UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Semi-skylines and skyline snippets

Markus Endres, Werner Kießling

Angaben zur Veröffentlichung / Publication details:

Endres, Markus, and Werner Kießling. 2010. “Semi-skylines and skyline snippets.” Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

UNIVERSITÄT AUGSBURG

Semi-Skylines and Skyline Snippets

Markus Endres and Werner Kießling

Report 2010-01 March 2010

INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © Markus Endres and Werner Kießling
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Semi-Skylines and Skyline Snippets

Markus Endres
Institute for Computer Science
University of Augsburg
86135 Augsburg, Germany
endres@informatik.uni-augsburg.de

Werner Kießling
Institute for Computer Science
University of Augsburg
86135 Augsburg, Germany
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 the data set that dominate them. To date already a variety of sophisticated skyline evaluation techniques are known, hence skylines are considered a well researched area. Nevertheless, in this paper we come up with interesting new aspects. Our first contribution proposes so-called semi-skylines as a novel building stone towards efficient algorithms. Semi-skylines can be computed very fast by a new Staircase algorithm. Semi-skylines have a number of interesting and diverse applications, so they can be used for constructing a very fast 2-dimensional skyline algorithm. We also show how they can be used effectively for algebraic optimization of preference queries having a mixture of hard constraints and soft preference conditions. Our second contribution concerns so-called skyline snippets, representing some fraction of a full skyline. For very large skylines, in particular for higher dimensions, knowing only a snippet is often considered as sufficient. We propose a novel approach for efficient skyline snippet computation without using any index structure, by employing our above 2-d skyline algorithm. All our efficiency claims are supported by a series of performance benchmarks. In summary, semi-skylines and skyline snippets can yield significant performance advantages over existing techniques.

1. INTRODUCTION

The skyline operator has emerged as an important and very popular summarization technique for multi-dimensional data sets. For a data set D consisting of data points $p_1, ..., p_n$ the skyline S is the set of all p_i such that there is no p_j that dominates p_i. p_i is said to *dominate* p_j if p_i is better than p_j in at least one dimension and not worse than p_j in all other dimensions, for a defined comparison function. The skyline introduced in [3] is related to several other problems, including maximal vectors [17], convex hulls [2] and *Pareto* sets [11]. As Pareto preference queries form a superset of the skyline, we refer to this general approach of skyline queries. An example of a preference query is shown in Figure 1, using the *Preference SQL* language from [13].

```
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 5
```

Figure 1: Sample Preference SQL query

In this example a user expresses its preferences after the keyword `PREFERRING`. It is a Pareto preference (`AND` in the `PREFERRING`-clause) consisting of preferences on soups, meat, and beverages, i.e. all preferences are equally important. The keyword `IN` denotes a preference for members of a given set, a POS-preference. Hence, the user preferes *Chicken* and *Noodle* soups over all others. Furthermore, the user wants *Beef* and a drink with a maximum of vitamin C, but a deviation of 5 does not matter (`HIGHEST 5`). The result of a preference query consists of *best matches only* (BMO-set, [11]). Skyline queries are a special case of this BMO approach: Basically, they only allow HIGHEST and LOWEST base preference constructors to participate in a Pareto preference ([11]). Therefore all results of this paper apply to skyline queries as well. A lot of algorithms have been developed in the context of skyline and Pareto queries. Generally, there are two types of algorithms: those depending on index structures ([15, 20]) and generic ones ([3, 7, 19, 22]).

Index based algorithms tend to be faster, but are less flexible. They are designed for flat query structures and have a high maintenance overhead associated with database updates. On the other hand, the generic algorithms show linear average-case and quadratic worst-case running times w. r.t. the size of the input relation. Recently, algorithms with linear worst-case complexity have been developed, cp. [22, 19]. Both use the structure of the lattice imposed by the Pareto 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. For high-cardinality domains LESS from [7] is currently considered to be the most efficient skyline algorithm that does not require indexing or preprocessing.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '10, September 13-17, 2010, Singapore
Copyright 2010 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

We will introduce the *semi-skyline* operator as one key contribution of this paper. Based on this operator we will develop a very fast 2d-skyline algorithm called *Staircase-Intersection* with logarithmic worst-case running time. The second key contribution are *skyline snippets* for a rapid computation of a subset of the skyline (or the skyline under some circumstances). The third application shows the benefit of our semi-skyline operator in relational preference query optimization. It 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. We conduct an extensive performance evaluation using both real and synthetic data sets. Our evaluation shows the enormous benefit of the semi-skyline operator in all three applications.

The remainder of this paper is organized as follows: In Section 2 we discuss the formal background used in this paper. In Section 3 we introduce our *semi-skylines* and the *Staircase* algorithm for its evaluation. Section 4 presents the *Staircase-Intersection* algorithm for very fast 2d-skyline computation. In Section 5 we present *skyline snippets* for skyline pieces and in Section 6 we show the benefit for multi-criteria query optimization. In Section 7 we discuss related work and Section 8 contains our concluding remarks.

2. FORMAL BACKGROUND

Preferences and their integration into databases have been in focus for some time, leading to diverse approaches, e.g. [11, 4]. We depict the preference model from [11] and look at a preference $P = (A, <_P)$, where A is a set of attributes and $<_P$ is a *strict partial order* on the domain of A. The term $x <_P y$ is interpreted as "I like y more than 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* in [4], *BMO*-set in [11]):

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

It finds all *best matching* tuples t for the preference P with $A \subseteq \text{attr}(R)$, where attr($R$) denotes all attributes of a relation R. If none exists, it delivers *best-matching alternatives*, but *nothing worse*.

An important subclass of preferences are *weak order preferences* (WOP, [12]), i.e. strict partial orders for which negative transitivity holds. For WOPs $P = (A, <_P)$ the dominance test can be efficiently done by a numerical *level*-function which depends on the type of preference, [22]:

$$level : dom(A) \to \mathbb{R}_0^+$$
$$x <_P y \iff level_P(x) > level_P(y)$$

For WOPs two domain values x and y having the same level are either equal or *indifferent*, i.e. $\neg(x <_P y) \wedge \neg(y <_P x)$. All domain values mapping to the same level value are considered as substitutable and are treated as one *equivalence class* (EC, regular SV-semantics, cp. [12]).

Preferences on single attributes are called *base preferences*. There are base preference constructors for continous 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 level function reflects the preference of elements of the POS-set over other values.

$$level_{POS}(x) := \begin{cases} 0 & \text{iff } x \in \textit{POS-set} \\ 1 & \text{iff } x \notin \textit{POS-set} \end{cases}$$

The sample query in Figure 1 shows two POS preferences with POS-sets for *Soup* and *Meat*. The generalization of the POS preference is *LAYERED*, introduced in [12]. It enables users to specify any number of different sets. Each set is preferred more or less than another set (thus has a unique integer level value). For all discrete base preferences, the maximum level value is settled by the definition of the preference constructor. For POS preferences, this value is 1.

Continuous numerical domains need a different type of preferences. We will use the advanced version of [12], allowing the partitioning of the range of domain values. For this purpose the so-called *d-value* ($d \geq 0$) was introduced. All numerical preference constructors are defined in [12], e.g. LOWEST, HIGHEST, AROUND, BETWEEN and the SCORE preference. To determine the level function for numerical preference constructors, we use the function *dist*, which has to be defined individually for every type of numerical base preference and is interpreted as the numerical distance from a perfect value:

$$level_P(x) := \begin{cases} dist(x) & \text{iff } d = 0 \\ \lceil dist(x)/d \rceil & \text{iff } d > 0 \end{cases}$$

The extremal preferences HIGHEST and LOWEST allow users to express easily their desire for values as high or as low as possible.

$\triangleright$ HIGHEST$_d(A)$: The distance function has to map higher inputs to lower function values. The best possible is the maximum value of the domain of A, max.

$$dist_{HIGHEST}(x) := max - x$$

$\triangleright$ LOWEST$_d(A)$: LOWEST$_d$ is the dual preference of HIGHEST. The best possible value is the minimum value of the domain of A, min.

$$dist_{LOWEST}(x) := x - min$$

The query in Figure 1 shows a HIGHEST preference for the amount of Vc in the *Beverage* products. It should be as high as possible; differences of up to $d = 5$ do not matter.

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**
For WOPs $P_i = (A_i, <_{P_i})$, $i = 1, ..., m$, a Pareto preference

$$P := \otimes(P_1, ..., P_m) = (A_1 \times ... \times A_m, <_P)$$

is defined as:

$$(x_1, ..., x_m) <_P (y_1, ..., y_m) \iff$$
$$\exists i \in \{1, 2, ..., m\} : level_{P_i}(x_i) > level_{P_i}(y_i) \ \wedge$$
$$\forall j \in \{1, 2, ..., m\},\ j \neq i : level_{P_j}(x_j) \geq level_{P_j}(y_j)$$

We restrict our attention to WOPs as input preferences for a Pareto preference P and consider tuples having the same level value as substitutable. In this sense, Pareto preference queries model the semantics of the traditional skyline queries.

3. SEMI-SKYLINES

In this section we introduce the *Semi-Pareto* preference to compute semi-skylines, focusing on the 2-dimensional case ($m = 2$ in Definition 1). At first sight the reader might find this not very exciting, but it will become clear later on and will surprisingly have very interesting applications.

3.1 Formal Background

Comparing the definition of Semi-Pareto to Pareto, it is evident that Semi-Pareto is the half of a Pareto preference for two preferences and therefore computes a 'semi-skyline'.

DEFINITION 2. **Semi-Pareto preference**
Let $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$ be WOPs.

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

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

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

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

The proof that Semi-Pareto is a preference, i.e. it is irreflexive and transitive, can be done straightforward.

PROOF. We present it for Left-Semi-Pareto, since Right-Semi-Pareto can be done analogously.

- irreflexive:

$$\begin{aligned}
&(x_1, x_2) <_{P_1 \ll\otimes P_2} (x_1, x_2) \\
&\iff \quad level_{P_1}(x_1) \geq level_{P_1}(x_1) \ \wedge \\
&\qquad\quad level_{P_2}(x_2) > level_{P_2}(x_2) \\
&\iff \quad false
\end{aligned}$$

- transitive:

$$\begin{aligned}
&x <_{P_1 \ll\otimes P_2} y \ \wedge \ y <_{P_1 \ll\otimes P_2} z \\
&\iff \quad [level_{P_1}(x_1) \geq level_{P_1}(y_1) \ \wedge \\
&\qquad\quad level_{P_2}(x_2) > level_{P_2}(y_2)] \ \wedge \\
&\qquad\quad [level_{P_1}(y_1) \geq level_{P_1}(z_1) \ \wedge \\
&\qquad\quad level_{P_2}(y_2) > level_{P_2}(z_2)] \\
&\implies \quad level_{P_1}(x_1) \geq level_{P_1}(z_1) \ \wedge \\
&\qquad\quad level_{P_2}(x_2) > level_{P_2}(z_2) \\
&\iff \quad x <_{P_1 \ll\otimes P_2} z
\end{aligned}$$

□

Similar to a Pareto preference, the Semi-Pareto constructor is not a WOP even if it consists of WOPs.

