split the entire set of prepared jobs up into logical and smaller units of lets say a few hundred files. The jobs from this directories where moved to jobs/ once there were no jobs left here. This is done by feed.pl. The same is true for the sibling directories s1/ and s2/.
ws/	This one for example holds the jobs, when we were analyzing the effect of varying window size.
.../	
s1/	
s2/	
SKY[xx]/	While processing a job it resides inside a directory with the same name as the machine which is processing the job.
log/	
*.log	the raw log files
done/	Processed jobs end up here.
*.sql	processed jobs

Table 5.1: Directory layout for the experiments

noteworthy trick used in this script is to skip the generation of a Job if the file already exists, which gives an enormous speed up and the archive bit of existing files was not touched which made it easier to operate with `xcopy /m` on that files.

Job Scheduling

The concept behind this is simple but very effective. We run a process called *job scheduler*, realized in the Perl script `js.pl`, on each machine which is available to run some Jobs. The script `js.pl` waits to get an exclusive file lock on `js.lock`, checks for `*.sql` files, i.e. Jobs in the `jobs/` directory, moves the first of these files, given one was found, to a directory with the same name as the machine `js.pl` is running one, next releases the exclusive lock on `js.lock` and finally starts to work in the dequeued job by piping it through PostgreSQL's interactive SQL processor `psql`. The output is redirected to a log file in the `log/` directory. These steps are performed in an endless loop.

A simple lesson learned here is not to use Perl's `glob` function to read the contents of a directory with thousands of `*.sql` files, when you actually just want to pick one of them. Globbing a directory with thousands of files could take longer than running one of these jobs and instead of running them in parallel, they are processed serially. Instead rely on `opendir`, `readdir` and `closedir`.

The Perl script `feed.pl` is another simple script engaged with job scheduling. Its task is to move batches of Jobs into the `jobs` directory once this got empty. The idea is to have more control in which sequence the Jobs are processed, as we were a little bit more interested in some results than in some others. The access to the `jobs` directory was once again synchronized via file locking `js.lock`. A typical invocation of `feed.pl` looked like this:

```
$ feed.pl prep/s0/ws prep/s1/ws prep/s2/ws
```

Once the `jobs` directory did not contain any Jobs (`*.sql`-files), the Jobs from `prep/s0/ws`, `prep/s1/ws` and `prep/s2/ws` were moved in turn to the `jobs` directory.

There is one more little thing to tell, `js.pl` and `feed.pl` pause their operation if a file named `js.pause` exists in the `jobs/` directory. This is very handy to reconfigure the system, like temporarily remove some Jobs from the queue to schedule others first.

By means of this simple arrangement of files, directory and scripts we were able to add and remove machines as we liked.

To CSV Format

We wrote a little Perl script `qp2log` which transforms the raw log files (cf. Figure 5.5 into CSV format with the columns `runid`, `name` and `value`, and the content of the raw log file is preserved as comment prefixed with "`#`" (cf. Figure 5.6). This makes it easy to use `grep -v "^#"` as filter in the pipeline, while on the other hand being able to verify `qp2log` is working correctly, which was helpful during the development of `qp2log`.

The script `qp2log` makes extensive use of Perl's regular expressions to do its task. All the fields that are extracted from the queries' execution plans are listed in Table 5.2. We would like to point out a feature of Perl's regular expression (regex) support which was very handy for us, it is the combination of "`m//g`" and "`\G`". "`\G`" matches only at the end-of-match position of the prior `m//g`. By means of that one can build a very `lex`/`flex`-like scanner.

Pivoting the CSV Files

Our tool `pivot_log` transforms a parse log file (cf. Figure 5.6) into CSV format, where each Run is represented by a single line and values for the selected names are the columns (cf. Figure 5.7). For a list of fields with description see Table 5.2.

Aggregating and Analyzing in R

For further aggregation and analysis we imported the CSV files generated in the last step into the statistics software R (`http://www.r-project.org/`). The results of this effort can be seen in section 5.4

5.2.3 Total Computational Workload

In total we scheduled 301,196 runs batched up in 18,072 jobs consisting of 454,782 queries in total with a total CPU time of 17,154,401,205.424ms ≈ 200days. These figures do not include the work we had to repeat due to a bug we discovered in our code and the Runs that have been skewed by automatic software update, screen savers and so forth (cf. section 5.1.1). The workload was spread across seven computers (SKY01, . . ., SKY07), and these computers have been busy for about six weeks.

5.3 Parameter Space

The parameter space that can be analyzed is quite large, the parameters and their ranges are as follows:

- *size of dataset* (e.g. 100, 1000, 10000, 100000, 1000000)
- *number of dimensions* (e.g. 1-15)

Field name	*Description*
`[ANYNODE].est.cost`	For any node type in the execution plan we output these quantities. The node names are mapped to `skyline`, `elimfilter`, `seqscan` and `idxscan`. Although other node types do exist in general, they have not been of interest in our studies. The fields with `.est` correspond to the estimated values, while the other are actual values. The fields with `.cost` are costs split up into startup costs `.start`, the time needed to return the first tuple, and total costs `.total`, the time needed to return all the tuples. Please note that the estimated costs are in an arbitrary unit and that the actual costs are in milliseconds, see the PostgreSQL documentation on *Using EXPLAIN*[a]. The estimated average width of a row `.width` is measured in bytes. The field `.rows` corresponds to the number of rows output by this node. It can happen that in certain query plans a node is executed more than once, e.g. as part of a subplan, then this number is reported by `.loops`.
`[ANYNODE].est.cost.start`	
`[ANYNODE].est.cost.total`	
`[ANYNODE].est.rows`	
`[ANYNODE].width`	
`[ANYNODE].cost`	
`[ANYNODE].cost.start`	
`[ANYNODE].cost.total`	
`[ANYNODE].rows`	
`[ANYNODE].loops`	
	For a skyline node either “Skyline” or “Elimination Filter” the following fields are reported:
`[SKYLINENODE].method`	Currently one of `1dim`, `presort`, `mnl`, `bnl`, `sfs` and `elimfilter`. Note there are actually different methods for 1 dimensional distinct and non-distinct, but we distinguish them only by the next field here.
`[SKYLINENODE].distinct`	`SKYLINE OF` **`DISTINCT`**? 0/1
`[SKYLINENODE].dim`	# of dimensions: $1, 2, \ldots$
`[SKYLINENODE].attr`	The attributes the skyline is computed on.
`[SKYLINENODE].passes`	# of passes for BNL and SFS
`[SKYLINENODE].passes.rows`	# of rows in the input for each pass for BNL and SFS
`[SKYLINENODE].windowsize`	tuple window size in KB
`[SKYLINENODE].windowslots`	tuple window size in # of slots
`[SKYLINENODE].windowpolicy`	tuple window policy, one of: `append`, `prepend`, `entropy` and `random`
`[SKYLINENODE].cmps.tuples`	# of comparisons between two tuples
`[SKYLINENODE].cmps.fields`	# of comparisons between two fields
`[SKYLINENODE].cmps.ratio`	= `.cmps.fields / .cmps.tuples`
`seqscan{.ALIAS}.on`	Name of the base table, when an alias is used for the table it is given as `.ALIAS`, e.g. when we express a skyline query only with SQL, we have an inner `.i` and an outer `.o` relation.
`seqscan{.ALIAS}.tablesize`	Here we do a little hack and derive this value from the table name (# rows).
`sort.method`	typically `quicksort` or `external merge`
`sort.key`	sort is performed on this expressions
`idxscan.idx`	name of the index used
`idxscan.table`	name of the table the index is on
`total`	The total runtime of the query in milliseconds (ms).

