Optimization of Relational Preference Queries

Bernd Hafenrichter, Werner Kießling

Faculty of Applied Computer Science
University of Augsburg
Universitätsstraße 14, D-86159 Augsburg, Germany

{hafenrichter, kiessling}@informatik.uni-augsburg.de

Abstract

The design and implementation of advanced personalized database applications requires a preference-driven approach. Representing preferences as strict partial orders is a good choice in most practical cases. Therefore the efficient integration of preference querying into standard database technology is an important issue. We present a novel approach to relational preference query optimization based on algebraic transformations. A variety of new laws for preference relational algebra is presented. This forms the foundation for a preference query optimizer applying heuristics like 'push preference'. A prototypical implementation and a series of benchmarks show that significant performance gains can be achieved. In summary, our results give strong evidence that by extending relational databases by strict partial order preferences one can get both: good modelling capabilities for personalization and good query runtimes. Our approach extends to recursive databases as well.

Keywords: personalization, preference, query optimization, relational algebra.

1 Introduction

Preferences are an integral part of our private and business–related lives. Thus preferences must be a key element in designing personalized applications and Internet-based information systems. Personal preferences are often expressed in the sense of wishes: Wishes are free, but there is no guarantee that they can be satisfied at all times. In case of failure for the perfect match people are often prepared to accept worse alternatives or to negotiate compromises. Thus preferences in the real world require a paradigm shift from exact matches towards matchmaking, which means to find the best possible matches between one's wishes and the reality. In other words, preferences are soft constraints. On the other hand, declarative query languages like SQL don't offer convenient ways to express preferences. This deficiency has been the source for inadequate database support in many important application areas, in particular for search engines for e-commerce or m-commerce. As pointed out in Kießling (2002) and Kießling and Köstler (2002) extending SQL or XML by preferences will enable better personalized search engines that don't suffer from the notorious empty-result and flooding effects. Preferences have played a big role in other academic disciplines for many decades, notably within the economic and social sciences, in particular for multi-attribute decision-making in operations research (Fishburn 1999, Keeney and Raiffa 1993). The first encounter of preferences in databases is due to Lacroix and Lavency in 1987. Early work on cooperative databases includes the Cobase system (Chu et al. 1996). Despite the undeniable importance of preferences for real world applications preference research in databases did not receive a more widespread attention until around 2000 (Agrawal and Wimmers 2000, Börzsönyi, Kossmann and Stocker 2001, Chomicki 2002, Hristidis, Koudas and Papakonstantinou 2001, Tan, Eng and Ooi (2001)). Starting already in 1993 there has been the long-term research vision and endeavour of "It's a Preference World" at the University of Augsburg. Salient milestones so far include the design and implementation of Datalog-S (Kießling and Güntzer 1994, Köstler, Kießling, Thöne and Güntzer 1995), extending recursive deductive databases by preference queries. By 1997 the experiences gained from Datalog-S inspired the design of Preference SQL, its first commercial product release being in 1999 (Kießling and Köstler 2002). These experiences have been compiled into a comprehensive framework for preferences in database systems as defined by Kießling (2002). Preferences are modeled as strict partial orders, providing also a unifying framework for approaches of other research groups like those cited above. As an essential feature the use of intuitive preference constructors is promoted.

In this paper we focus on the critical issue of preference query performance. In particular, we investigate the challenge of optimizing preference queries in relational databases. To set the stage, in section 2 we revisit the preference framework from Kießling (2002). In section 3 we introduce preference relational algebra and discuss architectural aspects for a preference query optimizer. Novel results for algebraic optimization of preference queries are presented in section 4, followed by performance experiments in section 5. Related works in section 6 and a summary and outlook in section 7 ends this paper.

2 The Preference Query Model

2.1 Strict Partial Order Preferences

People express their wishes intuitively in the form "*I like A better than B*". Mathematically such preferences are

strict partial orders. Let us revisit those concepts of this preference model from Kießling (2002) that are important here.

Let A = {A_1, A_2, …, A_k} denote a set of attributes A_j with domains dom(A_j). Considering the order of components within a cartesian product as irrelevant, we define:

- dom(A) = $\times_{Aj \in A}$ dom(A_j)
- A ***preference*** *P* is a strict partial order **P = (A, <$_P$)**, where <$_P \subseteq$ dom(A) × dom(A).
- "x <$_P$ y" is interpreted as "I like y better than x".

For ease of use a choice of ***base preference constructors*** is assumed to be predefined. This choice is *extensible*, if required by the application domain. Commonly useful constructors include the following:

- For *categorical* attributes:
  `POS, NEG, POS/POS,POS/NEG, EXP`
- For *numerical* attributes:
  `AROUND, BETWEEN, LOWEST, HIGHEST, SCORE`

POS specifies that a given set of values *should be* preferred. Conversely, NEG states a set of disliked values *should be* avoided if possible. POS/POS and POS/NEG express certain combinations, EXP explicitly enumerates 'better-than' relationships.

`AROUND` prefers values closest to a stated value, `BETWEEN` prefers values closest to a stated interval. `LOWEST` and `HIGHEST` prefer lower and higher values, resp. `SCORE` maps attribute values to numerical scores, preferring higher scores.

Compound preferences can be gained inductively by ***complex preference constructors***:

- *Pareto* preferences: **P := P_1 ⊗ P_2 ⊗ ... ⊗ P_n**
  P is a combination of *equally important* preferences, implementing the pareto-optimality principle.
- *Prioritized* preferences: **P := P_1 & P_2 & ... & P_n**
  P evaluates *more important* preferences earlier, similar to a lexicographical ordering. P_1 is most important, P_2 next, etc.
- *Numerical* preferences: **P := rank$_F$(P_1,P_2, ...,P_n)**
  P combines `SCORE` preferences P_i by means of a numerical ranking function F.

Concerning the formal definitions of preference constructors the reader is referred to Kießling (2002).

Example 1: Preference constructors

"Julia wishes to buy a used car. It *must* be a BMW. Moreover, it *should* have a red or black color and *equally important* is a price around 10,000. Highest fuel economy matters too, but is *less important*."

