

Specification and optimization of preference (SV-)grouping queries

Markus Endres, Werner Kießling, Patrick Rooks

Angaben zur Veröffentlichung / Publication details:

Endres, Markus, Werner Kießling, and Patrick Rooks. 2013. "Specification and optimization of preference (SV-)grouping queries." Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

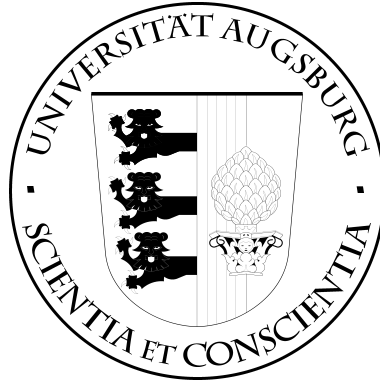
Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



UNIVERSITÄT AUGSBURG

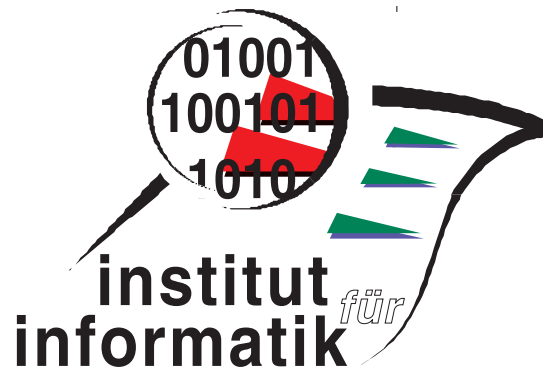


Specification and Optimization of
Preference (SV-)Grouping Queries

P. Rooks M. Endres W. Kießling

Report 2013-01

March 2013



INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

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

Specification and Optimization of Preference (SV)-Grouping Queries

Patrick Roocks Markus Endres Werner Kießling
Institute for Computer Science
University of Augsburg
86135 Augsburg, Germany
firstname.lastname@informatik.uni-augsburg.de

Abstract

Preference queries become more and more important in applications like OLAP, data warehousing, or decision support systems. In these environments the Preference SQL GROUPING operation and aggregate functions are extensively used in formulating queries. In this report we present the full specification of the GROUPING operation in Preference SQL. This specification describes the grouping and aggregation known from standard SQL as well as the grouping with substitutable values (SV) semantics to allow a flexible and powerful grouping functionality in comparison to standard SQL. Furthermore, we introduce novel algebraic transformation laws for grouped preference queries and numerical ranking which are one of the most intuitive and practical type of queries. We explain how Preference SQL can be modified to integrate these optimization laws into the existing rule-based query optimizer. Our study upon the well-known TPC-H benchmark dataset shows that significant performance gains can be achieved.

1 Introduction

Database queries containing grouping constructs are highly important for a wide field of applications, e.g., in business intelligence, the OLAP approach or data warehouses. Such decision support systems use the SQL operation of Group-by and aggregate functions extensively in formulating queries. For example, queries that create summary data are of great importance in such applications. These queries partition data in several groups (e.g., in business sectors) and aggregate on some attributes (e.g., sum of total sales).

Beyond this, the concept of Preference queries [7, 8, 9] has been established in the database community and was intensively studied in the last decade. Preference queries become more and more important in decision support environments, because they are an effective method to reduce very large datasets to

a small set of highly interesting results and to overcome the empty result set and flooding effect as described in [7]. In general, a preference query selects those objects from the database that are not dominated by any others. Therefore, preferences have shifted retrieval models from exact matching of attribute values to the notion of best matching database objects. Since Group-by and aggregating are essential in decision support systems, it thus makes sense to extend preference queries by grouping and aggregating functionality. In the following we show an example of a simple preference query using grouping.

Example 1. The wish for a car having the highest power and the lowest price can be expressed in Preference SQL as follows.

```

SELECT id, power, price
FROM car
PREFERRING power HIGHEST AND price LOWEST
GROUPING make;

```

Thereby the part of the query beginning with **PREFERRING** is called *preferring clause*. The connection of two preferences by **AND** is called *Pareto composition* and states the equal importance of two preferences. The query returns all cars, which belong to the maximal power / minimal price preference respective to their make.

Assume the sample dataset in Table 1. Then, each make forms its own group. Thus, in the BMW group the tuple with ID 1 is dominated by tuple 3, because the latter has more power and is cheaper than tuple 1. Therefore, the result is given by the IDs {3,2,5,4} where BMW, Mercedes, and Audi each forms one group.

Table 1: Sample dataset of cars.

<i>car</i>	id	make	power	price	ownerid
	1	BMW	180	35000	1
	2	Mercedes	200	38000	2
	3	BMW	230	34000	2
	4	Audi	170	32000	3
	5	Mercedes	220	40000	1

We generalize this concept by allowing arbitrary equivalence relations instead of the grouping by single attributes. While grouping on attributes is completely analogous to the *group by* clause in standard SQL, grouping on equivalence classes is described with the *substitutable values* (SV) construct, where the grouping is based on distinct values of an attribute. Furthermore we allow aggregating over the result, for example we can count the size of the BMO-set for every make. We show this in the next example, where a ranking function is used to specify the preference.

Example 2. Assume one wants to retrieve those cars, which are minimal with respect to the following ranking function: $f(\text{power}, \text{price}) = 0.01 \cdot \text{price} - 1 \cdot \text{power}$, i.e., a weighted compromise of high power and low price is preferred. This is done with the subsequent query, where `identityScoreF` denotes the identity and `sumRankF` is the weighted sum rank function. Furthermore, we want to retrieve the best cars for each federal state of Germany. This means, group all cars together which are made in Bavaria (Audi, BMW), in Baden-Württemberg (Mercedes), and so on. This is done using the **SV** keyword, cp. Section 4.2. Then, Audi and BMW are grouped together, and Mercedes builds one group in our sample dataset. This leads to the following query.

```

SELECT state, id
FROM car
PREFERRING (power SCORE 'identityScoreF' |
             price SCORE 'identityScoreF')
             RANK 'sumRankF' '-1,0.01'
GROUPING make SV (
    ('Audi', 'BMW') AS 'bavarian',
    ('Mercedes') AS 'baden-wuerttemberg') as state;

```

The query returns tuple 3 in the group of Bavarian cars, and tuples 2 and 5 in the group of Baden-Württemberg cars. In the latter one both cars have the same ranked score value.

Nowadays the size of databases increases drastically and complex business analysis becomes progressively more important. Motivated by this and the high significance of preferences, the question of efficient processing and optimization of such (SV-)grouping queries arises. Unfortunately, this problem has so far received little attention.

In this report we discuss Group-by and aggregation in preference queries. We provide a specification for (SV-)grouping which allows user-defined SV-relations for a flexible and powerful grouping functionality. Furthermore, we present significant novel optimization techniques of grouping preference queries which exploits the knowledge about the grouping clause in a query. For this we introduce transformation laws that make it possible to push a preference grouping operation over one or more joins and can potentially reduce the cost of processing a query significantly.

It turned out that the grouping constructs are also useful by means of optimization for non-grouping queries. Thereby the enrichment of queries with “redundant” grouping constructs allows the application of the optimization rules which require the occurrence of grouping constructs in the query. Hence the optimization of grouping preference queries is also fruitful for non-grouping preference queries and thus has high significance for the field of preferences in general.

In addition to these practical advances, the algebraic approach presented here demonstrates how to find clear and compact proofs by using the very general concept of a point-free preference algebra.

The remainder of this report is organized as follows: Section 2 contains the formal background used in this report and an introduction to Preference SQL. Section 3 discusses a point-free algebra for preference relations. Based on this, we specify grouped preference queries in Section 4. Afterwards we will develop relational query transformation laws and show the benefit for preference query optimization in Section 5. We conduct an extensive performance evaluation on the TPC-H benchmark dataset [1] in Section 6. Section 7 contains our concluding remarks.

2 Preference Background

Preference queries have been in focus for some time, leading to diverse approaches, e.g. [3, 7, 8]. We follow the preference model from [8] which is a direct mapping to relational algebra and declarative query languages, e.g., Preference SQL which is discussed in Section 2.2. It is semantically rich, easy to handle and very flexible to represent user preferences which are ubiquitous in our life.

Definition 1 (Preference). A preference $P = (A, <_P)$, where A is a set of attributes, is a strict partial order on the domain of A . Thus $<_P$ is irreflexive and transitive. The term $\mathbf{x} <_P \mathbf{y}$ is interpreted as “I like \mathbf{y} more than \mathbf{x} ”. Two tuples x and y are *indifferent*, if $\neg(x <_P y) \wedge \neg(y <_P x)$, i.e., neither x is better than y nor y is better than x .

The result of a preference is computed by the *preference selection* [7, 8], also called *winnow* by [3].

Definition 2 (Preference Selection, BMO-SET). The Best-Matching-Objects (BMO-set) of a preference $P = (A, <_P)$ on an input database relation R contains all tuples that are not dominated w.r.t. the preference. It is computed by the preference selection operator $\sigma[P](R)$ and finds all best matching tuples t for P , where $t[A]$ is the projection to the attribute set A .

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

Best-Matches-Only offers a cooperative query answering behavior by automatic matchmaking: The BMO query result adapts to the quality of the data in the database, defeating the empty result effect and reducing the flooding effect by filtering out worse results.

Note that the projection to the attribute set A in Definition 2 is often omitted. For a preference $P = (A, <_P)$ the expression $t <_P t'$ is equivalent with $t[A] <_P t'[A]$, i.e., whenever two tuples are compared w.r.t. a preference they have to be projected to the respective domain.

2.1 Preference Constructors

To specify a preference, a variety of intuitive base preference constructors together with some complex preference constructors has been defined. Subsequently, we present some selected preference constructors used in this report. More preference constructors as well as their formal definition can be found in [7, 8, 9].

2.1.1 Base Preference Constructors

Preferences on single attributes are called *base preferences*. There are base preference constructors for *discrete (categorical)* and for *continuous (numerical)* domains. Figure 1 shows the taxonomy of several frequently occurring base preferences [9]. Subsequently we describe some *numerical base preferences*.

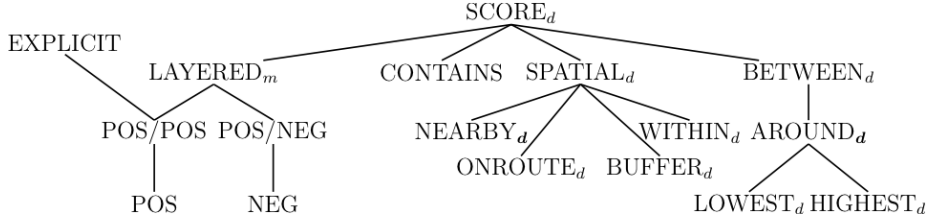


Figure 1: Taxonomy of base preference constructors

Definition 3 (SCORE_d Preference). Given a scoring function $f : \text{dom}(A) \rightarrow \mathbb{R}_0^+$, and some $d \in \mathbb{R}_0^+$. Then P is called a SCORE_d preference, iff for $x, y \in \text{dom}(A)$:

$$x <_P y \iff f_d(x) > f_d(y)$$

where $f_d : \text{dom}(A) \rightarrow \mathbb{R}_0^+$ is defined as:

$$f_d(v) := \begin{cases} f(v) & \text{if } d = 0 \\ \left\lceil \frac{f(v)}{d} \right\rceil & \text{if } d > 0 \end{cases}$$

Note that in the case of $d = 0$ the function $f(v)$ models the *distance* to the best value. That means $f_d(v)$ describes how far the domain value v is away from the optimal value. A d -parameter $d > 0$ represents a discretization of $f(v)$, which is used to group ranges of scores together. The d -parameter maps different function values to a single number. Choosing $d > 0$ effects that attribute values with identical $f_d(v)$ value become indifferent.