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

PROOF. We give a counterexample. Consider the sample data set from Table 1 holding a relation on beverages.

Given two preferences $P_1 = LOWEST(Cal)$ and $P_2 = LOWEST(Fat)$ and $P = P_1 \ll\otimes P_2$. Then 'B1' is indifferent to 'B5', since they are not comparable due to P. Furthermore, 'B5' is not comparable 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 in general. □

Table 1: A sample data set for beverages.

Beverages	*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

Therefore, a level-based domination between tuples is not possible [22].

So far, the apprehension arises that no better algorithm exists for evaluation than the well known block-nested-loop algorithm (BNL) from [3]. But *Staircase* comes to the rescue.

3.2 The Staircase Algorithm

We now introduce the *Staircase* (SC) algorithm for the evaluation of Semi-Pareto with guaranteed worst case complexity of $O(n \log n)$. SC is a variant of the BNL algorithm, but the candidate window will be a *SkipList* ([23]).

Have a look at the definition of the Left-Semi-Pareto operator from Definition 2 (analogously Right-Semi-Pareto), where P_1 and P_2 are arbitrary WOPs:

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

A tuple $x := (x_1, x_2)$ is worse than a tuple $y := (y_1, y_2)$, iff the level value is worse or equal in the first component, i.e. $level_{P_1}(x_1) \geq level_{P_1}(y_1)$ and worse in the second one, i.e. $level_{P_2}(x_2) > level_{P_2}(y_2)$. Since we map tuples (x_1, x_2) to equivalence classes represented by $(level_{P1}(x_1), level_{P_2}(x_2))$, we can state directly dominance using these equivalence classes. For a graphical interpretation have a look at Figure 2a. All equivalence classes in the pruning region PR, i.e. below and right of the equivalence class $[y] = (2, 2)$ are worse than itself, because their $level_{P_1}$ value is greater or equal to 2 and worse 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 $[x] = (level_{P1}(x_1), level_{P_2}(x_2))$ leads to the following possibilities:

a) If $[x]$ falls into the pruning region PR (Figure 2a) we directly can state dominance using the equivalence classes. For example consider $[x] = (3, 3)$. Since $3 \geq 2$ and $3 > 2$ 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 second preference.

b) If an equivalence class is left below of $[y]$ it is not dominated, but extends our staircase, cp. Figure 2b. 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.

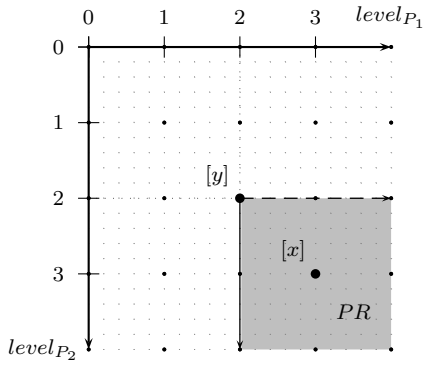

Figure 2a: Adding a worse equivalence class.

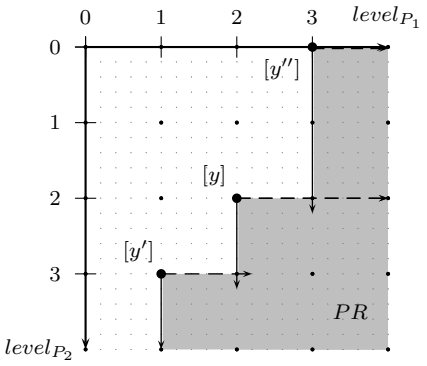

Figure 2b: Extending the staircase.

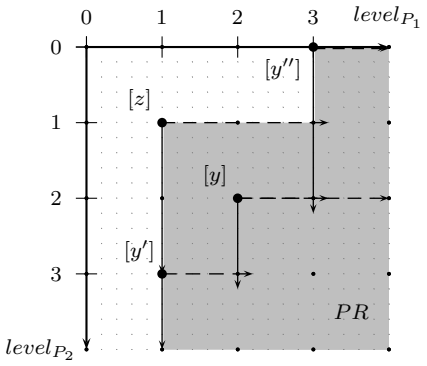

Figure 2c: Updating the staircase.

c) Only an equivalence class $[z]$ left above $[y]$ is better than $[y]$ and therefore dominates it, Figure 2c. 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 points have to be deleted. For example, consider $[z] = (1, 1)$ in Figure 2c. $[z]$ dominates $[y]$ and $[y']$ and therefore we have to delete these equivalence classes and change the staircase to its new form consisting of $\{[z] = (1, 1), [y''] = (3, 0)\}$.

In the worst-case of BNL ([3]) all tuples in the candidate window have to be compared to the new tuple to decide dominance. In contrast, 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 dominance criterion is only applicable if the equivalence classes on the staircase are comparable and ordered. Well, our staircase, i.e. the points on it, build a total order using the Manhattan distance [16].

DEFINITION 3. **Manhattan distance, L_1 norm**
The Manhattan distance d_1 for an equivalence class $[x] = (x_1, x_2)$ in our 2-dimensional staircase space is the distance from the *fixed* point $(0, max(level_{P_2}))$ to the point $[x]$. We denote $max(level_{P_2})$ as the maximum level value for P_2. Formally:

$$d_1([x]) = x_1 - x_2 + max(level_{P_2})$$

The Manhattan distance from Definition 3 can be also defined using $(max(level_{P_1}), 0)$ as origin. This would lead to a contrary staircase, but does not change anything in our algorithm.

LEMMA 2. ***Total Order of Staircase Points***
The points on the staircase build a total order concerning the Manhattan distance from Definition 3.

PROOF. We give the proof using a Left-Semi-Pareto preference $P := P_1 \lhd\otimes P_2$ 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 $d_1([y]) = y_1 - y_2 + max(level_{P_2})$ 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 $max(level_{P_2})$ 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 $x_i = y_i,\ i \in \{1, 2\}$, then 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 $[z] = (1, 1)$ and $[y''] = (3, 0)$ from Figure 2c. We get a distance $d_1([z]) = max(level_{P_2})$ and $d_1([y'']) = 3 + max(level_{P_2})$.

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 level values with the existing equivalence class and find out dominance (or add the tuple to the equivalence class if it has the same level 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 after 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 an alternative to binary trees and provied *insert*, *delete* and *search* in $O(\log n)$ ([23]). They 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.

A SkipList ([23]) is a collection of sorted linked lists, each at a given "level", that mimics the behavior of a search tree. The list at each level, other than the bottom level, is a sublist of the list at the level beneath it. Each node is assigned a random level, up to some maximum, and participates in the

lists up to that level. Figure 3 shows a SkipList with equivalence classes as keys ordered by the Manhattan distance (number below the classes, $max(level_{P_2}) = 10$). Among other points it contains the classes from Figure 2b.

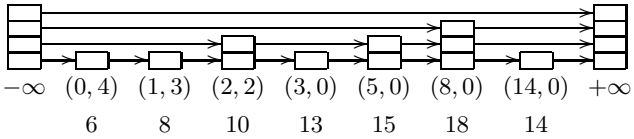

Figure 3: A SkipList with maximum level 4. The keys are equivalence classes ordered by the Manhattan distance (number at the bottom).

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 downward until a node with the desired key is found, or the bottom level is reached. Thus, the expected time complexity of a SkipList operations is logarithmic in the length of the list, cp. [23].

It is convenient to have left sentinel and right sentinel nodes, at the beginning and end of the lists respectively. These nodes have the maximum allowed level, and initially, when the SkipList is empty, the right sentinel is the successor of the left sentinel at every level. The left sentinels key is smaller, and the right sentinels key is greater, than any key that may be added to the set. Searching the skiplist thus always begins at the left sentinel.

For the complexity of a Semi-Pareto evaluation using the SC algorithm we can conclude the following theorem:

THEOREM 1. ***SC Complexity***
For n input tuples we get:

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

PROOF. For each input tuple we either have to insert or remove it from the staircase. These operations can be done in logarithmic time, i.e. we get a worst-case complexity of $n \cdot O(\log n)$. Since our SC algorithm is a specialized BNL algorithm, we can guarantee a best-case runtime of $O(n)$, cp. [7, 8]. □

We now show 3 interesting applications of Semi-Pareto.

4. VERY FAST 2-D-SKYLINES

In this section we introduce a novel method called *Staircase Intersection* (SCI) to evaluate 2-dimensional skylines. SCI significantly shows better performance than all known algorithms up to now.

4.1 Formal Background

The key to our very fast 2-d skyline computation is a theorem stated below. The proof of this theorem requires additional formal background. First, we want to define the *union* of preferences, which assembles a preference P from separate pieces $P_1, ..., P_n$ all acting on the same set of attributes.

DEFINITION 4. **Union preference:** $P_1 + P_2$
We assume preferences $P_1 = (A, <_{P_1})$ and $P_2 = (A, <_{P_2})$. Then $P = (A, <_{P_1+P_2})$ is called union preference, iff:

$$x <_{P_1+P_2} y \iff x <_{P_1} y \vee x <_{P_2} y$$

Using the union preference the following theorem holds:

THEOREM 2. ***Pareto decomposition***
Given preferences $P_1 = (A_1, <_{P_1})$ *and* $P_2 = (A_2, <_{P_2})$ *then:*

$$P_1 \otimes P_2 = (P_1 \lhd\otimes P_2) + (P_1 \otimes\rhd P_2)$$

PROOF. The proof follows directly from Defintion 1. □

Note, in general the union of preferences does not lead to a strict partial order. But if the preferences are disjoint a strict partial order is guaranteed ([11]). In our case, the union of the Semi-Paretos forms Pareto and therefore is a preference, even if the Semi-Pareto preferences are not disjoint. Furthermore, preference union can be evaluated by the intersection of the single preference selections.

THEOREM 3. $\sigma[P_1 + P_2](R) = \sigma[P_1](R) \cap \sigma[P_2](R)$
Given preferences $P_1 = (A_1, <_{P_1})$ *and* $P_2 = (A_2, <_{P_2})$ *with* $A \subseteq attr(R)$ *on a relation R. Then*

$$\sigma[P_1 + P_2](R) = \sigma[P_1](R) \cap \sigma[P_2](R)$$

PROOF. In [11] this has been shown already for disjoint preferences. It turns out that this also holds for non-disjoint preferences.