Wanting a BMW is a hard condition. Julia's preferences, i.e. soft conditions, can be extracted right away from this natural language description as follows:

P_{Julia} = (POS(color,{'red','black'})
⊗ AROUND(price,10000))
& HIGHEST(fuel_economy) ☼

Preference construction is inductively closed under strict partial order semantics (Kießling 2002). Due to a well-known theorem in Fishburn (1991) (see also Chomicki 2002) numerical preferences have a *limited* expressiveness. Many strict partial order preferences cannot be described by numerical preference constructors only. Therefore the support of the full preference constructor spectrum as described is a practical necessity.

2.2 The BMO Query Model

Extending declarative query languages by preferences leads to *soft selection conditions*. To combat the empty-result and the flooding effects the ***Best-Matches-Only* (BMO)** query model has been proposed in Kießling (2002). Assuming a preference P = (A, <$_P$) and a database relation R, BMO query answering conceptually works as follows:

- Try to find *perfect matches* in R wrt. P.
- If none exist, deliver *best-matching alternatives*, but *nothing worse*.

Efficient BMO query evaluation requires two new relational operators. Assuming a relation R where A ⊆ attributes(R), we define:

***Preference selection* σ[P](R):**

$\sigma[P](R) := \{w \in R \mid \neg\exists v \in R : w[A] <_P v[A]\}$

A preference can also be evaluated in grouped mode, given some B ⊆ attributes(R). According to Kießling (2002) this can be expressed as a preference itself:

$w <_{P \text{ groupby } B} v$ iff $w[A] <_P v[A] \wedge w[B] = v[B]$

***Grouped preference selection* σ[P groupby B](R):**

$\sigma[P \text{ groupby } B](R) := \{w \in R \mid \neg\exists\, v \in R : w[A] <_P v[A] \wedge w[B] = v[B]\}$

σ[P](R) and σ[P groupby B](R) can perform the match-making process as required by BMO semantics.

2.3 Practical Preference Query Languages

Existing implementations are Preference SQL for SQL environments and Preference XPATH for XML (Kießling, Hafenrichter, Fischer and Holland 2001). Subsequently we will use Preference SQL syntax for our case studies. Preference queries are specified as follows:

```
SELECT     <projection list L>
FROM       <R1, …, Rn>
WHERE      <hard conditions H>
PREFERRING <soft conditions P>
GROUPING   <B1, …, Bm>;
```

`SELECT-FROM-WHERE` is standard SQL, whereas `PREFERRING-GROUPING` cares for preferences.

Example 2: A preference query and its BMO result

"Under no circumstances Michael can afford to spend more than 35,000 Euro for a car. Other than that he wishes that the car *should be* a BMW or a Porsche, it *should be* around 3 years old and the color *shouldn't be* red. All these preferences are *equally important* to him."

Michael's overall preference is specified by:

```
P_Michael = POS(brand,{'BMW','Porsche'}) ⊗
   AROUND(age,3) ⊗ NEG(color,{'red'})
```

Given the Preference SQL query asking for Michael's best matching cars is as follows:

```
SELECT u.price, c.brand, u.age, u.color
FROM   used_cars u, category c
WHERE  u.ref = c.ref AND
       u.price <= 35000
PREFERRING c.brand IN ('BMW', 'Porsche')
       AND u.age AROUND 3
       AND u.color NOT IN ('red');
```

Note that 'AND' in the WHERE-clause means Boolean conjunction, whereas 'AND' in the PREFERRING-clause denotes pareto preference construction.

3 Relational Preference Query Optimization

3.1 Preference Relational Algebra

For the scope of this paper we restrict our attention to the non-recursive relational case, although our preference model is applicable to the general case of recursive deductive databases as well (Kießling and Güntzer 1994).

Let ***preference relational algebra*** denote the following two sets of operations:

- Standard positive relational algebra:
  hard selection $\sigma_H(\mathbf{R})$ given a Boolean condition H, projection $\pi(\mathbf{R})$, union $\mathbf{R} \cup \mathbf{S}$, Cartesian product $\mathbf{R} \times \mathbf{S}$
- *Preference operations:*
  preference selection $\sigma[P](\mathbf{R})$
  grouped preference selection $\sigma[\mathbf{P}\ \mathbf{groupby}\ \mathbf{B}](\mathbf{R})$

Due to a theorem in Chomicki (2002) the *expressive power* of preference relational algebra is the same as classical relational algebra (i.e. positive relational algebra plus \". Consequently preference queries under BMO can be rewritten into relational algebra (which is done by Preference SQL, see Kießling and Köstler 2002). It also implies that the optimization problem for preference relational algebra is not harder than for classical relational algebra, i.e. in principle preferences can be integrated into SQL with sufficient performance.

3.2 Operational Semantics of a Preference Query

Defining the semantics of a declarative query language in general requires a model-theoretic semantics (to capture the declarative aspects) and an equivalent fixpoint semantics (to capture the operational aspects of query evaluation). For Preference SQL this task can be embedded into the larger framework of Datalog-S, employing *subsumption models* and *subsumption fixpoints*, resp. (Köstler, Kießling, Thöne and Güntzer 1995).

Since our focus is here on algebraic optimization of *relational* preference queries, we can exploit the equivalence of relational algebra and preference relational algebra to define the operational semantics. Let's consider a preference query Q in Preference SQL syntax. Then the ***operational semantics*** of Q is defined as:

If <grouping clause does not exist>
then $\pi_L(\sigma[P](\sigma_H(R_1 \times ... \times R_n))$)
else $\pi_L(\sigma[P\ \text{groupby}\{B_1, \ldots, B_m\}](\sigma_H(R_1 \times ... \times R_n)))$

This canonically extends the familiar relational case. Referring back to our example 2, the operational semantics of this preference query is as follows.