Next, we discuss some sub-constructors of SCORE_d .

Definition 4 (Numerical Base Preferences). We define the following preferences on a continuous domain:

- The $\text{BETWEEN}_d(A, [low, up])$ preference expresses the wish for a value between a *lower* and an *upper* bound. If this is infeasible, values having the smallest distance to $[low, up]$ are preferred, where the distance is discretized by the discretization parameter d . The scoring function equals $f(v) = \max\{low - v, 0, v - up\}$.
- Specifying $low = up (=: z)$ in BETWEEN_d yields the $\text{AROUND}_d(A, z)$ preference, where the desired value should be z , i.e., $f(v) = |z - v|$.
- Furthermore, the $\text{LOWEST}_d(A, \inf_A)$ constructor and the $\text{HIGHEST}_d(A, \sup_A)$ constructor prefer the minimum and maximum of the domain of A , where \inf_A and \sup_A are the infimum and supremum of $\text{dom}(A)$. The scoring function equals $f(v) = v - \inf_A$ and $f(v) = \sup_A - v$, respectively.

Example 3. In the introductory Example 1 the HIGHEST and LOWEST preference constructors were used to express the wish for the maximal power and the minimal price of a car. Using the described preference constructors this leads to $\text{HIGHEST}_0(\text{power}, \sup_{\text{power}})$ and $\text{LOWEST}_0(\text{price}, \inf_{\text{price}})$, both with $d = 0$.

We now describe some *categorical base preferences* for frequently occurring cases.

Definition 5 (Categorical Base Preferences). We define the following preferences on a discrete domain:

- The positive preference $\text{POS}(A, \text{POS-set})$ expresses that a user has a set of preferred values, the *POS-set*, in the domain of A . For $d = 0$ we have

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

- The negative preference $\text{NEG}(A, \text{NEG-set})$ is the counterpart to the POS preference, formally

$$\text{NEG}(A, \text{NEG-set}) := \text{POS}(A, \text{dom}(A) \setminus \text{NEG-set}).$$

It is possible to combine the POS and NEG preferences to POS/POS or POS/NEG. For the $\text{POS/POS}(A, \text{POS1-set}, \text{POS2-set})$ preference a desired value *should be* amongst a finite set *POS1-set*. Otherwise it *should be* from a disjoint finite set of *alternatives POS2-set*. If this is also not feasible, any other value is acceptable; this is still better than getting nothing. There are many more base preference constructors (cp. Figure 1), all described in [7, 8, 9].

Example 4. The wish for an Audi or BMW leads to a POS preference

$$P_1 := \text{POS}(\text{make}, \{\text{'Audi'}, \text{'BMW'}\})$$

The preference selection $\sigma[P_1](\text{car})$ on the car dataset in Table 1 leads to the cars with IDs $\{1,3,4\}$.

If we prefer a horsepower around 170, where a difference of 5 does not matter, i.e. $d = 5$, this can be expressed by

$$P_2 := \text{AROUND}_5(\text{power}, 170)$$

The result is given by the tuple with ID 4 because it has a perfect match of 170 hp.

2.1.2 Complex Preference Constructors

If one wants to combine several preferences into more *complex preferences*, one has to decide the relative importance of these given preferences. Intuitively, people speak of “this preference is more important to me than that one” or “these preferences are all equally important to me”. Equal importance is modeled by the so-called *Pareto preference*.

Complex preferences are built of constructs like “Better w.r.t. P_1 , equal w.r.t. to P_2 ”, where P_1 and P_2 are preferences. Hence we need a notion of equality w.r.t. a preference. A simple approach for this is to use strict equality of the domain values. But often we have base preferences where values x, x' are equally good in the sense that $x <_P y \Leftrightarrow x' <_P y$ for all y . For example, this is the case if $f_d(x) = f_d(x')$, i.e., the tuples have the same score. This behavior is called *regular substitutable values semantics* (SV semantics), denoted by \sim_P . In contrary, requiring strict equality leads to the *trivial SV-semantics*, denoted by $=_P$. We formalize this in the following definition.

Definition 6 (SV-Relations). Let $P = (A, <_P)$ be a preference, and A an attribute set. We define the following:

- a) A general SV-relation (\cong, A) on attribute set A , where \cong has to be an equivalence relation on $\text{dom}(A)$.
- b) A SV-relation (\cong_P, A) for a preference $P = (A, <_P)$ has to be compatible to P which means that we have for all $x, y, z \in \text{dom}(A)$:
 - $x \cong_P y \Rightarrow \neg(x <_P y \vee y <_P x)$
 - $x <_P y \wedge y \cong_P z \Rightarrow x <_P z$
 - $x \cong_P y \wedge y <_P z \Rightarrow x <_P z$

Note that the attribute set is often omitted, i.e. we just write \cong_P for (\cong_P, A) . By convention, if P is a preference, then \cong_P is always the SV-relation associated with preference P .

- c) The identity relation on attribute set A , denoted by id_A . For a preference $P = (A, <_P)$ this is also called the *trivial SV-relation* and denoted by $=_P$, i.e. we always have

$$=_P := \text{id}_A .$$

- d) The *regular SV-relation* \sim_P for a preference P , the equivalence relation induced by the equivalence classes of $f_d(v)$.

Note that the above definition is well defined in the sense that the trivial and regular SV-relation are indeed compatible with their respective preferences. Also note that this definition of SV-semantics is needed to preserve the strict order property of complex preferences, cp. [8]. In the following we define two important complex preference constructors.

Definition 7 (Pareto). In a *Pareto preference* $P := P_1 \otimes P_2 = (A_1 \times A_2, <_P)$ all preferences $P_i = (A_i, <_{P_i})$ on the attributes A_i are of equal importance, i.e., for two tuples $x = (x_1, x_2), y = (y_1, y_2) \in \text{dom}(A_1) \times \text{dom}(A_2)$ we define:

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

If we restrict the attention to LOWEST/HIGHEST as input preferences for a Pareto preference P , then Pareto preference queries coincide with the traditional *Skyline queries* [2], corresponding to MIN/MAX. The BMO-set (Definition 2) of a Pareto preference query P is generally referred to as *the Skyline of P*.

Definition 8 (Prioritization). The *Prioritization preference* allows the modeling of combinations of preferences that have different importance. Assume preferences $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$, then prioritization denoted by $P := P_1 \& P_2$ is defined as:

$$(x_1, x_2) <_P (y_1, y_2) \iff x_1 <_P y_1 \vee (x_1 \cong_{P_1} y_1 \wedge x_2 <_{P_2} y_2)$$

For base preferences regular SV semantics does not affect $<_P$ itself, but expresses that it is admissible to substitute values for each other. A complex constructor using \sim_P does affect $<_P$, as we can see in the next example.

Example 5. Reconsider the preferences P_1 and P_2 from Example 4. In the Pareto preference both preferences are equally important and we write

$$\text{POS}(\text{make}, \{\text{'Audi'}, \text{'BMW'}\}) \otimes \text{AROUND}_5(\text{power}, 170)$$

Using *trivial* SV-semantics for P_1 and P_2 (i.e. $\cong_{P_i} = \text{id}_{A_i}$) the tuples ('BMW', 180) and ('Audi', 170) would be the best-matches, although a horsepower of 170 is better than 180. Due to the trivial SV-semantics BMW and Audi are *not* substitutable.

Having *regular* SV-semantics, BMW and Audi become substitutable. Therefore ('Audi', 170) is equally good as ('BMW', 180) concerning the make, but 170 hp is better than 180 concerning the AROUND preference. This means, ('Audi', 170) is the only tuple in the result set.

Another form of preference combination is by associating numerical scores to each individual preference and then applying a combining function to compute a single score, which decides the “better-than” relationship. Such *weighted importance* between preferences is realized by the numerical ranking preference, cf. [7, 8, 9].

Definition 9 (Rank). Given regular preferences P_1 and P_2 with scoring functions f_1 and f_2 and a d-value $d > 0$. Then the *Rank preference* $\text{RANK}_{F,d}$ with a combining function $F : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{N}_0$ is defined as:

$$(x_1, x_2) <_P (y_1, y_2) \iff \left\lceil \frac{F(f_1(x_1), f_2(x_2))}{d} \right\rceil > \left\lceil \frac{F(f_1(y_1), f_2(y_2))}{d} \right\rceil$$

Note that all sub-constructors of SCORE_d are defined by a scoring functions, thus can be used in a Rank preference. Also note that the SV-semantics of P_1 and P_2 are not relevant to the Rank preference of those. Furthermore, a generalization of all complex preferences to more than two preferences is straight forward, cp. [13].

2.2 Preference SQL

Preference SQL is a declarative extension of standard SQL by strict partial order preferences, behaving like soft constraints under the BMO query model. The BMO-set as result of a preference query contains all database tuples which are not dominated by any other tuple concerning the users preferences, cp. [7]. Syntactically, Preference SQL extends the `SELECT` statement of SQL by an optional **PREFERRING** clause leading to the (basic) schematic design of Figure 2.

```

SELECT      ... <projection>
FROM        ... <table_reference>
WHERE       ... <hard_conditions>
PREFERRING ... <soft_conditions>
TOP       ... <number>
ORDER BY   ... <attribute_list>

```

Figure 2: Preference SQL query block.

The keywords `SELECT`, `FROM`, `WHERE`, and `ORDER BY` are treated as the standard SQL query keywords. The **PREFERRING** clause specifies a preference by means of the preference constructors given in Section 2.1 and [9].

A preference is evaluated on the result of the hard conditions stated in the WHERE clause, returning the BMO-set. Empty results can only occur if the WHERE clause returns an empty result. If TOP- k is specified, only the k best tuples according to the preference order are returned.

Note that this does not represent the entire schema of Preference SQL. The additional syntactical elements for grouping queries will be explained in Section 4. *Preference SQL* currently supports most of the SQL-92 standard as well as all base and complex preference constructors from the previous section. For a full overview we refer to [7, 8, 9].

Example 6. The Pareto preference from Example 5 can be expressed in Preference SQL as follows:

```
SELECT id
FROM   car
PREFERRING make IN ('BMW', 'Audi') REGULAR
          AND power AROUND 170, 5 REGULAR;
```

AROUND expresses an AROUND-preference with d-value $d = 5$ and **IN** is a POS preference with preferred values BMW and Audi, cp. Definition 4. The **AND** in the **PREFERRING** clause denotes a Pareto preference, whereas a Prioritization is expressed using **PRIOR TO**. The keyword **REGULAR** defines a preference using regular SV-semantics whereas **TRIVIAL** denotes trivial SV-semantics, cp. Definition 6.

3 Point-free Algebra for Preference Relations

The definitions of the Pareto-composition and the Prioritization in the Definitions 7 and 8 are point-wise in the manner that explicit tuples with their components (i.e., “points”) appear on both sides of the equation. Additionally all variables have to be allquantified.

In [11, 12] we introduced a semiring-based point-free algebraic calculus for preferences. It turned out that many proofs can be shortened using this calculus and these techniques pave the way for automatic theorem provers. As the theory of grouped preferences also benefits from this formalism we will introduce it in this report.

