Semi-Skyline Optimization of Constrained Skyline Queries

Markus Endres **Werner Kießling**

Institute for Computer Science
University of Augsburg,
86135 Augsburg, Germany,
Email: {endres, kiessling}@informatik.uni-augsburg.de

Abstract

Skyline evaluation techniques (also known as Pareto preference queries) follow a common paradigm that eliminates data elements by finding other elements in a data set that dominate them. Nowadays already a variety of sophisticated skyline evaluation techniques are known, hence skylines are considered a well researched area. On the other hand, the skyline operator does not stand alone in database queries. In particular, the skyline operator may commute with the selection operator which may express hard constraints on the skyline. In this paper, we address skyline queries that satisfy some hard constraints, so-called *constrained skyline queries*. We will present novel optimization techniques for such queries, which allow more efficient computation. For this, we propose *semi-skylines* which can be used effectively for algebraic optimizations of skyline queries having a mixture of hard constraints and soft preference conditions. All our efficiency claims are supported by a series of performance benchmarks.

Keywords: Constrained skyline queries, Skyline optimization, Semi-Skylines, Preference

1 Introduction

The skyline operator has emerged as an important and very popular summarization technique for multi-dimensional data sets. The skyline, or Pareto operator selects those objects from the database that are not dominated by any others. An object dominates another object, if it is *as good or better in all dimensions and better in at least one dimension*. Therefore, skyline queries have shifted retrieval models from exact matching of attribute values to the notion of best matching database objects.

But skyline queries may return too many objects if the dimensionality of the data space is large, or, the dimensions are anti-correlated. Therefore, one may put *constraints* on some dimensions (or attributes of a database relation) to express hard restrictions, i.e., one defines a *skyline query that satisfy some hard constraints*. These hard constraints could be expressed by the *selection operator* in a database query. There exists two different types of such queries in the literature: *constrained skyline queries* (Hafenrichter & Kießling 2005, Papadias et al. 2005, Dellis et al. 2006) and *skyline queries with constraints* (Zhang & Alhajj 2010). Given a set of hard constraints, a *constrained skyline query* returns the most interesting objects in the data space defined by these constraints. In contrary, *skyline queries with constraints* return the objects from the skyline restricted by the constraints. In general the result of *skyline queries with constraints* is different from the result reported by *constrained skyline queries*. Thus, an object in the former result is not necessarily a object in the latter.

Constrained skyline queries often occur in real world applications, e.g. in a *diet planning service*. The United States Department of Agriculture (USDA, http://www.nal.usda.gov/fnic/) published a food database which contains nutritional facts for more than 7000 types of food. A user may be interested in finding meals that satisfy nutritional requirements such as a restriction on the number of calories (`Cal`), the amount of vitamin C (`Vc`), and the amount of total lipid fat (`Fat`). However, in the context of a diet, each user has preferences concerning its meal. Such a *constrained preference query* is shown in Figure 1, using the *PreferenceSQL* language from Kießling and Köstler (2002).

```
SELECT *
FROM  Soup S, Meat M, Beverage B
WHERE  S.Cal + M.Cal + B.Cal ≤ 1100
  AND  S.Vc + M.Vc + B.Vc    ≥ 38
  AND  S.Fat + M.Fat + B.Fat ≤ 9
PREFERRING
    S.Name IN ('Chicken', 'Noodle') AND
    M.Name IN ('Beef')              AND
    B.Vc HIGHEST
```

Figure 1: Sample Preference SQL query

In our running example a user expresses his or her preferences after the keyword `PREFERRING`. It is a Pareto preference (`AND` in the `PREFERRING`-clause[1]) consisting of preferences on soups, meat, and beverages. The keyword `IN` denotes a preference for members of a given set, a POS-preference. Hence, the user prefers *Chicken* and *Noodle* soups over all others. Furthermore, the user wants *Beef* and a drink with a maximum of vitamin C (`B.Vc HIGHEST`). The preferences are evaluated after the selection operator as stated by *constrained skyline queries*, following the operational semantics of skyline queries from Börzsönyi et al. (2001) and Hafenrichter & Kießling (2005), where the skyline operator is logically applied after the selection operation and joins, respectively.

The result of a preference query consists of *best matches only* (BMO-set, (Kießling 2002)). Skyline queries are a special case of this BMO approach: Basically, they only allow HIGHEST (MAX) and LOWEST (MIN) preference constructors to participate in

[1] Note that `AND` in the `WHERE`-clause means Boolean conjunction, whereas `AND` in the `PREFERRING`-clause denotes Pareto preference construction, i.e., all preferences are equally important.

a Pareto preference (Kießling 2002). Therefore all results of this paper apply to skyline queries as well.

The optimization of constrained skyline queries as given in Figure 1 is essential to support fast result computation (Börzsönyi et al. 2001, Papadias et al. 2005, Hafenrichter & Kießling 2005, Zhang & Alhajj 2010). However, the optimization of such queries is far from being a trivial task. The query in Figure 1 contains hard constraints referring several relations and user preferences on some attributes. Conventional approaches implement such queries by a set of binary join operators and evaluate the hard constraints. Afterwards the user preferences as soft selection combined with the Pareto operator are evaluated by a skyline algorithm to retrieve the best matching objects. This can be seen in Figure 2.

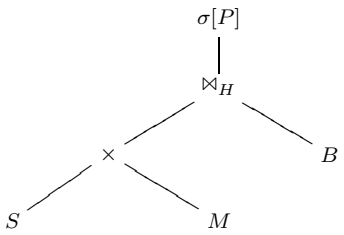

Figure 2: Unoptimized operator tree.

In this operator tree $\sigma[P]$ denotes Pareto, or skyline, evaluation and $\bowtie_H$ the join with the hard constraints from the query in Figure 1. The query evaluation process must evaluate the cartesian product of all tuples of all relations (about $120 \cdot 10^6$ combinations using the USDA database), which leads to *high memory and computation costs*, particularly for large relations.

It would be attractive to apply the skyline operator before the selection operator, or the join, because the skyline operator reduces the size of the intermediate results. Therefore, a skyline operator before a join makes the join cheaper. Also, non-reductive joins tend to increase the size of intermediate results, so that the skyline operator itself becomes cheaper if it is pushed through the join. Furthermore, pushing the skyline operator through a join might make it possible to use an index to compute the skyline, as described by Börzsönyi et al. (2001).

In this paper we present *semi-skylines* which allow an algebraic optimization of constrained skyline queries. Semi-skylines eliminate tuples from relations which definitely can never be in the result set before building the costly join. This reduces the relation sizes and therefore the computation costs and needed memory for the join.

The remainder of this paper is organized as follows: In Section 2 we discuss some related work. Section 3 contains the formal background used in this paper and defines semi-skylines. Based on semi-skylines we will develop relational query transformation laws and show the benefit for constrained skyline query optimization in Section 4. Afterwards, we introduce the *Staircase* algorithm for an efficient evaluation of semi-skylines in Section 5. We conduct an extensive performance evaluation on real and synthetic data sets in Section 6. Section 7 contains our concluding remarks.

2 Related Work

The evaluation of Pareto preference queries and finding the skyline was brought in a database context with the skyline operator by Börzsönyi et al. (2001). Since then, a lot of algorithms have been developed in this context. Index based algorithms are designed for flat query structures and have a high maintenance overhead associated with database updates, e.g. Kossmann et al. (2002), or Papadias et al. (2005). Nested-loop algorithms are the generic way of computing a skyline, e.g. Godfrey et al. (2005). Despite lower performance compared to index algorithms, they are capable of processing arbitrary data without any preparations necessary. Recently, algorithms have been developed exploiting the structure of the lattice imposed by the skyline operator on the data space to identify the skyline, but they keep this lattice structure in main memory and are only applicable on low-cardinality domains, see the works of Preisinger and Kießling (2007), and Morse et al. (2007).