Example 2 (cont'd): Operational semantics

$\pi_{\text{u.price, c.brand, u.age, u.color}}$
$(\sigma[\text{POS(c.brand,\{'BMW','Porsche'\}) } \otimes$
$\text{ AROUND(u.age,3) } \otimes \text{ NEG(color,\{'red'\})}]$
$(\sigma_{\text{u.ref = c.ref } \wedge \text{ u.price <= 35000}}$
$(\text{used\_cars u} \times \text{category c}))$ ☼

Such an initial preference relational algebra expression will be subject to algebraic optimization methods developed subsequently.

3.3 Architectural Design Issues

Extending an existing SQL implementation by preference queries requires some crucial design decisions for the preference query optimizer.

- **The loosely coupled pre-processor architecture:**
  Preference queries are processed by rewriting them to standard SQL and submitting them to the SQL database. The current version of Preference SQL follows such a loose coupling, achieving acceptable performance in many e-commerce applications (Kießling and Köstler 2002).
- **The tightly coupled architecture:**
  Integrating the preference query optimizer and the SQL optimizer more tightly promises an even much better performance. A practical obstacle might be that this implementation requires the close cooperation with a specific SQL database manufacturer.

Here the optimization problem for preference queries can be mapped onto preference relational algebra. In fact we can follow the two classical approaches of database query optimization:

- **Hill climbing optimization:**
  Hereby an initial operator tree T_{start} is optimized in a top down manner by applying some set of algebraic transformation laws. Which laws to apply, and in what order, has to be controlled by some *heuristics* that intelligently prune the usually exponentially large search space.
- **Dynamic programming optimizer:**
  The idea of this algorithm is to construct an operator tree in an bottom up manner. A set of alternative plans is computed in parallel whereby only the best plans re-

garding to a cost function are used in the further steps. Overall a plan is produced which is optimal according to the given cost metric.

In the next sections we will focus on the tight integration. Hereby we will demonstrate how preference optimization could be integrated in both categories of optimization approaches.

4 Preference Relational Algebra Laws

For the enhancement of the previous mentioned optimization approaches a deeper knowledge about algebraic transformation laws based on preference relational algebra is required. For example, successful heuristic strategies of algebraic relational query optimization are '*push hard selection*' and '*push projection*'. We extend this idea by developing transformation laws for preference relational algebra that allow us to perform '*push preference*' within operator trees.

4.1 Transformation Laws

Some annotations of the subsequent theorems refer to left-to-right 'push' transformations. We use attr(R) to denote all attributes of a relation R. Note, that in this work due to space limitations only the most important laws are given. For a full overview of all algebraic laws and the according proofs please refer to Kießling and Hafenrichter (2003) and Hafenrichter (2004). Also the numbering of the following theorems has been adopted from this work.

Theorem L1: Push preference over projection

Let $P = (A, <_P)$ and $A, X \subseteq attr(R)$.

a) $\sigma[P](\pi_X(R)) = \pi_X(\sigma[P](R))$ if $A \subseteq X$

b) $\pi_X(\sigma[P](\pi_{X \cup A}(R))) = \pi_X(\sigma[P](R))$ otherwise

Theorem L2: Push preference over Cartesian product

Let $P = (A, <_P)$ and $A \subseteq attr(R)$.

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

Pushing the Preference $\sigma[P]$ over the cartesian product $R \times S$ reduces the input size of $\sigma[P]$ enormous, since only $|R|$ tuples instead of $|R \times S|$ tuples have to be compared. Since $\sigma[P]$ reduced the input size of the cartesian product too, even this operation runs faster.

Theorem L3: Push preference over union

Let $P = (A, <_P)$ and $A \subseteq attr(R) = attr(S)$.

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

Theorem L3 creates two new $\sigma[P]$ operators for the child relations R and S. At a first glance, this leads to a performance loss, since the overhead for the two new $\sigma[P]$ operators have to be paid. On the other side, if R and S are complex queries itself, this transformation enables further optimization steps, which improve the performance of $\sigma[P](R)$ and $\sigma[P](S)$.

Theorem L6: Push preference over a join

Let $P = (A, <_P)$, $A \subseteq attr(R)$ and $X \subseteq attr(R) \cap attr(S)$.

a) $\sigma[P](R \bowtie_{R.X = S.X} S) = \sigma[P](R) \bowtie_{R.X = S.X} S$, if each tuple in R has at least one join partner in S

Let $R \ltimes_{R.X = S.X} S$ denote a semi-join operation.

b) $\sigma[P](R \bowtie_{R.X = S.X} S) = \sigma[P](R \ltimes_{R.X = S.X} S) \bowtie_{R.X = S.X} S$

c) $\sigma[P](R \bowtie_{R.X = S.X} S)) = \sigma[P](\sigma[P \text{ groupby } X](R) \bowtie_{R.X = S.X} S)$

Further let $B \subseteq attr(S)$.

d) $\sigma[P \text{ groupby } B](R \bowtie_{R.X = S.X} S) = \sigma[P \text{ groupby } B](\sigma[P \text{ groupby } X](R) \bowtie_{R.X = S.X} S))$

Note that as a special case law L6a is applicable, if R.X is a foreign key referring to S.X. If no such meta-information is available, then L6b explicitly computes all join partners. Therefore, L6a should always be applied before L6b because of the additional semi-join operation. Note that in the case of L6b not the whole semi-join has to be computed. Only tuples which dominate others have to be tested, whether they satisfy the join-condition or not.

Sometimes, a preference expression can become very complex and use attributes of different input relations. The preference can not be pushed as a whole over the join. Due to this reason, it is important to *split* a preference into pieces which can be further optimized.

Theorem L7: Split pareto preference and push over join

Let $P_1 = (A_1, <_{P1})$ where $A_1 \subseteq attr(R)$, $P_2 = (A_2, <_{P2})$ where $A_2 \subseteq attr(S)$. Further let $X \subseteq attr(R) \cap attr(S)$.

$$\sigma[P_1 \otimes P_2](R \bowtie_{R.X = S.X} S) = \sigma[P_1 \otimes P_2](\sigma[P_1 \text{ groupby } X](R) \bowtie_{R.X = S.X} S)$$

Theorem L7 splits the pareto preference $P := P_1 \otimes P_2$ into a grouped preference which is pushed over the join. The selectivity of a groupby-Preference depends on the distribution of the attribute X. In the worst case, X is a unique attribute and no tuples are removed by the grouping operation. Otherwise, if the distribution of X is very poor, a good selectivity can be achieved.