[a] `http://www.postgresql.org/docs/8.3/static/using-explain.html`

Table 5.2: Fields/Values from the execution plan

```
$ pivot_log --help
Usage: pivot_log [OPTIONS] [FILE] ...
Pivot each parsed log file FILE or standard input.

Options:
        --c=NAME[=ALIAS]
        --column=NAME[=ALIAS]
                        include the VALUE of field NAME, if ALIAS is given
                        use this as column name instead of NAME.

        --[no]header    output the CSV header. default is output it.

        --help          display this help message and exit

The columns are in the same order as specified by --c. Nevertheless the first
column is always RUNID.

$ unzip -p sql.zip | qp2log | ↩
→ pivot_log --c=method --c=seqscan.o.on=table [...] --c=total
"runid","method","table","outrows","dim","inrows","skyline.method", [...]
"sql.a15d1e1s0.10","sql","a15d1e1s0","10","10","10","sql","","","", [...]
"sql.a15d5e1s0.10","sql","a15d5e1s0","50","10","50","sql","","","", [...]
"sql.a15d1e2s0.10","sql","a15d1e2s0","100","10","100","sql","","","", [...]
[...]
```

Figure 5.7: Typical tool chain for `pivot_log`. Note that all the log files for "SQL-only" Runs are kept inside `sql.zip`.

- *data distribution* (indep, corr, anti)
- *data source* (disk, set returning function)

  in case the data are stored on disk:

  – *sequential scan vs. index scan* SFS can benefit from index scan (omit sort phase), BNL and SFS could benefit from a index scan if DIFF groups are used[6].

  – *sequence of tuples in tuple stream* (random, high entropy first/last) for BNL and EF if a very good tuple is very early in the tuple stream it will eliminate a lot of tuples in a very early stage

- *methods = algorithms*

  – as SQL statement

  – special case 1 dim distinct

  – special case 1 dim

  – special case 2 dim (needs sort presort or access path)

  – MNL

  – BNL

  – SFS

  – EF+SFS ≈ LESS

  – EF+BNL

[6] not yet implemented, see section 6.3.4

- window size in KB or # slots (e.g. 1 KB, 2 KB, 4 KB, 8 KB, ...)
- window policy (prepend, append, entropy[7], random) In case EF is used the window policy for EF and BNL/SFS can be selected independently, resulting in 16 possible combinations.

Exploring the entire parameter space is practically infeasible. The subset we selected and the according results are presented in the following section.

5.4 Comparisons and Analysis

We measure the time performance with respect to three parameters: dimension, data cardinality, and data distribution. We tested on the following three data distributions: independent (i), correlated (c), and anti-correlated (a). For the other two factors "dimension" and "cardinality", one has been fixed during each experiment. Thus all the plots are two dimensional.

5.4.1 Time performance w.r.t. dimension

For this set of experiments we fix the number of tuples of the input data. Note that generally the absolute time measurements are not proportional throughout the dimensions, we thus deploy the relative values by setting one measurement as the reference. In Figure 5.9 we illustrate the absolute time measurements which are corresponding to the relative values in Figure 5.8. In the rest of the experiments, we only provide relative time measurements.

Special case: 1 dimensional skylines

We believe the one dimensional case either distinct or non-distinct is of so limited use, that we decided not to perform any performance studies for this case. Furthermore the one dimensional case is not even mentioned most of the time in literature.

Special case: 2 dimensional skylines

For the two dimensional case the specialized algorithms PRESORT is available. Nevertheless we did exclude it from the performance study, because given that the tuple window size for SFS is large enough the behavior of PRESORT is very much like SFS with tuple window placement policy *prepend*.

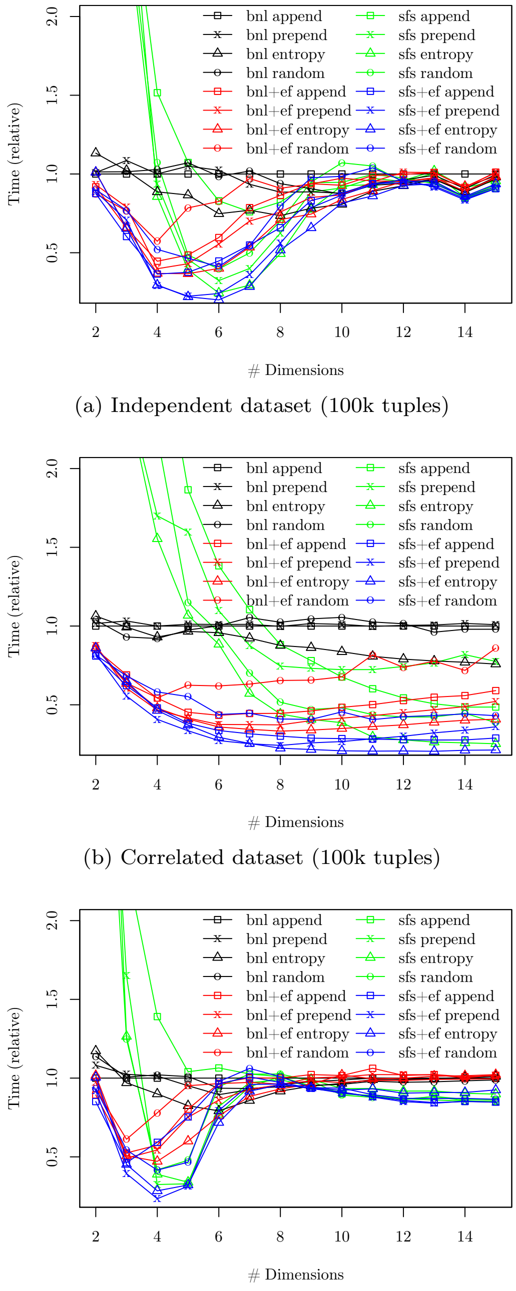
Big datasets

Figure 5.8 shows the time performance of the skyline algorithms vs. dimension with 100k tuples. It shows clearly that the SFS+EF algorithm has the best time performance with all the data distributions. This conforms with the results in [Godfrey et al., 2007], where it was shown that LESS outperforms the SFS algorithm with 1M tuples.

We also note that SFS performs extremely poor as the dimension is less than six for independent and anti-correlated data, and even worse for correlated data. This can be justified that if the dimension is low, the skyline selectivity factor[8] is low as well. Therefore, EF operation is effective, i.e., a number of tuples could be removed at this stage. However, this does not explain why SFS performs worse than BNL on datasets of low dimensions. This can be clarified when considering the absolute time measurement in Figure 5.9. It is shown

[7] policy entropy needs table stats for scaling to $[0, 1]$

[8] selectivity factor = skyline tuples / input tuples. A low selectivity factor means high selectivity.