Although, the evaluation of skyline queries is a well researched area, constrained skyline queries have not been intensively researched in the last years. There exists some algorithms for constrained skyline computation, e.g. BBS by Papadias et al. (2005), STA introduced by Dellis et al. (2006), or a modification of SaLSa by Sun et al. (2008). But all of them check the hard constraints 'inside' their algorithms, i.e., they are modified join operators. All these algorithms have the disadvantage that in the case of a cartesian product, or a join, still all tuple combinations have to be computed before the skyline join can be applied. Also, Raghavan and Rundensteiner (2010) consider such queries. They present a progressive algorithm by translating a tuple-level processing into a job-sequencing problem. Jin et al. (2007 and 2010) integrate different join methods into skyline computation. (Jin et al. 2007) covers the skyline operator on joins and aggregates. In (Jin et al. 2010) only equijoins are considered. Nevertheless, no tuple reduction can be done before the join is evaluated. Cui et al. (2008) propose the PaDSkyline algorithm in distributed environments. However, PaDSkyline does not address algebraic optimization issues, which are the main contribution of Semi-Skylines.

There are many publications on traditional database query optimization involving joins, but none of them deal with skylines, e.g. (Agarwal et al. 1998, Guha et al. 2003, Liu et al. 2005, Nestorov et al. 2007).

3 Semi-Skylines

Skyline queries, in particular preference queries and their integration into databases have been in focus for some time, leading to diverse approaches, e.g. (Kießling 2002, Chomicki 2003). We shortly review the preference model from Kießling (2002). Afterwards we define the Semi-Pareto preference which is the key idea of this paper.

3.1 Preference Background

A preference $P = (A, <_P)$, where A is a set of attributes, is a *strict partial order* on the domain of A. The term $\mathbf{x} <_\mathbf{P} \mathbf{y}$ is interpreted as "*I like y more than x*". Two tuples x and y are *indifferent*, if $\neg(x <_P y) \wedge \neg(y <_P x)$. The *skyline* of a preference $P = (A, <_P)$ on an input relation R are all tuples that are not dominated w.r.t. the preference. It is computed by the *preference selection operator* $\sigma[P](R)$ (called *winnow* by Chomicki (2003) , *BMO*-set by Kießling (2002)):

$$\sigma[P](R) := \{t \in R \mid \neg\exists t' \in R : t <_P t'\}$$

It finds all *best matching* tuples t for the preference P with $A \subseteq \text{attr}(R)$, where $\text{attr}(R)$ denotes all attributes of a relation R.

Preferences on single attributes are called *base preferences*. There are base preference constructors for continuous and for discrete domains. The discrete

POS-preference POS(A, POS-set) for example states that the user has a set of preferred values, the POS-set, in the domain of A. The sample query in Figure 1 shows two POS preferences with POS-sets for *Soup* and *Meat*. For example, *Chicken* and *Noodle* soups are preferred to all other soups.

Continuous numerical domains need a different type of preferences. For this purpose Kießling (2005) defined many numerical preference constructors, e.g., HIGHEST, LOWEST, ANTICHAIN. The extremal preferences HIGHEST and LOWEST allow users to express easily their desire for values as high or as low as possible. ANTICHAIN considers each tuple as equally good. Formally:

$\triangleright$ P:=HIGHEST(A): A tuple x is worse than a tuple y if the value of x is lower than y:

$$x <_P y \iff x < y$$

$\triangleright$ P:=LOWEST(A): A tuple x is worse than a tuple y if the value of x is higher than y:

$$x <_P y \iff x > y$$

$\triangleright$ P:=ANTICHAIN(A): All values are considered as equally good, i.e., $<_P = \emptyset$.

The query in Figure 1 shows a HIGHEST preference for the amount of Vc in the *Beverage* products.

There is the need to combine several base preferences into more *complex preferences*. One way is to list a number of preferences that are all equally important to the user. This is the concept of *Pareto* preferences.

Definition 1. Pareto Preference (Kießling 2002)
For preferences $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$, a Pareto preference $P := P_1 \otimes P_2 = (A_1 \times A_2, <_P)$ is defined as:

$$\begin{aligned}(x_1, x_2) <_P (y_1, y_2) \iff \\ (x_1 <_{P_1} y_1 \ \wedge \ (x_2 <_{P_2} y_2 \ \vee \ x_2 = y_2)) \ \vee \\ (x_2 <_{P_2} y_2 \ \wedge \ (x_1 <_{P_1} y_1 \ \vee \ x_1 = y_1)\end{aligned}$$

A generalization to more than two preferences is straightforward.

If we restrict the attention to LOWEST (MIN), HIGHEST (MAX) and ANTICHAIN (DIFF) as input preferences for a Pareto preference P, then Pareto preference queries coincide with the traditional *skyline queries*.

An important subclass of preferences are *weak order preferences* (WOP, (Kießling 2005)), i.e., strict partial orders for which negative transitivity holds: $\neg(x <_P y) \ \wedge \ \neg(y <_P z) \Rightarrow \neg(x <_P z)$ for a preference P. For a WOP $P = (A, <_P)$ the dominance test can be efficiently done by a numerical *utility function* f_P which depends on the type of preference. Dominated tuples have higher function values.

$$\begin{aligned}f : dom(A) \to \mathbb{R}_0^+ \\ x <_P y \iff f_P(x) > f_P(y)\end{aligned}$$

For WOPs two domain values x and y having the same function value are considered as *substitutable* and are treated as one *equivalence class* (regular SV-semantics, cp. (Kießling 2005)).

The utility function has to be defined individually for every type of base preference. For numerical base preferences the utility function is interpreted as the numerical distance from a perfect value, e.g., $f_{LOWEST(A)}(x) := x - min$, where min is the minimum value of the domain of A (Kießling 2005).

3.2 Semi-Pareto Preference

The *Semi-Pareto* preference is the key to our relational algebraic transformation laws for constrained skyline queries. Firstly introduced as Cutoff preference by Endres & Kießling (2008), our Semi-Pareto preference provides an optimization technique for preference queries in combination with the selection operator. Since this workshop paper was a first approach, and only covers the fundamentals of constrained skyline optimization, we now complete the theoretical background for our optimization techniques which we introduce in Section 4.

Comparing the definition of Semi-Pareto to Pareto, it is evident that Semi-Pareto is "the half of a Pareto preference".

Definition 2. Semi-Pareto Preference
Let $P_1 = (A_1, <_{P_1})$, $P_2 = (A_2, <_{P_2})$ be preferences.

a) Left-Semi-Pareto: $P_1 \lessdot\otimes P_2 = (A_1 \times A_2, <_{P_1 \lessdot\otimes P_2})$

$$\begin{aligned}(x_1, x_2) <_{P_1 \lessdot\otimes P_2} (y_1, y_2) \iff \\ x_1 <_{P_1} y_1 \ \wedge \ (x_2 <_{P_2} y_2 \ \vee \ x_2 = y_2)\end{aligned}$$

b) Right-Semi-Pareto: $P_1 \otimes\gtrdot P_2 = (A_1 \times A_2, <_{P_1 \otimes\gtrdot P_2})$

$$\begin{aligned}(x_1, x_2) <_{P_1 \otimes\gtrdot P_2} (y_1, y_2) \iff \\ x_2 <_{P_2} y_2 \ \wedge \ (x_1 <_{P_1} y_1 \ \vee \ x_1 = y_1)\end{aligned}$$

The proof that Semi-Pareto is a preference, i.e., it is irreflexive and transitive, can be done straightforward, cp. the extended version of this paper (Endres & Kießling 2010). We now define *Semi-Skylines*.

Definition 3. Semi-Skyline
The Semi-Skyline of a Semi-Pareto preference P ($P := P_1 \lessdot\otimes P_2$ or $P := P_1 \otimes\gtrdot P_2$) on an input relation R are all tuples that are not dominated w.r.t. P.