Theorem L8: Split prioritization and push over join

Let $P_1 = (A_1, <_{P1})$ where $A_1 \subseteq attr(R)$, $P_2 = (A_2, <_{P2})$ where $A_2 \subseteq attr(R) \cup attr(S)$ and let $X \subseteq attr(R) \cap attr(S)$.

a) $\sigma[P_1 \,\&\, P_2](R \bowtie_{R.X = S.X} S) = \sigma[P_2 \text{ groupby } A_1](\sigma[P_1](R) \bowtie_{R.X = S.X} S)$, if each tuple in R has at least one join partner in S

b) $\sigma[P_1 \,\&\, P_2](R \bowtie_{R.X = S.X} S) = \sigma[P_2 \text{ groupby } A_1](\sigma[P_1](R \ltimes_{R.X = S.X} S) \bowtie_{R.X = S.X} S)$

Concerning the precondition on join partners the same remark as for L6 applies here too. In contrast to Theorem L7, the splitting of $P := P_1 \,\&\, P_2$ is easier, since the whole Preference P_1 can be pushed over the join. The transformation L8 can be applied recursively to P_1, if P_1 is a prioritized preference too.

4.2 Integration with a Hill Climbing Optimizer

Because preference relational algebra extends relational algebra, we can construct a ***preference query optimizer*** as an extension of a classical relational query optimizer. Importantly, we can inherit all familiar laws from relational algebra given by Ullman (1989). Thus we can apply well-established heuristics aiming to reduce the sizes of intermediate relations, e.g. '*push hard selection*' and '*push projection*'. Let's consider this basic hill-climbing algorithm defined by Ullman (1989), given a suitable repertoire of relational algebra transformations:

Algorithm Pass(T):
{ update T:
Step 1-1 <split hard selections>;
Step 1-2 <push hard selections as far as possible>;
Step 2-1 <push projections as far as possible>;
Step 2-2 <combine hard selections and Cartesian products into joins>;
return T **}**

Expanding the given repertoire of relational transformations by our new laws L1 to L13 raises the issue of how to integrate them into the above procedure. In particular, we want to add the heuristic strategy of '*push preference*'. This strategy is based on the assumption, that early application of σ[P] reduces intermediate results. This leads to a better performance in subsequent operators like join and cartesian product. The test cases in given in section 5 and Hafenrichter (2004) give strong evidence that the heuristic "push preference" holds.

Finding a 'good' ordering of transformations is as usual a difficult, heuristic task, since the application of a rule can depend on the previous application of other rules. In case of preference relational algebra, L6 can only be applied if join operator have already been generated. Due to this knowledge the order defined in the following extended hill-climbing algorithm seems to be adequate for optimization of preference relational algebra statements.

Algorithm PassPreference(T):
{update T:
Step A-1 <push preference over union, projection>
Step A-2 <simplify preferences by hard selection>
Step 1-1 <split hard selections>;
Step 1-2 <push hard selections as far as possible>;
Step 2-1 <combine hard selections and Cartesian products into joins>
Step B-1: <*simplify prioritization*>
Step B-2: <*push preference over join*, apply L6a;L6b>
Step B-3: <s*plit preference and push over join* , apply L7; L9e; L8a; L8b>
Step B-4: <push preference over cartesian product, apply L2>
Step B-5: <split and push preference over cartesian product, apply L12, L13>
Step 2-2*: Step <push projections as far as possible>;
return T**}**

In PassPreference the Steps A and B have been added to the algorithm Pass. Step A is executed first and pushes σ[P] over ∪ to all sub-operator-trees. As a result, σ[P] is placed over the hard selection operator in each sub-tree. Due to this reason σ[P] can be refined in Step A-2 because of interactions between preference selection and hard selection. After Step A the operator-tree is transformed by traditional rules of relational algebra (Step 1-1 to Step 2-1). The resulting operator-tree forms the basis for further applications of preference laws, hence Step 2-1 generates join operators. This is a prerequisite for the application of laws L6, L7, L8 and L9. Starting with Step B-1 should always be ok, because σ[P] is simplified and therefore more subsequent rules can be applicable. Also in any case Step B-2 should come before Step B-3 because of a better filter effect. Note that L7 and L8a,b relate to different situations, hence their relative order within Step B-3 does not matter. B-4 should always come before Step B-5 because of a better filter effect. T is repeatedly traversed until no more preferences can be pushed. To prevent an infinite firing of laws like L7, proper context information is maintained.

Finally in Step 2-2* the projection operator π is pushed as far as possible. Note that this step has been extended by rules for pushing π over σ[P] (L1). This extension is straightforward.

Now we demonstrate the potential of such a preference query optimizer by a practical case study.

4.3 Case Studies

Let's revisit our example 2, adding a third relation `seller`. In used_cars we assume that *ref* and *sid* are foreign keys from category and seller.

Example 3: Complex preference query Q_1

```
SELECT s.name, u.price
FROM   used_cars u, category c, seller s
WHERE  u.sid = s.sid AND u.ref = c.ref AND
      (u.color = 'red' OR u.color = 'gray')
PREFERRING
       (LOWEST u.age AND
        u.price BETWEEN 5000, 6000)
       PRIOR TO c.brand IN ('BMW')
       PRIOR TO s.zipcode AROUND 86609;
```

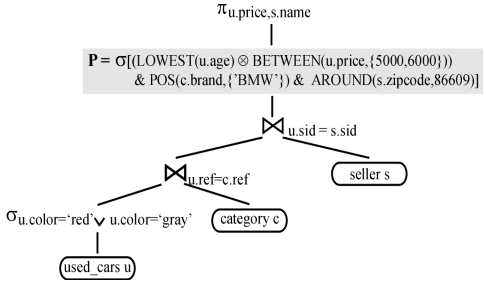

Figure 1: T_C after Step 2-1 (complex query)

The tree T_C as output after Step 2-1 is depicted in figure 1. The following sequence of preference transformations are performed to produce the final operator tree T_{end} in figure 2: **`L8a; L8a; L1b; L1a;`**

Let's compare T_{end} vs. T_C:

- *Join costs*: The lower join's left operand in T_{end} is reduced by the preference selection marked P_A in figure 2. This in turn, intensified by the preference selec-

tion P_B, reduces the size of the upper join's left operand in T_{end}.