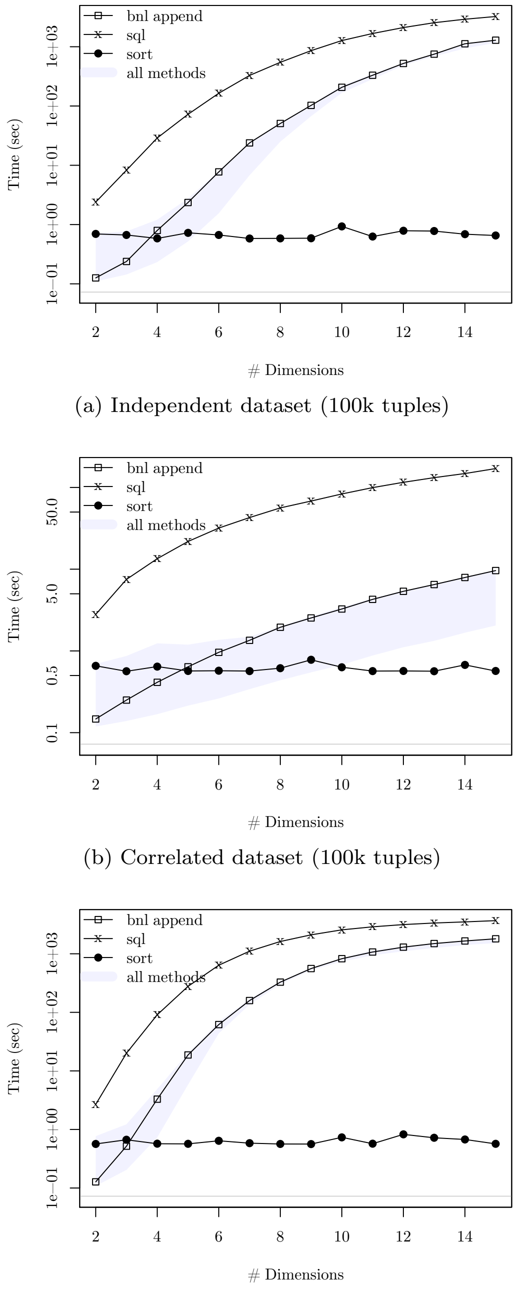


(a) Independent dataset (100k tuples)

(b) Correlated dataset (100k tuples)

(c) Anti-correlated dataset (100k tuples)

Figure 5.8: Comparing runtime for different methods (100k tuples)



(a) Independent dataset (100k tuples)

(b) Correlated dataset (100k tuples)

(c) Anti-correlated dataset (100k tuples)

Figure 5.9: Absolute timing for different physical skyline operators (100k tuples)

that the sorting routine of SFS has a nearly constant cost for all dimensions. Therefore, for low dimensional datasets, the sorting cost is dominant.

BNL+EF performs consistently better than BNL, due to the low skyline selectivity factor for datasets with 100k tuples (cf. Figure 5.16). As far as the window policy is concerned, *entropy* has consistently better performance for all the algorithms.

Small datasets

Figure 5.11 depicts the time-dimensionality performance regarding to datasets with 1k number of tuples. One interesting observation is that the performance of SFS is surprisingly better than others, except for the extremely low dimension values (three for independent, anti-correlated, and six for correlated distributions). As illustrated by Figure 5.16, there exists a causal relation with the skyline selectivity factors. It shows clearly that as the data cardinality decreases, the skyline selectivity factor increases. Therefore, in general, the EF operator is less effective with lower data cardinalities. To be more accurate, if the skyline selectivity factor is higher than 0.1, the benefit gained from EF is marginal. This is confirmed by the results shown in Figure 5.8 as well.

The result is somehow contradictory to the claim that SFS+EF should perform consistently better than SFS, because at the EF stage there should always be some tuples eliminated. However, one should not ignore the cost of the EF operation, where comparisons are executed. Moreover, if the *entropy* window policy is applied, the time consumption is even higher. Hence, there exist data settings where EF does not pay off anymore.

5.4.2 Time performance w.r.t. cardinality

To better understand the time performance of the skyline algorithm with respect to the data cardinality, we conducted experiments for each dimension $k \in \{2, \ldots, 15\}$ with the number of tuples ranging from 100 to 100k. For each data distribution we have chosen four to five representative dimension values. Figure 5.12, 5.13, and 5.14 illustrate the test results for independent, correlated, and anti-correlated datasets, respectively.

Surprisingly, the *rule of the selectivity factor* is proven to be true in all the datasets. That is, if the selectivity factor is higher than 0.1, the elimination filter does not have any advantage.

Independent and anti-correlated datasets

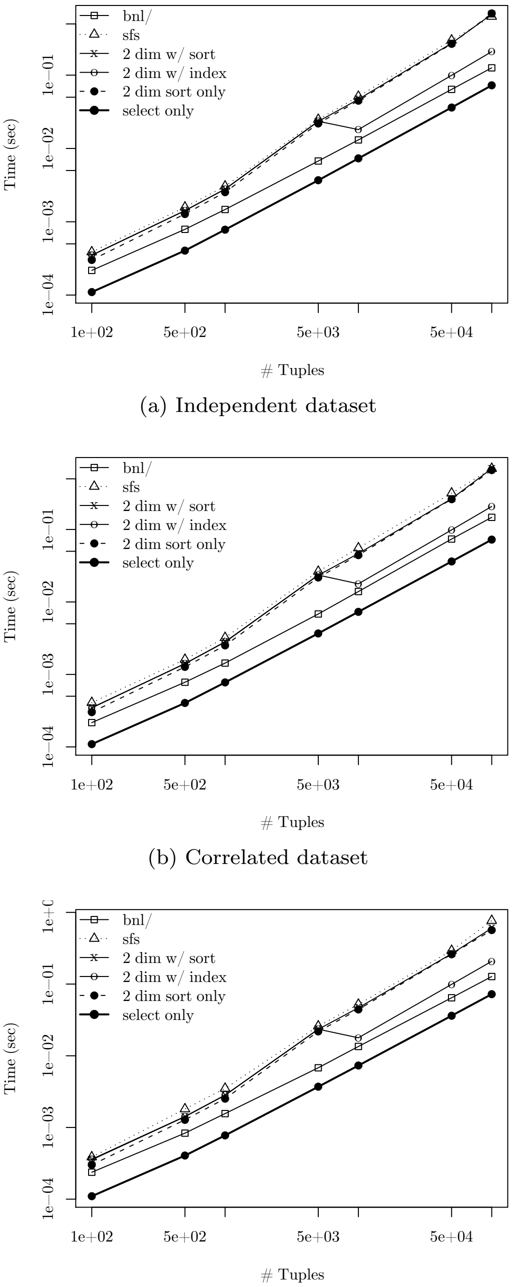
Let us first consider the independent datasets (Figure 5.12). As the dimension is higher than four, SFS is superior for all datasets, except for data cardinality of 50k and 100k, where SFS+EF outperforms SFS. If we examine again the selectivity factor for independent datasets in Figure 5.16, these datasets are in the category where the selectivity factor is higher than 0.1. This result confirms our observation in section 5.4.1. Note that the same observation holds for BNL+EF vs. BNL as well.

With dimension three there is an exception, where BNL performs best. This can be justified as follows: the sorting cost remains constant w.r.t. dimensions, thus it becomes dominant with low dimension (such as three).

The performance of anti-correlated datasets in Figure 5.14 behaves similarly to that of independent datasets, thus the above analysis can be applied as well.

Correlated datasets

For correlated datasets in Figure 5.13, SFS+EF is superior with a few exceptions. This, again, can be perfectly explained by the selectivity factors for correlated datasets (cf. 5.16b). Except for the data cardinalities of 100 and 500, all the selectivity factors are remarkably low (most of them are less than 0.1), thus the EF operation pays off.

