Grouping Queries with SV-Semantics in Preference SQL

Markus Endres and **Patrick Roocks** and **Manuel Huber** and **Werner Kießling**
University of Augsburg
86135 Augsburg, Germany
{endres, roocks, kiessling}@informatik.uni-augsburg.de
manuel.huber@aisec.fraunhofer.de

Abstract

Preference database queries become more and more important in Data Warehousing or Decision Support Systems. In these environments the standard SQL **GROUP BY** operation with aggregate functions is extensively used in formulating queries. In this paper, we focus on the novel **GROUPING** functionality of Preference SQL which substantially extends the common aggregation features of SQL **GROUP BY**. It fully supports the Substitutable Values semantics for preference queries, enriched by comfortable aliasing mechanisms. This yields to an increased intuitive readability of complex queries, which in turn can reduce clearly the cognitive load for OLAP programmers. In addition we show how a correlated subquery in SQL can be written simpler and evaluated faster using the **GROUPING** clause.

Introduction

Database queries containing grouping constructs are highly important in applications like Business Intelligence, or Data Warehouses (Gupta, Harinarayan, and Quass 1995). Such Decision Support Systems use SQL aggregate functions and the **GROUP BY** operation extensively to formulate queries. For example, queries that create summary data are of great importance in such applications. These queries partition data into several groups (e.g., in business sectors) and aggregate on some attributes (e.g., sum of total sales).

Beyond this, preferences in databases – as shown by a recent survey (Stefanidis, Koutrika, and Pitoura 2011) – as well as preferences in artificial intelligence and social choice theory (Rossi, Venable, and Walsh 2011; Kaci 2011) are a well established framework to create personalized information systems. Moreover, preference database queries become more and more important in decision support environments (Golfarelli and Rizzi 2009), because they are an effective method to reduce very large datasets to a small set of highly interesting results. In general, a preference query selects those objects from the database that are not dominated by any others. Since operations like *grouping* and *aggregating* have proven to be very helpful for the analysis of large datasets and are essential in decision support systems, it seems to be a fruitful approach to extend preference queries by these concepts. Subsequently, we show a simple example of a preference database query using the *Preference SQL* language from (Kießling, Endres, and Wenzel 2011).

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Example 1. *The wish for a car having the highest power and a price around 35000 Euro for each group of registration year (reg_date) can be expressed in Preference SQL as:*

```
SELECT id, power, price, y
FROM car
PREFERRING
  power HIGHEST AND price AROUND 35000
  GROUPING EXTRACT(YEAR FROM reg_date) AS y
ORDER BY y;
```

Thereby **PREFERRING** *initiates the* preferring clause. *The keyword* **AND** *expresses a* Pareto composition *and states the equal importance of two preferences. The query returns all cars, for which no dominating cars w.r.t. the preference exist in the same year of registration.*

In our sample dataset in Table 1 there are three groups (built by the **GROUPING** *keyword), namely 2012, 2013, and NULL. In the year 2012 the tuples with ID 4 and 5 are dominated by tuple 3, because the latter has more power and is closer to 35000 Euros than the other two. In the year 2013 both tuples are incomparable as the tuple with ID 1 is cheaper and the tuple with ID 2 has more power.*

Table 1: Sample dataset of cars.

car	id	make	power	price	reg_date
	1	BMW	180	35000	2013-03-03
	2	Mercedes	200	38000	2013-02-15
	3	Mercedes	215	40000	2012-11-17
	4	Geely	120	22000	2012-10-03
	5	Geely	110	23000	2012-01-27
	6	Chery	120	25000	NULL

In this paper we generalize the concept of *grouping* in preference database queries by allowing *grouping w.r.t. arbitrary equivalence relations* instead of *single grouping attributes*. While for the latter the **GROUPING** construct works similar to the **GROUP BY** clause in standard SQL, grouping on *equivalence classes* is described with the *Substitutable Values* (SV) construct, where the partitions are induced by the equivalence classes of the attribute values.

Based on this, we extend the syntax of the Preference SQL query language and provide some common use cases. Furthermore, we show how *preference grouping* can be efficiently used to simplify *correlated nested queries*, and therefore reduce the computation costs of such queries.