3.1 Point-free Compositions of Preferences

At first we introduce some basic definitions of our framework, starting with the intersection of two preferences or SV relations. The intersection was already introduced in [7] and will be generalized to SV relations.

Definition 10 (Intersection with SV). Let P_1, P_2 be preferences and \cong_1, \cong_2 be SV relations. For tuples x, y we define the following intersection preferences:

- $P := P_1 \blacklozenge P_2$: $x <_P y \Leftrightarrow x <_{P_1} y \wedge x <_{P_2} y$
 $x \cong_P y \Leftrightarrow x \cong_{P_1} y \wedge x \cong_{P_2} y$
- $P := P_1 \blacklozenge \cong_1$: $x <_P y \Leftrightarrow x <_{P_1} y \wedge x \cong_1 y$
 $(= \cong_1 \blacklozenge P_1)$ $x \cong_P y \Leftrightarrow x \cong_{P_1} y \wedge x \cong_1 y$
- $\cong_3 := \cong_1 \blacklozenge \cong_2$: $x \cong_3 y \Leftrightarrow x \cong_1 y \wedge x \cong_2 y$

Remember that \cong_P for a preference P is always the SV-relation of P , as defined in Definition 6.

Preferences with “ \blacklozenge ” as “join operator” form a concrete relational instance of the abstract *join algebra*, defined in [12]. This means that all theorems which hold for join algebras also hold for the concrete instance. We give some further definitions to express the Pareto-composition and the Prioritization point-free:

Definition 11 (Union of Preferences). Let P_1, P_2 be preferences. The union of two preferences $P := P_1 + P_2$ is defined for all tuples x, y as:

$$x <_P y \Leftrightarrow x <_{P_1} y \vee x <_{P_2} y$$

Furthermore we define $x \cong_P y \Leftrightarrow x \cong_{P_1} y \wedge x \cong_{P_2} y$, i.e., we use the logical conjunction, despite of $P_1 + P_2$ behaves like the disjunction. But note that the disjunction of SV relations would not yield an equivalence relation. By convention, \blacklozenge binds tighter than $+$.

Definition 12 (Point-free Compositions). For preferences P_1 and P_2 we give point-free definitions for the Prioritization and the Pareto Composition.

- $P_1 \& P_2 = P_1 + \cong_{P_1} \blacklozenge P_2$
- $P_1 \otimes P_2 = (P_1 + \cong_{P_1}) \blacklozenge P_2 + (P_2 + \cong_{P_2}) \blacklozenge P_1$

The well-definedness of these definitions, i.e., their compatibility with Definitions 7 and 8 is done by easy verification.

Expressions like “ $P_1 \otimes P_2$ ” or “ $(P_1 \otimes P_2) \blacklozenge \cong_1$ ” are *preference terms*. We aim to work with them in a point-free fashion, which is especially useful for the proofs of our theorems. Therefore we need formal definitions of equivalence and inclusion w.r.t. preference terms.

Definition 13 (Preference Term (In)Equations). For preferences $P_1 = (A, <_{P_1})$, $P_2 = (A, <_{P_2})$ we define:

- $P_1 = P_2$ if for all $x, y \in \text{dom}(A)$ we have that

$$x <_{P_1} y \Leftrightarrow x <_{P_2} y$$

- $P_1 \subseteq P_2$ if for all $x, y \in \text{dom}(A)$ we have that

$$x <_{P_1} y \Rightarrow x <_{P_2} y$$

- $P_1 \supseteq P_2$ iff $P_2 \subseteq P_1$

Note that the inclusion order \subseteq corresponds to the subsumption order in [12].

3.2 Properties of Preference Operations

In the following we present some properties based on the previous definitions.

Lemma 1 (Fundamental Properties). *We have the following properties for intersection and union, where P_1 is a preference, and X, Y are preferences or SV relations.*

- $+$ and \diamond are associative and commutative
- \diamond distributes over $+$
- $+, \diamond$ are isotone w.r.t. \subseteq , i.e. we have:

$$X \subseteq X' \Rightarrow X + Y \subseteq X' + Y \quad \wedge \quad X \diamond Y \subseteq X' \diamond Y$$

- For $+, \diamond$ and \subseteq we have: $X \subseteq X + Y$ and $X \diamond Y \subseteq X$
- For $P := P_1 \diamond \cong_1$ we have: $\cong_P = \cong_{P_1} \diamond \cong_1$

Proof. a-d) directly follow from the definitions and the well-known properties of the logical \wedge, \vee and \Rightarrow operators. e) follows directly from the definitions of $P_1 \diamond \cong_1$ and $\cong_1 \diamond \cong_2$. \square

4 Grouping Queries

For preferences queries there exists a grouping-construct, which allows to split the dataset into several groups according to the grouping attributes. Afterwards the preference is evaluated on each group separately. If no aggregate function is given, the BMO-sets of all groups are put together and returned. If an aggregate function (e.g., SUM(...), COUNT(...)) is specified in the projection of the query, then each BMO-set collapses to the aggregation result. In this case, due to the grouping-modifier HAVING groups can also be excluded from the result with a having-condition, e.g., COUNT(*) > 1.

4.1 Specification of Grouped Preferences

4.1.1 Grouped Preference Selection

The simplest grouping queries split the dataset according to distinct values of one or more attributes. For a preference P and an attribute set $G \subseteq \text{attrib}(R)$, where $\text{attrib}(R)$ are all attributes of a database relation R , we define:

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

According to [7] this can be expressed as a preference itself:

$$t <_{P \text{ grouping } G} t' \iff t <_P t' \wedge t[G] = t'[G]$$

We also allow grouping w.r.t. to equivalence relations on attributes. Let \cong_X be an equivalence relation on an attribute $X \in G = \{A, B, \dots\}$. We define the direct product of the SV relations in the attribute set $G = \{A, B, \dots\}$:

$$\cong_G = \cong_A \diamond \cong_B \diamond \dots$$

The grouped preference selection according to \cong_G is defined as:

$$\sigma[P \text{ grouping } \cong_G](R) := \{t \in R \mid \neg \exists t' \in R : t <_P t' \wedge t \cong_G t'\} \quad (1)$$

As the identity relation id_A on $\text{dom}(A)$ is an equivalence relation, this is a generalization of the grouping on distinct values of attributes. To see this, consider that $P \text{ grouping } G$ can be expressed as $P \text{ grouping } \cong_G$ with $\cong_G = \text{id}_A \diamond \text{id}_B \diamond \dots$ where $G = \{A, B, \dots\}$.

4.1.2 Point-free Characterization of Grouped Preferences

We can characterize a grouped preference by the direct product of the grouping relation and the preference relation. This yields our point-free definition of the grouping construct:

Definition 14 (Grouping). For a preference P and an attribute A we define the following grouped preferences:

a) Grouping w.r.t. A , where id_A is the identity on $\text{dom}(A)$:

$$P \text{ grouping } A := \text{id}_A \diamond P$$

b) Grouping w.r.t. an arbitrary SV relation (\cong_A, A) , which operates on $\text{dom}(A)$:

$$P \text{ grouping } \cong_A := \cong_A \diamond P$$

By convention, `grouping` binds tighter than `+` and `♦`.

Note that this is well-defined, i.e., we have

$$\sigma[\cong_A \blacklozenge P](R) = \sigma[P \text{ grouping } \cong_A](R)$$

where **grouping** in the right hand side is interpreted according to Equation (1). To see this, one has simply to apply the definitions of “ \blacklozenge ” and $\sigma[\dots]$ to the left hand side.

Now we will present some results. At first we show that some grouping constructs serve as lower estimates for complex preferences. This will be helpful later on for proving optimization theorems.

Lemma 2 (Grouping as Lower Bound for Complex Compositions). *For preferences P_1, P_2 with SV relations \cong_{P_1} and \cong_{P_2} we have:*

- a) $P_1 \& P_2 \supseteq P_2 \text{ grouping } \cong_{P_1}$
- b) $P_1 \otimes P_2 \supseteq P_1 \text{ grouping } \cong_{P_2} + P_1 \text{ grouping } \cong_{P_2}$
- c) $P_1 \otimes P_2 \supseteq P_1 \text{ grouping } \cong_{P_2}$

Proof. Using the definitions of $\otimes, \&$, **grouping** and additionally some basic properties (Lemma 1) we infer:

- a) $P_1 \& P_2 = P_1 + \cong_{P_1} \blacklozenge P_2$
 $\supseteq \cong_{P_1} \blacklozenge P_2$
 $= P_2 \text{ grouping } \cong_{P_1}$
- b) $P_1 \otimes P_2 = (P_1 + \cong_{P_1}) \blacklozenge P_2 + (P_2 + \cong_{P_2}) \blacklozenge P_1$
 $\supseteq \cong_{P_1} \blacklozenge P_2 + \cong_{P_2} \blacklozenge P_1$
 $= P_2 \text{ grouping } \cong_{P_1} + P_1 \text{ grouping } \cong_{P_2}$

c) Follows immediately from b) □

For compositions of grouping and preferences we yield the following results:

Lemma 3 (Simplify Grouping Constructs). *Let $P = (A, <_P)$, $P_1 = (A_1, <_{P_1})$, $P_2 = (A_2, <_{P_2})$ be preferences with $A, A_1, A_2 \subseteq \text{attrib}(R)$ and \cong_1, \cong_2 be equivalence relations. Then the following holds:*

- a) $(P \text{ grouping } \cong_1) \text{ grouping } \cong_2 = P \text{ grouping } (\cong_1 \blacklozenge \cong_2)$
- b) $(P_1 \text{ grouping } \cong_1) \& P_2 = (P_1 \& P_2) \text{ grouping } \cong_1$
- c) $P_1 \otimes (P_2 \text{ grouping } \cong_1) = (P_1 \otimes P_2) \text{ grouping } \cong_1$
- d) $(P_1 \text{ grouping } \cong_1) \otimes (P_2 \text{ grouping } \cong_2) = (P_1 \otimes P_2) \text{ grouping } (\cong_1 \blacklozenge \cong_2)$

Proof. The following derivations are justified by applying the definitions and by using Lemma 1.