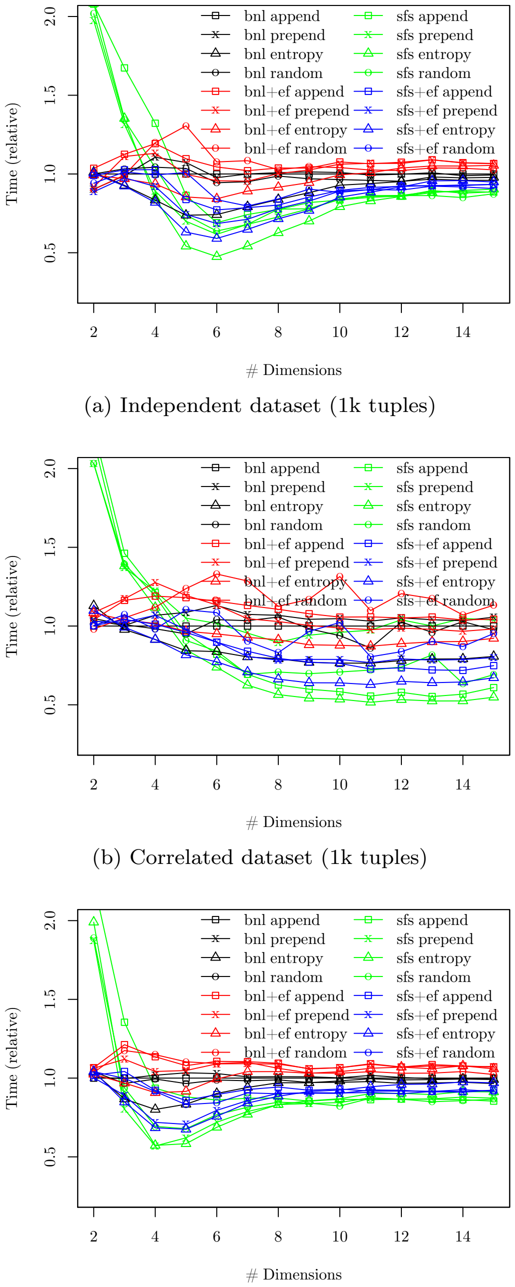


(a) Independent dataset

(b) Correlated dataset

(c) Anti-correlated dataset

Figure 5.10: Absolute timing for different physical skyline operators in the 2 dimensional case

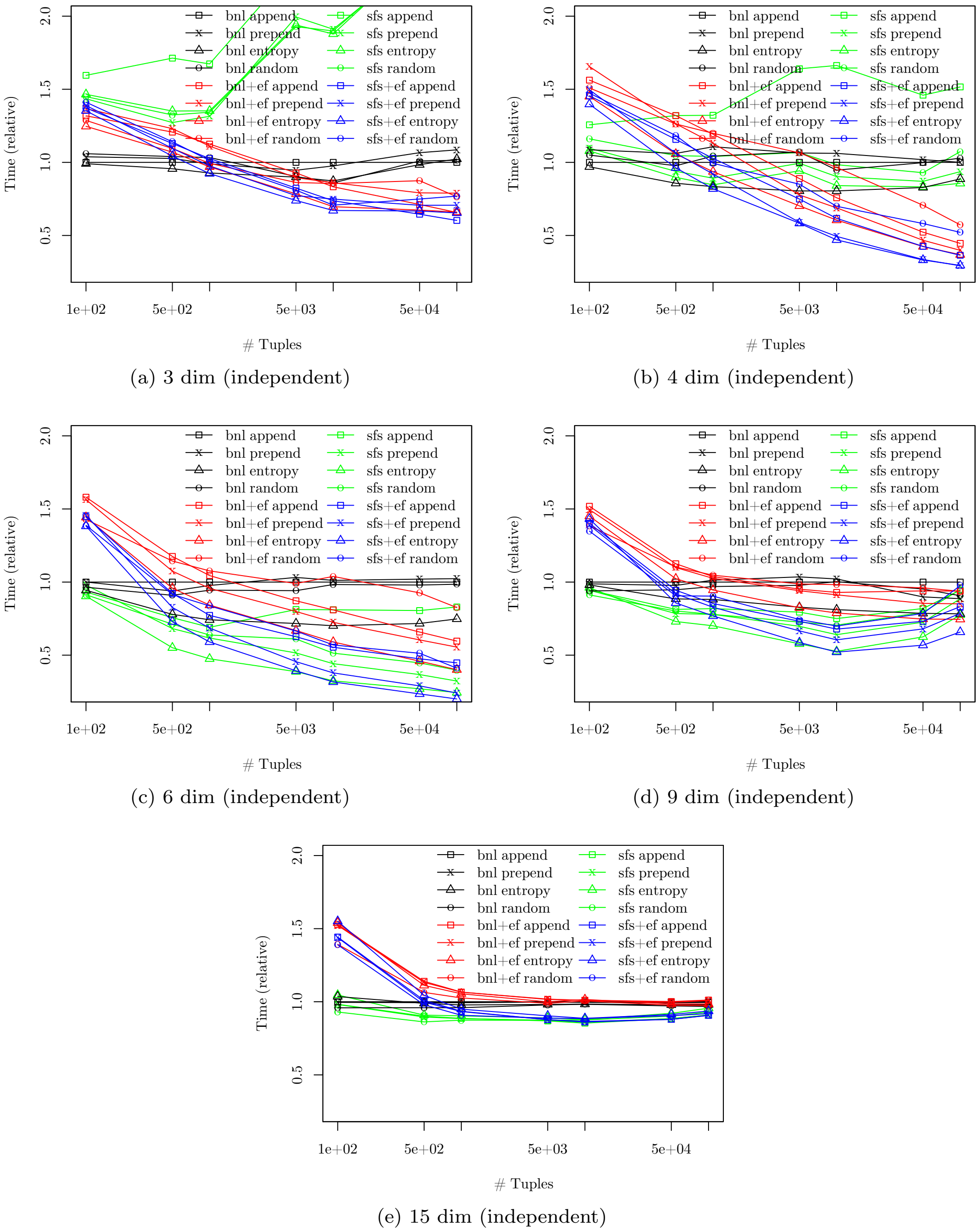


(a) Independent dataset (1k tuples)

(b) Correlated dataset (1k tuples)

(c) Anti-correlated dataset (1k tuples)

Figure 5.11: Comparing runtime for different methods (1k tuples)



(a) 3 dim (independent)