Example 1. *Consider two preferences on Table 1:*

- $P_1 = POS(B.Name, \ 'Red\ Wine')$
- $P_2 = HIGHEST(Vc)$

B	*ID*	*Name*	*Cal*	*Vc*	*Fat*
	B1	Red Wine	85	1	0
	B2	Red Wine	181	14	0
	B3	Coke	220	21	2
	B4	Lemonade	281	17	2
	B5	Red Wine	400	4	0

Table 1: A sample data set for *Beverages* B.

Then we have the following semi-skylines:

- $\sigma[P_1 \lessdot\otimes P_2](B) = \{B1, B2, B3, B4, B5\}$
  No tuple is dominated due to $P_1 \lessdot\otimes P_2$. All Red Wines are indifferent concerning P_1, and therefore cannot be worse than another tuple. B3 and B4 are worse than the others concerning P_1, but have a higher value for Vc, and therefore are not dominated.
- $\sigma[P_1 \otimes\gtrdot P_2](B) = \{B2, B3\}$
  B2 dominates all other Red Wines because it has the highest amount of Vc, and concerning P_1 they are substitutable. B4 is dominated by B3 (substitutable in the kind of drink, but worse in the amount of Vc), but B2 does not dominate B3 (Red Wine is better than Coke, but Coke has a higher value for Vc).

We conclude the following lemma:

Lemma 1. *Semi-Pareto is not a WOP.*

To show that Semi-Pareto is not a WOP we give a counterexample.

Example 2. *Consider the sample data set from Table 1 holding a relation on beverages.*

Given two preferences $P1 = LOWEST(Cal)$ and $P2 = LOWEST(Fat)$ and $P = P1 \otimes\!\!> P2$.

Then 'B1' is indifferent to 'B5', since none of them is better than the other concerning P. Furthermore, 'B5' is indifferent to 'B3'. It has more calories, but is better in the amount of fat. But 'B1' is better than 'B3', because B1' has less calories and less fat than 'B3'. Hence, negative transitivity does not hold.

Note that Semi-Pareto in contrast to Pareto is neither commutative nor associative. However, the following laws hold:

Corollary 1. Algebraic Laws for Semi-Pareto

a) $P_1 \otimes\!\!< P_2 = P_2 \otimes\!\!> P_1$

b) $(P_1 \otimes\!\!< P_2) \otimes\!\!< P_3 = P_1 \otimes\!\!< (P_2 \otimes P_3)$

Proof. The proof is straightforward and can be done by applying Boolean algebraic transformations. □

4 Constrained Skyline Optimization

In this section we study *constrained Skyline queries* and their algebraic optimization. However, we present a more general approach for arbitrary preferences.

4.1 Formal Background

Given a preference $P = (A, <_P)$ on a database schema R with $A \subseteq attr(R)$, and a Boolean condition H, i.e., a hard constraint. Then, using preference selection from Section 3, we define a *constrained preference (skyline) query* as $\sigma[P](\sigma_H(R))$. Under some conditions, preference selection $\sigma[P]$ and hard selection σ_H are commutative, cp. Chomicki (2003) and Hafenrichter & Kießling (2005).

Theorem 1. Push Preference over Hard Selection
For a preference $P = (A, <_P)$ with $A \subseteq attr(R)$ and a hard constraint H the following holds:

$$\sigma[P](\sigma_H(R)) = \sigma_H(\sigma[P](R)) \iff$$
$$\forall w \in R : H(w) \ \wedge \ \exists v \in R : w <_P v \rightarrow H(v)$$

If a tuple w is dominated by a tuple v, and w fulfills the hard constraint, then v has to fulfill the hard constraint, too. This guarantees the reduction of a tuple w only if for each dominating tuple v the condition $H(v)$ is fulfilled. Hence, a commutation due to Theorem 1 is possible.

Example 3. *Consider $P := HIGHEST(Vc)$ and the sample data set of Beverages in Table 1.*

We put a hard constraint $H := Vc \geq 15$ on the preference, i.e., $\sigma[P](\sigma_{Vc \geq 15}(B))$. A commutation of the preference and the hard selection is possible, because tuples which fulfill the hard constraint also fulfill the preference.

Therefore, $\sigma[P](\sigma_{Vc \geq 15}(B)) = \sigma_{Vc \geq 15}(\sigma[P](B))$, both leading to the result 'B3'.

4.2 Algebraic Optimizations

In this section we look at a database relation $R = (A_1, ..., A_l, B_1, ..., B_k)$ where the B_i's are numerical attributes. We consider a hard constraint H on R as follows:

$$H := h(b_1, ..., b_k) \ \Theta \ c, \quad \Theta \in \{\leq, <, >, \geq, =, \neq\}$$

where $h : dom(B_1) \times ... \times dom(B_k) \rightarrow \mathbb{R}$ is a *monotone function*, $b_i \in dom(B_i)$, and $c \in \mathbb{R}$ a constant. An instance of such a query is our running example, see Figure 1.

Applying Theorem 1 in the presence of hard constraints like H is a non-trivial task. In the following we introduce a novel algebraic optimization idea that achieves to satisfy the complex conditions required for Theorem 1 to hold. The key idea is to syntactically derive an *induced preference* from the given query, which then can be used as a *prefilter preference*.

Definition 4. Prefilter Preference
A preference $Q = (A, <_Q)$ is a prefilter preference for a preference $P = (A, <_P)$ w.r.t. a relation R iff

$$\sigma[P](R) = \sigma[P](\sigma[Q](R))$$

Definition 5. *Induced Preference*
Given a database relation R, and a hard constraint H as above. Then we define a preference $\overleftarrow{H}_{B_i}$ induced by B_i as:

$$\overleftarrow{H}_{B_i} := \begin{cases} LOWEST(B_i) & \text{if} \quad \Theta \in \{<, \leq\} \\ HIGHEST(B_i) & \text{if} \quad \Theta \in \{>, \geq\} \\ ANTICHAIN(B_i) & \text{if} \quad \Theta \in \{=, \neq\} \end{cases}$$

For all induced preferences $\overleftarrow{H}_{B_i}, \ i = 1, \ldots, k$ we define the overall induced preference $\overleftarrow{H}$ as

$$\overleftarrow{H} := \overleftarrow{H}_{B_1} \otimes \ldots \otimes \overleftarrow{H}_{B_k}$$

Example 4. *Consider the following query*

$$\sigma_{S.Cal+M.Cal+B.Cal \leq 1100}(S \times M \times B)$$

where H is the sum of the calories. Since Θ is "$\leq$" our induced preferences are

- $\overleftarrow{H}_{S.Cal} := LOWEST(S.Cal)$
- $\overleftarrow{H}_{M.Cal} := LOWEST(M.Cal)$
- $\overleftarrow{H}_{B.Cal} := LOWEST(B.Cal)$

The overall induced preference is

$$\overleftarrow{H} = \overleftarrow{H}_{S.Cal} \otimes \overleftarrow{H}_{M.Cal} \otimes \overleftarrow{H}_{B.Cal}$$

Using these definitions, our Semi-Pareto preference can be pushed over the hard selection operator.

Theorem 2. Push Semi-Pareto over Hard Selection
Given an arbitrary preference P and a hard constraint H on a relation R. Then

$$\sigma[P \otimes\!\!< \overleftarrow{H}](\sigma_H(R)) = \sigma_H(\sigma[P \otimes\!\!< \overleftarrow{H}](R))$$

where $\overleftarrow{H}$ is an induced preference from Definition 5.

Proof. Consider a relation R and tuples t and t'. Assume [2]

$$t <_{P \otimes\!\!< \overleftarrow{H}} t' \ \wedge \ H(t) \overset{Def. \otimes\!\!<}{\iff}$$
$$t <_P t' \ \wedge \ (t <_{\overleftarrow{H}} t' \ \vee \ t =_{\overleftarrow{H}} t') \ \wedge \ H(t)$$

[2] If two tuples have equal values concerning the attributes of a preference P we write $t =_P t'$.