$$\begin{aligned}
\text{a)} \quad & (P \text{ grouping } \cong_1) \text{ grouping } \cong_2 \\
& = \cong_2 \diamond (\cong_1 \diamond P) \\
& = (\cong_1 \diamond \cong_2) \diamond P \\
& = P \text{ grouping } (\cong_1 \diamond \cong_2) \\
\text{b)} \quad & (P_1 \text{ grouping } \cong_1) \& P_2 \\
& = (\cong_1 \diamond P_1) \& P_2 \\
& = \cong_1 \diamond P_1 + \cong_{P_1} \diamond \cong_1 \diamond P_2 \\
& = \cong_1 \diamond (P_1 + \cong_{P_1} \diamond P_2) \\
& = \cong_1 \diamond (P_1 \& P_2) \\
& = (P_1 \& P_2) \text{ grouping } \cong_1 \\
\text{c)} \quad & P_1 \otimes (P_2 \text{ grouping } \cong_1) \\
& = P_1 \otimes (\cong_1 \diamond P_2) \\
& = (P_1 + \cong_{P_1}) \diamond (\cong_1 \diamond P_2) + (\cong_1 \diamond P_2 + \cong_1 \diamond \cong_{P_2}) \diamond P_1 \\
& = \cong_1 \diamond ((P_1 + \cong_{P_1}) \diamond P_2 + (P_2 + \cong_{P_2}) \diamond P_1) \\
& = \cong_1 \diamond (P_1 \otimes P_2) \\
& = (P_1 \otimes P_2) \text{ grouping } \cong_1 \\
\text{d)} \quad & (P_1 \text{ grouping } \cong_1) \otimes (P_2 \text{ grouping } \cong_2) \\
& = (\cong_1 \diamond P_1) \otimes (\cong_2 \diamond P_2) \\
& = (\cong_1 \diamond P_1 + \cong_1 \diamond \cong_{P_1}) \diamond (\cong_2 \diamond P_2) + \\
& \quad (\cong_2 \diamond P_2 + \cong_2 \diamond \cong_{P_2}) \diamond (\cong_1 \diamond P_1) \\
& = (\cong_1 \diamond \cong_2) \diamond ((P_1 + \cong_{P_1}) \diamond P_2 + (P_2 + \cong_{P_2}) \diamond P_1) \\
& = (\cong_1 \diamond \cong_2) \diamond (P_1 \otimes P_2) \\
& = (P_1 \otimes P_2) \text{ grouping } (\cong_1 \diamond \cong_2)
\end{aligned}$$

□

4.2 Grouping and Preference SQL

4.2.1 Specifying the Grouping Attributes

Grouping on distinct values of attributes A, B, \dots is specified with `GROUPING A, B, \dots` completely analogous to the `GROUP BY` clause in SQL. Grouping on equivalence classes is described with the construct **sv** short for “Substitutable Values”. This is because the Grouping-SV functionality behaves similar to the SV relations on preferences, introduced in [8]. The equivalence classes forming

the groups are from now on called (*grouping*) *SV classes*. The syntactic schema of a SV classes specification is as follows (where the [...] braces indicate optional elements):

```

GROUPING A SV ((A_1, A_2, ...) [as 'class1_a'],
                 (A_3, A_4, ...) [as 'class2_a'], ...,
                 [OTHERS [as 'oth_class_a']] ) [as sv_a],
  B SV ((B_1, B_2, ...) [as 'class1_b'], ...,
        [OTHERS [as 'oth_class_b']] ) [as sv_b], ...

```

Thereby A_i and B_i are domain values according to the attributes A and B. The domain values (A_1, A_2, \dots) are considered equal w.r.t. the SV relation \cong_A for attribute A (and analogous for B and \cong_B). The names `class...` are aliases for the SV classes which occur in the projection. If not specified, (A_1, A_2, \dots) is the default name for an SV class with values A_1, A_2, \dots . The keyword `OTHERS` puts all elements which were not be mentioned before, in one “default” SV class. If `OTHERS` is not given, all tuples $t \in \text{dom}(A) \setminus \{A_1, A_2, \dots, A_3, A_4, \dots\}$ form a SV class (t) of its own.

The name of an SV-attribute (e.g. `sv_a`) is also optional, but should be given to reference the SV grouping attributes in the projection. Note that the original attributes `A, B, ...` stay unchanged; they can still be referenced in the projection. Finally the entire term generates an SV relation $\cong_G = \cong_A \blacklozenge \cong_B \blacklozenge \dots$ where $G = \{A, B, \dots\}$ are the grouping attributes.

4.2.2 The Entire Grouped Preference Query

The schema of an entire Preference SQL query containing the grouping construct is depicted in Fig. 3, where `aggregation` symbolizes an aggregation function, which is optional in the projection.

```

SELECT      ... <projection, aggregation>
FROM        ... <table-reference>
WHERE       ... <hard.conditions>
PREFERRING ... <soft.conditions>
GROUPING  ... <attribute.list>
TOP       ... <number>
BUT ONLY  ... <but.only.condition>
HAVING    ... <aggregating hard.conditions>
ORDER BY   ... <attribute.list>

```

Figure 3: Preference SQL query block with `grouping`.

Such a query is evaluated as follows:

- If no aggregation function (such as `SUM(...)`, `COUNT(...)`, etc.) is given, this query is evaluated as a grouped preference query, i.e., for every group the BMO-Set w.r.t. the preference is calculated. The additional specifier `TOP k` selects the k best tuples according to the preference order per group, i.e., $k \cdot g$ tuples are maximally returned if g groups exist. With the `BUT ONLY` construct it is possible to specify a hard post-selection.

- If an aggregation function occurs (either in the projection or in the having-clause) the following process is done: At first, the groups are calculated as above. Afterwards the having condition is evaluated, i.e., some groups might be excluded. Finally the aggregation function of the projection is applied; hence only one line is returned per group in the result set.

Note that TOP, BUT ONLY, HAVING, and the use of aggregation is optional. Even the PREFERRING keyword is optional, which allows us to use the Grouping-SV syntax for conventional aggregations. Thus we are able to simplify GROUP BY queries with the lengthy CASE ... THEN ... END statements. These can be mostly be replaced by our Grouping-SV-syntax which we will illustrate in Example 7.

Example 7. Assume the sample dataset given in Table 2.

Table 2: Sample dataset

<i>cars</i>	<u>id</u>	make	power	price
	1	Smart	60	15000
	2	Mercedes	200	38000
	3	Audi	180	29000
	4	VW	110	25000
	5	Bugatti	1000	500000

We give an example to calculate the average price of the two cheapest cars of every car manufacturer. We assume that only an attribute “make” is given for every tuple in the relation, hence we have to retrieve the manufacturer by an appropriate SV grouping on “make”.

```

SELECT manufacturer, AVG(price) FROM car
PREFERRING price LOWEST TOP 2
GROUPING make SV (('Mercedes', 'Smart') as 'Daimler',
                    ('VW', 'Audi', 'Bugatti') as 'Volkswagen'
                    ) AS manufacturer

```

The result of this query on the given sample dataset is: {(Daimler, 26500), (Volkswagen, 27000)}. Note that the Bugatti is not in the Top-2 set of the manufacturer “Volkswagen”, hence not included in the average price calculation.

If we omit the preference, i.e., just consider the aggregating query

```

SELECT manufacturer, AVG(price) FROM car GROUPING make SV ...

```

we could express this in standard SQL in a more lengthy way:

```

SELECT manufacturer, AVG(price) FROM
  SELECT (CASE
    WHEN make IN ('Mercedes', 'Smart') THEN 'Daimler'
    WHEN make IN ('VW', 'Audi', 'Bugatti') THEN 'Volkswagen'
    ELSE make
  END) AS manufacturer, fuel FROM car) tmp_car
GROUP BY manufacturer

```

Hence the Grouping-construct allows us to denote group-by-case statements in a more concise way and without subqueries.

The order of execution of a preference query is as follows, wherein all those steps are skipped which are not specified in the query:

1. the hard conditions in the WHERE-clause are evaluated
2. the remaining tuples are split into groups
3. the preference is evaluated on any group; thereby TOP k is considered
4. the hard post-selection via BUT ONLY is processed (which may lead to less than k tuples for TOP k queries)
5. the aggregation functions in the HAVING clause are evaluated and groups are excluded which do not fulfil the condition
6. the aggregation functions in the projection are processed and the results are returned

5 Optimization Theorems

Some results in the following have been also presented in [10], but in a simpler form without SV-semantics. In the following theorems we cover full support of SV semantics. The proofs in this report become more readable due to the use of – where appropriate – our point-free formalism introduced in Section 3. Note that the parts concerning joins are shown point-wise, as to the best of our knowledge, we cannot handle them in a point-free manner.

5.1 Decompositions of Preferences

We show some Lemmas which are helpful for the proofs of the optimization theorems:

Lemma 4 (Decomposition of the Union Preference). *For preferences P_1, P_2 and a relation R we have:*

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

Proof. We show this using the definition of σ and $(P_1 + P_2)$:

$$\begin{aligned} \sigma[P_1 + P_2](R) &= \{w \in R \mid \exists v : w <_{P_1} v \vee w <_{P_2} v\} \\ &= \{w \in R \mid \exists v : t <_{P_1} v \wedge \exists u : w <_{P_2} u\} \\ &= \sigma[P_1](R) \cap \sigma[P_2](R) \quad \square \end{aligned}$$

Note that a more general result, denoted as Lemma 4.2.7 in [12], could be shown there in a completely point-free fashion.

Lemma 5 (Subset Preferences). *For preferences P_1, P_2 with $P_2 \subseteq P_1$ and a relation R we have:*

- a) $\sigma[P_1](R) \subseteq \sigma[P_2](R)$
- b) $\sigma[P_1](R) = \sigma[P_1](\sigma[P_2](R))$

Proof. a) Note that $P_2 \subseteq P_1$ implies for tuples x, y : $x <_{P_2} y \Rightarrow x <_{P_1} y$. This yields $\neg(x <_{P_1} y) \Rightarrow \neg(x <_{P_2} y)$ and hence the definition of σ leads immediately to the claim.

b) We deduce:

$$\begin{aligned}
& \sigma[P_1](R) \\
&= \quad \llbracket \text{From a) we have: } \sigma[P_1](R) \subseteq \sigma[P_2](R) \rrbracket \\
& \quad \sigma[P_1](R) \cap \sigma[P_2](R) \\
&= \{w \in R \mid \nexists v \in R : w <_{P_1} v\} \cap \sigma[P_2](R) \\
&= \{w \in \sigma[P_2](R) \mid \nexists v \in R : w <_{P_1} v\} \\
&= \quad \llbracket \text{all candidates for } \exists \dots \text{ can be substituted by } v \in \sigma[P_1](R) \\
& \quad \text{due to transitivity of } <_{P_1} \rrbracket \\
& \quad \{w \in \sigma[P_2](R) \mid \nexists v \in \sigma[P_1](R) : w <_{P_1} v\} \\
&= \quad \llbracket \sigma[P_1](R) \subseteq \sigma[P_2](R) \rrbracket \\
& \quad \{w \in \sigma[P_2](R) \mid \nexists v \in \sigma[P_2](R) : w <_{P_1} v\} \\
&= \sigma[P_1](\sigma[P_2](R)) \quad \square
\end{aligned}$$

Note that for part a) a more general result, denoted as Lemma 4.2.8 in [12], could be shown there in a point-free way.

Lemma 6 (Push Preference over Union). *Let $P = (A, <_P)$ and $A \subseteq \text{attrib}(R) = \text{attrib}(S)$. Then we have:*

$$\sigma[P](R \cup S) = \sigma[P](\sigma[P](R) \cup \sigma[P](S))$$

Proof. Let $R^\sigma := \sigma[P](R)$, $S^\sigma := \sigma[P](S)$ and $T := R^\sigma \cup S^\sigma$

$$\begin{aligned}
& \sigma[P](\sigma[P](R) \cup \sigma[P](S)) = \sigma[P](T) \\
&= \{w \in T \mid \nexists v \in T : w <_P v\} \\
&= \{w \in T \mid \neg((\exists v \in R^\sigma : w <_P v) \vee (\exists v \in S^\sigma : w <_P v))\} \\
&= \quad \llbracket \text{all candidates for } \exists \dots \text{ can be substituted by } v \in R^\sigma \\
& \quad \text{(or } v \in S^\sigma, \text{ resp.) due to transitivity of } <_P \rrbracket \\
& \quad \{w \in T \mid \neg((\exists v \in R : w <_P v) \vee (\exists v \in S : w <_P v))\} \\
&= \{w \in R \cup S \mid \neg((\exists v \in R : w <_P v) \vee (\exists v \in S : w <_P v)) \wedge w \in R^\sigma \cup S^\sigma\} \\
&= \quad \llbracket \nexists v \in R : \dots \text{ implies } w \in R^\sigma \text{ and } \nexists v \in S : \dots \text{ implies } w \in S^\sigma \rrbracket
\end{aligned}$$

$$\begin{aligned}
& \{w \in R \cup S \mid \neg((\exists v \in R : w <_P v) \vee (\exists v \in S : w <_P v))\} \\
&= \{w \in R \cup S \mid \nexists v \in R \cup S : w <_P v\} \\
&= \sigma[P](R \cup S) \quad \square
\end{aligned}$$

Note that this is also an important lemma for a distributed computation of the BMO-Set: A large dataset R can be split into (disjoint) sets $R_1 \cup \dots \cup R_n = R$ where the BMO-sets can be calculated independently. Afterwards merging the partial results and finally applying $\sigma[P](\dots)$ leads to the result, cp. [3, 6].