Preferences in Database Systems

We follow the preference model from (Kießling 2005), where a preference $P = (A, <_P)$ is a strict partial order on the domain of A. Thereby A is a set of attributes. The term $\mathbf{x} <_\mathbf{P} \mathbf{y}$ is interpreted as "*I like y more than x*". Two tuples x and y are *indifferent*, if $\neg(x <_P y) \ \wedge \ \neg(y <_P x)$, i.e., neither x is better than y nor y is better than x. 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 $\sigma[P](R)$ (called *winnow* by (Chomicki 2003)) and finds all best matching tuples t for P, where $t.A$ is the projection to the attribute set A.

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

To specify a preference, a variety of intuitive preference constructors have been defined.

Base Preference Constructors

Preferences on single attributes are called *base preferences*. There are base preference constructors for *continuous* (*numerical*), *discrete* (*categorical*), *temporal*, and *spatial* domains. Most of them can be specified by a SCORE function $f_d : \mathrm{dom}(A) \to \mathbb{R}_0^+$, and some $d \in \mathbb{R}_0^+$ (Kießling 2005). In the case of $d = 0$ the function $f_d(v)$ 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 $f_d(v)$, 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.

There are several sub-constructors of SCORE$_d$: In the *numerical* AROUND$_d(A,\, z)$ the desired value should be z. If this is infeasible, values having the smallest distance to z are preferred, where the distance is discretized by the parameter d. The LOWEST$_d(A, \inf_A)$ constructor and the 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 $\mathrm{dom}(A)$. The *categorical base preference* POS(A, *POS-set*) expresses that a user has a set of preferred values, the positive *POS-set*. The negative preference NEG(A, *NEG-set*) is the counterpart to the POS preference. For the *temporal* preferences LATEST / EARLIEST, LATER / EARLIER THAN the desired date or time should be as late/early as possible or later/earlier than the denoted date or time.

Example 2. *The wish for a BMW or Geely is expressed with a positive preference:* $P_1 :=$ POS(make, {'BMW', 'Geely'}). *The preference selection* $\sigma[P_1]$(car) *on the car dataset in Table 1 leads to the cars with IDs* {*1,4,5*}.

If we prefer a horsepower around 170, where a difference of 5 does not matter, i.e., $d = 5$*, this can be expressed by* $P_2 :=$ AROUND$_5$(power, 170). *The result is given by the tuple 1 because its power of 180 hp is nearest to 170 hp.*

Complex Preference Constructors

Complex preferences determine the relative importance of preferences. Intuitively, people speak of "this preference is more important to me than that one" or "these preferences are all equally important". 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 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 happens 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 1 (SV-Relations). *Let* $P = (A, <_P)$ *be a preference, and A an attribute set. We define the following:*

a) A general SV-relation $(A, \cong)$ *on attribute set A, where* $\cong$ *has to be an equivalence relation on* $\mathrm{dom}(A)$.

b) A SV-relation $(A, \cong_P)$ *for a preference* $P = (A, <_P)$ *has to be compatible to P which means that we have for all* $x, y, z \in \mathrm{dom}(A)$*:*
- $x \cong_P y \ \Rightarrow \ \neg(x <_P y \vee y <_P x)$
- $(x <_P y \wedge y \cong_P z) \ \Rightarrow \ x <_P z$
- $(x \cong_P y \wedge y <_P z) \ \Rightarrow \ x <_P z$

Note that the attribute set is often omitted, i.e., we just write $\cong_P$ *for* $(A, \cong_P)$*. 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* $=_P := \mathrm{id}_A$.

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

Note that this definition of SV-semantics is required to preserve the strict order property of complex preferences, cp. (Kießling 2005). Within Preference SQL the use of trivial or regular SV-semantics can be specified with the keywords **`TRIVIAL`** and **`REGULAR`**.