Since t fulfills the hard constraint H and is worse or as good as t' concerning the induced preference, it follows that t' fulfills the hard constraint, too, i.e., $H(t')$ is valid. Therefore we can apply Theorem 1 and push the preference over the hard constraint. □

If P is an induced preference, i.e., LOWEST, HIGHEST or ANTICHAIN we can simplify Theorem 2 as follows:

Corollary 2. *Consider a preference query where $\overleftarrow{H}$ is an induced preference, then:*

$$\sigma[\overleftarrow{H}](\sigma_H(R)) = \sigma_H(\sigma[\overleftarrow{H}](R))$$

Semi-Pareto can be used as a *prefilter preference* to eliminate tuples from the underlying relation which are definitely no candidates for the skyline. Particularly, this is a crucial step in queries involving joins as we will see later.

Theorem 3. Prefilter Preference and Hard Selection *Consider a preference query $\sigma[P](\sigma_H(R))$ with a hard constraint H on a relation R. Then $P \lessdot\!\otimes \overleftarrow{H}$ is a prefilter preference for P, i.e.,*

$$\sigma[P](\sigma_H(R)) = \sigma[P](\sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)))$$

Proof. We prove "$\subseteq$" and "$\supseteq$".

"$\subseteq$": Let $t \in \sigma[P](\sigma_H(R))$, i.e. $\neg\exists t' \in \sigma_H(R) : t <_P t'$.
Assume, $t \notin \sigma[P](\sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)))$. Then either

$$\begin{aligned} & \exists t' \in \sigma_H(R) : t <_{P \lessdot\!\otimes \overleftarrow{H}} t' \\ \Longrightarrow\ & t <_P t' \Longrightarrow t \notin \sigma[P](\sigma_H(R)) \end{aligned}$$

or

$$\begin{aligned} & \exists t' \in \sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)) : t <_P t' \\ \Longrightarrow\ & \exists t' \in \sigma_H(R) : t <_P t' \end{aligned}$$

A contradiction to $t \in \sigma[P](\sigma_H(R))$

"$\supseteq$": Let $t \in \sigma[P](\sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)))$.
Assume, $t \notin \sigma[P](\sigma_H(R))$, i.e.,

$$\exists t' \in \sigma_H(R) : t <_P t'$$

But then, t' must be dominated w.r.t. $P \lessdot\!\otimes \overleftarrow{H}$ since $t \in \sigma[P](\sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)))$, i.e.,

$$\begin{aligned} \exists t'' \in \sigma_H(R) \ &: \ t' <_{P \lessdot\!\otimes \overleftarrow{H}} t'' \qquad \text{i.e.,} \\ \exists t'' \in \sigma_H(R) \ &: \ t' <_P t'' \wedge \\ & \quad (t' <_{\overleftarrow{H}} t'' \ \vee \ t' =_{\overleftarrow{H}} t'') \end{aligned}$$

In sum, we have $t' <_P t''$ and $t <_P t'$. By transitivity it follows $t <_P t''$, hence a contradiction to $t \notin \sigma[P](\sigma[P \lessdot\!\otimes \overleftarrow{H}](\sigma_H(R)))$. □

Now we can push Semi-Pareto over the hard constraint.

Corollary 3. Insert and Push Semi-Pareto over H

$$\sigma[P](\sigma_H(R)) = \sigma[P](\sigma_H(\sigma[P \lessdot\!\otimes \overleftarrow{H}](R)))$$

Proof. By Theorem 2 and Theorem 3. □

Of course, our results are not restricted to a single input relation. They are also valid for cartesian products having a hard constraint concerning attributes of all participating relations.

Corollary 4. Push Semi-Pareto over Cart. Prod. *Let $P_1 = (A_1, <_{P_1})$, $P_2 = (A_2, <_{P_2})$ be two preferences, and $H := h(b_1, b_2)$ a hard constraint, $A_1, B_1 \subseteq attr(R)$ and $A_2, B_2 \subseteq attr(S)$, $b_1 \in dom(B_1)$, and $b_2 \in dom(B_2)$. Then:*

a) $\sigma[P_1](\sigma_H(R \times S)) = \sigma[P_1](\sigma_H(\sigma[P_1 \lessdot\!\otimes \overleftarrow{H}_{B_1}](R) \times S))$

b) $\sigma[P_1 \otimes P_2](\sigma_H(R \times S)) = \sigma[P_1 \otimes P_2](\sigma_H(\sigma[P_1 \lessdot\!\otimes \overleftarrow{H}_{B_1}](R) \times \sigma[P_2 \lessdot\!\otimes \overleftarrow{H}_{B_2}](S)))$

Proof. This is a consequence of Corollary 3 and law L2 from Hafenrichter & Kießling (2005), "Push Preference Over Cartesian Product". □

This theorem allows us to push the Semi-Pareto prefilter preference over a hard constraint as presented in the next example.

Example 5. *Revisit the preference query from Figure 1 with preferences*

- $P_1 = POS(S.Name, \{'Chicken', 'Noodle'\})$
- $P_2 = POS(M.Name, 'Beef')$
- $P_3 = HIGHEST(B.Vc)$

combined to a Pareto preference $P = P_1 \otimes P_2 \otimes P_3$. Furthermore, consider the single hard constraint on the sum of calories (Cal) that must be less or equal to 1100 kcal, $H = S.Cal + M.Cal + B.Cal \leq 1100$. The induced preferences are the same as in Example 4. Using Corollary 4b) we can insert the prefilter preferences and push them over the hard constraint down to the relations, cp. Figure 3.

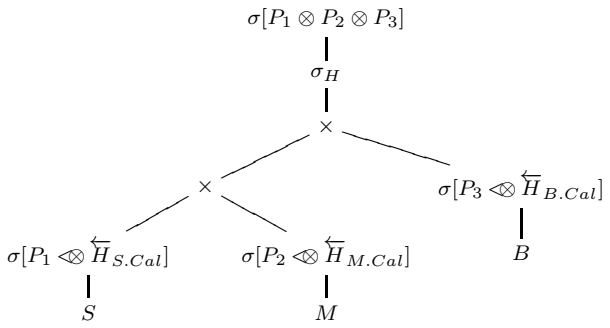

Figure 3: Push Semi-Pareto over Cart. Prod.

The insertion of our Semi-Pareto prefilter preferences lead to an elimination of tuples from the relations before building the cartesian products and hence reduces memory and computation costs.

In the case of a hard constraint in combination with Theta-joins (or Equi-joins) like $R \bowtie_{R.X=S.X} S$ for $X \subseteq attrR \cap attrS$ we have to ensure that we do not eliminate join partners, i.e., for each tuple in the first relation there must exist a join partner in the second relation. To get rid of this problem we have to evaluate the Semi-Pareto preference as a *grouped preference selection*. Therefore, Semi-Pareto is only evaluated for tuples in the same equivalence class, i.e., grouped by X.

Corollary 5. Push Semi-Pareto over Join *Let $P = (A, <_P)$ be a preference and $H := h(b_1, b_2)$ a hard constraint on two relations R and S. Furthermore $A, B_1 \subseteq attr(R)$ and $B_2 \subseteq attr(S)$, as well as $X \subseteq attr(R) \cap attr(S)$, $b_1 \in dom(B_1)$, $b_2 \in dom(B_2)$, and $\Theta \in \{<, \leq, >, \geq, =, \neq\}$. Then*

$$\sigma[P](\sigma_H(R \bowtie_{R.X \Theta S.X} S)) = \sigma[P](\sigma_H(\sigma[(P \lessdot\!\otimes \overleftarrow{H}_{B_1})\ groupby\ X](R) \bowtie_{R.X \Theta S.X} (S)))$$

Proof. This is a consequence from Corollary 3 and Hafenrichter and Kießling (2005), law L6, which discusses "Push Preference Over Join". □

Until now, we have only considered one hard constraint H in combination with preference selection. However, in database queries multiple hard constraints composed by the SQL keywords `AND` (conjunction) or `OR` (disjunction) are often involved in the selection conditions. The next theorem will introduce a prefilter preference for such queries. For disjunctive constraints like $H := H_1\ \Theta_1\ c_1\ \ \texttt{OR}\ \ H_2\ \Theta_2\ c_2$, $\Theta_i \in \{<, \leq, <, \geq, =, \neq\}$, the condition $\sigma_H(R)$ could be regarded as the union of two operators $\sigma_{H_1\Theta_1 c_1}(R)$ and $\sigma_{H_2\Theta_2 c_2}(R)$. Therefore, we only consider conjunctive constraints.