5.2 Preferences and Joins

In this section we study the interplay of preferences and (semi-)joins. At first we present the definition of an *extension* which specializes the database join:

Definition 15 (Extension). Let $f(r, s)$ be a left-total Boolean-valued function, i.e., for every r there exists at least one s satisfying $f(r, s) = \text{true}$. The *extension of R by S through f* is defined as:

$$R \text{ext}_f S = \{(r, s) \in R \times S \mid f(r, s) = \text{true}\}$$

One immediate result is:

Corollary 1 (Preference over Extension). Let $P = (A, <_P)$ and $A \subseteq \text{attrib}(R)$, then we have:

$$\sigma[P](R \text{ext}_f S) = \sigma[P](R) \text{ext}_f S$$

Proof.

$$\begin{aligned}
& \sigma[P](R \text{ext}_f S) \\
&= \{w \in R \text{ext}_f S \mid \nexists v \in R \text{ext}_f S : w <_P v\} \\
&= \{ \{ P \text{ operates only on } R \} \\
&\quad \{w \in R \text{ext}_f S \mid \nexists v \in R : w <_P v\} \\
&= \{ \{ f \text{ is left-total} \} \\
&\quad \{w \in R \mid \nexists v \in R : w <_P v\} \text{ext}_f S \\
&= \sigma[P](R) \text{ext}_f S \quad \square
\end{aligned}$$

To simplify an evaluation of a preference over a join we can use the following Lemma. Thereby we declare $e(X) \Leftrightarrow R.X = S.X$ for the join condition on X of two relations R and S .

Lemma 7 (Push Preference over Join). Let $P = (A, <_P)$, $A \subseteq \text{attrib}(R)$, $X \subseteq \text{attrib}(R) \cap \text{attrib}(S)$. Then we have:

- a) $\sigma[P](R \bowtie_{e(X)} S) = \sigma[P](R) \bowtie_{e(X)} S$
if each tuple in R has at least one join partner in S , e.g., if X is a foreign key from R to S .
- b) $\sigma[P](R \bowtie_{e(X)} S) = \sigma[P](R \bowtie_{e(X)} S) \bowtie_{e(X)} S$

Proof. Let

$$j(r, s) = \begin{cases} \text{true} & \text{if } r[X] = s[X] \\ \text{false} & \text{otherwise} \end{cases}$$

- a) Note that since each tuple in R has a join partner in S , the join can be rewritten in terms of an extension function.

$$\begin{aligned} & \sigma[P](R \bowtie_{e(X)} S) \\ &= \sigma[P](R \text{ ext}_j S) \\ &= \llbracket \text{Corollary 1} \rrbracket \\ & \sigma[P](R) \text{ ext}_j S \\ &= \sigma[P](R) \bowtie_{e(X)} S \end{aligned}$$

$$\begin{aligned} \text{b)} \quad & \sigma[P](R \bowtie_{e(X)} S) \\ &= \sigma[P]((R \ltimes_{e(X)} S) \bowtie_{e(X)} S) \\ &= \llbracket j \text{ is left-total on the semi join } (R \ltimes_{e(X)} S) \rrbracket \\ & \sigma[P]((R \ltimes_{e(X)} S) \text{ ext}_j S) \\ &= \llbracket \text{Corollary 1} \rrbracket \\ & \sigma[P](R \ltimes_{e(X)} S) \text{ ext}_j S \\ &= \sigma[P](R \ltimes_{e(X)} S) \bowtie_{e(X)} S \end{aligned} \quad \square$$

To rewrite a preference evaluation on a semi-join in connection with a normal join we have the following lemma:

Lemma 8. *Let $P = (A, <_P)$, $A \subseteq \text{attrib}(R) \cap \text{attrib}(S)$. Then we have:*

$$\sigma[P \text{ grouping } X](R \ltimes_{e(X)} S) \bowtie_{e(X)} S = \sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S$$

Proof. Let $T := R \ltimes_{e(X)} S$. At first we show

$$\sigma[P \text{ grouping } X](T) = \sigma[P \text{ grouping } X](R) \cap T \quad (\text{h1})$$

by

$$\begin{aligned} & \sigma[P \text{ grouping } X](T) \\ &= \{w \in T \mid \exists v \in T : w <_P v \wedge w[X] = v[X]\} \\ &= \llbracket P \text{ operates only on } R \rrbracket \\ & \{w \in R \mid \exists v \in R : w <_P v \wedge w[X] = v[X] \wedge v, w \in T\} \\ &= \llbracket w[X] = v[X] \wedge w \in T \text{ implies } v \in T \rrbracket \\ & \{w \in R \mid \exists v \in R : w <_P v \wedge w[X] = v[X] \wedge w \in T\} \\ &= \{w \in R \mid \exists v \in R : w <_P v \wedge w[X] = v[X]\} \cap T \\ &= \sigma[P \text{ grouping } X](R) \cap T \end{aligned}$$

Now we show the claim by:

$$\begin{aligned}
& \sigma[P \text{ grouping } X](R \bowtie_{e(X)} S) \bowtie_{e(X)} S \\
= & \quad \{ \{ \text{Equation (h1)} \} \} \\
& (\sigma[P \text{ grouping } X](R) \cap (R \bowtie_{e(X)} S)) \bowtie_{e(X)} S \\
= & \quad \{ \{ \bowtie \text{ distributes over } \cap \} \} \\
& (\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S) \cap ((R \bowtie_{e(X)} S) \bowtie_{e(X)} S) \\
= & (\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S) \cap (R \bowtie_{e(X)} S) \\
= & \sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S
\end{aligned}$$

□

5.3 Optimization Theorems for Grouping

Remember that we write $e(X)$ instead of $R.X = S.X$. First we present a lemma which is necessary for further proofs: a preference which is grouped w.r.t. to the join attribute can be directly pushed over a join.

Lemma 9 (Push Grouped Preference over Join). *Let $P = (\langle P, A \rangle)$ and $A \subseteq \text{attrib}(R)$, $X \subseteq \text{attrib}(R) \cap \text{attrib}(S)$. Then we have:*

$$\sigma[P \text{ grouping } X](R \bowtie_{e(X)} S) = \sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S$$

Proof. Let $T := R \bowtie_{e(X)} S$

$$\begin{aligned}
& \sigma[P \text{ grouping } X](R \bowtie_{e(X)} S) \\
= & \{ w \in T \mid \exists v \in T : w \langle_P v \wedge w[X] = v[X] \} \\
= & \{ w \in R \mid \exists v \in R : w \langle_P v \wedge w[X] = v[X] \wedge v, w \in R \bowtie_{e(X)} S \} \bowtie_{e(X)} S \\
= & \quad \{ \{ v \in R \bowtie_{e(X)} S \wedge w[X] = v[X] \text{ implies } v \in R \bowtie_{e(X)} S \} \} \\
& \{ w \in R \mid \exists v \in R : w \langle_P v \wedge w[X] = v[X] \wedge w \in R \bowtie_{e(X)} S \} \bowtie_{e(X)} S \\
= & \quad \{ \{ w \notin R \bowtie_{e(X)} S \text{ implies } w \notin \{w\} \bowtie_{e(X)} S \} \} \\
& \{ w \in R \mid \exists v \in R : w \langle_P v \wedge w[X] = v[X] \} \bowtie_{e(X)} S \\
= & \sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S
\end{aligned}$$

□

Theorem 1 (Push Preference Over Join). *For a preference $P = (A, \langle_P)$ with $A \subseteq \text{attrib}(R)$ and $X \subseteq \text{attrib}(R) \cap \text{attrib}(S)$ the following holds:*

a) $\sigma[P](R \bowtie_{e(X)} S) = \sigma[P](\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S)$

b) *For an SV relation (\cong_B, B) with $B \subseteq \text{attrib}(S)$ we have:*

$$\sigma[P \text{ grouping } \cong_B](R \bowtie_{e(X)} S) = \sigma[P \text{ grouping } \cong_B](\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S)$$

$$\begin{aligned}
\text{Proof. a) } & \sigma[P](R \bowtie_{e(X)} S) \\
& = \{ \text{Lemma 5 b) using } P \text{ grouping } R.X = \text{id}_{R.X} \blacklozenge P \subseteq P \} \\
& \quad \sigma[P](\sigma[P \text{ grouping } R.X](R \bowtie_{e(X)} S)) \\
& = \{ \text{Lemma 9} \} \\
& \quad \sigma[P](\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S)
\end{aligned}$$

b) Let $S = \{s_1, \dots, s_n\}$

$$\begin{aligned}
& \sigma[P \text{ grouping } \cong_B](R \bowtie_{e(X)} S) \\
& = \sigma[P \text{ grouping } \cong_B] \left(\bigcup_{i=1}^n (R \bowtie_{e(X)} \{s_i\}) \right) \\
& = \{ \text{Lemma 6} \} \\
& \quad \sigma[P \text{ grouping } \cong_B] \left(\bigcup_{i=1}^n \sigma[P \text{ grouping } \cong_B](R \bowtie_{e(X)} \{s_i\}) \right) \\
& = \{ \text{Since } B \in \text{attrib}(S): v \in R \bowtie_{e(X)} \{s_i\} \text{ implies } v \cong_B s_i \} \\
& \quad \sigma[P \text{ grouping } \cong_B] \left(\bigcup_{i=1}^n \sigma[P](R \bowtie_{e(X)} \{s_i\}) \right) \\
& = \{ v \in R \bowtie_{e(X)} \{s_i\} \text{ implies } v[X] = s_i[X] \} \\
& \quad \sigma[P \text{ grouping } \cong_B] \left(\bigcup_{i=1}^n \sigma[P \text{ grouping } X](R \bowtie_{e(X)} \{s_i\}) \right) \\
& = \{ \text{Lemma 9} \} \\
& \quad \sigma[P \text{ grouping } \cong_B] \left(\bigcup_{i=1}^n (\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} \{s_i\}) \right) \\
& = \sigma[P \text{ grouping } \cong_B](\sigma[P \text{ grouping } X](R) \bowtie_{e(X)} S)
\end{aligned}$$

□

Theorem 2 (Split Pareto and Push Over Join). *Let $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$ with $A_1 \subseteq \text{attrib}(R)$, $A_2 \subseteq \text{attrib}(S)$ and $X \subseteq \text{attrib}(R) \cup \text{attrib}(S)$. Then:*

$$\begin{aligned}
a) & \sigma[P_1 \otimes P_2](R \bowtie_{e(X)} S) = \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} S) \\
b) & \sigma[P_1 \otimes P_2](R \bowtie_{e(X)} S) \\
& \quad = \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} \sigma[P_2 \text{ grouping } X](S)) \\
c) & \sigma[P_1 \otimes P_2](R \bowtie_{e(X)} S) = \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R \bowtie_{e(X)} S) \bowtie_{e(X)} S)
\end{aligned}$$

$$\begin{aligned}
\text{Proof. a) } & \sigma[P_1 \otimes P_2](R \bowtie_{e(X)} S) \\
& = \{ \text{Lemma 5 b) together with Lemma 2 c)} \} \\
& \quad \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } \cong_{P_2}](R \bowtie_{e(X)} S)) \\
& = \{ \text{Theorem 1 b)} \} \\
& \quad \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } \cong_{P_2}](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} S))
\end{aligned}$$

$$= \quad \{ \text{Again Lemma 5 b) together with Lemma 2 c) } \}$$

$$\sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} S)$$

b) Follows from a) and an symmetric argument of Theorem 1 b)

$$\text{c) } \quad \sigma[P_1 \otimes P_2](R \bowtie_{e(X)} S)$$

$$= \quad \{ \text{Part a) } \}$$

$$\sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} S)$$

$$= \quad \{ \text{Lemma 8 } \}$$

$$\sigma[P_1 \otimes P_2](\sigma[P_1 \text{ grouping } X](R \bowtie_{e(X)} S) \bowtie_{e(X)} S)$$

□

Subsequent we present a simplification of a Prioritization into a grouping expression.