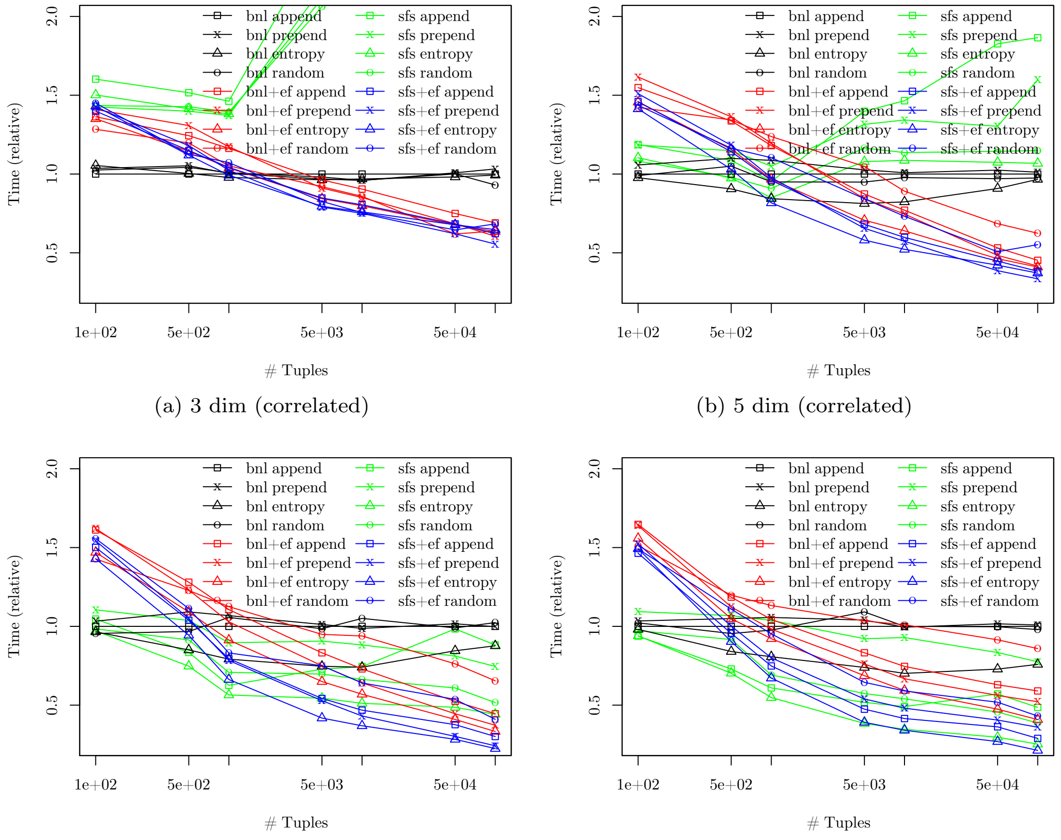
(b) 4 dim (independent)

(c) 6 dim (independent)

(d) 9 dim (independent)

(e) 15 dim (independent)

Figure 5.12: Comparing runtime for independent datasets



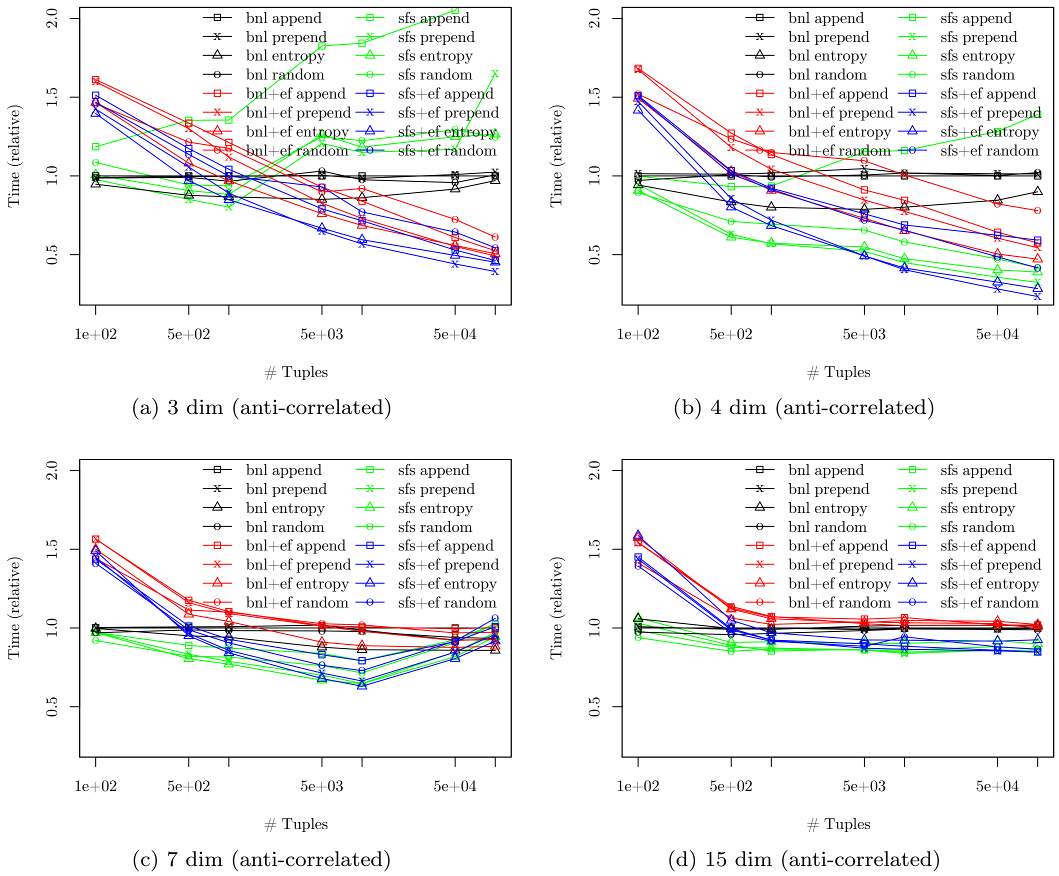
(a) 3 dim (correlated)

(b) 5 dim (correlated)

(c) 8 dim (correlated)

(d) 15 dim (correlated)

Figure 5.13: Comparing runtime for correlated datasets



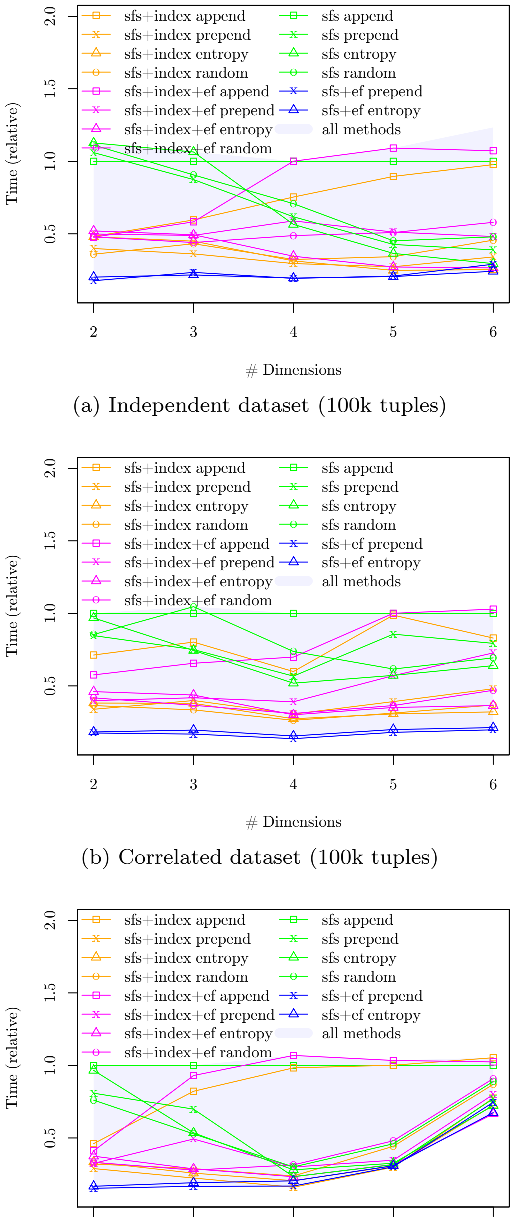
(a) 3 dim (anti-correlated)

(b) 4 dim (anti-correlated)

(c) 7 dim (anti-correlated)

(d) 15 dim (anti-correlated)