Corollary 6. *Consider a preference query $\sigma[P](\sigma_H(R))$ with k conjunctive hard constraints $H := H_1 \wedge \cdots \wedge H_k$ on a relation R. Then:*

$$\sigma[P](\sigma_{H_1\wedge\cdots\wedge H_k}(R)) = \sigma[P](\sigma_{H_1\wedge\cdots\wedge H_k}(\sigma[P \lessdot\!\!\otimes (\overleftarrow{H_1} \otimes \ldots \otimes \overleftarrow{H_k})](R)))$$

Proof. Relational algebra leads to

$$\sigma[P](\sigma_{H_1\wedge\cdots\wedge H_k}(R)) = \sigma[P](\sigma_{H_1}(\ldots(\sigma_{H_k}(R))\ldots)$$

By Corollary 3 we know

$$\sigma[P](\sigma_{H_1\wedge\cdots\wedge H_k}(R)) = \sigma[P](\sigma_{H_1}(\sigma[P \lessdot\!\!\otimes \overleftarrow{H_1}](\sigma_{H_2}(\ldots(\sigma_{H_k}(R))\ldots))))$$

for the single hard constraint H_1. Now by further applying Corollary 3 on $\sigma[P \lessdot\!\!\otimes \overleftarrow{H_1}](\sigma_{H_2}(\ldots))$ we get

$$\sigma[P](\sigma_{H_1\wedge\cdots\wedge H_k}(R)) = \sigma[P](\sigma_{H_1}(\sigma_{H_2}(\ldots(\sigma_{H_k}(\underbrace{\sigma[(\ldots((P \lessdot\!\!\otimes \overleftarrow{H_1}) \lessdot\!\!\otimes \overleftarrow{H_2}) \lessdot\!\!\otimes \ldots \lessdot\!\!\otimes \overleftarrow{H_k})]}_{=\sigma[P \lessdot\!\!\otimes (\overleftarrow{H_1}\otimes\ldots\otimes\overleftarrow{H_k})] \text{ by Corollary 1}}(R)))\ldots)))$$

Note that the intermediate preferences $\sigma[P \lessdot\!\!\otimes \overleftarrow{H_i}]$ can be removed because they are only prefilter preferences and need not to be evaluated.

□

Example 6. *Consider the skyline query from Figure 1, and Example 5 with our preference $P := P_1 \otimes P_2 \otimes P_3$. Now, for all three hard sum constraints on 'Cal', 'Vc' and 'Fat' we have $H := H_1 \wedge H_2 \wedge H_3$ for*

- $H_1 := S.Cal + M.Cal + B.Cal \leq 1100$
- $H_2 := S.Vc + M.Vc + B.Vc \geq 38$
- $H_3 := S.Fat + M.Fat + B.Fat \leq 9$

The overall induced preferences are

$$\overleftarrow{H_S} := LOWEST(S.Cal) \otimes HIGHEST(S.Vc) \otimes LOWEST(S.Fat)$$
$$\overleftarrow{H_M} := LOWEST(M.Cal) \otimes HIGHEST(M.Vc) \otimes LOWEST(M.Fat)$$
$$\overleftarrow{H_B} := LOWEST(B.Cal) \otimes HIGHEST(B.Vc) \otimes LOWEST(B.Fat)$$

Applying Corollary 6 leads to:

$$\sigma[P](\sigma_H(S \times M \times B)) = \sigma[P](\sigma_H(\sigma[(P_1 \otimes P_2 \otimes P_3) \lessdot\!\!\otimes (\overleftarrow{H}_S \otimes \overleftarrow{H}_M \otimes \overleftarrow{H}_B)](S \times M \times B)$$

This is equal to

$$\sigma[P](\sigma_H(S \times M \times B)) = \sigma[P](\sigma_H(\sigma[P_1 \lessdot\!\!\otimes \overleftarrow{H}_S)](S) \times \sigma[P_2 \lessdot\!\!\otimes \overleftarrow{H}_M)](M) \times \sigma[P_3 \lessdot\!\!\otimes \overleftarrow{H}_B)](B)$$

5 Efficient Evaluation of Semi-Skylines

Many algorithms have been developed for skyline evaluation, cp. Section 2. Since the concept of semi-skylines is totally new, no specialized algorithm for its computation exists. Therefore, only BNL with a worst-case complexity of $\mathcal{O}(n^2)$ remains to evaluate semi-skylines. Thus, the question for an efficient algorithm arises. We introduce the *Staircase* algorithm for the evaluation of semi-skylines with guaranteed worst case complexity of $\mathcal{O}(n \log n)$. Staircase is a variant of the BNL algorithm, but the candidate window will be a *Skiplist* (Pugh 1990).

5.1 The Staircase Algorithm

For simplicity we restrict the attention to weak order preferences P_1 and P_2 as input for Semi-Pareto. Then we can define Left-Semi-Pareto from Definition 2 (analogously Right-Semi-Pareto) by utility functions:

$$(x_1, x_2) <_{P_1 \lessdot\!\!\otimes P_2} (y_1, y_2) \iff f_{P_1}(x_1) > f_{P_1}(y_1) \wedge f_{P_2}(x_2) \geq f_{P_2}(y_2)$$

A tuple $x := (x_1, x_2)$ is worse than a tuple $y := (y_1, y_2)$, iff the utility function value is worse in the first component, i.e., $f_{P_1}(x_1) > f_{P_1}(y_1)$ and worse or equal in the second one, i.e., $f_{P_2}(x_2) \geq f_{P_2}(y_2)$. Since we map tuples (x_1, x_2) with the same function values to equivalence classes represented by $(f_{P1}(x_1), f_{P_2}(x_2))$, we can state directly dominance using these equivalence classes. For a graphical interpretation have a look at Figure 4a. All equivalence classes in the pruning region PR, i.e., below and right of the equivalence class $[y] = (2, 2)$ are worse than y, because their f_{P_1} value is greater to 2 and worse or equal than 2 in the second preference P_2. Note that the equivalence classes on the dashed line are not dominated. Therefore, a tuple belonging to the equivalence class $(2, 2)$ dominates all tuples belonging to an equivalence class lying in PR. This dominance test is only possible if the underlying preferences are WOPs. Comparing a new tuple $x = (x_1, x_2)$ with equivalence class $[x] = (f_{P1}(x_1), f_{P_2}(x_2))$ leads to the following possibilities:

a) If $[x]$ falls into the pruning region PR (Figure 4a) we directly can state dominance using the equivalence classes. For example consider $[x] = (3, 3)$. Since $3 > 2$ and $3 \geq 2$ the equivalence class $[x]$ is worse then $[y]$, thus the tuple x is dominated. If an equivalence class falls directly on the dashed line it is not dominated, since tuples in such a class are not worse concerning the first preference.

b) If an equivalence class is left below of $[y]$ it is not dominated, but extends our staircase, cp. Figure 4b. Equally if an equivalence class is right above $[y]$ (or on the dashed line) our staircase will be extended. For example, $[y'] = (1, 3)$ and $[y''] = (3, 0)$ extends our staircase and therefore the pruning region PR. Inserting these equivalence classes all tuples lying in an equivalence class of the gray area are dominated, i.e., dominance can now be decided by the dichotomy of the staircase.

c) Only an equivalence class $[z]$ left above $[y]$ is better than $[y]$ and therefore dominates it, cp. Figure 4c. In this case, we have to *update* our staircase, since it is possible that $[z]$ dominates other equivalence classes (and their containing tuples), too. But updating is an easy step, because $[z]$ only can dominate equivalence classes right below of itself. All equivalence classes right below of $[z]$ (without the dashed line) in the order of our staircase

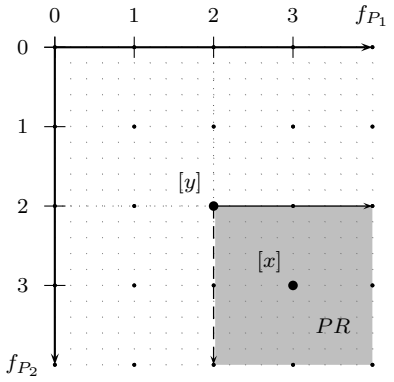

Figure 4a: Comparing a dominated equivalence class.