Theorem 3 (Split Prioritization Into Grouping). *Let $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$ two preferences with $A_1, A_2 \subseteq \text{attrib}(R)$ and SV-semantics \cong_{P_1} and \cong_{P_2} . Then:*

$$\text{a) } \sigma[P_1 \& P_2](R) = \sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R))$$

$$\text{b) } \sigma[P_1 \& P_2](R) = \sigma[P_1](\sigma[P_2 \text{ grouping } \cong_{P_1}](R))$$

Proof. $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R))$

$$= \{w \in \sigma[P_1](R) \mid \nexists v \in \sigma[P_1](R) : w \cong_{P_1} v \wedge w <_{P_2} v\}$$

$$= \{w \in R \mid w \in \sigma[P_1](R) \wedge \nexists v \in R : v \in \sigma[P_1](R) \wedge w \cong_{P_1} v \wedge w <_{P_2} v\}$$

$$= \quad \{ \{ w \in \sigma[P_1](R) \wedge w \cong_{P_1} v \text{ implies } v \in \sigma[P_1](R) \} \}$$

$$\quad \{w \in R \mid w \in \sigma[P_1](R) \wedge \nexists v \in R : w \cong_{P_1} v \wedge w <_{P_2} v\}$$

$$= \sigma[P_1](R) \cap \{w \in R \mid \nexists w \in R : w \cong_{P_1} v \wedge w <_{P_2} v\}$$

$$= \sigma[P_1](R) \cap \sigma[P_2 \text{ grouping } \cong_{P_1}](R)$$

$$= \quad \{ \text{Lemma 4} \}$$

$$\sigma[P_1 + (P_2 \text{ grouping } \cong_{P_1})](R)$$

$$= \quad \{ \{ P_1 + (P_2 \text{ grouping } \cong_{P_1}) = P_1 + \cong_{P_1} \blacklozenge P_2 = P_1 \& P_2 \} \}$$

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

This shows part a) and an analogous argument holds for part b). □

The next theorem splits Prioritization and pushes the preferences over the join. Note that the following rules depend on the used SV-semantics.

Theorem 4 (Split Prioritization and Push Over Join). *Let $P_1 = (A_1, <_{P_1})$ and $P_2 = (A_2, <_{P_2})$ preferences with $A_1 \subseteq \text{attrib}(R)$, $A_2 \subseteq \text{attrib}(R) \cup \text{attrib}(S)$, $X \subseteq \text{attrib}(R) \cap \text{attrib}(S)$. Then*

- a) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S) = \sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R) \bowtie_{e(X)} S)$
if each tuple in R has at least one join partner in S , e.g., if X is a foreign key from R to S .
- b) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S) = \sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R \bowtie_{e(X)} S) \bowtie_{e(X)} S)$
- c) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S) = \sigma[P_1 \& P_2](\sigma[P_1 \text{ grouping } X](R) \bowtie_{e(X)} S)$

Proof.

- a) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S)$
 $= \{ \text{Theorem 3} \}$
 $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R \bowtie_{e(X)} S))$
 $= \{ \text{Lemma 7 b) (since tuples in } R \text{ have join partners in } S) \}$
 $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R) \bowtie_{e(X)} S)$
- b) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S)$
 $= \{ \text{Theorem 3} \}$
 $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R \bowtie_{e(X)} S))$
 $= \{ \text{Lemma 7 a) } \}$
 $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R \bowtie_{e(X)} S) \bowtie_{e(X)} S)$
- c) $\sigma[P_1 \& P_2](R \bowtie_{e(X)} S)$
 $= \{ \text{Lemma 5 b) using } P_1 \& P_2 \subseteq P_1 \subseteq P_1 \blacklozenge \text{id}_X \}$
 $\sigma[P_1 \& P_2](\sigma[P_1 \text{ grouping } X](R \bowtie_{e(X)} S))$
 $= \{ \text{Lemma 9} \}$
 $\sigma[P_2 \text{ grouping } \cong_{P_1}](\sigma[P_1](R \bowtie_{e(X)} S) \bowtie_{e(X)} S)$

□

Example 8. Consider the Prioritization $P := P_1 \& P_2$ as

$$P = \text{POS}(\text{make}, \{ \text{'Audi'}, \text{'BMW'} \}, =_{P_1}) \& \text{AROUND}_5(\text{power}, 170, \sim_{P_2})$$

where P_1 has *trivial* and P_2 has *regular SV*-semantics.

The preference selection $\sigma[P_1 \& P_2](\text{car})$ on our sample dataset (Table 1) results in the tuple $\{(\text{'BMW'}, 180), (\text{'Audi'}, 170)\}$, since *BMW* and *Audi* are not substitutable concerning P_1 having *trivial SV*-semantics.

Now, if we apply Theorem 3a), we first evaluate $\sigma[P_1](\text{cars})$ to $\text{tmp} := \{(\text{'BMW'}, 180), (\text{'BMW'}, 230), (\text{'Audi'}, 170)\}$ and afterwards we compute

$$\sigma[P_2 \text{ grouping } \cong_{P_1}](\text{tmp}),$$

that means we group the results in *tmp* using trivial SV-semantic from P_1 (we get groups on BMW and Audi) and evaluate P_2 on these groups. This leads to the same result as above: $\{('BMW', 180), ('Audi', 170)\}$. Here, P_1 acts as pre-filter preference, cp. [4].

Theorem 5 (Simplify Grouped Preference Selection). *Let $P = (A, <_P)$, $P_1 = (A_1, <_{P_1})$, $P_2 = (A_2, <_{P_2})$ be preferences with $A, A_1, A_2 \subseteq \text{attrib}(R)$ and \cong_1, \cong_2 be equivalence relations. Then the following holds:*

- a) $\sigma[(P \text{ grouping } \cong_1) \text{ grouping } \cong_2](R) = \sigma[P \text{ grouping } (\cong_1 \blacklozenge \cong_2)](R)$
- b) $\sigma[(P_1 \text{ grouping } \cong_1) \& P_2](R) = \sigma[(P_1 \& P_2) \text{ grouping } \cong_1](R)$
- c) $\sigma[P_1 \otimes (P_2 \text{ grouping } \cong_1)](R) = \sigma[(P_1 \otimes P_2) \text{ grouping } \cong_1](R)$
- d) $\sigma[(P_1 \text{ grouping } \cong_1) \otimes (P_2 \text{ grouping } \cong_2)](R) = \sigma[(P_1 \otimes P_2) \text{ grouping } (\cong_1 \blacklozenge \cong_2)](R)$
- e) $\sigma[P \text{ grouping } B](R) = R$ where $B \in \text{attrib}(R)$ is unique for all tuples in R

Proof. Parts a-d) follow immediately from Lemma 3. For part e) we infer:

$$\begin{aligned}
& \sigma[P \text{ grouping } B](R) \\
&= \{w \in R \mid \nexists v \in R : w[X] = v[X] \wedge w <_P v\} \\
&= \{ \{ w[X] = v[X] \text{ implies } w = v \text{ since } X \text{ is unique} \} \\
&\quad \{w \in R \mid \neg(w <_P w)\} \\
&= \{ \{ w <_P w \text{ is always false since } P \text{ is irreflexive} \} \\
&\quad R
\end{aligned}$$

□

6 Performance Benchmarks

In this section we discuss the application of our algebraic optimization laws by giving a implementation sketch and a comprehensive benchmark.

6.1 Integration into Preference SQL

Because *preference relational algebra* extends *relational algebra*, it is possible to construct a preference query optimizer as an extension of a classical relational query optimizer [6, 3]. Importantly, one can inherit all familiar laws from relational algebra given by [14]. Thus, well-established heuristics can be applied aiming to reduce the sizes of intermediate relations, e.g. *Push Hard Selection* and *Push Projection*.

Preference SQL implements a rule based query optimizer based on the Hill Climbing approach from [14] and the preference transformation laws from [6].

Expanding this repertoire of optimization techniques by our new *grouped preferences theorems* raises the issue how to integrate them into a query optimizer. Finding a good ordering of transformation is as usual a difficult, heuristic task, since the application of a rule can depend on the previous application of other rules. Most of our optimization laws on grouped preferences can only be applied if the join operator have already been generated. Due to this knowledge it seems to be adequate for the optimization of preference relational algebra statements if our grouped transformation laws are applied after *Push Hard Selections* as well as *Combine Hard Selections and Cartesian Products into Joins*. Afterwards the operator tree can be transformed using our novel grouped preference optimization laws. For more details on the integration and ordering of preference transformation laws we refer to [6].

6.2 Benchmark Framework

For the benchmarks we used our *Preference SQL* system [9], a Java SE 6 framework for preference queries on conventional database systems. Preference SQL is a middleware which parses the query and performs query optimization as described in [3, 6, 4] and in Section 5. Furthermore, we implemented several preference evaluation algorithms, e.g. BNL [2], Hexagon [13], and LESS [5]. Note that until now Preference SQL does not support index based preference algorithms. The implementation of the GROUPING evaluation is based on hash buckets. For further details we refer to [9].

For the benchmarks we used the TPC-H dataset designed for decision support system queries [1]. We generated several preference queries to present the advantage of our optimization laws from Section 5. All experiments are performed on a 2.53GHz Intel Xeon machine running Linux with 8 GB RAM for the JVM. We used a PostgreSQL 8.4 database to store all generated data tuples.

In our framework a query is mapped onto its initial operator tree \mathbf{T}_{init} . This initial tree will be transformed in a **standard optimized tree** \mathbf{T}_{std} using the laws from [6, 3], this means for example *Push Preference* or *Push Selection*. Additionally, we use our optimization laws from Section 5 concerning the GROUPING laws to transform \mathbf{T}_{std} into the **final operator tree** $\mathbf{T}_{\text{grouping}}$.