Definition 2 (Pareto). *In a* Pareto preference $P := P_1 \otimes P_2 = (A_1 \times A_2, <_P)$ *all preferences* $P_i = (A_i, <_{P_i})$ *are of equal importance, i.e., for two tuples* $x = (x_1, x_2)$, $y = (y_1, y_2) \in \mathrm{dom}(A_1) \times \mathrm{dom}(A_2)$ *we define:*

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

Definition 3 (Prioritization). *In a* Prioritization preference $P := P_1 \,\&\, P_2$ *the preference* $P_1 = (A_1, <_{P_1})$ *is more important than* $P_2 = (A_2, <_{P_2})$*, i.e.*

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

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 3. *Reconsider the preferences* P_1 *and* P_2 *from Example 2. If both preferences are equally important and we use* trivial *SV-semantics for* P_1 *and* P_2 *we write*

$$P_t := \text{POS}(\text{make}, \{\text{'BMW'}, \text{'Geely'}\}, =_{P_1}) \otimes$$
$$\text{AROUND}_5(\text{power}, 170, =_{P_2})$$

Using the dataset given in Table 1 on page 1 the tuples ('BMW', 180) and ('Geely', 120) would be the best-matches for P_t*, although a horsepower of 180 is better w.r.t. the preference than 120. Due to the trivial SV-semantics BMW and Geely are* not *substitutable.*

Having regular *SV-semantics we write*

$$P_r := \text{POS}(\text{make}, \{\text{'BMW'}, \text{'Geely'}\}, \sim_{P_1}) \otimes$$
$$\text{AROUND}_5(\text{power}, 170, \sim_{P_2})$$

Here BMW and Geely become substitutable. Therefore ('Geely', 120) is equally good as ('BMW', 180) concerning the make, but 180 hp is better than 120 concerning the AROUND *preference. This means, ('BMW', 180) is the only tuple in the result set.*

Grouping Queries

To conjunct preferences and our target applications (e.g., business intelligence, decision support) it is highly important to define a grouping functionality for preferences. Therefore we define the grouped preference selection and the syntactical GROUPING schema for Preference SQL queries. Finally we present some use cases including a subquery optimization via preferences and GROUPING.

Grouped Preference Selection in Preference SQL

Basically, the grouping-construct allows us to split the dataset into several groups w.r.t. the grouping attributes. Afterwards the preference is evaluated on each group separately. The simplest grouping queries split the dataset according to distinct values of one ore more attributes. We also allow grouping w.r.t. to *equivalence relations* on attributes. Formally the grouped preference selection is described in the following definition.

Definition 4. *Let* $G \subseteq$ attrib(R) *and* $\cong_G$ *an equivalence relation on* $\text{dom}(G)$*, then the grouped preference selection w.r.t.* $\cong_G$ *is defined as:*

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

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, we refer to (Roocks, Endres, and Kießling 2013). According to (Kießling 2005) this can be expressed as a preference itself:

$$t <_{P \text{ grouping } G} t' \iff t <_P t' \;\wedge\; t \cong_G t'$$

Preference SQL (Kießling, Endres, and Wenzel 2011) is a declarative extension of 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. Syntactically, Preference SQL extends the **SELECT** statement of SQL by an optional **PREFERRING** clause, cp. Figure 1.

```
SELECT        ...  <projection, aggregation>
FROM          ...  <table_reference>
WHERE         ...  <hard_conditions>
 PREFERRING   ...  <soft_conditions>
 GROUPING     ...  <attribute_list>
  TOP         ...  <number>
  BUT ONLY    ...  <but_only_condition>
  HAVING      ...  <aggregating conditions>
ORDER BY      ...  <attribute_list>
```

Figure 1: Preference SQL query block with **GROUPING**.

The keywords **SELECT**, **FROM**, **WHERE**, and **ORDER BY** are treated as the standard SQL keywords. The **PREFERRING** clause specifies a preference which is evaluated separately on each of the groups induced by **GROUPING**. 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. If **TOP**-k is specified, only the k best tuples according to the preference order are returned. Preference SQL currently supports most of the SQL-92 standard (SQL-92 1992) as well as all preference constructors from (Kießling, Endres, and Wenzel 2011). Note that **TOP**, **BUT ONLY**, **HAVING**, and the use of *aggregation* is optional. Even **PREFERRING** is optional, which allows us to use Grouping-SV for conventional aggregations.

A *grouped preference query* is evaluated as follows:

1. The result of the **WHERE**-clause is partitioned according to the grouping attributes or their SV classes, respectively.
2. If a preference is given, the BMO-set is calculated separately for every group.
3. The additional **TOP**-k specifier selects the k best tuples according to the preference order per group. The maximum amount of tuples returned is $k \cdot g$, if g groups exist.
4. The **BUT ONLY** clause is applied, where all tuples are excluded which do not fulfill this hard selection (i.e. similar to **WHERE**, but *after* the preference selection).
5. The **HAVING** condition, e.g., **COUNT**`(*)>1`, is evaluated, i.e., those groups are excluded in which **HAVING** evaluated to false.
6. If an aggregation function occurs in the projection, it is applied to any group separately; hence only one line is returned per group in the result set.