We define the set of *non-maximal values* for a database relation R and a preference $P(A, <_P)$ as

$$Nmax(P) := R[A] - max(P)$$

For $w \in R[A]$ we get:

$$\begin{aligned} & x \in Nmax(P_1 + P_2) \\ & \iff \exists y \in R[A] : x <_{P_1+P_2} y \\ & \iff \exists y \in R[A] : x <_{P_1} y \vee x <_{P_2} y \\ & \iff (\exists y \in R[A] : x <_{P_1} y) \vee (\exists y \in R[A] : w <_{P_2} y) \\ & \iff x \in Nmax(P_1) \vee x \in Nmax(P_2) \end{aligned}$$

Thus: $Nmax(P_1 + P_2) = Nmax(P_1) \cup Nmax(P_2)$
Then:

$$\begin{aligned} & \sigma[P_1 + P_2](R) \\ & = \{t \in R \mid t[A] \in max(P_1 + P_2)\} \\ & = \{t \in R \mid t[A] \in R[A] - Nmax(P_1 + P_2)\} \\ & = \{t \in R \mid t[A] \in R[A] - (Nmax(P_1) \cup Nmax(P_2))\} \\ & = \{t \in R \mid t[A] \in (R[A] - Nmax(P_1)) \cap (R[A] - Nmax(P_2))\} \\ & = \{t \in R \mid t[A] \in max(P_1) \cap max(P_2)\} \\ & = \sigma[P_1](R) \cap \sigma[P_2](R) \end{aligned}$$

□

Now we present the key to our very fast 2-d skylines.

THEOREM 4. ***Skyline by Semi-Pareto Intersection***
Given preferences $P_1 = (A_1, <_{P_1})$ *and* $P_2 = (A_2, <_{P_2})$ *with* $A_1, A_2 \subseteq attr(R)$ *on a relation R. Then*

$$\sigma[P_1 \otimes P_2](R) = \sigma[P_1 \lhd\otimes P_2](R) \cap \sigma[P_1 \otimes\rhd P_2](R)$$

PROOF. Using Pareto decomposition and theorem 3 we get:

$$\begin{aligned}\sigma[P_1 \otimes P_2](R) &= \sigma[(P_1 \lessdot\otimes P_2) + (P_1 \otimes\gtrdot P_2)](R) \\ &= \sigma[P_1 \lessdot\otimes P_2](R) \cap \sigma[P_1 \otimes\gtrdot P_2](R)\end{aligned}$$

□

This means, a 2-dimensional Pareto preference can be computed by the intersection of both Semi-Pareto preferences. Therefore, we call this method *Staircase-Intersection* (SCI). Since all equivalence classes in the staircase of a Semi-Pareto preference evaluation are ordered, the intersection of both Semi-Paretos can be done by sorted-based intersection in linear time ([1]). In particular, the parallel computation of the Left-Semi-Pareto and the Right-Semi-Pareto preferences enormously speeds up the Pareto evaluation and obviously can be applied on multi-core processor architectures.

EXAMPLE 1. *Consider two preferences on Table 1:* $P_1 = POS(B.Name, \text{'RedWine'})$, $P_2 = HIGHEST(Vc)$. *We want to compute* $\sigma[P_1 \otimes P_2](B)$. *Then*

- $\sigma[P_1 \lessdot\otimes P_2](B) = \{B2, B3, B4\}$
- $\sigma[P_1 \otimes\gtrdot P_2](B) = \{B1, B2, B3, B4, B5\}$

Using Theorem 4 we get

$$\begin{aligned}\sigma[P_1 \otimes P_2](R) &= \sigma[P_1 \lessdot\otimes P_2](R) \cap \sigma[P_1 \otimes\gtrdot P_2](R) \\ &= \{B2, B3, B4\}\end{aligned}$$

4.2 Performance Benchmarks

We now present results from an experimental study designed to compare the performance of the SCI algorithm with the best existing methods. The results of our extensive performance tests show the benefit of our SCI, in particular for high cardinality domains.

▷ Algorithms and Data Sets

We have implemented the following algorithms: our SCI, the *Hexagon* algorithm from [22] (also known as *Lattice skyline* in [19]) using the structure of the lattice imposed by the Pareto operator on the data space to identify the skyline. We also implemented *LESS* described in [7] which average-case running time is linear in the number of data points for fixed dimensionality. For 2-d skylines an algorithm using sorted data was proposed in [3] (*BNL2d*). If the data is sorted according to the two attributes of the Pareto clause, the test of whether a tuple is part of the skyline is very cheap: one simply needs to compare a tuple with its predecessor. More precisely, one needs to compare a tuple with the last previous tuple which is part of the skyline.

The reason for choosing these algorithms is as follows: The SCI algorithm is a Pareto evaluation technique that does not require an index for evaluation. All evaluated algorithms do not require preprocessing or an index to be pre constructed on the data, which makes them very appealing when the skyline operation is part of a complex query (for example computing a Pareto preference over a subset of the base relation or over a join condition). Of course, *Hexagon* works best for low-cardinality domains. Nevertheless, we compare it to our SCI to show the high performance even in low-cardinality domains. *LESS* is currently the best known general skyline algorithm for high-cardinalty domains.

All algorithms are implemented in our *Preference SQL* system ([9]), a Java SE 6 framework for preference queries on real database systems using a JDBC-connection. All experiments are performed on a 2.53GHz Core 2 Duo machine running Max OS X with 4 GB RAM used for the JVM. We used an Oracle 11g database to store all generated data. We used a buffer pool large enough for all operations to fit into main-memory for all tests. 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.

We use synthetic data sets, since these are commonly used for skyline evaluation. These data sets allow us to carefully explore the effect of various data characteristics. We generate the synthetic data sets with correlated (*COR*), independent (*IND*) and anti-correlated (*ANTI*) distributions using an implementation of the popular data set generator of [3]. We have modified the generator to generate data sets with attributes from high-cardinality domains with domain size c. We generate a number of synthetic data sets by varying three parameters: the data cardinality n, the number of distinct values for each high-cardinality attribute domain c and the d-value, which controls the number of values considered as substitutable. The data dimensionality is fixed to two. In all synthetic experiments, the tuple size is 100 bytes (also used in [7] in their experiments). A tuple has two attributes of type *Integer* and one *bulk* attribute with "garbage" characters to ensure that each tuple is 100 byte long.

▷ Results

We run four tests, each using correlated, independent and anti-correlated data, but different data cardinality n and domain cardinality c to represent the power of our SCI algorithm.

Test 1: Figures 7a through and 7c show runtimes for different distributions, containing $n = 10$ to 10^6 tuples. We fixed $c = 100K$, i.e. the domain contains 100.000 different values, and $d = 0$ to represent the conventional skyline queries. Obviously, SCI performs better than the competitors. Note that the axes are logarithmical scaled. The difference to the other algorithms using correlated or independent data is small. For example, consider 500K correlated tuples (Figure 7a). This leads to a runtime of 5 seconds for our SCI algorithm, whereas LESS takes about 8 seconds (nearly twice as SCI), and BNL2d exceeds 20 seconds to evaluate the query. Similar runtimes are presented in Figure 7b for an independent data distribution. For anti-correlated data we can achieve a very fast skyline retrieval. For example, SCI takes about 20 seconds to retrieve the skyline for a relation containing 10^6 tuples. The Bnl2d algorithm using sorted data takes about 50 seconds, whereas LESS needs about 3 minutes. The performance of generic skyline algorithms varies greatly depending on the underlying data distribution; specifically, the performance of these algorithms degrades if the distribution tends towards an anti-correlated distribution. Note that many skyline applications involve data sets that tend to be anti-correlated, see e.g. [19].

Test 2: In our second test series we increase the domain size to $c = 500$, fix $d = 0$, and have a look at different data cardinality n to see how our algorithm will scale for larger

domain sizes. The results can be found in the Figures 8a to 8c. Obviously our SCI algorithm is not affected at all by changing the domain size. As already seen in our first test, SCI is only a little bit faster than LESS or BNL2d for correlated and independent data, but for an anti-correlated data distribution it computes the skyline much more faster than its competitors.

Test 3: The third test series compares the runtimes for different domain cardinality n, but using $d = 10K$ to represent the influence of the d-value in preference queries to our SCI algorithm. Using this d-value the Hexagon algorithm from [22] can be applied, since we 'achive' low-cardinality. The d-parameter allows the partitioning of the range of domain values into equivalence classes, i.e. it controls the number of substitutable values. Figure 9a and Figure 9b show the runtimes of all presented algorithms for correlated and independent data sets. As one can see, even if Hexagon has a linear runtime, our SCI algorithm performs only a little bit worse. Using anti-correlated data (cp. Figure 9c), it is evident that SCI becomes worse with higher domain cardinality n, and Hexagon can use its advantage from the lattice structure imposed by the Pareto operator to identify the skyline.

Test 4: In this test we evaluated the influence of the d-value on the different algorithms. We fixed $n = 500K$ (this value is also used in [7]) and set $c = 500K$ for all type of data sets. We varied the d-value for our preference queries to control the number of substitutable values. This is similar to vary the domain cardinality. The result are shown in Figure 10a through 10c. For all data distributions our SCI algorithm shows its excellent performance until a d-parameter of $d = 500$, since lower d-values represent high-cardinality domains. From $d \geq 1000$ on, the Hexagon algorithm turns out to be better since we achieve 'low-cardinalty' for exploiting the lattice structure of a Pareto preference ([22]). Hexagon was not able to compute the results for $d < 50$, because the lattice structure did not fit into main memory for high-cardinality domains.

5. SKYLINE SNIPPETS

In many applications knowing only a piece of the skyline is sufficient. This problem has been addressed before by so called progressive skyline algorithms ([20]). However, all of them have to use predefined index structures. In this section we introduce the novel approach of *skyline snippets* for very fast retrieval of a skyline subset.

Since Pareto is associative and commutative ([11]), each m-dimensional Pareto preference can be k-partitioned. For example, consider $P = \otimes(P_1, P_2, P_3, P_4)$, $m = 4$ and $k = 2$. Then a few 2-partitions of $P = P^{[1]} \otimes P^{[2]}$ are:

- $P^{[1]} = P_1 \otimes P_2$ and $P^{[2]} = P_3 \otimes P_4$
- $P^{[1]} = P_1 \otimes P_3$ and $P^{[2]} = P_2 \otimes P_4$
- $P^{[1]} = P_1 \otimes P_4$ and $P^{[2]} = P_2 \otimes P_3$

$P^{[1]}$ and $P^{[2]}$ are called *partition preferences*, forming special subspace preferences ([21]). The number of k-partitions of an m-dimensional Pareto preference is given by the *Stirling* numbers of second kind $\left\{ {m \atop k} \right\}$ ([14]). For the example above we get $\left\{ {4 \atop 2} \right\} = 7$ partitions; 3 of them are shown above.