Figure 5.14: Comparing runtime for anti-correlated datasets

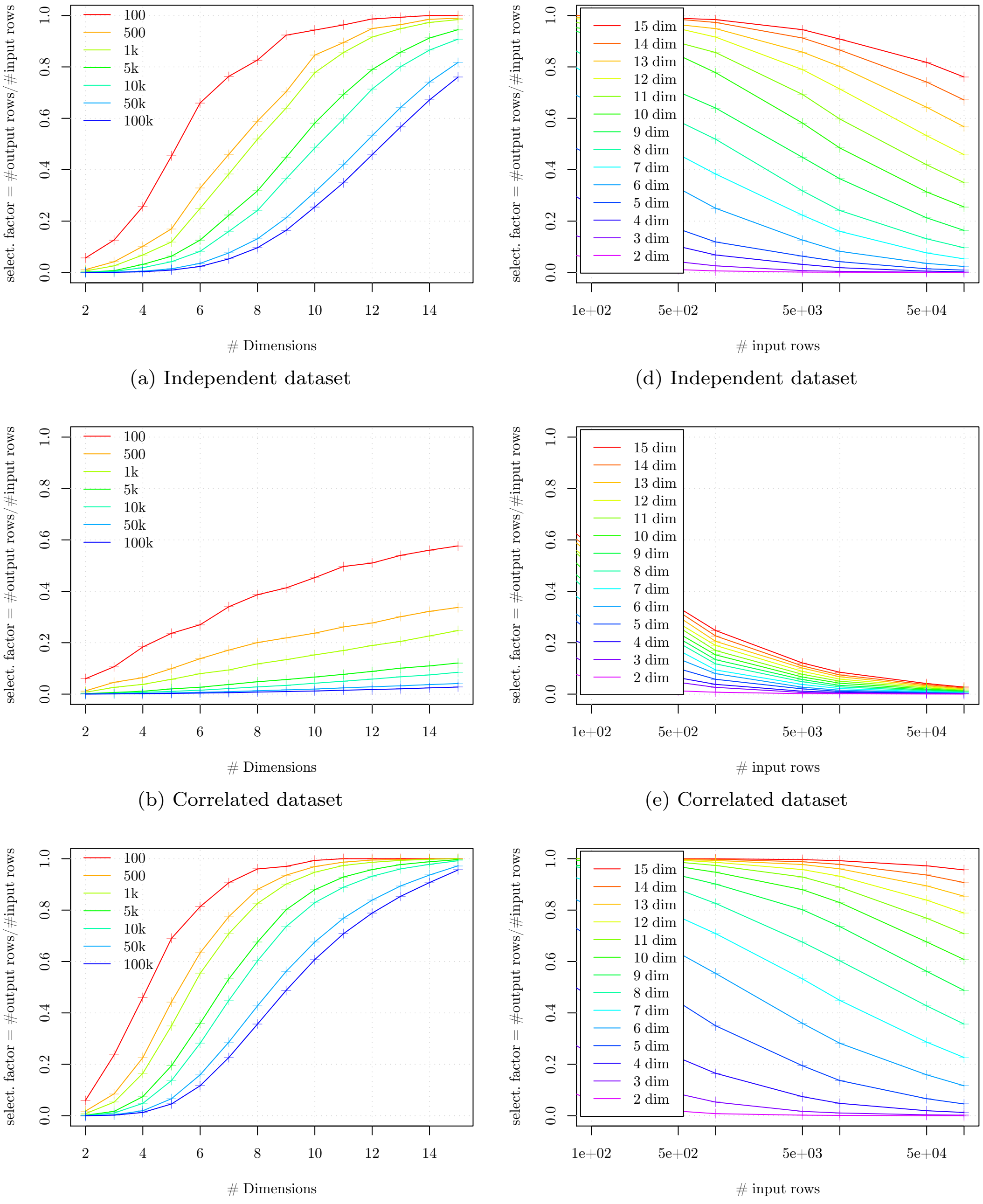


(a) Independent dataset (100k tuples)

(b) Correlated dataset (100k tuples)

(c) Anti-correlated dataset (100k tuples)

Figure 5.15: Relative timing for SFS/SFS+EF when using index access paths (100k tuples)



(a) Independent dataset

(d) Independent dataset

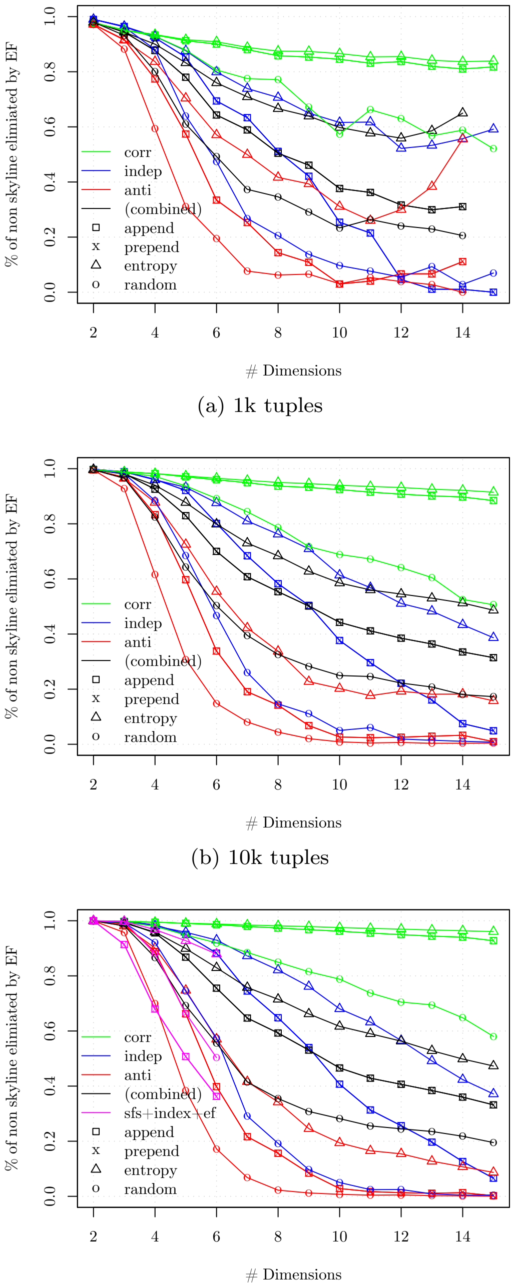
(b) Correlated dataset

(e) Correlated dataset

(c) Anti-correlated dataset

(f) Anti-correlated dataset

Figure 5.16: Skyline operator selectivity factor on our test datasets



(a) 1k tuples

(b) 10k tuples

(c) 100k tuples

Figure 5.17: Effectiveness of elimination filter (EF) for varying dimensions

5.4.3 SFS using indexes

The idea is to leverage the index structure for SFS in order to avoid an explicit sort. For SFS+EF w/ index this is illustrated in 4.4d. It is up to the query optimizer to decide by means of cost estimation whether to use an index or not. We analyzed the usage of indexes for SFS in detail for 100k tuples (see Figure 5.15).

It turns out that SFS w/ index consistently outperforms SFS, independent from the window policy used, with a single exception (cf. 5.15c when dim=6). This is clearly an argument for using an index.