Syntax for Grouping with Substitutable Values

Grouping on *distinct values of attributes $A, B, ...$* is specified with **GROUPING** `A, B,...` completely analogous to the **GROUP BY** clause in SQL. Grouping on *equivalence classes* is described with the construct **SV**, short for "Substitutable Values". This stems from the SV relations on preferences, introduced in (Kießling 2005) which behave similar to the Grouping-SV functionality. The equivalence classes forming the groups are from now on called *(grouping) SV classes*. The syntactic schema of a SV class' specification is as follows (where [...] indicates optional elements):

```
GROUPING
A SV (
   (A_1, A_2, ...) [AS 'class1_a'],
   (A_3, A_4, ...) [AS 'class2_a'], ...,
   [OTHERS [AS 'others_class_a']]) [AS sv_a],
B SV (
   (B_1, B_2, ...) [AS 'class1_b'], ...,
   [OTHERS [AS 'others_class_b']]) [AS sv_b],
   ...
```

Thereby the meaning of the syntactical parts is as follows:

- `A_`i and `B_`i represent domain values according to the attributes `A` and `B`.
- The domain values `A_1, A_2,...` are considered equal w.r.t. the SV relation $\cong_A$ for an 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,...` is the default name for a SV class with values `A_1, A_2,...`
- The keyword **OTHERS** puts all elements, which were not be mentioned before, in one "default" SV class. If **OTHERS** is not given, all tuples $t \in \text{dom}(\texttt{A}) \setminus \{$`A_1, A_2, ..., A_3, A_4,...`$\}$ form a SV class (t) of its own.

Note that the name of a SV-attribute (e.g. `sv_a`) is optional, but should be given to reference the SV grouping attributes in the projection. The original attributes `A, B,...` remain unchanged; they can still be referenced in the projection. Finally the entire term generates a SV relation $\cong_G$ where $G = \{A, B, ...\}$ are the grouping attributes. The handling of NULL values or NULL SV classes follows the specification in (Endres et al. 2012). The implementation of **GROUP BY** in standard database systems usually rely either on sorting or on hashing techniques. For the **GROUPING** of SV-equivalent values we use a combination of sets for the SV values and hashing techniques to perform an efficient grouping.

Use Cases

Subsequently we show some common use cases to demonstrate the expressive power of our preference grouping.

Example 4. *Consider the sample data set given in Table 1 on page 1. We want to retrieve the best cars for each country. Therefore we group all cars which are made in one country, i.e., Germany or China, together. This is done using the* **SV** *keyword. For every group we are interested in the cheapest cars. This is performed by a* LOWEST *preference in the following query:*

```
SELECT country, id, price FROM car
PREFERRING price LOWEST
GROUPING make SV (
  ('BMW', 'Mercedes') AS 'Germany',
  ('Geely', 'Chery')  AS 'China') AS country;
```

The result of this query on the given sample dataset is: $\{(\text{'Germany'}, 1, 35000), (\text{'China'}, 4, 22000)\}$.

In the next example we consider an aggregating query in combination with TOP-k.

Example 5. *We are interested in the average price of the two cheapest cars of every group of country. Therefore we state the following query:*

```
SELECT country, AVG(price)
FROM car
PREFERRING price LOWEST
GROUPING make SV (
  ('BMW', 'Mercedes') AS 'Germany',
  ('Geely', 'Chery')  AS 'China') AS country
TOP 2;
```

For the calculation of the average price the cars with IDs $\{1, 2, 4, 5\}$ *are considered. The final result of this query is* $\{(\text{'Germany'}, 36500), (\text{'China'}, 22500)\}$.

Note that, if we omit the preference in the example above, we could express this in standard SQL in a more lengthy way, cp. the next example.