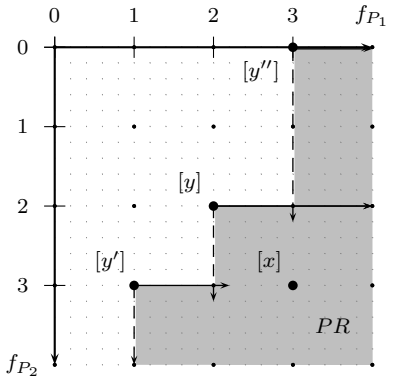

Figure 4b: Extending the Staircase.

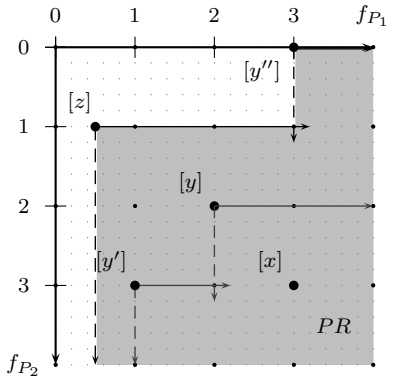

Figure 4c: Updating the Staircase.

points have to be deleted. For example, consider $[z] = (0.5, 1)$ in Figure 4c. $[z]$ dominates $[y]$ and $[y']$ and therefore we have to delete these equivalence classes and change the staircase to its new form consisting of $\{[z] = (0.5, 1), [y''] = (3, 0)\}$.

In the worst-case of BNL all tuples in the candidate window have to be compared to the new tuple to decide dominance. In contradiction, using the staircase we only have to decide if a tuple is left of the staircase, i.e., it dominates, or a tuple is right of the staircase, i.e., it is dominated. This leads to the question how to represent the staircase data structure?

5.2 The Staircase Data Structure

We observe that the dominance decision based on the staircase is only applicable if the equivalence classes on the staircase are comparable and ordered, yielding a total order using the Manhattan distance, cf. (Krause 1987).

Definition 6. Manhattan distance, L_1 norm
The Manhattan distance d_1 for an equivalence class $[x] = (x_1, x_2)$ in our staircase space is the distance from the point $(0, max_P)$ to the point $[x]$, where $max_P := max(max(f_{P_1}), max(f_{P_2}))$. $max(f_{P_i})$ is the maximum function value for P_i.

$$d_1([x]) = x_1 - x_2 + max_P$$

Lemma 2. Total Order of Staircase Points
The points on the staircase build a total order concerning the Manhattan distance from Definition 6.

Proof. We give the proof using a Left-Semi-Pareto preference $P := P_1 \ltimes P_2$ with weak order preferences $P_1 = (A_1, <_{P_1})$ and $P_1 = (A_2, <_{P_2})$. The proof for Right-Semi-Pareto can be done analogously.

Consider an equivalence class $[y] = (y_1, y_2)$ with Manhattan distance $d_1([y]) = y_1 - y_2 + max_P$ already on the staircase. We want to insert a newly tuple x with equivalence class $[x] = (x_1, x_2)$ and the same distance $d_1([x])$. Since max_P is fixed, they only have the same Manhattan distance if $y_1 - y_2 = x_1 - x_2$. This leads to the following possibilities:

- if $x_i = y_i,\ i \in \{1, 2\}$, both fall in the same equivalence class, i.e., tuple x will be added to $[y]$.
- if $x_1 > y_1$ and $x_2 > y_2$, then $[x]$ is dominated by $[y]$. Therefore tuple x is dominated, too.
- if $x_1 < y_1$ and $x_2 < y_2$, then $[y]$ is dominated and replaced by $[x]$.

Since these are all possibilities for $d_1([y]) = d_1([x])$ the staircase points build a total order concerning the Manhattan distance. □

As an example consider the equivalence classes $[z] = (0.5, 1)$ and $[y''] = (3, 0)$ from Figure 4c. We get a distance $d_1([z]) = -0.5 + max_P$ and $d_1([y'']) = 3 + max_P$.

Using the Manhattan distance the dominance decision in the staircase is easy. Search $d_1([x])$ of a tuple x. If $d_1([x])$ exists, just compare the utility function values with the existing equivalence class and find out dominance (or add the tuple to the equivalence class if it has the same function values). If $d_1([x])$ does not exists, compare the equivalence class of $[x]$ to the one with next lower distance. If $[x]$ is not dominated *insert* it into the staircase and *update* the staircase, i.e., *delete* all equivalence classes dominated by $[x]$.

Our first idea to use balanced search trees to store the staircase failed on finding the dominated equivalence classes for an *update* action, i.e., in a binary tree it is not easy to find all points which are worse than the newly inserted point. Skiplists are a collection of sorted linked lists, each at a given "level", that mimic the behavior of a binary search tree.

Example 7. *Figure 5 shows a Skiplist with equivalence classes as keys ordered by the Manhattan distance (number below the classes, $max_P = 10$). Among other points it contains the classes from Figure 4b.*

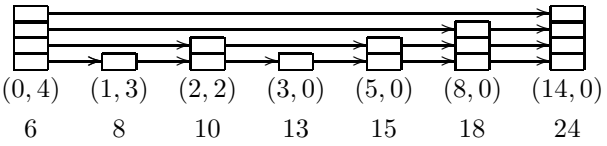

Figure 5: A Skiplist with maximum level 4.

The number of nodes in each list decreases with the level, implying that we can find a key quickly by searching first at higher levels, skipping over large numbers of shorter nodes, and progressively working downwards until a node with the desired key is found, or the bottom level is reached. Thus, the time complexity of a Skiplist operation (*insert*, *delete*, *search*) is logarithmic in the length of the list, $\mathcal{O}(\log n)$, cp. (Pugh 1990). Skiplists also provide easy access to all equivalence classes for the *update* action. Begin at the inserted point and run through the list until dominance fails.

Theorem 4. Staircase Time Complexity
Let P be a Semi-Pareto preference. Given n input tuples, then the semi-skyline of P can be evaluated in

- *worst-case runtime:* $\mathcal{O}(n \log n)$
- *best-case runtime:* $\mathcal{O}(n)$

Proof. For each input tuple we either have to insert or remove it from the staircase. These operations can be done in logarithmic time. Therefore, we get a worst-case complexity of $n \cdot \mathcal{O}(\log n)$.

Since our Staircase algorithm is a specialized BNL algorithm, we can guarantee a best-case runtime of $\mathcal{O}(n)$, cp. for instance (Godfrey et al. 2005). □

Note that for simplicity we have only presented the Staircase algorithm for WOPs. Staircase can be generalized to arbitrary preferences, but due to length restrictions this can not be presented here.

6 Performance Benchmarks

We now present results from an experimental study designed to show the benefit of our techniques.

All compared algorithms are implemented in our *PreferenceSQL* system (Hafenrichter & Kießling 2005), a Java SE 6 framework for preference queries. All experiments are performed on a 2.53GHz Core 2 Duo machine running Mac OS X with 4 GB RAM for the JVM. Moreover a buffer pool large enough for all operations to fit into the main memory for all tests was used. The input sets and the skyline points are kept in main memory, too. Performing all operations in main memory is the best case for all used algorithms since no external operation is necessary.