- *Preference costs*: P_A is simpler than the original P in T_{rel}, but it's still a compound preference. However, both P_A and P_C are simpler, i.e. grouped *base* preferences.

Now the tradeoff is more apparent. Our heuristics of 'push preference' -in particular over joins- will pay off, if the savings for join computation outweigh the preference costs of P_A, P_B, P_C vs. the original P. ☼

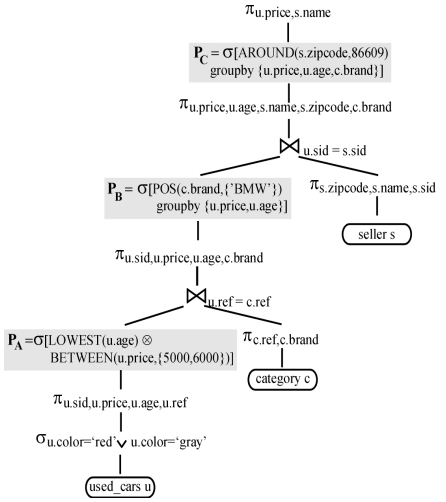

Figure 2: T_{end} after pass 2 (complex query)

4.4 Join Order Optimization

A major problem in our extended hill-climbing optimizer is the fact that the produced join order is predetermined by the initial alignment of the base relations. The produced join-order has an great impact on the application of preference laws like L6. Therefore, one might think of join orders, which allow the early application of the σ[P] operator. In the following, we will extend our basic optimizer by new bottom up algorithm, that computes such a join order.

Example 4: Complex join query Q_2

```
SELECT a.x, b.x
  FROM a, b, c, d, e, f
 WHERE a.x=b.x and b.y=c.y and b.x=d.x and
       d.y=e.y and d.y=f.y
PREFERRING a.val lowset and c.val highest
```

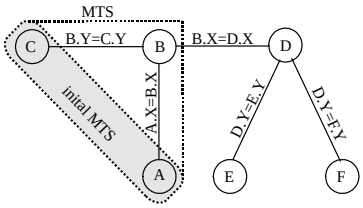

Figure 3: Query graph for Example 4

As initial input, the algorithm receives a query graph Q=(V,E) whose nodes are the relations of the query and edges are induced by the join conditions (figure 3). As first step the initial minimal table set is computed. This consists of all relations whose attributes are referenced by the preference-selection σ[P]. The minimal tables set induces a sub-graph Q_{MTS} of Q with V_{MTS} = MTS. If this graph is interconnected we proceed with the next step. Otherwise, we add new Nodes from Q to Q_{MTS} until the resulting Graph is interconnected. This is very important, since not connected nodes would result in a cartesian product. After this step, the query graph Q is collapsed, whereby all nodes of Q_{MTS} are combined to a single node called Q_P. Starting with this new node as root, it is tested whether the resulting Graph is a tree or not. In the first case, the algorithm terminates. Otherwise we have to eliminate cycles within the graph. This is done by iteratively collapsing adjacent nodes of Q_P with Q_P. For this choice nodes are preferred, which reside on a way to a cycle in Q. The resulting query graph is again tested for it's tree membership. If this test becomes true, the algorithm terminates and otherwise the iteration proceeds.

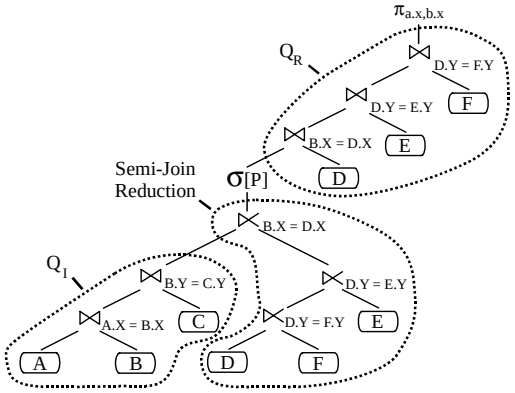

Figure 4: Querytree for Example 4

The resulting query graph is a tree with root Q_P. From this query graph, the new query is constructed in four steps. First of all, a query Q_I that joins all relations contained in the collapsed node Q_P is constructed. Second, a full-semi-join reduction for Q is computed that receives Q_I as input. The full semi-join reduction guarantees, that all resulting tuples have at least one join partner in the resulting query, which is a prerequesite for the application of σ[P]. Please refer to Bernstein and Chiu (1981) for further details of semi-join reductions. Third, the σ[P]-operator is added to the query and as last step the remaining nodes of Q are joined in an appropriate manner with the current query (Q_R). The semi-joins introduced at step 2, could be further refined due the existence of foreign key constraints.

4.5 Integration with a Dynamic Programming Optimizer

Current databases systems use a cost based dynamic programming algorithm for query optimization. This type of algorithm has been first introduced by Selinger, Astrahan, Chamberlin and Lorie (1979). It computes query plans in a bottom up manner by combining simple plans into complexer ones. Such algorithms doesn't explicitly apply transformation rules as known from the hill-climbing algorithm. Rather these rules are implicitly utilized by the bottom-up construction rules. Therefore the preference transformation laws can not be directly transferred to this kind of algorithm. As a starting point, the idea given by the previous mentioned join order optimization can be used. In the following section a short sketch about such an integration will be given.