5.1 Formal Background

We give the formal background to our skyline snippets.

THEOREM 5. ***k-Snippets of a Skyline***
Consider a Pareto preference $P = \otimes(P_1, ..., P_m)$, its k-partition $\otimes(P^{[1]}, ..., P^{[k]})$ and the skyline $S = \sigma[P](R)$ on some relation $R = (A_1, ..., A_m)$.

a) Let $S_k = \bigcup_{j=1}^{k} \sigma[P^{[j]}](R)$, then

- $\sigma[P](S_k) \neq \emptyset$
- $\sigma[P](S_k) \subseteq S$

$\sigma[P](S_k)$ is called a k-snippet of the skyline S.

b) Let $L_k = \bigcap_{j=1}^{k} \sigma[P^{[j]}](R)$. If $L_k \neq \emptyset$, then $L_k = S$. L_k is called a lucky k-snippet of skyline S.

PROOF.

a) $\sigma[P](S_k) \neq \emptyset$ is obvious since preference selection never will be empty. We proove $\sigma[P](S_k) \subseteq S$.
Let $t = (a_1, ..., a_m, ...) \in \sigma[P](S_k)$, i.e.

$$\neg\exists t' = (a'_1, ..., a'_m, ...) \in S_k : (a_1, ..., a_m, ...) <_P (a'_1, ..., a'_m, ...)$$

Furthermore, $t \in S_k$, i.e.

$$\neg\exists t'' = (a''_1, ..., a''_m, ...) \in R : (\exists P^{[j]},\ 1 \leq j \leq k : (a_1, ..., a_m, ...) <_{P^{[j]}} (a''_1, ..., a''_m, ...))$$

It follows

$$\neg\exists t' = (a'_1, ..., a'_m, ...) \in R : (a_1, ..., a_m, ...) <_P (a'_1, ..., a'_m, ...) \Leftrightarrow t \in S = \sigma[P](R)$$

b) Let $t = (a_1, ..., a_m, ...) \in L_k$. Then

$$\neg\exists t' = (a'_1, ..., a'_m, ...) \in R : (a_1, ..., a_m, ...) <_{P^{[j]}} (a'_m, ..., a'_m, ...),\ \forall 1 \leq j \leq k \iff t \in S = \sigma[P](R)$$

□

We give a short example for a better understanding.

EXAMPLE 2. *Given preferences $P_1 = AROUND(A_1, 0)$ and $P_2 = AROUND(A_2, 0)$ and a relation R as in Table 2.*

Table 2: Sample data for relation R.

R	*ID*	A_1	A_2
	1	-2	2
	2	0	5
	3	-1	3
	4	2	2
	5	2	3
	6	0	4

The skyline S of the preference $P = P_1 \otimes P_2$ is

$$S := \pi_{ID}(\sigma[P](R)) = \{1, 3, 4, 6\}$$

There is one 2-partition with $P^{[1]} = P_1$ and $P^{[2]} = P_2$:

$$\pi_{ID}(\sigma[P^{[1]}](R)) = \{2, 6\} \quad and \quad \pi_{ID}(\sigma[P^{[2]}](R)) = \{1, 4\}$$

Thus, our 2-snippet has the following tuples:

$$\pi_{ID}(\sigma[P](\{2, 6\} \cup \{1, 4\})) = \{1, 4, 6\} \subset S$$

Now, assume that luckily a new tuple $t = (7, 0, 2)$ is inserted into R. Then we get $S := \pi_{ID}(\sigma[P](R)) = \{7\}$ and

$$\pi_{ID}(\sigma[P^{[1]}](R)) = \{2, 6, 7\} \quad and \quad \pi_{ID}(\sigma[P^{[2]}](R)) = \{1, 4, 7\}$$

Thus $L_2 = \{2, 6, 7\} \cap \{1, 4, 7\} = \{7\} \neq \emptyset$. Therefore we can conclude: $S = L_2 = \{7\}$. Sometimes tuples like this are called "killer tuples" in the literature.

5.2 Equal-sized $\frac{m}{2}$-Snippets

Since for the purpose of this paper we want to promote the power of the Semi-Pareto preference and our SCI algorithm, we now present a case study using $\frac{m}{2}$-snippets such that every partition is of dimension 2, hence can be evaluated using our novel semi-skyline algorithm. Assume m is even. We want to partition m into $\frac{m}{2} = n$ pieces, however each piece is supposed to have cardinality 2. Due to this extra constraint we can not use $\left\{ {m \atop m/2} \right\}$ as the number of partitions, since $\left\{ {m \atop m/2} \right\}$ also accounts for unequal cardinalities amongst the partitions.

LEMMA 3. *The number N of different equal-sized $\frac{m}{2}$-snippets of a Pareto preference $\otimes(P_1, ..., P_m)$ is given by*

- $N(1) = N(0) = 1$
- $N(m) = (m - 1) \cdot N(m - 2),\ m > 1$

PROOF. We proof it by induction on m.

- Induction start:

  $N(1) = 1$ is clear, $N(0) = 0$ by definition.

- Induction step:

  Assume we know $N(m - 2),\ m \geq 2$. Proceeding to the next even number, let us consider $m - 2 + 2 = m$ The newly added attributes A_{m-1} and A_m can be combined with the $N(m - 2)$ possibilities as follows:

  a) (A_{m-1}, A_m) form a new 2-partition, which can be combined with $N(m - 2)$ possibilities yielding $N(m - 2)$ possibilities.

  b) Assume A_{m-1} is combined with $A_1, ..., A_{m-2}$, respectively, to form a 2-partition. Each one can be combined with $N(m - 2)$ possibilities, where A_m replaces $A_1, ..., A_{m-2}$, respectively. In total this leads to $(m - 2) \cdot N(m - 2)$ possibilities.

  Since no other cases can occur, we get

  $N(m) = N(m-2)+(m-2)\cdot N(m-2) = (m-1)\cdot N(m-2)$

□

In our introductory example we have $N(4) = 3$ possible 2-snippets, exactly those mentioned in this example. For 6 preferences combined by Pareto we get $N(6) = 15$ and for 8 preferences we reach $N(8) = 105$ possible snippets.

Note, in the case of $m = 2n + 1, n \in \mathbb{N}$, we generate n partitions for a very fast evaluation using our SCI algorithm and compute the results of the remaining preference using a level-based algorithm in linear time.

5.3 Performance Benchmarks

For the performance benchmarks of our skyline snippets we use the same set-up as described in Section 4.2. We varied the number of dimensions *dim* from 2 to 10. The data cardinality is fixed to $n = 500K$ as in [7] and the domain cardinality is fixed to $c = 100K$. We run several tests and present two of them with different d-values. We run tests for independent, correlated and anti-correlated data. We choose BNL, LESS and Hexagon (if applicable) for comparison, because these algorithms are the best competitors.

Test 1: For the first test series we set the d-value to $d = 0$ to represent the conventional skyline queries. The results are shown in the Figures 11a, 11b and 11c. Since $d = 0$ represents a high-cardinality domain, the algorithm Hexagon can not be applied (the lattice structure does not fit into main memory). As one can see, the computation of some skyline points can be done extremely fast with our skyline-snippets method, profiting from parallel computations of the 2-partitions. Although, our skyline snippets method does not compute the complete skyline, it returns enough skyline points for a user in real-world applications as can be seen in the case of anti-correlated data, cp. Figure 11c. For $dim = 6$ LESS needs more than 2 minutes to compute the whole skyline (14024 points), whereas our snippets method gets 342 points in about 10 seconds. Considering 10 dimensions, we have a runtime for LESS of 20 minutes (about 33.000 skyline points). Our snippets method computes 719 skyline points in less than 20 seconds. Table 3 shows the number of skyline points found by LESS (the complete skyline) and our snippets algorithm.

Table 3: #Points in the skyline and the partitions.

dim	*LESS*	$\sigma[P](S_k)$	$P^{[1]}$	$P^{[2]}$	$P^{[3]}$	$P^{[4]}$	$P^{[5]}$	P^s
2	619	619	619	-	-	-	-	-
3	1174	278	291	-	-	-	-	84
4	2808	311	266	278	-	-	-	-
5	5495	327	312	319	-	-	-	78
6	14024	342	378	311	299	-	-	-
7	18112	456	489	412	401	-	-	112
8	20333	586	411	419	458	403	-	-
9	26851	658	379	441	390	471	-	98
10	32973	719	480	429	369	391	451	-

For comparison, we also show the number of skyline points found by each partition $P^{[j]}, j = 1, ..., k$, of dimension 2. For each dimension we generated new test data. In the case of an odd number of preferences we also present the result size of the single preference selection P^s. As in Theorem 5 we write $\sigma[P](S_k)$ for the result of our snippets algorithm, where $S_k = \bigcup_{j=1}^{k} \sigma[P^{[j]}](R)$. For dimension $d = 2$ our snippets algorithm always finds all skyline points because the union of the both Semi-Pareto selections also contains the

intersection of them (cp. Theorem 4). Therefore the complete skyline will be computed. Furthermore, we observe that the result sizes of the different partitions are nearly the same for each dimension.

Test 2: Our second test series demonstrates the influence of the d-parameter to our snippets algorithm. As above, we fixed the data cardinality to $n = 500K$ and the domain size to $c = 100K$. Furthermore, we varied the number of dimensions from 2 to 10 and fixed the d-value to $d = 10K$. The results are represented in the Figures 12a, 12b and 12c. Interestingly, for correlated data (Figure 12a) the Hexagon algorithm from [22] nearly performs as good as our snippets algorithm. The reason is the worse reduction from the equal-sized $\frac{m}{2}$ snippets, this means the union of all snippets nearly contains as much tuples as the complete data set. The second reason is the high d-value. The Hexagon algorithm gains its speed from the lattice structure induced by a Pareto preference using a high d-value. Using independent or anti-correlated data sets, the snippets algorithm shows particularly its high performance in higher dimensions. LESS and Hexagon show nearly the same runtime as our skyline snippets.

The reader might ask why we do not compute each single preference separately, i.e. use a m-partition and compute $\sigma[P](\bigcup_{j=1}^{m} \sigma[P^{[j]}](R))$. Our experiments outline that the runtime is nearly the same, but the number of skyline points is much less than using $\frac{m}{2}$-partitions. For example, computing each preference separately for 4 dimensions we get 84 skyline points (anti-correlated data set). Using 2-partitions we have 311 skyline points [6]. On the other hand, using less partitions leads to more skyline points, but to longer computation times, too.

In many applications it is sufficient to know only a piece of the skyline. If all points are necessary the complete skyline has to be computed requiring much more time.