However, SFS+EF w/ index performs worse than SFS w/ index. This is somehow counter intuitive, because for 100k tuples SFS+EF performs better than SFS, as we have discussed in section 5.4.1 (cf. Figure 5.8). We argue that the effectiveness of EF deteriorates in the presence of the index structure. The data is physically ordered according to one of the indexed attributes, which is not guaranteed to be beneficial for the elimination filtering. Therefore, one should be careful with applying EF in the presence of index, even on the datasets with low selectivity factors. The only significant exception is for correlated data, window policy append, and dim ≤ 3 (cf. 5.15b). We argue this is due to the general low selectivity factor and high EF effectiveness in this case (cf. 5.16b and 5.17c).

In all aspects, SFS w/ index and SFS+EF w/ index are outperformed by SFS+EF using window policy entropy or prepend. This can be explained with the same argument as above, EF eliminates so many tuples prior to sorting that this becomes cheaper than using an index.

5.4.4 Window policy

All experimental results show that the *entropy* is the most efficient and effective window policy. One might notice that on correlated datasets, the *prepend* window policy performs the worst in the SFS algorithm. The explanation is obvious: if the correlated data is sorted, the most effective skyline tuples (that is, the tuples dominate most of the other tuples) tend to enter the widow earlier. However with the prepend policy, these tuples are successively pushed to the end of the window, which results in high time costs for checking whether a new tuple is dominated by the tuples in the window. Figure 5.17 shows that *entropy* is the most effective EF window policy.

Chapter 6

Future work

In this chapter we have collected various directions of future work. This chapter is organized as the stages a query takes during execution.

6.1 Syntax

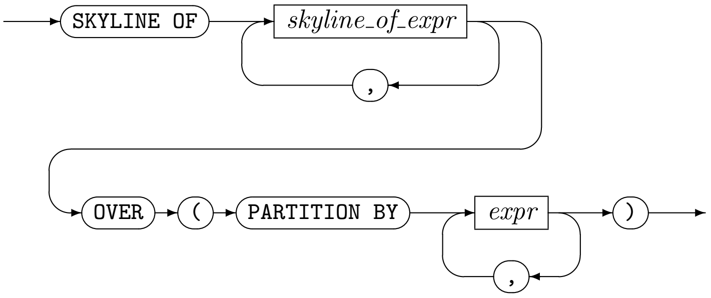
6.1.1 More General Syntax

The syntax for the skyline operator as proposed in [Börzsönyi et al., 2001] is not part of any standard, hence our extended syntax is not part of a standard either. Other extensions to SQL have been proposed, like Preference SQL [Kießling and Köstler, 2002]. A subset of Preference SQL was used in [Chaudhuri et al., 2006]. Our current syntax is quite limited to a special purpose, a more general syntax for preference queries is desirable.

6.1.2 Different Syntax for `SKYLINE OF DIFF`

We believe the syntax of `SKYLINE OF DIFF` is a bit cumbersome for the intended semantics to partition the input relation, so that the skyline is computed within these partitions. Instead of `DIFF` a construct like this could be used:

skyline_clause



With the intended semantics to compute the skyline within the partitions as induced by the `PARTITION BY` subclause. This syntax was inspired by Oracle's analytic functions [Oracle, 2005, Page 5-10].

6.1.3 **`USING`** ***op*** for `SKYLINE OF DIFF`

While we do provide syntax to use user-defined ordering operators instead for `SKYLINE OF MIN/MAX`, we do not do so yet for `SKYLINE OF DIFF`, which is used to partition the input into groups. For the moment only equality for the given data type is used, but in general any equality relation could be used. E.g. a syntax like ... `SKYLINE OF d3 DIFF USING myOp` could be used but this is subject to future work.

6.2 Query Planner/Optimizer

6.2.1 Cost-based operator selection

To let the skyline operator be a full-blown relational operator one has to solve the associated cost- and cardinality estimation problems. The results from Chaudhuri et al. [2006] could be incorporated in our work. Nevertheless the query optimizer of PostgreSQL is not fully generic nor only cost-based, so it would take quite some effort to build an optimizer that is only controlled by rules when a specific transformation is applicable and the associated costs. See also section 4.7.3 and 4.7.5.

6.2.2 Algebraic Optimizations

Chomicki [2003] and Kießling and Hafenrichter [2003] show in their papers algebraic optimizations of the preference (skyline) operator, e.g. when and how to push a skyline operator through a join, how to simplify multiple preference operators, and so forth, together with sufficient conditions when the rules are applicable. It is future work to integrate such optimizations into the query planner/optimizer.

6.2.3 Sampling

In the absence of statistics (histograms) for the relation in questions, sampling could be engaged. Chaudhuri et al. [2006] did derive some results on this topic.

6.3 Physical Operators

6.3.1 Index-based Algorithms

In the future it would interesting to implement and evaluate index-based algorithms such as *Index* [Tan et al., 2001], *Nearest Neighbor* (NN) [Kossmann et al., 2002], and *Branch and Bound* (BBS) [Papadias et al., 2003, 2005].

6.3.2 Index Scan for one Dimensional Skylines

Since one dimensional skylines are a little bit of limited use anyway, we did not yet make an attempt to make use of a suitable index for skyline computation in the one dimensional case. The computation of distinct and non-distinct one dimensional skylines could be improved by performing an index scan. In the distinct case we start at the minimum or maximum respectively and read a single tuple. For the non-distinct case we start at the right end of the index and read as long as the value does not change.

6.3.3 SFS

Index Scan and SFS

As for SFS algorithms using indexes, what remains open is to investigate the behavior in case all skyline attributes and attributes in the select clause are part of the index, such that it suffices to perform an index-scan. We believe this could yield good results.

Project Tuples before inserting into SFS Tuple Window

SFS emits a tuple directly after it has survived the test against all tuples in the tuple window and the dominance check is only performed on the skyline attributes. Because of this property it is possible to project the tuples on the skyline attributes before inserting into the tuple window to save space in the tuple window. But there is no free lunch, this projection comes at some CPU cost as memory has to be copied and manipulated. Some cost estimation or some heuristics could be developed to decide when to perform this projection.

6.3.4 Speedups for `SKYLINE OF DIFF`

One can use a separate tuple window for each `SKYLINE OF DIFF` group, so only the tuples with the same values on the `SKYLINE OF DIFF` attributes have to be compared to each other. This reduction of comparison would yield an overall speedup. In order to avoid an unbounded number of tuple windows and instead of having a tuple window for each `SKYLINE OF DIFF` group hashing could be engaged, i.e. allocate a fixed number of tuple windows and hash each `SKYLINE OF DIFF` group into one tuple window.

For SFS differentiate between `CMP_INCOMPARABLE_SAME_GROUP` and `CMP_INCOMPARABLE_DIFFERENT_GROUP`, because once we have a group change we can flush the tuple window.

6.3.5 Speedups for low Cardinality Domains

A couple of algorithms for low cardinality domains, e.g. the “stars ranking of a hotel”, only five discrete values (one to five stars), have been proposed [Preisinger et al., 2006; Preisinger and Kießling, 2007; Morse et al., 2007]. Based on histograms for the base relations the query optimizer could select such specialized algorithms.

6.3.6 Speedup Tuple Comparison

We believe the concept of elimination filter (EF) is very promising and we believe that it is possible to further improve its efficiency and effectiveness, e.g. to integrate it as a filter in the index scan code, or to use other data structures and policies for the EF window.

Order in which Columns are compared

Investigate the question, whether the number of comparisons and the related costs can be reduced by choosing a specific order for the columns. Can histograms for the base relations be used for this? On the other hand compare integer data, floating point and in turn floating point before string data.