It has to be noted that this main memory requirement restricts the general applicability of our Staircase algorithm. However, as can be seen from the subsequent benchmarks, the Staircase algorithm can nevertheless be used for very large data sets.

6.1 BNL vs. Staircase

From Section 5 we know that BNL is the only algorithm to evaluate arbitrary preference queries. Therefore we compare BNL and our Staircase algorithm for semi-skyline evaluation. Both algorithms have been integrated in our PreferenceSQL framework to produce performance benchmarks.

We use synthetic data sets, since this is commonly used for skyline evaluation and allow us to carefully explore the effect of various data characteristics. For this, we generate data sets with correlated (*COR*), independent (*IND*) and anti-correlated (*ANTI*) distributions using an implementation of the popular data set generator of Börzsönyi et al. (2001), and vary the data cardinality n, and the number of distinct domain values c. In all synthetic experiments, the tuple size is 100 bytes (also used by Godfrey et al. (2005) in their experiments). A tuple has attributes of type *Integer* and *bulk* attributes with "garbage" characters to ensure that each tuple is 100 byte long. For simplicity we generated a Semi-Pareto preference consisting of LOWEST preferences (MIN in skylines queries).

Figure 6 shows runtimes for BNL and Staircase on an anti-correlated data set containing up to $n = 500K$ tuples. We fixed $c = 100K$, i.e., the domain contains 100.000 different values. Figure 7 contains the comparison of BNL and Staircase for an independent distributed data set. Due to the limited space we only present these benchmarks. Further tests with different cardinalities and number of distinct domain values can be found in the extended version of this paper, cp. (Endres & Kießling 2010). In all benchmarks it turns out that Staircase for semi-skyline evaluation performs much better than BNL.

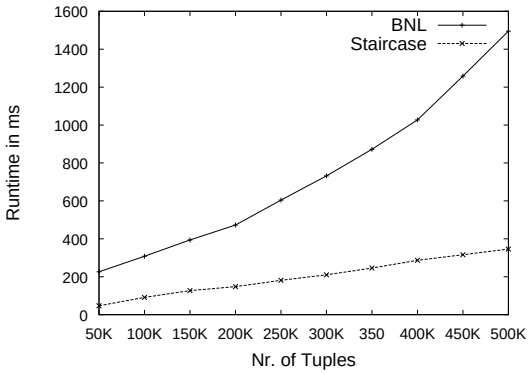

Figure 6: BNL vs. Staircase, ANTI

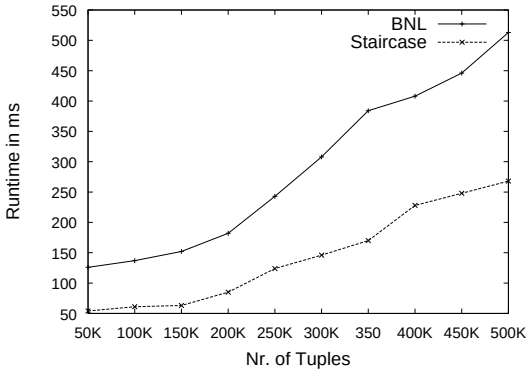

Figure 7: BNL vs. Staircase, IND

6.2 Constrained Skyline Optimization

To evaluate the Pareto prefilter preference, we performed several experiments. For this we integrated our optimization rules into the preference query optimizer of PreferenceSQL. This allows the evaluation of constrained preference queries. For the evaluation of the semi-skylines we used our Staircase algorithm from Section 5. The evaluation of the remaining preference was done by *Hexagon* (Preisinger & Kießling 2007). We evaluated the efficiency of our Semi-Pareto prefilter preference (abbr. *pref-prefilter*) by comparing the response times of several constrained preference queries to an evaluation of the queries with *standard optimization* (e.g., building joins, cp. Hafenrichter & Kießling (2005)), without prefilter preferences. We abbreviate this with *no-pref-prefilter*.

Remark: The reason for choosing the comparison between an *optimization with prefilter preferences* and *without prefilter preferences* is as follows: We developed generic optimization rules which can be easily integrated into a preference query optimizer. We do **not** rely on any index structures. Furthermore, the mentioned skyline join algorithms from Section 2 (Related Work) check the hard constraints 'inside' their algorithms, i.e., these works present modified join operators for constrained skyline evaluation. All these algorithms have the disadvantage that in the case of a cartesian product, or a join, still all tuple combinations have to be considered for the skyline join. Therefore, a comparison to these skyline join algorithms is not a good representation of the benefit of our optimization laws. However, these algorithms can be part of the operator repertoire to choose from.

For the evaluation we used the food database mentioned in the introduction. From this database we created several relations, e.g., *Soup*, *Meat*, and *Beverage* containing information about their eponymous types of food. The sizes of these relations are as follows: There are max. 500 soups, 680 meats, and 350 beverages available, i.e., there are about 120 Mio. possible combinations. We also evaluated the influence of various data distributions to our prefilter preference. In all tests it turns out, that our optimization laws yield significant benefits for constrained skyline computation. Due to limited space we only present tests on the USDA data set. For further benchmarks we refer to Endres and Kießling (2010).

Test 1: The first test is based on the query in Figure 1 and contains three constraints. For representation we varied the amount of calories *Cal*, which must be less than or equal to a value called *max_cal*. The amount of vitamin C (Vc) and the fat value (fat) are fixed values. Notice, modifying the parameter *max_cal* changes the selectivity of the query, while varying the size of the relations changes the size of the problem to be solved. We varied the *max_cal* value in a range from 500 to 1600 calories, see Figure 8. Since the prefilter preference only depends on the preferences and not on the hard constraints, the response time for the preference query with different *max_cal* is nearly constant for each approach. In contrast, the approach without prefilter preference takes much more time to evaluate the whole query, since it must build the full cartesian product to evaluate the join conditions and the Pareto preference.

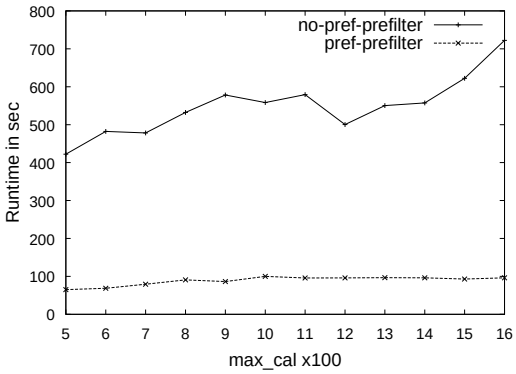

Figure 8: Different amount of calories.

Test 2: In our second test we run the query from Figure 1 with different relation sizes, i.e., a different number of possible tuple combinations (but fixed constraints) on our USDA database and demonstrate the results in Figure 9. Again, the prefilter preference eliminates tuples before building the cartesian product and therefore speeds up the evaluation of the join and the overall Pareto preference, respectively.

Test 3: We regard a preference query without a cartesian product, i.e., a query containing only one relation, namely the *Meat* table, but with a hard constraint on it ($M.Cal \leq 300$). Since the hard constraint is a strong filter for the data set, the assumption arises that inserting the prefilter preference and pushing it over the hard selection will not speed up the computation. This is due to the prefilter preference is not such a strong criteria as the hard selection. This result is verified by the benchmarks in Figure 10, where we run the query on the *Meat* table with a modified tuple count.

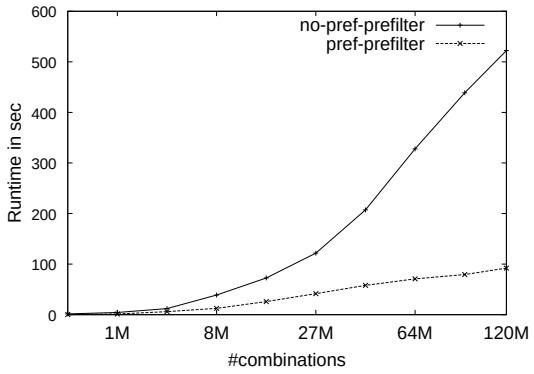

Figure 9: Different relations sizes.