6. MULTI-CRITERIA OPTIMIZATION

The optimization of queries with multiple preferences and hard constraints is essential to support fast result computation ([4, 9]). The query in Figure 1 is a query containing hard constraints 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. As in Figure 1 the hard constraints refer to attributes from more than two relations, pair-wise join operators cannot test the satisfiability of an intermediate tuple until all variables have been determined. The query evaluation process must evaluate the cartesian product of all tuples of all join relations, which leads to *high memory and computation costs*, particularly for large relations.

Firstly introduced as Cutoff preference constructor ([5]), our Semi-Pareto preference provides an optimization technique for preference queries in combination with hard constraints over several relations. It allows us to 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. Since the workshop paper [5] was a first approach and only covers the fundamentals of the Semi-Pareto preference, we now complete the theoretical background for the developed optimization techniques.

6.1 Formal Background

The basis of our preference optimization technique is the following theorem published in [9, 4]:

THEOREM 6. ***Push Preference over Hard Selection*** *For a preference $P = (A, <_P)$ with $A \subseteq attr(R)$ and a hard constraint H the following holds:*

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

If a tuple w is dominated by a tuple v, a tuple $v' \in \sigma_H(R)$ must exists which dominates w. This guarantees the reduction of a tuple w only if it is dominated transitive by a tuple from $\sigma_H(R)$. Hence, a commutation due to Theorem 6 is possible, if for each dominating tuple $v[A]$ the condition $H(v)$ is fulfilled. We now define the *prefilter preference*:

DEFINITION 5. **Prefilter Preference**
A preference $Q = (Q, <_Q)$ is a *prefilter preference* for a preference $P = (P, <_P)$ w.r.t. a relation R iff

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

For the rest of this section we look at database relations $R = (A_1, ..., A_l, B_1, ..., B_k)$ 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 and $c \in \mathbb{R}$ a constant. An example of such a query is given in the introduction, see Figure 1. For our optimization techniques we need the idea of an *induced preference*.

DEFINITION 6. **Induced Preference**
Given a database relation R and a hard constraint H as above: $H = h(b_1, ..., b_k) \ \Theta \ c, \quad c \in \mathbb{R} \ \Theta \in \{\leq, <, >, \geq, =, \neq\}$.

a) If $\Theta \in \{<, \leq\}$ and h is monotone in b_i, then:
$H_i := LOWEST(b_i)$ is called *induced preference*.

b) If $\Theta \in \{>, \geq\}$ and h is monotone in b_i, then:
$H_i := HIGHEST(b_i)$ is called *induced preference*.

c) If $\Theta \in \{=, \neq\}$ and h is monotone, then:
$H_i := ANTICHAIN(b_i)$ is called *induced preference*. The $ANTICHAIN$ preference on the attribute b_i returns all elements of the input relation ([11]).

Using Definition 6 we get:

THEOREM 7. ***Push Semi-Pareto over Hard Sel.*** *Given a preference query with hard constraints H on a relation R, $P = (A, <_P)$ a preference. Then*

a) $\sigma[P \ll\!\otimes H_i](\sigma_H(R)) = \sigma_H(\sigma[P \ll\!\otimes H_i](R))$

b) $\sigma[P \otimes\!\gg H_i](\sigma_H(R)) = \sigma_H(\sigma[P \otimes\!\gg H_i](R))$

where H_i is an induced preference from Definition 6.

PROOF. We only prove a) since b) can be done analog. Consider a relation $R = (A_1, ..., A_k, B_1, ..., B_k)$, B_i numerical attributes and tuples $t = (a_1, ..., a_j, ..., b_i, ...)$ and $t = (a'_1, ..., a'_j, ..., b'_i, ...)$. For all $t, t' \in R$ assume

$$t <_{P \otimes H_i} t' \ \wedge \ H(t) \overset{Def. \otimes}{\iff} t <_{H_i} t' \ \wedge \ t <_P t' \ \wedge \ H(t)$$

Since t fulfills the hard constraint H and is worse than t' concerning the induced preference, it follows that also t' fulfills the hard constraint, i.e. $H(t')$ is valid. Therefore we can apply Theorem 6 and push the preference over the hard constraint.

□

If P is an induced preference, i.e. LOWEST, HIGHEST or ANTICHAIN we can simplify Theorem 7 as follows:

COROLLARY 1. ***Special Cases***
Consider a preference query where H_i is an induced preference, then:

$$\sigma[H_i](\sigma_H(R)) = \sigma_H(\sigma[H_i](R))$$

PROOF. $H_i = H_i \otimes H_i$ □

THEOREM 8. ***Prefilter Preference for*** $P_1 + P_2$
Let $P_1 = (A, <_{P_1})$, $P_2 = (A, <_{P_2})$ and $P = P_1 + P_2$ be preferences. Then P_1 and P_2 are prefilter preferences of P.

$$\sigma[P](R) = \sigma[P](\sigma[P_1](R)) = \sigma[P](\sigma[P_2](R))$$

PROOF. The first step of the following equation follows from [12, 4].

$$\begin{aligned} \sigma[P_1 + P_2](\sigma[P_1](R)) = & \ \sigma[P_1](\sigma[P_1](R)) \cap \sigma[P_2](\sigma[P_1](R)) \\ = & \ \sigma[P_1](R) \cap \sigma[P_2](\sigma[P_1](R)) \\ = & \ \sigma[P_2](\sigma[P_1](R)) \end{aligned}$$

Thus, a result tuple t has to be P_1-maximal w.r.t. R and P_2-maximal w.r.t. $\sigma[P_1](R)$. On the other hand, consider

$$\sigma[P_1](R) \cap \sigma[P_2](R)$$

A result tuple t has to be P_1-maximal w.r.t. R and P_2-maximal w.r.t. R as well. A result tuple t that is P_2-maximal in R is P_2-maximal w.r.t. $\sigma[P_1](R)$, too. For the opposite, by contradiction let's assume that t is P_2-maximal in $\sigma[P_1](R)$, but not in R:

$$\neg \exists v \in \sigma[P_1](R) : t <_{P_2} v$$

but

$$\exists v' \in R : t <_{P_2} v'$$

Since v' is not in $\sigma[P_1](R)$, it is P_1-dominated by t. However, for $P = P_1 + P_2$ this would contradict irreflexivity. Consequently:

$$\sigma[P_2](\sigma[P_1](R)) = \sigma[P_1](R) \cap \sigma[P_2](R) = \sigma[P_1 + P_2](R)$$

□

We now formulate a prefilter preference for Pareto.

THEOREM 9. ***Pareto Prefilter Preference***
Consider a Pareto preference query $\sigma[P](\sigma_H(R))$ with a hard constraint H on a relation R, $P := \otimes(P_1, ..., P_m)$. Then $P_i \otimes H_i$, $i \in \{1, ..., m\}$ is a prefilter preference for P with H_i an induced preference as in Definition 6 and it follows

$$\sigma[P](\sigma_H(R)) = \sigma[P](\sigma_H(\sigma[P_i \otimes H_i](R)))$$

PROOF. Let $t := (a_1, ..., a_i, ..., a_m)$ and $t' := (a'_1, ..., a'_i, ..., a_m)$ two tuples in $R := (A_1, ..., A_m)$ which only differ in a and a' and

$$level_{P_i}(t) > level_{P_i}(t') \ \wedge \ level_{H_i}(t) \geq level_{H_i}('t)$$

Then, it is evident that:

1) if the hard constraint H fails for t', also the hard constraint fails for t, since $level_{P_i}(t) \geq level_{P_i}('t)$ (H is monotone). Therefore t is not an element of the solution.

2) if t' fulfills the hard constraint, then

   a) if t fails the hard constraint, then t is not an element of the solution.

   b) if t also fulfills the hard constraint, then we know $level_{P_i}(t) > level_{P_i}(t')$, i.e. tuple t' is preferred to t. It follows from the Pareto preference $P := \otimes(P_1, ..., P_m)$ that t' is preferred over t since t' is preferred w.r.t P_i and all others are equal.

□

This theorem allows us to push the induced prefilter preference "down to the relation", e.g. over an cartesian product applying the rule "*push preference*" from [9].

EXAMPLE 3. *Revisit the preference query from Figure 1 with preference $P_1 = POS(S.Name, \{'Chicken', 'Noodle'\})$, $P_2 = POS(M.Name, 'Beef')$ and $P_3 = HIGHEST_5(B.Vc)$ combined to a Pareto preference $P = P_1 \otimes P_2 \otimes P_3$. Furthermore, consider the sum of calories (Cal) that must be less or equal to 1100 kcal as hard constraint H. Since Θ is "$\leq$" our induced preference are*

- $H_S = LOWEST(S.Cal)$ *for Soup*
- $H_M = LOWEST(M.Cal)$ *for Meat*
- $H_B = LOWEST(B.Cal)$ *for Beverage*

Using Theorem 9 we can insert the induced preferences and push them over the hard constraint. Since the prefilter preferences H_S, H_M and H_B only have preferences on attributes of Soup Meat and Beverage, respectively, we can push these prefilter preferences over the cartesian product as introduced by [9] and get the optimized operator tree in Figure 4.

The insertion of induced prefilter preferences leads to an elimination of tuples from the relations before building the join and hence reduces memory and computation costs for the cartesian product. For fast evaluation semi-skylines and our staircase algorithm can be used.

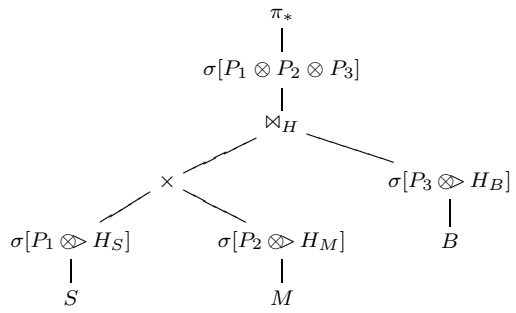

Figure 4: Operator tree with induced preference.

6.2 Performance Benchmarks

In order to evaluate the Pareto prefilter preference, we performed several experiments. For this we integrated the optimization rule from Theorem 9 into the preference query optimizer of Preference SQL ([9]). We used a real-world food database published by the United States Department of Agriculture (USDA, http://www.nal.usda.gov/fnic/), and synthetic data sets to explore the effect of various data characteristics.

For the evaluation of the Semi-Pareto preferences induced by the prefilter preference described in theorem 9 we used our SC algorithm from Section 4. The evaluation of the remaining Pareto preference was done by LESS. We evaluated the efficiency of our Pareto prefilter preference (abbr. *pref-prefilter*) by comparing the response times of several preference queries with hard constraints to an unoptimized evaluation (abbr. *no-opt*).