In our performance evaluation we carried out the following performance measurements for a query Q (in **seconds**):

- Runtime t_{init} for evaluating \mathbf{T}_{init}
- Runtime t_{std} for evaluating \mathbf{T}_{std}
- Runtime t_{grouping} for evaluating $\mathbf{T}_{\text{grouping}}$

We used 7 iterations for each query and skipped the best and the worst value. From the remaining runtimes we computed the average value.

As an indicator for the optimization impact of our new laws we choose the **speedup factor**

$$\text{Speedup factor} := t_{\text{std}} / t_{\text{grouping}}$$

In the **TPC-H** benchmark dataset we used different scaling factors: 0.001, 0.01, 0.1, 0.5, 1, i.e., **1 MB**, **10 MB**, **100 MB**, **500 MB** and **1 GB** of data. Additionally, we present the **BMO-size** of each query.

6.3 Benchmark Queries

We now present our benchmarks to demonstrate the advantage of our algebraic optimization techniques.

Benchmark Theorem 1a) (*Push Preference Over Join*)

In our first benchmark we apply Theorem 1a) to the following Preference SQL query.

```
-- Query for Theorem 1a)
SELECT p_partkey, p_size, ps_supplycost
FROM partsupp, part
WHERE ps_partkey = p_partkey
PREFERRING ps_supplycost BETWEEN 300, 400, 50 REGULAR;
```

This is a typical query with a preference on a join. The query retrieves all parts (`p_partkey`) with its size (`p := p_size`) and the supplier costs (`ps := ps_supplycost`) from the relations `partsupp` and `part`. The preference

$$P := \text{BETWEEN}_{50}(\text{ps_supplycost}, 300, 400, \sim_P)$$

has *regular* SV-semantics. The algebraic form is given as:

$$\sigma[P](\text{ps} \bowtie_{\text{ps_partkey} = \text{p_partkey}} \text{p}) \quad (2)$$

where `ps` is an alias for `ps_supplycost` and `p` corresponds to `part`. Note that we omit the projection π to simplify the algebraic laws. Applying Theorem 1a) to Equation (2) the operator tree in Figure 4 will be produced.

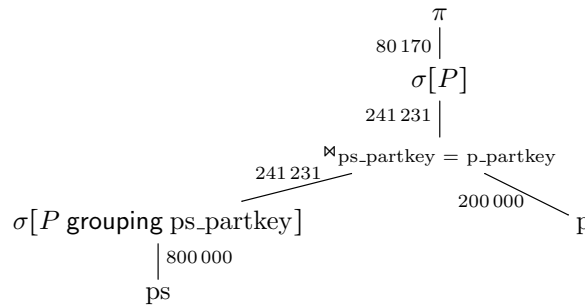


Figure 4: Operator tree T_{grouping} for query 1a).

Table 3 shows the performance results for the 1 MB, 10 MB, 100MB, 500 MB and 1 GB dataset. In the 1 MB dataset we have no advantage of our grouping optimization T_{grouping} in comparison to the standard optimization T_{std} due to the costly grouping operation. However, if the dataset grows, the additional grouping operation $\sigma[P \text{ grouping ps_partkey}]$ acts as a strong pre-filter ([4]) and reduces the intermediate result sizes. For example, in the 1 GB dataset the evaluation of the initial operator tree T_{init} takes about 10 hours, whereas the standard optimization techniques like *Push Projection* and *Push Selection* in T_{std} leads to a runtime of just one minute. The T_{grouping} operator tree can be evaluated 15 times faster and clearly outperforms T_{std} .

Table 3: Results for query 1a).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	92	782	7980	40151	80170
t_{init} in sec	0.10	2.56	374.65	8524.67	36188.75
t_{std} in sec	0.09	0.25	9.43	24.79	69.47
t_{grouping} in sec	0.09	0.06	1.94	1.77	4.51
Speedup factor	1.0	4.12	4.86	14.01	15.40

Benchmark for Theorem 1b) (*Push Preference Over Join*)

In this benchmark we apply Theorem 1b), *Push Preference Over Join*, to the following Preference SQL query.

```
-- Query Theorem 1b):
SELECT p_partkey, p_size, ps_supplycost
FROM partsupp, part
WHERE ps_partkey = p_partkey
PREFERRING ps_supplycost BETWEEN 200, 400, 100 REGULAR
GROUPING p_brand;
```

The query retrieves all parts ($p_partkey, p_size, ps_supplycost$) where the supplier costs ($ps_supplycost$) should be between 200 and 400 ($d=100$), i.e. using a preference constructor we write

$$P := \text{BETWEEN}_{100}(ps_supplycost, 200, 400, \sim P)$$

with *regular* SV-semantics. Note that in contrast to last benchmark this preference is evaluated on the groups of p_brand . The algebraic form is given by

$$\sigma[P \text{ grouping ps_brand}](ps \bowtie_{ps_partkey = p_partkey} P) \quad (3)$$

Applying Theorem 1b) we get the optimized operator tree (without *Push Projection*) in Figure 5. Table 4 shows the performance results of our optimization.

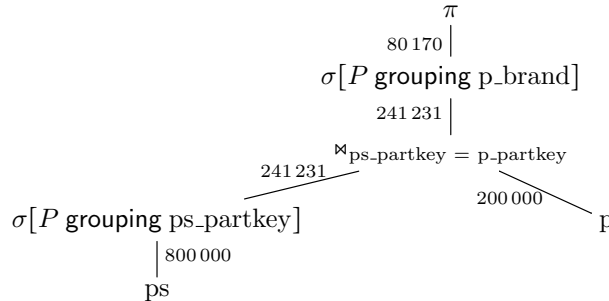


Figure 5: Operator tree T_{grouping} for query 1b).

Table 4: Results for query 1b).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	94	782	7980	40151	80170
t_{init} in sec	0.09	2.54	403.62	8631.05	> 10h
t_{std} in sec	0.09	0.09	8.31	77.02	188.38
t_{grouping} in sec	0.08	0.02	2.33	12.90	47.34
Speedup factor	1.13	4.5	3.57	5.97	3.98

Again, in the 1 MB dataset we only have a small speedup due to the costly grouping operation in comparison to the data size. However, in the 100 MB dataset the evaluation of the standard optimized tree T_{std} takes about 8.31 seconds, whereas our grouped optimization T_{grouping} leads to a runtime of 2.33 seconds, this is a speedup more than 3.5. The runtime for the unoptimized operator tree T_{init} in the 1 GB dataset is more than 10 hours. Furthermore, the grouped operator tree T_{grouping} in the 1 GB dataset is about 4 times faster than the standard optimization T_{std} . This is a strong evidence that the heuristics of *Push Grouped Preference* reduce intermediate results. This is also verified by considering the intermediate number of tuples in the operator tree. In the original preference query (3) the join produces 800 000 tuples (on the 1 GB dataset) which must be compared by the grouped preference selection. On the other hand, the reduction advantage based on our optimization is shown in Figure 5, where the tree is annotated with the number of intermediate results. After evaluating the left branch of the tree which contains the grouped preference on partsupp there are only 241 231 tuples left. This leads to 241 231 tuples after the join¹. Therefore the grouped preference selection $\sigma[P \text{ grouping } p_brand]$ at the top must be applied only on a fraction of the original join tuples leading

¹Note that $p_partkey$ is a primary key, whereas $ps_partkey$ is a foreign key to $p_partkey$.

to a BMO-set of 80 170 tuples in about 4 seconds. Furthermore, one may note that the speedup factor is not constant but depends on the data size. This is due to the different group sizes for the grouped preference selections.

Benchmark for Theorem 2b) (*Split Pareto and Push Over Join*)

Since Theorem 2a) is a special case of Theorem 2b), we only sum up the performance results for the latter one. We used the following query to demonstrate the advantage of *Split Pareto and Push Over Join*.

```
-- Query Theorem 2b):
SELECT p_partkey, p_size, ps_supplycost
FROM partsupp, part
WHERE ps_partkey = p_partkey
PREFERRING ps_supplycost LOWEST 0, 100 REGULAR
AND p_size HIGHEST 50, 5 REGULAR;
```

The query consists of a Pareto preference based on two preferences

$$P_1 := \text{LOWEST}_{100}(\text{ps_supplycost}, \text{inf} = 0, \sim_{P_1})$$

and

$$P_2 := \text{HIGHEST}_5(\text{p_size}, \text{sup} = 50, \sim_{P_2})$$

both having regular SV-semantics. The query joins the relations `part` and `partsupp` on the attribute `partkey` and evaluates the Pareto preference. Note, this is similar to a traditional Skyline query. Figure 6 shows the standard operator tree for the preference query above.

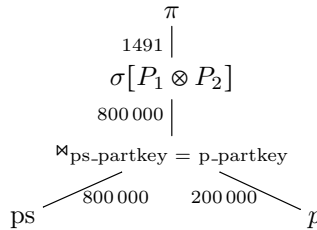


Figure 6: Operator tree T_{std} for query 2b).

Optimizing the operator tree in Figure 6 using Theorem 2b) leads to the optimized operator tree T_{grouping} in Figure 7.

The performance results can be found in Table 5. The evaluation of the grouping optimized tree T_{grouping} in the 1 MB dataset is slower than T_{init} or T_{std} because of the costly grouping operations in both branches of the join, cp. Figure 7. The speedup for the 10 MB dataset is not noteworthy. However, for larger datasets we have a speedup up to 11 because the induced grouping preferences act as pre-filters and reduce intermediate result sizes.

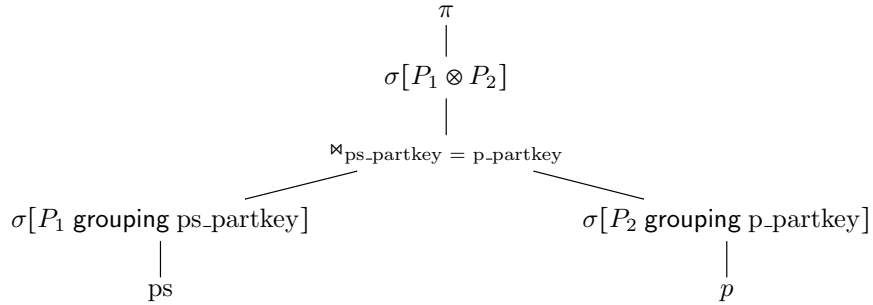


Figure 7: Operator tree T_{grouping} for query 2b).

Table 5: Results for query 2b).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	9	16	161	1900	1491
t_{init} in sec	0.08	2.65	400.82	8392.19	> 10h
t_{std} in sec	0.06	0.27	25.71	668.9	1072.82
t_{grouping} in sec	0.16	0.22	3.6	60.57	176.4
Speedup factor	0.38	1.23	7.14	11.04	6.08

Benchmark for Theorem 2c) (*Split Pareto and Push Over Join*)

In this benchmark we used the following query.

```
-- Query Theorem 2c):
SELECT p_partkey, p_size, ps_supplycost
FROM partsupp, part
WHERE ps_partkey = p_partkey AND
      p_size <= 20
PREFERRING ps_supplycost LOWEST 0, 100 REGULAR
      AND p_size HIGHEST 50, 5 REGULAR;
```

The query is the same as in the last benchmark, however the additional filter $p_size \leq 20$ reduces the size of the relation `part`. Figure 8 shows the standard optimization of the preference query using *Push Hard Selection*.