Example 6. *Consider the aggregating query from Example 5 without the preference, i.e., calculate the average car price for every country:*

```
SELECT manufacturer, AVG(price) FROM car
GROUPING make SV (...) AS country;
```

This can be expressed in standard SQL using the lengthy **CASE**... **THEN**... *statement:*

```
SELECT country, AVG(price) FROM
 (SELECT (CASE
    WHEN make IN ('BMW', 'Mercedes')
         THEN 'Germany'
    WHEN make IN ('Geely', 'Chery')
         THEN 'China'
 END) AS country, price FROM car) tmp_car
GROUP BY country;
```

Thus we are able to simplify **GROUP BY** *queries with case-statements by our Grouping-SV construct. Note that this kind of grouping could be simplified, if the country is stored as an attribute in the dataset. However, if this is not the case, it would be a high effort to alter the table and add the country for each make.*

Projection functions for hierarchical data types are also supported, e.g., the function **EXTRACT**`(datepart` **FROM** `...)` for date.

Example 7. *We are looking for the most powerful cars, where groups are built from the year of registration.*

```
SELECT y, id, power
FROM car
PREFERRING power HIGHEST
GROUPING
 EXTRACT(YEAR FROM reg_date) AS y AVOID NULL;
```

This yields the result $\{(2012, 3, 215), (2013, 2, 200)\}$. *Following (Endres et al. 2012),* **`AVOID NULL`** *specifies that the grouping preference does not consider a NULL-group.*

Note that the aliasing mechanism of our syntactic scheme allows for a renaming via **`AS`** `...` in the grouping-clause. This is in contrast to standard SQL where such declarations would be stated in the projection. As in the logical order of processing, the partitioning into groups is done first and the projection at last, we think that it is intuitive to state the aliasing in the grouping-clause.

Expressing Subqueries with Preferences

Beyond the powerful and flexible grouping functionality, grouped preferences allow for the elimination of correlated subqueries in some cases. A common practice in data warehouses is to relate a nested query to the outer query such that the subquery only processes values relevant to the rows of the outer query (Elhemali et al. 2007). This introduces a so-called *correlation* between the outer query and the subquery. Correlations make it challenging to find well-performing execution plans for queries with subqueries, cp. (Ganski and Wong 1987; Elhemali et al. 2007).

Subsequently, we illustrate this for a variant of the TPC-H Q2 query[1], reduced to its essence. The TPC-H benchmark is a decision support benchmark, which consists of a suite of business oriented ad-hoc queries with a broad industry-wide relevance. Assume one is looking for the cheapest supplier for each part in the relation *partsupp*(ps_partkey, ps_suppkey, ps_supplycost, ...):

```
SELECT ps_partkey, ps_suppkey
FROM partsupp ps
WHERE ps.ps_supplycost = (
  SELECT MIN(p.ps_supplycost)
  FROM partsupp p
  WHERE p.ps_partkey = ps.ps_partkey);
```

Following (Elhemali et al. 2007), this is a *Type-JA Nested Query*, which can be optimized by transforming it into a canonical equivalent which references a new temporary relation `Rt`.

```
-- Query 1: Non correlated query.
SELECT ps.ps_partkey, ps.ps_suppkey
FROM partsupp ps,
 (SELECT p.ps_partkey AS C1,
         MIN(p.ps_supplycost) AS C2
  FROM partsupp p
  GROUP BY p.ps_partkey) AS Rt
WHERE ps.ps_supplycost = Rt.C2
  AND Rt.C1 = ps.ps_partkey;
```

This can be rewritten as the following query:

```
-- Query 2: Preference SQL query.
SELECT ps_partkey, ps_suppkey FROM partsupp
PREFERRING ps_supplycost LOWEST
GROUPING ps_partkey;
```

The subquery searching for the minimal price is replaced with **`GROUPING`** by the auto-correlation parameter together with a **`LOWEST`** preference on price.

[1]*http://www.tpc.org/tpch/spec/tpch2.15.0.pdf* (2013)

The latter query is a more *intuitive semantic representation*. Beyond that, no temporary space for the subquery is necessary, such that the grouped preference query outperforms the computation, too. This can be seen in Table 2, where we used the Preference SQL system to evaluate the queries above. The Preference SQL framework is a Java 1.6 middleware for preference queries on conventional database systems. It parses the query and performs a logical *query optimization* as described in (Hafenrichter and Kießling 2005) inside the middleware. Afterwards it evaluates the preference query using, e.g., BNL (Börzsöny, Kossmann, and Stocker 2001). The queries were performed on the 10 MB, 100 MB, 500 MB, and 1 GB TPC-H dataset. The preference query (Query 2) outperforms the conventional optimized query (Query 1) by a factor of more than 2. Therefore, our preference rewriting may speed up the query answer time for such nested queries.