Test 1: The first test is based on the query in Figure 1 and contains three constraints. We used the real-world-database from the USDA. This database contains nutritional facts for more than 7000 types of food. 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 500 soups, 680 meats, and 350 beverages available, i.e. about 120 Mio. possible combinations. 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, varying the parameter *max_cal* varies 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 13. Since the Pareto 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 optimizer 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. In all our tests with different *Cal*, *Vc* and *Fat* it performed out, that our Pareto prefilter preference performs best for all, since it eliminates tuples before building the cartesian product and the join, respectively.

Test 2: In our second test we run the query from Figure 1 with different relation sizes (but fixed max_cal = 1100) on our USDA database and demonstrate the performance results in Figure 14. 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. Since the join up to 1 Mio. tuples can be done very fast, the runtimes for the standard approach without optimizer and the prefilter evaluation are almost the same. With the beginning of larger relations sizes, the prefilter preference reduces the number of tuples for each relation and therefore speeds up the computation of the cartesian product and the join, respectively.

Test 3: The third tests demonstrates our induced prefilter preference on one relation with one single hard constraint and a base preference (P=HIGHEST$_{d=0}$(B)) on it. We consider the following query on a relation $R(A, B)$, where A, B are numerical attributes.

```
SELECT *
FROM  R
WHERE A ≤ 150
PREFERRING B HIGHEST
```

Figure 5: Preference SQL query on one relation

The query expresses the wish for B as high as possible, but A must be less than 150. Without the induced preference the HIGHEST$_{d=0}$(B) preference will be computed after the evaluation of the hard selection. Since $A \leq 150$ filters the relation R, the assumption arise that the insertion of the prefilter preference and pushing it over the hard selection will not speed up the computation, since the prefilter preference is not such a strong criteria as the hard selection. This assumption is verified in Figure 15. In this test we use an anti-correlated data distribution and vary the domain cardinality n from 10 to 10^6 tuples. We set the domain size to $c = 500K$ as in [7]. Furthermore, we fixed the constant for the hard selection to $const = 150$ as in the query above. The computation runtimes without the prefilter preference are much better, because using the prefilter preferences implies the evaluation of it self, the evaluation of the hard selection and the computation of the original preference HIGHEST$_{d=0}$(B). In contrast, without using the prefilter preference only the hard selection and the original preference HIGHEST$_{d=0}$(B) must be evaluated, whereas the hard selection is a much better filter than the prefilter preference.

Test 4: In our last tests series we evaluated the influence of various data distributions to our prefilter preference. We generated synthetic data sets with correlated (COR), independent (IND) and anti-correlated ($ANTI$) distributions. We used three relations $R_i(A_i, B_i)$, $i = 1, 2, 3$, A_i, B_i numerical attributes, each with 500 tuples, i.e. a total of 125 Mio. possible combinations for the join. The join condition is set to the sum of the different attributes, each belonging to another relation, i.e. we select $A_1 + A_2 + A_3 \leq const$. We varied the *const* value to present the behavior of the prefilter preference depending on the data distribution. Since we generated the numerical data for the attributes A_i, B_i with a maximum value of 100, the sum can not exceed $const = 300$ as it is represented in the results. We use a Pareto preference expressing the wish for lowest B_i values leading to the

following query:

```
SELECT *
FROM  R1, R2, R3
WHERE R1.A + R2.A + R3.A ≤ const
PREFERRING
     R1.B LOWEST AND
     R2.B LOWEST AND
     R3.B LOWEST
```

Figure 6: Preference SQL query for different data distributions

The results are shown in the Figures 16a, 16b and 16c. In all tests it performance out, that the induced prefilter preference enormously speeds up the evaluation of the query in contrast to the non optimized one. For correlated (Figure 16a) and independent (Figure 16b) data the pref-prefilter method takes less than 1 sec to evaluate the query, since the prefilter preference reduces the single relations to less than 10 tuples each, i.e. only 1000 possible combinations for the join, instead of 125 Mio. combinations. For the anti-correlated case (Figure 16c) the prefilter preference reduces the relations to about 80 tuples each, this means about 500K combinations, much less than the original cartesian product.

From our experimental results we observe that the proposed prefilter preference improves the query evaluation consistently for different types of multi-criteria queries, also known as constrained skyline queries ([24]).

7. RELATED WORK

The evaluation of Pareto preference queries and finding the skyline is a generalization of finding the best elements of a set of multi-dimensional values. This was brought in a database context with the skyline operator in [3].

Nested-loop algorithms are the generic way of computing a skyline [17]. Despite lower performance compared to index algorithms, they are capable of processing arbitrary data without any preparations necessary.

One well-known generic algorithm is LESS ([7, 8]). It uses combined sorting and discarding in linear time, but is bound to specific features of the input tuples, in particular un-correlated and uniformly distributed data with mostly distinct values. Nevertheless, it shows quadratic worst case performance. Recently, Hexagon [22] and Lattice skyline [19] have been developed independently, both with linear worst case complexity for any data distribution, but only for low cardinality domains. They gain its speed from the construction and analysis of the underlying lattice structure. A 2-d skyline can be computed by sorting the data ([3]). If the data is sorted according to the two attributes a simple comparison of a tuple with its predecessor states dominance. However, sorting is combined with high costs and this method only works for numerical data, i.e. is not applicable for categorical preference queries. Techniques to evaluate skylines in subspaces have been analyzed in [21]. Some methods compute skylines for every subspace, and interestingly, the studies suggest that a top-down depth-first search framework may favor efficient computation.

Multi-criteria query optimization with preferences has not been intensively researched in the last years. In [10] algorithms have been developed for top-k queries that can be extended to implement queries with a constraint on the value of a monotone function, but this is only valid for one constraint. [18] integrated constraint-programming techniques with traditional database techniques to solve sum constraint queries by modifiying existing nested loop-join operators, but not in combination with preferences.

The paradigm of our algorithms is the decomposition of the skyline into separate semi-skylines for parallel computation. To the best of our knowledge, semi-skylines and skyline snippets have not been studied until now.

8. SUMMARY AND OUTLOOK

In this paper we have proposed the novel concepts of semi-skylines and skyline snippets. Semi-skylines enable a new view on well researched research areas. For semi-skylines we have provided the highly efficient *Staircase* algorithm, employing SkipLists for on-the-fly dominance testing with a worst-case complexity of $O(n \log n)$. Then we showed how 2-d skylines can be computed very fast by the intersection of semi-skylines, beating other competing algorithms significantly. As a completely different setting we have demonstrated how semi-skylines can be used effectively in algebraic preference query optimization. Semi-skylines can also be used effectively for the computation of skyline snippets, which are the second novel contribution in this paper. Our approach relies on partitioning a m-dimensional skyline into k subspaces. Initial benchmarks demonstrate the enormous benefit of skyline snippets, which compute skyline points on multi-core architectures without any index support.

At this point there are several interesting open research questions. Naturally one might ask whether the *Staircase-Intersection* algorithm for 2-d semi-skylines can be generalized for higher dimensions. Our current opinion is that this attempt will probably not be successful. We also want an external algorithm that can handle input sets that are too large for main-memory. Certainly, skyline snippets deserve additional investigations. Instead of partitioning an m-dimensional skyline into very fast 2-d subspaces, exploring other ways of partitioning might be interesting as well. In addition, studying the relationships between existing progressive skyline algorithms using indexes with our skyline snippets approach sounds challenging.

9. REFERENCES

[1] R. A. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. In *CPM '04: Proceedings of Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer, 2004.

[2] C. Böhm and H.-P. Kriegel. Determining the Convex Hull in Large Multidimensional Databases. In *DaWaK '01: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, volume 2114/2001, pages 294–306, London, UK, 2001. Springer.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[4] J. Chomicki. Preference Formulas in Relational Queries. In *TODS '03: ACM Transactions on Database Systems*, volume 28, pages 427–466, New York, NY, USA, 2003. ACM Press.

[5] M. Endres and W. Kießling. 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)*, pages 25–32, 2008.

[6] M. Endres and W. Kießling. Semi-Skylines and Skyline Snippets. Technical Report 2010-1, Institute of Computer Science, University of Augsburg, 2010.

[7] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 229–240. VLDB Endowment, 2005.

[8] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and Analyses for Maximal Vector Computation. *The VLDB Journal*, 16(1):5–28, 2007.

[9] B. Hafenrichter and W. Kießling. Optimization of Relational Preference Queries. In *ADC '05: Proceedings of the 16th Australasian database conference*, pages 175–184, Darlinghurst, Australia, 2005. Australian Computer Society, Inc.

[10] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 754–765. VLDB Endowment, 2003.

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

[12] W. Kießling. Preference Queries with SV-Semantics. In J. R. Haritsa and T. M. Vijayaraman, editors, *COMAD '05: Advances in Data Management 2005, Proceedings of the 11th International Conference on Management of Data*, pages 15–26, Goa, India, 2005. Computer Society of India.

[13] W. Kießling and G. Köstler. Preference SQL - Design, Implementation, Experiences. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 990–1001, Hong Kong, China, 2002. VLDB Endowment.

[14] D. E. Knuth. *The art of computer programming, Vol. I-III*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[15] D. Kossmann, F. Ramsak, and S. Rost. 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*, pages 275–286. VLDB Endowment, 2002.

[16] E. F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Courier Dover Publications, 1987.

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

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

[19] M. Morse, J. M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 267–278. VLDB Endowment, 2007.

[20] D. Papadias, Y. Tao, G. Fu, and B. 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, New York, NY, USA, 2003. ACM.

[21] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, and Q. Liu. Towards multidimensional subspace skyline analysis. *ACM Trans. Database Syst.*, 31(4):1335–5915, 2006.

[22] T. Preisinger and W. Kießling. 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)*, 2007.