Chapter 7

Conclusion

In this thesis we present the extension of PostgreSQL with the skyline operator and extensive experiments, with the ultimate goal of building a skyline query optimizer. It is our strong belief that with this effort the skyline operator is in the habitat it belongs to.

First, we give a general introduction and motivate why the name "skyline operator" is a good choice. Secondly, we give a formal definition of the skyline operator and we treat in detail how skyline queries can be rewritten into standard SQL. Thirdly, we deal with existing non-index-based skyline algorithms. We describe them using pseudo-code and show some of their properties, which are called "physical properties" in the database context. We prove that the original version of BNL does not terminate in all cases and give a corrected version. Furthermore we talk about related works.

In chapter 4 we describe the details of our implementation. The way the pseudo-code is given fits into the pipelined architecture of the execution engine. The algorithms we have implemented so far are two specialized algorithms for one dimensional skyline queries, a special case for two dimensions (PRESORT), a naïve nested loop algorithm (MNL), BNL, SFS, and a variant of LESS. We extend the basic syntax of the `SKYLINE OF`-clause [Börzsönyi et al., 2001] to allow influence on the semantics (treatment of `NULL` values, etc.) and operational aspects (physical operator, tuple window size/policy, and so forth).

A noteworthy result of our efforts is the site `http://skyline.dbai.tuwien.ac.at/`, where it is possible to test-drive our implementation and get an intuition for the behavior of the skyline operator. Furthermore we provide a series of patches, applicable against the PostgreSQL 8.3 stable release. By means of that we hope to acquaint the skyline operator to a larger audience and lay the ground for further research in the direction of integrating preference query concepts into RDBMSs. We wish that at least one flavor of SQL preference clauses will make it into a future SQL standard.

We have conducted extensive experiments with a total workload of more than 200 days of CPU time on various datasets. The experimental environment we have designed and set up for this task is noteworthy on its own. It is well known that the performance of skyline queries is sensitive to a number of parameters. From our experimental results we were able to expose several findings which are difficult to be verified theoretically. They are: (1) the elimination filter is effective only if the selectivity factor of the skyline query is not more than 0.1; (2) for datasets up to 500 tuples and of relatively small dimensions (e.g. up to five) BNL performs the best in all aspects. Our findings are beneficial for developing heuristics for the skyline query optimization, and in the meantime provide some insight for a deeper understanding of the skyline query characteristics.

We have identified several directions of future work, see chapter 6.

Bibliography

Jon Louis Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors and Applications. *J. ACM*, 25(4):536–543, 1978.

Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *ICDE' 01: Proceedings of the 17th International Conference on Data Engineering*, pages 421–432, Heidelberg, Germany, April 2001.

Carmen Brando, Marlene Goncalves, and Vanessa González. Evaluating Top-k Skyline Queries over Relational Databases. In *DEXA 2007: 18th International Conference on Database and Expert Systems Applications*, volume 4653, pages 254–263, Regensburg, Germany, 2007. Springer.

Reinhard Braumandl, Jens Claussen, and Alfons Kemper. Evaluating Functional Joins Along Nested Reference Sets in Object-Relational and Object-Oriented Databases. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 110–122, San Francisco, CA, USA, 1998.

C. Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33 (2):63–65, 1989.

Chee-Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 203–214, Baltimore, Maryland, USA, 2005.

Surajit Chaudhuri, Nilesh Dalvi, and Raghav Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, Washington, DC, USA, 2006.

Jan Chomicki. Preference Formulas in Relational Queries. *ACM Trans. Database Syst.*, 28 (4):427–466, 2003.

Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting. Technical Report CS-2002-04, York University, Ontario, Canada, 2002.

Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting. In *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, pages 717–816, 2003.

Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting: Theory and Optimizations. In *Intelligent Information Systems*, pages 595–604, 2005.

Neil Conway and Gavin Sherry. Introduction to hacking PostgreSQL. `http://neilconway.org/talks/hacking/hack_slides.pdf`, presented at Anniversary Summit 2006, 2006.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

Hannes Eder. Random Dataset Generator for Skyline Operator Evaluation. `http://randdataset.projects.postgresql.org/`, October 2007.

Parke Godfrey. Cardinality Estimation of Skyline Queries: Harmonics in Data. Technical Report CS-2002-03, York University, Ontario, Canada, 2002.

Parke Godfrey. Skyline Cardinality for Relational Processing. *Foundations of Information and Knowledge Systems*, pages 78–97, 2004.

Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB '05: Proceedings of the 31th International Conference on Very Large Data Bases*, pages 229–240, Trondheim, Norway, 2005.

Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

Marlene Goncalves and Maria-Esther Vidal. Top-k Skyline: A Unified Approach. In *On the Move to Meaningful Internet Systems 2005: OTM Workshops*, volume 3762, pages 790–799. Springer, 2005a.

Marlene Goncalves and Maria-Esther Vidal. Preferred Skyline: A Hybrid Approach Between SQLf and Skyline. In *DEXA 2005: 16th International Conference on Database and Expert Systems Applications*, volume 3588, pages 375–384, Copenhagen, Denmark, 2005b. Springer.

Torsten Grust, Joachim Kröger, Dieter Gluche, Andreas Heuer, and Marc H. Scholl. Query Evaluation in CROQUE - Calculus and Algebra Coincide. In *BNCOD 15: Proceedings of the 15th British National Conference on Databases*, pages 84–100, London, UK, 1997. Springer.

Joseph M. Hellerstein and Michael Stonebraker. Anatomy of a Database System. In *Readings in Database Systems.* The MIT Press, 2005.

ISO/ANSI. *Database Language SQL ISO/IEC 9075:1992.* ISO/ANSI, 1991.

Werner Kießling and Bernd Hafenrichter. Algebraic Optimization of Relational Preference Queries. Technical Report 2003-1, Universität Augsburg, Augsburg, Germany, 2003.

Werner Kießling and Gerhard Köstler. Preference SQL - Design, Implementation, Experiences. In *VLDB '02: Proceedings of the 28th VLDB Conference on Very Large Data Bases*, pages 990–1001, Hong Kong, China, 2002.

Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB '02: Proceedings of the 28th VLDB Conference on Very Large Data Bases*, pages 275–286, Hong Kong, China, 2002.

H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *J. ACM*, 22(4):469–476, 1975.

Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 502–513, Washington, DC, USA, 2005.

Jim Melton, editor. *Database Language SQL ISO/IEC 9075:2003.* ISO/ANSI, 2003.

Michael Morse, Jignesh M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In *VLDB '07: Proceedings of the 31th International Conference on Very Large Data Bases*, pages 267–278, Vienna, Austria, 2007.

Oracle. *Oracle Database, SQL Reference, 10g Release 2 (10.2)*, Dezember 2005.

Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467–478, San Diego, California, USA, 2003.

Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

Timotheus Preisinger and Werner Kießling. The Hexagon Algorithm for Pareto Preference Queries. In *3rd Multidisciplinary Workshop on Advances in Preference Handling*, Vienna, Austria, 2007.

Timotheus Preisinger, Werner Kießling, and Markus Endres. The BNL^{++} Algorithm for Evaluating Pareto Preference Queries. In *ECAI 2006: Advances in Preference Handling*, 2006.

Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer, corrected and expanded second printing, 1988 edition, 1985.

Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient Progressive Skyline Computation. In *VLDB '01: Proceedings of 27th International Conference on Very Large Data Bases*, pages 301–310, Rome, Italy, 2001.

Yufei Tao and Dimitris Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, 2006.

Index