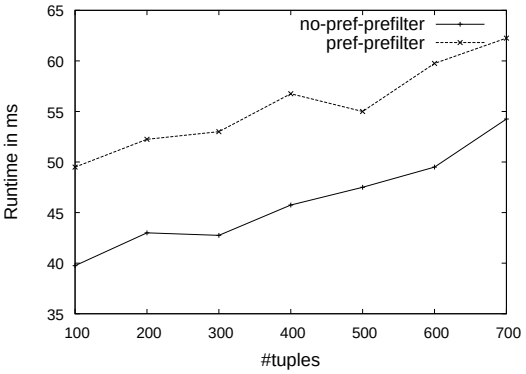

Figure 10: One relation, different tuple count.

The runtimes without the prefilter preference are much better, because using the prefilter implies the evaluation of itself, the evaluation of the hard selection, and the computation of the original preference on the top of the query. In contrast, without prefilter preference only the hard selection and the preference on top of the query has to be evaluated.

7 Summary and Outlook

In this paper we have proposed the novel concept of semi-skylines. For semi-skylines we have provided the *Staircase* algorithm, employing Skiplists for on-the-fly dominance testing with a worst-case complexity of $\mathcal{O}(n \log n)$. We do not rely on any pre-computed index structure. Hence, our methods are generally applicable. As an application of semi-skylines we presented algebraic preference query optimization techniques for constrained skyline queries. Our algebraic laws can be easily integrated into a database system by extending the query optimizer. After such an operator tree optimization a cost-based optimization phase can then choose the best available skyline evaluation algorithm. In particular, known skyline joins can be part of the operator repertoire to choose from. Thus our presented algebraic transformation approach using semi-skylines is not intended as a substitution for skyline joins, but may well complement each other. Our experimental results show the benefit of our algebraic optimizations. Although we reduced the query evaluation time from round about 10min to 1min (for $120 \cdot 10^6$ tuples), further work must be done for a very fast retrieval of all skyline points, e.g. in an online diet planning service.

References

Agarwal, P. K., Arge, L., Erickson, J., Franciosa, P. G. & Vitter, J. S. (1998), Efficient Searching with Linear Constraints, *in* 'PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems', ACM Press, pp. 169–178.

Börzsönyi, S., Kossmann, D. & Stocker, K. (2001), The Skyline Operator, *in* 'ICDE '01: Proceedings of the 17th International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 421–430.

Chomicki, J. (2003), Preference Formulas in Relational Queries, *in* 'TODS '03: ACM Transactions on Database Systems', Vol. 28, ACM Press, New York, NY, USA, pp. 427–466.

Cui, B., Lu, H., Xu, Q., Chen, L., Dai, Y. & Zhou, Y. (2008), Parallel Distributed Processing of Constrained Skyline Queries by Filtering, *in* 'ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 546–555.

Dellis, E., Vlachou, A., Vladimirskiy, I., Seeger, B. & Theodoridis, Y. (2006), Constrained subspace skyline computation, *in* 'CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management', ACM, New York, NY, USA, pp. 415–424.

Endres, M. & Kießling, W. (2008), Optimization of Preference Queries with Multiple Constraints, *in* 'PersDB '08: Proceedings of the 2nd International Workshop on Personalized Access, Profile Management, and Context Awareness: Databases (in conjunction with VLDB '08)', pp. 25–32.

Endres, M. & Kießling, W. (2010), Semi-Skylines and Skyline Snippets, Technical Report 2010-1, Institute of Computer Science, University of Augsburg.

Godfrey, P., Shipley, R. & Gryz, J. (2005), Maximal vector computation in large data sets, *in* 'VLDB '05: Proceedings of the 31st international conference on Very large data bases', VLDB Endowment, pp. 229–240.

Guha, S., Gunopulos, D., Koudas, N., Srivastava, D. & Vlachos, M. (2003), Efficient Approximation of Optimization Queries under Parametric Aggregation Constraints, *in* 'VLDB '03: Proceedings of the 29th international conference on Very large data bases', VLDB Endowment, pp. 778–789.

Hafenrichter, B. & Kießling, W. (2005), Optimization of Relational Preference Queries, *in* 'ADC '05: Proceedings of the 16th Australasian database conference', Australian Computer Society, Inc., Darlinghurst, Australia, pp. 175–184.

Jin, W., Ester, M., Hu, Z. & Han, J. (2007), 'The Multi-Relational Skyline Operator', *International Conference on Data Engineering* **0**, 1276–1280.

Jin, W., Morse, M. D., Patel, J. M., Ester, M. & Hu, Z. (2010), Evaluating skylines in the presence of equijoins, *in* 'ICDE '10: Proceedings of the 2010 IEEE International Conference on Data Engineering', IEEE, pp. 249–260.

Kießling, W. (2002), Foundations of Preferences in Database Systems, *in* 'VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases', VLDB Endowment, Hong Kong, China, pp. 311–322.

Kießling, W. (2005), Preference Queries with SV-Semantics, *in* J. R. Haritsa & T. M. Vijayaraman, eds, 'COMAD '05: Advances in Data Management 2005, Proceedings of the 11th International Conference on Management of Data', Computer Society of India, Goa, India, pp. 15–26.

Kießling, W. & Köstler, G. (2002), Preference SQL - Design, Implementation, Experiences, *in* 'VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases', VLDB Endowment, Hong Kong, China, pp. 990–1001.

Kossmann, D., Ramsak, F. & Rost, S. (2002), Shooting Stars in the Sky: An Online Algorithm for Skyline Queries, *in* 'VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases', VLDB Endowment, pp. 275–286.

Krause, E. F. (1987), *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*, Courier Dover Publications.

Liu, C., Yang, L. & Foster, I. (2005), Efficient Relational Joins with Arithmetic Constraints on Multiple Attributes, *in* 'IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium', IEEE Computer Society, Washington, DC, USA, pp. 210–220.

Morse, M., Patel, J. M. & Jagadish, H. V. (2007), Efficient Skyline Computation over Low-Cardinality Domains, *in* 'VLDB '07: Proceedings of the 33rd international conference on Very large data bases', VLDB Endowment, pp. 267–278.

Nestorov, S., Liu, C. & Foster, I. (2007), Efficient Processing of Relational Queries with Sum Constraints, *in* 'APWeb/WAIM '07: Advances in Data and Web Management, Joint 9th Asia-Pacific Web Conference, and 8th International Conference on Web-Age Information Management', Vol. 4505 of *Lecture Notes in Computer Science*, Springer, pp. 440–451.

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

Preisinger, T. & Kießling, W. (2007), The Hexagon Algorithm for Evaluating Pareto Preference Queries, *in* 'MPref '07: Proceedings of the 3rd Multidisciplinary Workshop on Advances in Preference Handling (in conjunction with VLDB '07)'.

Preisinger, T., Kießling, W. & Endres, M. (2006), The BNL++ Algorithm for Evaluating Pareto Preference Queries, *in* 'Proceedings of the 2nd Multidisciplinary Workshop on Advances in Preference Handling (in conjunction with ECAI '06)', pp. 114–121.

Pugh, W. (1990), 'Skip Lists: A Probabilistic Alternative to Balanced Trees', *Commun. ACM* **33**(6), 668–676.

Raghavan, V. & Rundensteiner, E. A. (2010), Progressive result generation for multi-criteria decision support queries, *in* 'ICDE '10: Proceedings of the 2010 IEEE International Conference on Data Engineering', IEEE, pp. 733–744.

Sun, D., Wu, S., Li, J. & Tung, A. K. H. (2008), Skyline-join in distributed databases, *in* 'ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering', IEEE Computer Society, pp. 176–181.

Zhang, M. & Alhajj, R. (2010), 'Skyline queries with constraints: Integrating skyline and traditional query operators', *Data Knowl. Eng.* **69**(1), 153–168.