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

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

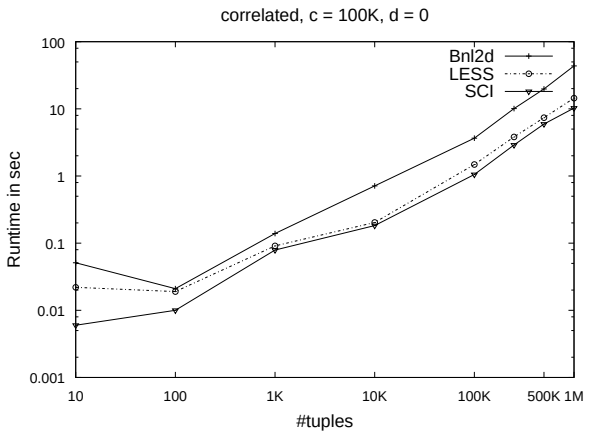

Figure 7a: Semi-Skylines: Test 1, COR

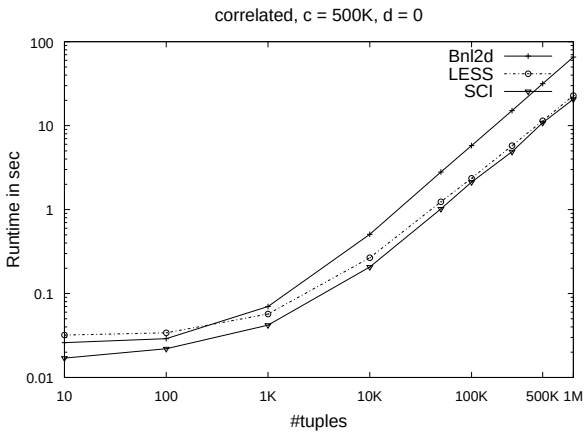

Figure 8a: Semi-Skylines: Test 2, COR

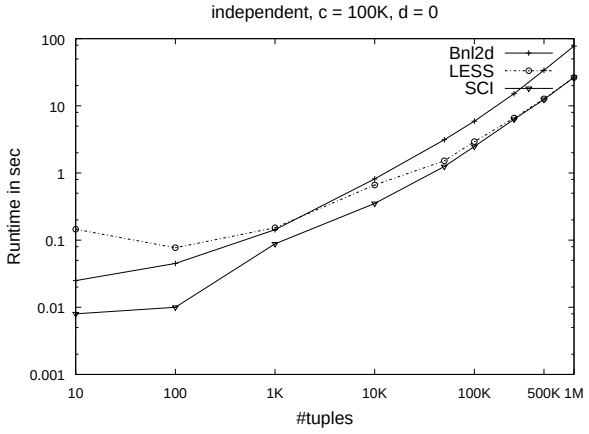

Figure 7b: Semi-Skylines: Test 1, IND

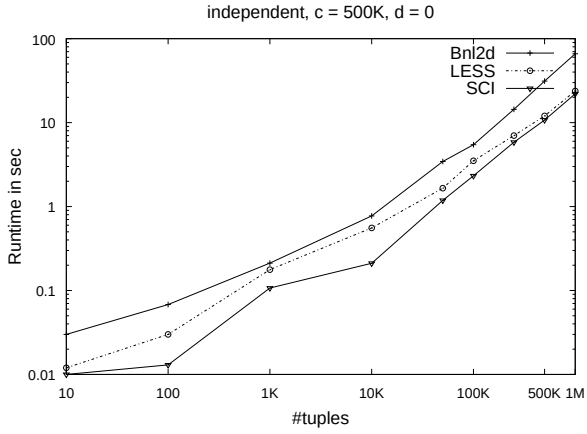

Figure 8b: Semi-Skylines: Test 2, IND

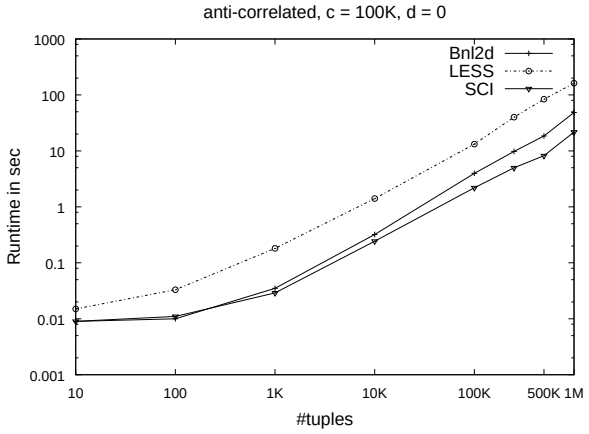

Figure 7c: Semi-Skylines: Test 1, ANTI

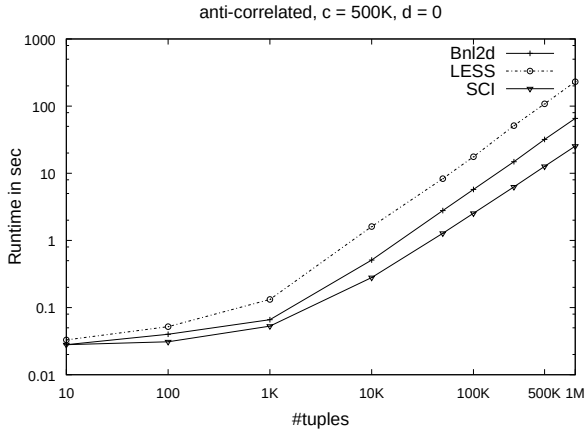

Figure 8c: Semi-Skylines: Test 2, ANTI

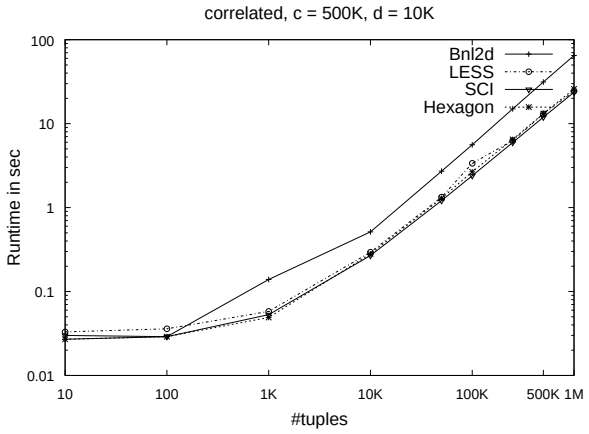

Figure 9a: Semi-Skylines: Test 3, COR

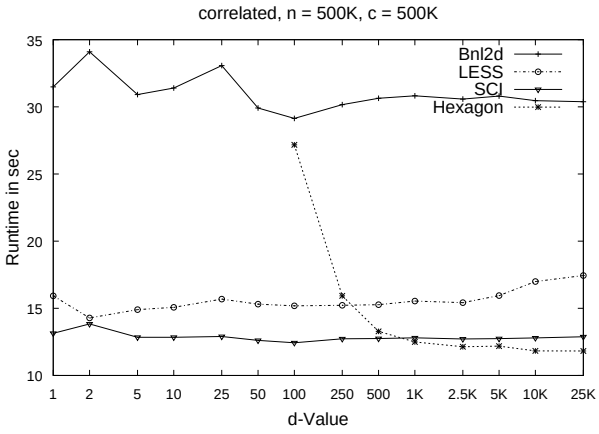

Figure 10a: Semi-Skylines: Test 4, COR

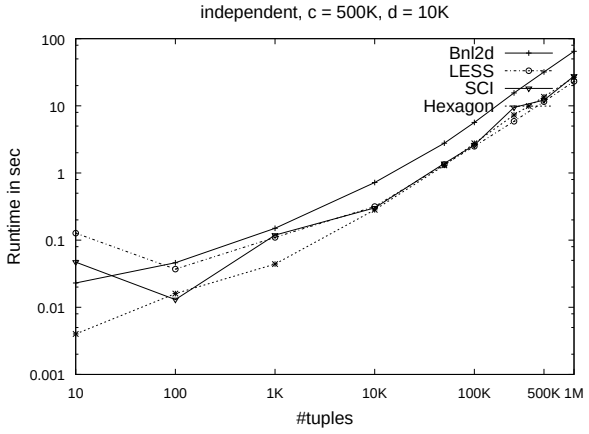

Figure 9b: Semi-Skylines: Test 3, IND

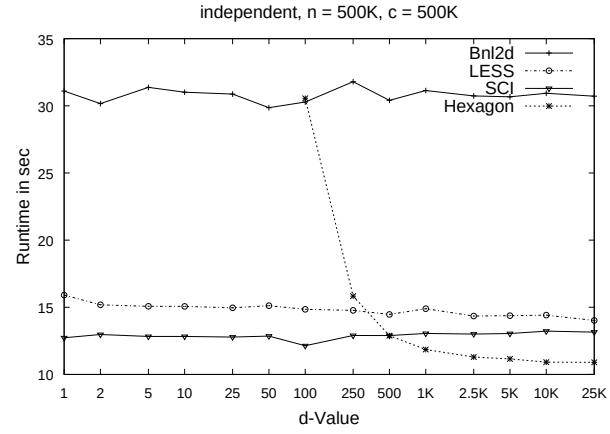

Figure 10b: Semi-Skylines: Test 4, IND

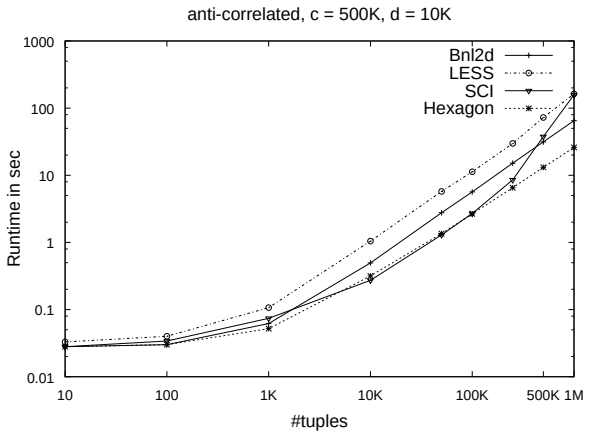

Figure 9c: Semi-Skylines: Test 3, ANTI

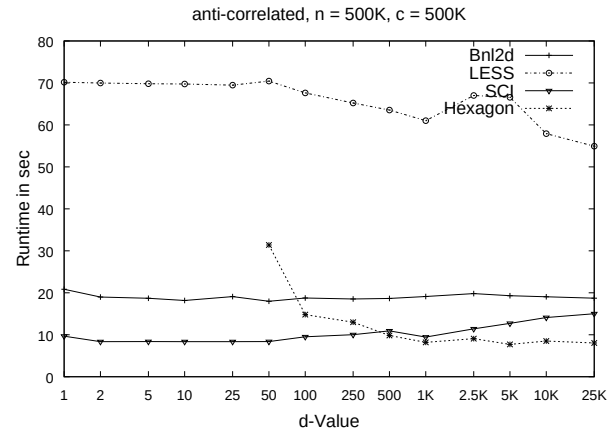