Table 2: Results for queries 1 and 2.

TPC-H	**10 MB**	**100 MB**	**500 MB**	**1 GB**
BMO-size	2000	20002	100006	200010
Query 1 in sec	2.5	129.8	1329.2	2092.2
Query 2 in sec	1.1	51.5	569.5	802.2
Speedup factor	**2.3**	**2.5**	**2.3**	**2.6**

In the following theorem we show that the above rewriting is correct.

Theorem 1 (Replace Nested Query by Preference). *Let R be a database relation where $G \in \text{attrib}(R)$ is a grouping attribute without NULLs. Let $A \in \text{attrib}(R)$ be some numerical attribute and $P = (A, <_P)$ a* LOWEST *(or* HIGHEST*) preference. Assume a subquery* $\text{sub}(s) = f(\sigma_{A=s}(R).A)$ *where $f = \min$ (or $f = \max$). Then*

$$\bigcup_{t \in R.G} \sigma_{A=\text{sub}(t)}(R) = \sigma[P \text{ grouping } G](R)$$

Note that the union on the left hand side is the relational transcription for a correlation (Elhemali et al. 2007).

Proof.

$$\begin{aligned}
& \textstyle\bigcup_{t \in R.G} \sigma_{A=\text{sub}(t)}(R) \\
= \; & \textstyle\bigcup_{t \in R.G} \sigma_{A=f(\sigma_{G=t}(R).A)}(R) \\
= \; & \{[f \in \{\min, \max\} \text{ is LOWEST or HIGHEST pref.}]\} \\
& \textstyle\bigcup_{t \in R.G} \sigma[P]\,(\sigma_{G=t}(R)) \\
= \; & \{[\text{Definition of selection and BMO set}]\}] \\
& \textstyle\bigcup_{t \in R.G} \{y \in \{x \in R \mid x.G = t\} \mid \\
& \qquad \nexists z \in \{x \in R \mid x.G = t\} : y <_P z\} \\
= \; & \{[\text{Reorganize set and quantifier conditions}]\} \\
& \textstyle\bigcup_{t \in R.G} \{y \in R \mid y.G = t \wedge \\
& \qquad \nexists z \in R : \underbrace{z.G = t \wedge y <_P z}_{y <_{P \text{ grouping } G} z}\} \\
= \; & \{[\text{For every } y \text{ there exists a } t \in R.G \text{ with } y.G = t]\} \\
& \{y \in R \mid \nexists z \in R : y <_{P \text{ grouping } G} z\} \\
= \; & \{[\text{Definition of BMO-Set}]\} \\
& \sigma[P \text{ grouping } G](R) \qquad \square
\end{aligned}$$

Related Work

Since the specification of grouping in the well-known SQL-92 standard, many papers were published on grouping and aggregation, e.g., (S. Chaudhuri 1994). Grouping was extended in SQL-99 to *grouping sets*, which allow more repeated grouping clauses in one query. However, to the best of our knowledge, there is no work which discusses the interplay between *grouping* and *preferences* in detail. However, *preference* queries containing grouping are one of the most intuitive and practical type of queries: they find the best results concerning the preference for each group of tuples.

In (Börzsöny, Kossmann, and Stocker 2001) the Skyline operator was introduced, which is a *special case of a Pareto preference query* (Chomicki, Ciaccia, and Meneghetti 2013). Although the Skyline operator may interact with the **`GROUP BY`** operation, the discussion of this issue was restricted to the question whether grouping and aggregating should be executed before or after the Skyline operator. Another interpretation of grouping in the context of Skylines is the partitioning of the data w.r.t. an attribute. Then the Skyline is executed on each partition separately, which is studied for example in (Luk, Yiu, and Lo 2009). However, they discuss Group-by algorithms using sorting and hashing, but do not specify the grouping functionality.