Whenever the dynamic programming algorithm computes a new sub-plan (Q_S), the following test can be applied:

1) Does Q_S contain all relations defined by the minimal table set ? If not, σ[P] couldn't be applied to Q_S
2) Otherwise, construct a query graph Q_G. Collapse all relations used in Q_S to one single node in Q_G. If the resulting query graph Q_G' belongs to the class of tree queries continue with step 3. Otherwise σ[P] couldn't be applied to Q_S
3) Based on Q_G' compute a semi-join reduction SJR. Add σ[P] and SJR to Q_S, so that the modified Query Q_S' = σ[P](SJR(Q_S)) is created. Add Q_S and Q_S' to the set of optimal plans.

In each construction step of the dynamic programming algorithm, the cost function is used to discard the most expensive plans. In order to be applicable for σ[P], the cost-function has to be extended by a preference based metric. The selectivity of σ[P] depends on the complexity of P. If P is a base preference (except explicit) traditional statistical information can be used. If P is a compound preference one has to estimate the selectivity based on the used complex-constructors and base-preferences. For more details please refer to Hafenrichter (2004).

5 Performance Evaluation

Finally we present performance experiments from a prototype implementation of our preference query optimizer. Since we did not have quick access to a commercial SQL query engine to tightly integrate our novel optimization techniques we decided to simulate it. For the sake of *rapid prototyping* we built a Java middleware on top of Oracle 9i. This task has been facilitated by the use of the XXL Java-library (Bercken, Blohsfeld, Dittrich 2001), offering a collection of relational algebra operators.

5.1 Prototype Implementation

Figure 5 illustrates how a preference query Q is evaluated:

- Q is mapped onto its initial operator tree T_{start}.
- T_{start} is algebraically optimized, employing PassPreference with final output T_{end}.
- T_{end} is submitted to an evaluation middleware, utilizing XXL for relational operations and providing own Java implementations of the preference operators σ[P](R) and σ[P groupby B](R).
- All persistent databases relations are managed by Oracle 9i, accessed from XXL via JDBC.

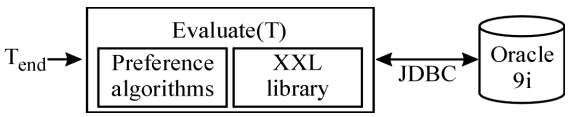

Figure 5: **Rapid prototyping for performance studies**

The development of efficient evaluation algorithms for σ[P](R) is considerably facilitated by the constructor-based approach of our preference model. Depending on the preference constructor P we can efficiently *custom-design* the evaluation method for σ[P](R) as follows.

Most specific evaluation algorithms for σ[P](R):

We generalized the basic *block nested loop* algorithm BNL, investigated by Börzsönyi, Kossmann and Stocker (2001) in the context of skyline queries, to arbitrary strict partial order preferences. However, the subsumption test 'x <$_P$ y' within BNL can be simplified significantly, if P is known to have special properties. This is the case for each of our *base* preference constructors (cmp. section 2.1). Therefore we have implemented the following evaluation policy for σ[P](R):

- If P is a *complex* preference (i.e. constructed e.g. by '⊗' and-or '&'), then the general BNL applies.
- Otherwise, if P is a base preference constructor, then a specialized algorithm for this P has to be chosen.

BNL is known to degrade to $O(n^2)$ in the worst case. In contrast, the complexity of specialized algorithms for base preference constructors is much better: *O(n)* if no index support is provided, otherwise *O(log n)*. Considering σ[P groupby B](R), directly exploiting its formal definition as a complex preference and applying BNL would risk an $O(n^2)$ performance penalty. More efficient is to implement the grouping effect e.g. by sorting and to invoke the most specific algorithm for P on each group. Therefore if P is a base preference, an *O(n log n)* behavior can be accomplished for σ[P groupby B](R).

The procedure `Evaluate(T)` in figure 5 traverses T and maps subtrees to XXL methods or *most specific* preference algorithms, returning JDBC ResultSets. Pipelining of XXL and of preference algorithms is enforced as much as possible. Expressions of type π(σ$_H$(R)), possibly extended by semi-joins generated by laws L6b or L8b, are evaluated by Oracle if R is a database relation. To expedite the evaluation of grouped preferences on π(σ$_H$(R)) the ResultSet is returned ordered by B. Moreover, we replaced the nested-loops join of XXL by a more efficient hash join. Doing so, our heuristics of 'push preference' has a tougher task to prove its benefits.

5.2 Performance Results

We have built up a test suite for performance evaluation. Given a preference query Q, we carried out the following performance measurements in our rapid prototype:

- Runtime $\mathbf{t_{rel}}$ (in sec.): Q is only optimized by traditional relational algebra laws as defined by algorithm "Pass"
- Runtime $\mathbf{t_{pref}}$ (in sec.): Q is optimized by preference relational algebra laws as defined by algorithm "PassPreference".

As an indicator for the optimization impact of our new approach we choose the *speedup factor* **SF1:= t_{rel} / t_{pref}**.

The experiments were performed on a standard PC (2 GHz CPU, 512 MB main memory) running XP and JDK 1.3. The relation `seller` has 5000 tuples, `category` has 1000, while `used_cars` varies from 1000 to 50000.

All data have been generated synthetically. Values are uniformly distributed. Attributes are uncorrelated except that higher `age` anti-correlates with higher `price`. We present selected characteristic tests, beginning with the query from example 3. Please note that due to space limitations we can't display all operator trees here.

Example 5: Performance results for Q_1

For the transformation sequence and the pushed and split preferences P_A, P_B and P_C please refer back to example 3.

used_cars	1000	5000	10000	50000
BMO size	15	59	138	518
t_{rel}	0.7	0.8	1.1	4.3
t_{pref}	0.2	0.3	0.5	2.8
SF1	3.5	2.7	2.2	1.5

The original P was split and pushed twice by L8a. P_B and P_C are grouped base preferences and can be evaluated reasonably fast. The net effect is a sizeable performance gain as indicated by SF1. ☼

Example 6: Performance results for Q_3

```
SELECT s.name, u.price
FROM   used_cars u, category c, seller s
WHERE  u.sid = s.sid AND u.ref = c.ref AND
       c.brand = 'BMW'
PREFERRING LOWEST u.age PRIOR TO
           (LOWEST u.price AND
           HIGHEST c.horsepower );
```