Figure 10c: Semi-Skylines: Test 4, ANTI

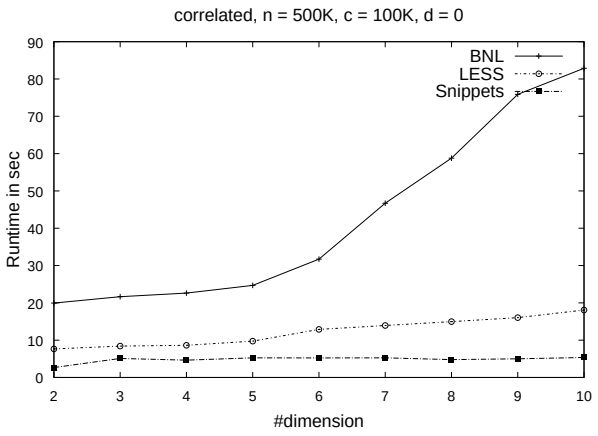

Figure 11a: Skyline-Snippets: Test 1, COR

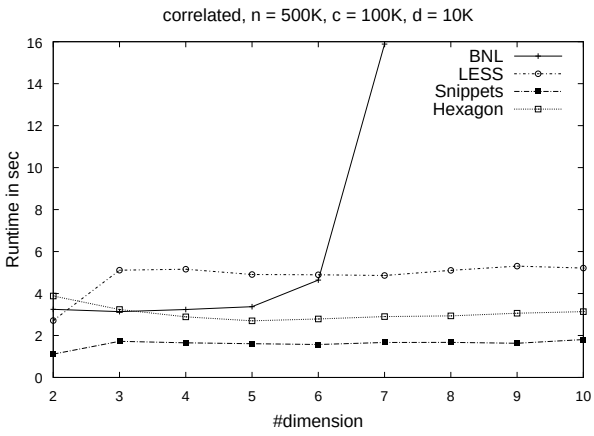

Figure 12a: Skyline-Snippets: Test 2, COR

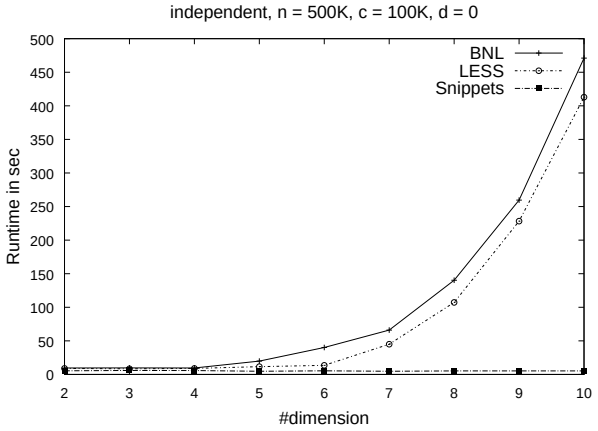

Figure 11b: Skyline-Snippets: Test 1, IND

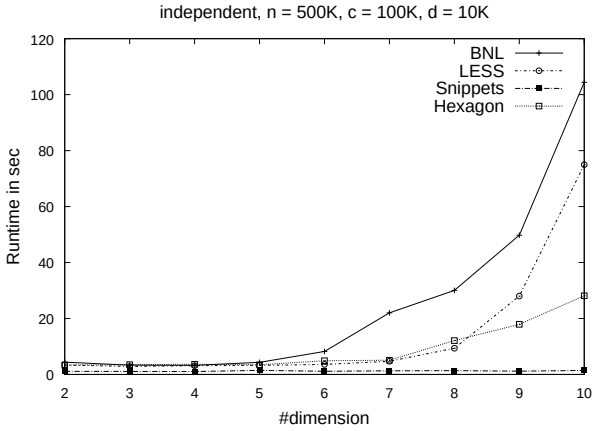

Figure 12b: Skyline-Snippets: Test 2, IND

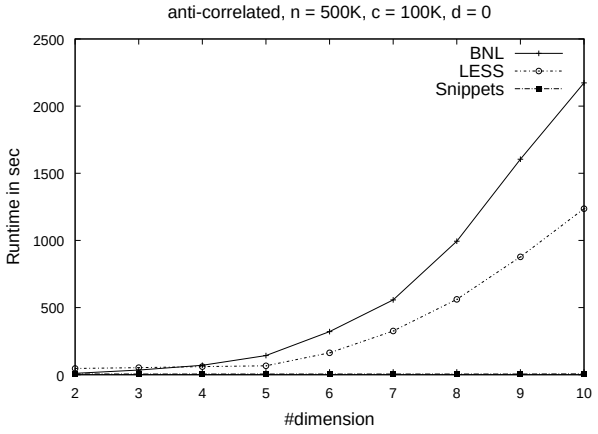

Figure 11c: Skyline-Snippets: Test 1, ANTI

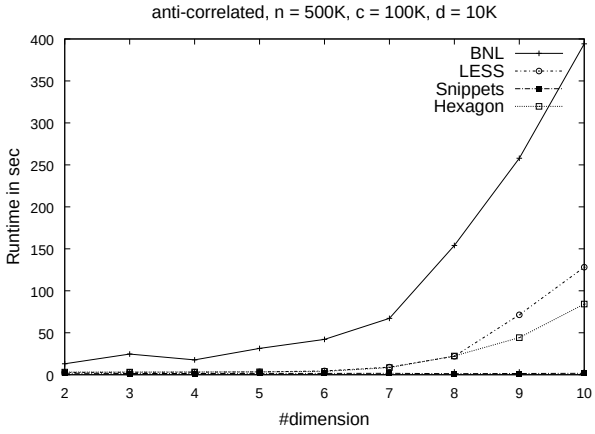

Figure 12c: Skyline-Snippets: Test 2, ANTI

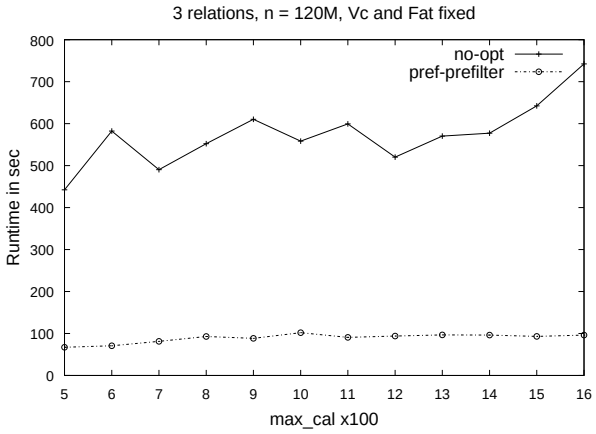

Figure 13: Pref-Filter: Test 1, max_cal

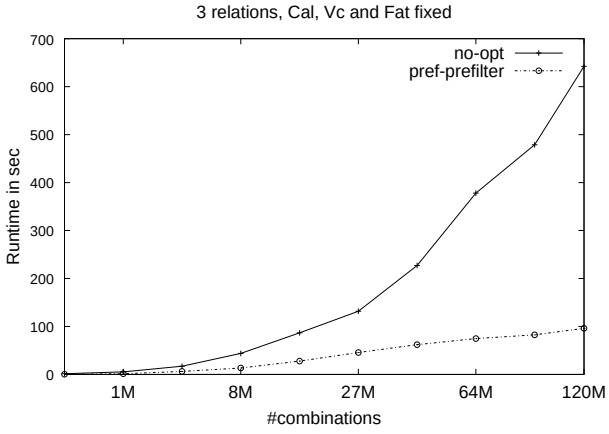

Figure 14: Pref-Filter: Test 2, relations

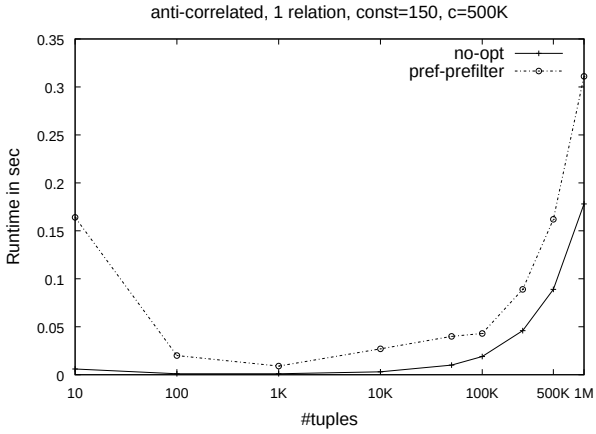

Figure 15: Pref-Filter: Test 3, 1 relation

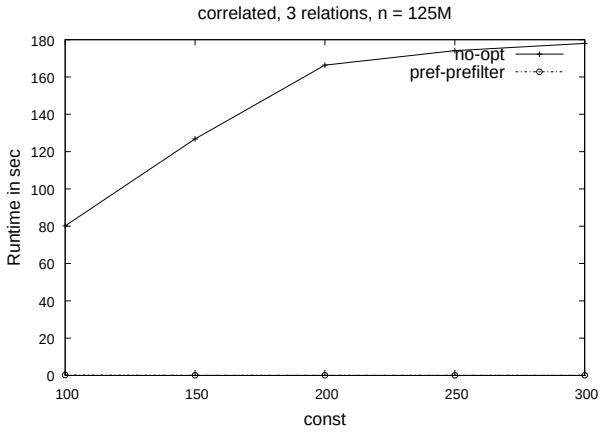

Figure 16a: Pref-Filter: Test 4, COR

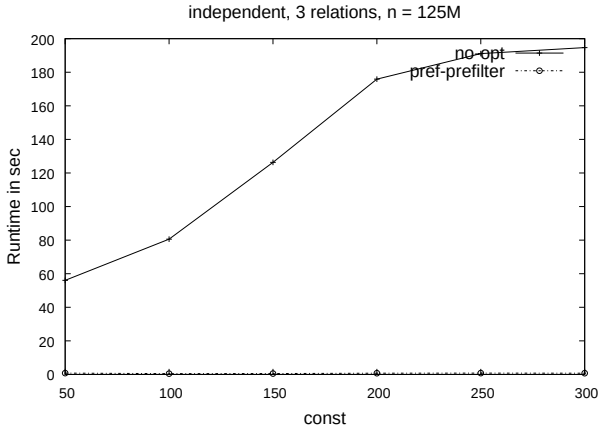

Figure 16b: Pref-Filter: Test 4, IND

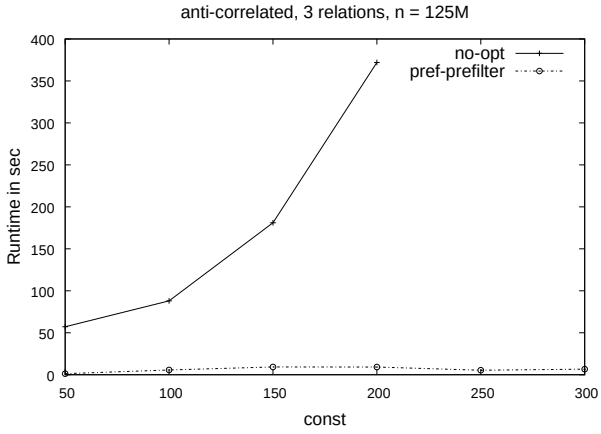

Figure 16c: Pref-Filter: Test 4, ANTI