In Figure 9 the Pareto preference is split by Theorem 2c), *Split Pareto and Push Over Join* and pushed over the join using a grouped preference and a semi-join on the attribute `partkey`. In both operator trees the edges show the number of intermediate results for the 1 GB dataset.

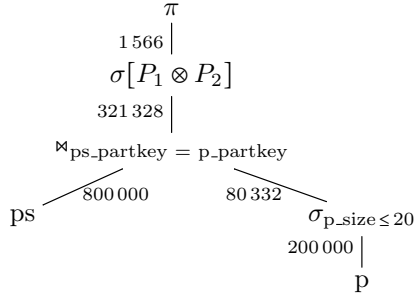


Figure 8: Operator tree T_{std} for query 2c).

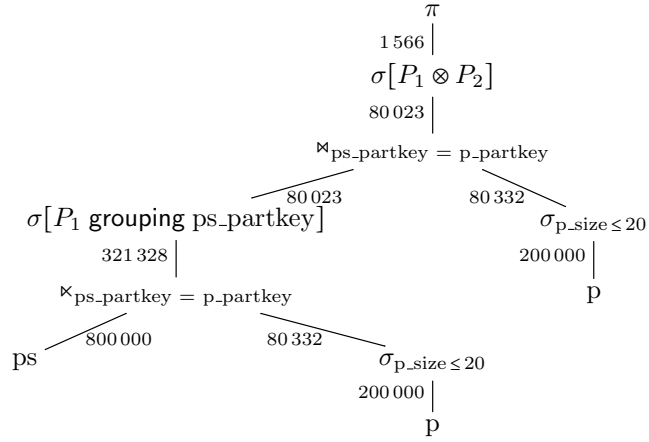


Figure 9: Operator tree $T_{grouping}$ for query 2c).

The net effect is a sizeable performance gain as indicated by the speedup factor, cp. Table 6. The grouped preference selection after the semi-join reduces the intermediate result such that the equi-join and the Pareto preference can be computed efficiently. Again, T_{init} takes several hours for the 1 GB dataset.

Table 6: Results for query 2c).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	5	33	182	1961	1566
t_{init} in sec	0.69	3.81	507.62	11739.65	> 10h
t_{std} in sec	0.23	1.12	27.50	137.86	1007.79
t_{grouping} in sec	0.11	0.23	9.31	35.22	287.36
Speedup factor	2.09	4.87	2.95	3.91	3.51

Benchmark Theorem 3a) (*Split Prioritization Into Grouping*)

This benchmark demonstrates the advantage of Theorem 3a) on the subsequent query. In this case the preference on `l_linestatus` acts as a pre-filter preference and afterwards the HIGHEST preference is evaluated on the intermediate result.

```
-- Query Theorem 3a):  
SELECT * FROM lineitem  
PREFERRING l_linestatus IN ('O')  
  PRIOR TO l_discount HIGHEST 0.1, 0.02 REGULAR;
```

We omit the operator tree, but show the performance results in Table 7. Up to the 100 MB dataset we have only a small speedup. However, on the 500 MB and 1 GB dataset we have a speedup factor of 4 and therefore the query can be evaluated in less than 2 minutes.

Table 7: Results for query 3a).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	256	2733	9200	10000	9200
t _{init} in sec	0.15	0.91	36.06	> 5h	> 10h
t _{std} in sec	0.13	0.84	9.41	222.95	480.31
t _{grouping} in sec	0.09	0.71	6.81	54.28	121.28
Speedup factor	1.44	1.18	1.38	4.11	3.96

Benchmark Theorem 4a) (*Split Prioritization and Push Over Join*)

This query demonstrates Theorem 4a), *Split Prioritization and Push Over Join*, where a foreign key from `ps_partkey` to `p_partkey` in the relation `part` is available, cp. [1]. This typically occurs in real world databases.

```
-- Query Theorem 4a):  
SELECT p_partkey, p_size, ps_supplycost  
FROM partsupp, part  
WHERE ps_partkey = p_partkey AND  
  p_size <= 20  
PREFERRING ps_supplycost LOWEST 0, 100 REGULAR  
  PRIOR TO p_size HIGHEST 50, 5 REGULAR;
```

The standard optimized tree T_{std} is similar to the one given in Figure 8. Only the Pareto preference is substituted by the Prioritization P_1 & P_2 . Therefore, the annotation of the tree with the intermediate results for the 1 GB TPC-H dataset is nearly the same. Figure 10 shows the optimized tree $T_{grouping}$ after applying Theorem 4a) as well as the intermediate result sizes.

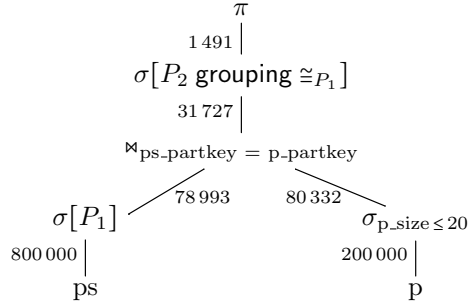


Figure 10: Operator tree T_{grouping} for query 4a).

Here P_1 acts as a pre-filter preference [4] to eliminate tuples before the costly join. This causes a faster preference selection and thus, leads to the runtimes in Table 8. Note that in the 100 MB, 500 MB and 1 GB dataset we have a speedup factor of over 17. This is because of the strong preference selection P_1 , e.g. from 800 000 tuples to 78 993 tuples in the 1 GB dataset, cp. Figure 10. In the 1 MB dataset we have no speedup. Here the additional costs for splitting the preference and the grouped preference evaluation are higher than the benefits of this optimization rule.

Table 8: Results for query 4a).

TPC-H	1 MB	10 MB	100 MB	500 MB	1 GB
BMO-size	9	16	161	1	1491
t_{init} in sec	0.09	2.56	505.23	> 5h	> 10h
t_{std} in sec	0.05	0.18	7.09	47.45	115.45
t_{grouping} in sec	0.07	0.06	0.31	2.64	4.67
Speedup factor	0.71	3.00	22.87	17.97	24.72

Benchmark 5 (*A Complex Preference Query*)

Our last benchmark shows a Pareto preference in combination with a Rank preference. The Rank preference weights AROUND with a weight of 2 and the HIGHEST preference with a weight of 0.5. Afterwards, the average function with a d-value of 25 is applied on the result of these weighted preferences.

```
-- Query 5:
SELECT COUNT(*)
FROM partsupp, part
WHERE ps_partkey = p_partkey
PREFERRING p_size LOWEST 0, 20 REGULAR
AND (ps_supplycost AROUND 200, 50 REGULAR |
ps_availqty HIGHEST 10000, 100 REGULAR)
RANK 'avgRankF' '2,0.5', 25
```

We omit the operator tree, but show the performance results in Table 9. Again, the evaluation of T_{init} for the 500 MB and 1 GB dataset takes several hours. Note that Theorem 2a) is applied: *Push Preference Over Join*.

Table 9: Results for query 5.

TPC-H	10 MB	100 MB	500 MB	1 GB
BMO-size	1	1	1	1
t_{init} in sec	2.36	346.6	> 5h	> 10h
t_{std} in sec	0.71	67.06	336.60	675.32
t_{grouping} in sec	0.27	12.32	39.87	89.92
Speedup factor	2.63	5.44	8.44	7.51

6.4 Observations and Discussion

To summarize our benchmarks experiences gained so far we can state that the application of the grouped preference optimization laws from Section 5 shows excellent performance advantages. The rules reduce intermediate result sizes in the evaluation and therefore speed up the computation of joins, Cartesian products, and last but not least the preference selection. One may note that a speedup factor up 4 is not good at all. However, we compared the grouped optimization rules with the standard preference optimization technique which already results in a high performance gain. Moreover, our grouping rules reduce intermediate results and therefore all test queries can be computed also in low memory environments.

In some cases the grouped preference optimization has no advantage in comparison to the standard optimization. For example, if the dataset is small (say 1 MB) the grouping operation is more costly than the evaluation without grouping. However, on large datasets the preference grouping optimization results in a high performance gain. Furthermore, if the preference grouping is based on a primary key, each tuple will form its own group. Then, the grouped preference computation does not eliminate any tuples from the dataset, thus does not reduce intermediate results. In this case Theorem 5e) can be applied to remove the unnecessary grouped preference selection from the operator tree.

7 Conclusion and Outlook

This work can be seen as an approach to merge the benefits of two different research lines. On the one hand we address the further development of preference evaluation algorithms by means of optimization theorems. Thereby we continue the work building up on [4, 7, 8, 10] and specially focus on grouping queries in this report.

On the other hand we developed a novel point-free algebraic calculus which allows for a more concise specification of preference constructs and shorter proofs (in [11, 12]). Applied to the theory of (SV-)grouping queries, we redefined the grouping-construct as the intersection of an SV-relation and a preference. Outgoing from this small formal improvement many simplifications arose in the specification and in several steps of the proofs.

Additionally we introduced the support of SV relations – originally introduced to be used in complex preferences in [8] – for grouping constructs. On the one hand there was a technical demand for this, as optimization theorems like “Split Prioritization into Grouping” shall be applicable to prioritizations with SV-semantics. With this we contribute to the field of algorithmic performance of preference evaluation. On the other hand we recognized the user-side benefits of an enriched specification of preferences queries. Thus we also allowed user-defined SV-relations for grouping queries, which primarily act as “syntactic sugar” for database queries in general by replacing the lengthy **CASE WHEN ... THEN ... END** statements of SQL. Beyond this they go hand in hand with the theory of SV semantics for preferences.

At the moment we do not have an adequate algebraic calculus which is able to handle theorems with database joins in a point-free fashion. This is a very sophisticated task as e.g., equijoins are point-to-point correspondences between data tables, hence they are somehow “inherently point-wise”. Nevertheless we are working on a concise algebraic axiomatisation of general join operations. In the algebraic aspects of the preference theory this is a major research challenge for us. It would also have an important practical impact because database queries containing joins offer a high potential for optimization by a plenty of “Push ... over join” operations.

This work is part of our approach to bring algebraic preference theory, query specification and preference algorithms more closely together; to make it easier to prove correctness of the evaluation algorithms, their used optimization theorems and to handle specification and implementation in a quite uniform way.

References

- [1] <http://www.tpc.org/tpch/spec/tpch2.15.0.pdf>, 2013.
- [2] 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.
- [3] 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.
- [4] M. Endres and W. Kießling. Semi-Skyline Optimization of Constrained Skyline Queries. In *ADC '11: Proceedings of the 22nd Australasian Database Conference*. Australian Computer Society, 2011.
- [5] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and Analyses for Maximal Vector Computation. *The VLDB Journal*, 16(1):5–28, 2007.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] W. Kießling, M. Endres, and F. Wenzel. The Preference SQL System - An Overview. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 34(2):11–18, 2011.
- [10] W. Kießling and B. Hafenrichter. Algebraic Optimization of Relational Preference Queries. Technical Report 2003-1, University of Augsburg.
- [11] B. Möller and P. Rooks. An algebra of layered complex preferences. In W. Kahl and T. Griffin, editors, *Relational and Algebraic Methods in Computer Science*, volume 7560 of *Lecture Notes in Computer Science*, pages 294–309. Springer Berlin Heidelberg, 2012.
- [12] B. Möller, P. Rooks, and M. Endres. An Algebraic Calculus of Database Preferences. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 241–262. Springer Berlin Heidelberg, 2012.

- [13] 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.
- [14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I + II. Computer Science Press, New York, NY, USA, 1988 / 1989.