The preference transformations carried out are:

```
Cor1;L6a;L6b;L6a;L7;L7;L9e;L1b;L1a;L1b
```

Pushed and split preferences in T_{end} are:
P_A = LOWEST(u.age)
P_B = LOWEST(u.price) groupby {u.ref}
P_C = LOWEST(u.price) ⊗ HIGHEST(c.horsepower)

used_cars	1000	5000	10000	50000
BMO size	3	2	6	4
t_{rel}	0.7	0.8	1.0	3.5
t_{pref}	0.2	0.2	0.3	0.4
SF1	3.5	4.0	3.3	8.8

☼

In the following examples the tables A - I are used. The data are synthetically generated whereas the columns vala and valb underlie a normal distribution in the range 1 to 50000. Within tables A-F the columns x, y and z are unique. For the tables G - I the column x is unique. The columns y and z are normal distributed between 1 and 2000. Therefore a one to many relationship can be modeled.

Example 7: ***Performance results for Q_2***

The parameter t_{join} describes the runtime for evaluating a query with join order optimization.

A,B,C,D,E,F	1000	5000	10000	50000
BMO size	7	7	13	6
t_{rel}	0.4	1.3	3.2	58.6
t_{pref}	0.4	1.3	2.8	58.0
t_{join}	0.4	0.9	2.0	30.7
SF1	1.0	1.0	1.1	1.0
$SF1_{Join}$	1.0	1.4	1.6	1.9

☼

Example 8: Performance results for Q_4

```
SELECT A.X, B.X, C.X, D.X, E.X
FROM   A, B, C, D, E
WHERE  B.X = A.X AND B.Y = C.Y AND
       B.Z = E.Z AND B.X = D.X
PREFERRING C.VALA LOWEST PRIOR TO A.VALA
HIGHEST
```

A,B,C,D,E,F	1000	5000	10000	50000
BMO size	7	7	13	6
t_{rel}	0.4	1.1	2.5	49.5
t_{pref}	0.2	1.0	2.6	40.6
t_{join}	0.2	0.3	0.5	2.4
SF1	2.0	1.1	1.0	1.2
$SF1_{Join}$	2.0	3.7	5.0	20.6

Due to the early application of σ[P], t_{join} is extremely better then t_{end}. Since the join-order algorithm takes care of the query structure, it is able to place σ[P] earlier as the rule base optimizer. Furthermore the join order algorithm has more degrees of rearranging the query. ☼

Example 9: Performance results for Q_5

```
SELECT G.X, H.X, I.X, A.X
FROM   G, H, I, A
WHERE  G.Y=H.Y AND H.Z=I.Y AND H.X=A.X
PREFERRING G.VALA LOWEST AND G.VALB HIGHEST
```

G	1000	5000	10000	50000
BMO size	150	200	375	225
t_{rel}	2.9	11.2	23.0	106.0
t_{pref}	0.4	0.6	0.9	2.0
t_{join}	0.4	0.6	0.9	2.0
SF1	7.3	18.7	25.6	53.0
$SF1_{Join}$	7.3	18.7	25.6	53.0

size of A, H, I = 10000 tuples ☼

In an extended test suite, defined in Hafenrichter (2004), over 50 preference queries have been tested with the given prototype.

We also executed some test queries on Preference SQL 1.3, running loosely coupled on Oracle 9i. SF2 as defined below compares it with our rapid prototype:

- Runtime $t_{Pref\text{-}SQL}$ (in sec.)
- Performance speedup factor SF2:= $t_{Pref\text{-}SQL}$ / t_{pref}

Though Preference SQL is known as reasonably fast, the subsequent numbers show already SF2 > 1, which is quite remarkable for a rapid prototype carrying this much overhead. (Note that SF2 could not be measured in each case, because prioritization P_1 & P_2 is supported in Preference SQL 1.3 only if P_1 is a chain.)

SF2 : Q_3	2.0	3.0	2.0	2.3
SF2 : Q_5	6.8	20.2	34.8	96.8

5.3 Lessons Learned so far

So far we have seen cases with 1 < SF2 and SF1 being already quite good. But there are many more tuning opportunities. For instance, recall that we intentionally have favored join costs by implementing a fast hash join in XXL, but using the sub-optimal BNL algorithms for general preference evaluation. Clearly, if the latter is replaced by advances like Tan, Eng and Ooi (2001), SF1

will increase considerably. All specialized evaluation algorithms for base preference constructors were implemented by plain O(n) methods. Obviously the use of indexing achieves another substantial performance boost. When migrating to a tight integration the JDBC overhead will be eliminated either. Implementing all these add-on improvements is straightforward and will upgrade performance by large extents, maybe by orders of magnitude. Options for further performance speedup concern the sophistication of our PassPreference and the topic of cost-based query optimization. Improving the quality of the 'push preference' heuristics is a learning process, similar to a classical SQL optimizers. Our algebraic approach to preference query optimization will be a good foundation to come up with cost estimation models for critical issues like preference selectivity. In summary our experiments give already strong evidence that a tightly integrated preference query optimizer can achieve excellent performance. Enabled by the foundations of algebraic optimization we believe that we have revealed the entrance to a performance tuning gold mine, having many more options up our sleeves.

6 Related Works

The optimization of preference queries with BMO semantics poses new research challenges. Based on a technical report in July 2002 several preference relational algebra laws have been published by the authors in Kießling and Hafenrichter (2002). J. Chomicki's independent work focuses on a relaxation of strict partial order semantics for preferences: Our laws L1 and L2 and L5 are special cases of Chomicki (2002). Since most practical database applications seem to comply with strict partial order semantics, relaxing it must be carefully weighed; transformations can become invalidated, which in turn decreases preference query performance. Here we have presented a further series of new laws plus, for the first time, performance evaluations with a carefully engineered preference query optimizer. In Börzsönyi, Kossmann, and Stocker (2001) a transformation for pushing skyline preferences through joins, being an instance of our law L6a, has been presented without a formal proof. Likewise a method for pushing skylines into a join, being an instance of law L7, can be found there. The notion of non-reductive joins is related to our laws L6 and L7.