Furthermore, the optimization of nested queries was done by (Ganski and Wong 1987), and extended e.g., by (Elhemali et al. 2007), to name a few. However, in all these papers temporary relations are used to optimize nested queries. We presented an approach to rewrite a correlated subquery into a preference query without temporary sets. This simplifies the modeling of nested queries and at the same time speeds up their computation.

Summary and Outlook

In Decision Support Systems, BI, OLAP or Data Warehouses the **`GROUP BY`** operation is extensively used to formulate queries. Moreover, preference database queries allow to analyze large data sets without the annoying empty result or flooding effect. In this paper we combined these two concepts to provide a flexible framework for preference data analytics. The novel concept of **`Grouping-SV`** allows user-defined SV-relations for a powerful grouping functionality. In particular, this unified approach presents a more intuitive representation of grouping and aggregating in Preference SQL. Using Decision Support Systems this means a faster, more intuitive and less error-prone query writing.

Furthermore, we demonstrated how a correlated subquery can be rewritten as a grouped preference query. Our benchmark shows that the expressive power of grouped preference queries leads to a faster computation of such nested queries.

In the future work we will investigate other types of nested queries for preference rewriting. Since the size of databases increases drastically and complex business analysis becomes progressively more important, the question of an efficient processing of preference Grouping-SV queries arises. Motivated by this and the high significance of preferences, we will look deeper into finding more efficient ways of processing and optimizing Grouping-SV queries.

References

Börzsöny, S.; Kossmann, D.; and Stocker, K. 2001. The Skyline Operator. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, 421–430.

Chomicki, J.; Ciaccia, P.; and Meneghetti, N. 2013. Skyline Queries, Front and Back. *SIGMOD Rec.* 42(3):6–18.

Chomicki, J. 2003. Preference Formulas in Relational Queries. In *TODS '03: ACM Transactions on Database Systems*, volume 28, 427–466.

Elhemali, M.; Galindo-Legaria, C. A.; Grabs, T.; and Joshi, M. 2007. Execution Strategies for SQL Subqueries. In *Proceedings of SIGMOD '07*.

Endres, M.; Roocks, P.; Wenzel, F.; Huhn, A.; and Kießling, W. 2012. Handling of NULL Values in Preference Database Queries. In *MPref '12: Proceedings of the 6th Multidisciplinary Workshop on Advances in Preference Handling*.

Ganski, R. A., and Wong, H. K. T. 1987. Optimization of Nested SQL Queries Revisited.

Golfarelli, M., and Rizzi, S. 2009. Expressing OLAP Preferences. In *Proceedings of SSDBM '09*, 83–91.

Gupta, A.; Harinarayan, V.; and Quass, D. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of VLDB '95*, 358–369.

Hafenrichter, B., and Kießling, W. 2005. Optimization of Relational Preference Queries. In *ADC '05: Proceedings of the 16th Australasian database conference*, 175–184. ACS.

Kaci, S. 2011. *Working with Preferences: Less Is More*. Springer.

Kießling, W.; Endres, M.; and Wenzel, F. 2011. The Preference SQL System - An Overview. *Bulletin of the Technical Commitee on Data Engineering, IEEE CS* 34(2):11–18.

Kießling, W. 2005. Preference Queries with SV-Semantics. In *In Proceedings of COMAD '05*, 15–26.

Luk, M.-H.; Yiu, M. L.; and Lo, E. 2009. Group-by Skyline Query Processing in Relational Engines. In *Proceedings of CIKM '09*, 1433–1436.

Roocks, P.; Endres, M.; and Kießling, W. 2013. Specification and Optimization of Preference (SV-)Grouping Queries. Technical Report 2013-01, University of Augsburg.

Rossi, F.; Venable, B.; and Walsh, T. 2011. *A Short Introduction to Preferences: Between Artificial Intelligence and Social Choice*. Morgan & Claypool.

S. Chaudhuri, S. K. 1994. Including Group-By in Query Optimization. In *VLDB '94: Proceedings of the 20th Int. Conference on Very Large Data Bases*.

SQL-92. 1992. Database Language SQL. Document ISO/IEC 9075:1992. ANSI Document X3.135-1992 (SQL92 Standard).

Stefanidis, K.; Koutrika, G.; and Pitoura, E. 2011. A Survey on Representation, Composition and Application of Preferences in Database Systems. *ACM Transaction on Database Systems* 36(4).