BMO is compatible with the *top-k* query model, which has been quite popular with numerical preferences. A theorem in Chomicki (2003) ensures that in principle top-k can be provided by means of BMO as follows: If the result res of $\sigma[P](R)$ has m tuples and $m \geq k$, return k of them; otherwise deliver those m and find the remaining ones by $\sigma[P](R \setminus res)$. In this way top-k query semantics can be provided for arbitrary strict partial order preferences, not only for numerical preferences. How skylines relate to top-k was addressed in Börzsönyi, Kossmann, and Stocker (2001) before.

Several algorithms have been proposed to efficiently implement numerical preferences for top-k querying, e.g. Agrawal and Wimmers (2000), Balke, Güntzer and Kießling (2002), Fagin, Lotem and Naor (2001), Güntzer, Balke and Kießling (2000), Hristidis, Koudas and Papakonstantinou (2001). Efficient skyline algorithms were investigated e.g. by Börzsönyi, Kossmann, and Stocker (2001), Kossmann, Ramsak and Rost (2002), Papadias, Tao, Fu and Seeger (2003), Tan, Eng and Ooi (2001).

7 Summary and Outlook

Assigning a strict partial order semantics to preferences is a good choice in many database applications. A rich repertoire of intuitive preference constructors can facilitate the design of personalized applications. Since numerical preferences are of limited expressiveness, constructors for pareto and prioritized preferences are required too. One might argue that such a high modeling convenience leads to runtime inefficiency. However, our contributions give strong evidence that for relational preference queries with BMO semantics this is not the case.

We have laid the foundations of a framework for preference query optimization that extends established query optimization techniques from relational databases. Preference queries can be evaluated by preference relational algebra, extending classical relational algebra by two new preference operators. We have provided a series of novel transformation laws for preference relational algebra that are the key to algebraic optimization. A preference query optimizer can be constructed as an extension of existing SQL optimizers, adding new heuristics like 'push preference'. In this way the decade-long investments and experiences with relational query optimizer can be inherited completely. We presented a rapid prototype of such a preference query optimizer and carried out a series of performance experiments. The performance speedups observed so far give already strong evidence that a tightly coupled implementation inside an existing SQL query engine can achieve excellent performance.

Currently there are several projects within our "It's a Preference World" program that use Preference SQL or Preference XPATH, e.g. Fischer, Kießling, Holland and Fleder (2002). Building on the fundamental insights developed here for heuristic preference query optimization, the important issue of cost-based optimization can be tackled next. We will also extend our optimization framework to recursive preference queries, where we can start over from research results on pushing preference selection over recursion established already in Köstler, Kießling, Thöne and Güntzer (1995). In summary we believe that we have contributed a crucial stepping stone towards efficient preference query optimization: Database technology can offer very good support for preferences and personalization, both in terms of ease of modeling and high efficiency.

References

Agrawal R., Wimmers E. L. (2000): A Framework for Expressing and Combining Preferences. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, USA, 297 - 306, ACM Press.

Balke W.-T. , Güntzer U., Kießling W. (2002): On Real-time Top k Querying for Mobile Services. *International Confererence on Cooperative Information Systems (CoopIS)*, Irvine CA,USA, 125-143.

Bercken J., Blohsfeld B., Dittrich J., et al. (2001): XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. *Proceedings of 27th International Conference on Very Large Data Bases*, Rome, Italy, 39 - 48.

Bernstein P. A., Chiu D.-M. (1981): Using Semi-Joins to Solve Relational Queries, *Journal of the ACM*, 28(1):25 - 40,

Börzsönyi S., Kossmann D., Stocker K. (2001): The Skyline Operator. *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, 421 - 430.

Chomicki J. (2002): Querying with Intrinsic Preferences. *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, Prague, Poland, 34 - 51.

Chomicki J. (2003): Preference formulas in relational queries. ACM Transactions on Database Systems (TODS), 28(4), 427 - 466.

Chu W., Yang H., Chiang K., Minock M., Chow G., Larson C.(1996): CoBase: A Scalable and Extensible Cooperative Information System. Journal of Intelligence Information Systems, 6(2-3): 223 - 259.

Fagin R., Lotem A., Naor M. (2001): Optimal Aggregation Algorithms for Middleware. *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Santa Barbara CA, USA, 102 - 113.

Fischer S., Kießling W., Holland S., Fleder M. (2002): The COSIMA Prototype for Multi-Objective Bargaining. *The First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, Bologna, Italy, 1364 - 1371.

Fishburn P. C. (1999): Preference Structures and their Numerical Representations. Theoretical Computer Science, 217(2): 359 - 383.

Güntzer U., Balke W.-T., Kießling W. (2000): Optimizing Multi-Feature Queries for Image Databases. *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, Sept. 2000, 419 - 428.

Hafenrichter B. (1994): Optimierung relationaler Präferenz-Datenbankanfragen. Ph.D. thesis. University of Augsburg, Germany.

Hristidis V., Koudas N., Papakonstantinou Y. (2001): PREFER : A System for the Efficient Execution of Multi-parametric Ranked Queries. *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara CA, USA, 259 - 269.

Keeney R., Raiffa H. (1993): Decisions with Multiple Objectives: Preferences and Value Tradeoffs. Cambridge University Press.

Kießling W. (2002): Foundations of Preferences in Database Systems. *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 311 - 322.

Kießling W., Güntzer U. (1994): Database Reasoning - A Deductive Framework for Solving Large and Complex Problems by means of Subsumption. *Proceedings of the 3rd Workshop on Information Systems and Artificial Intelligence*, LNCS 777, Hamburg, Germany, 118 - 138.

Kießling W., Hafenrichter B. (2002): Optimizing Preference Queries for Personalized Web Services. *In Proceedings of the IASTED International Conference, Communications, Internet and Information Technology (CIIT 2002)*, St. Thomas, Virgin Islands, USA, 461 - 466.

Kießling W., Hafenrichter B.(2003): Algebraic Optimization of Relational Preference Queries, Technical Report 2003-1, University of Augsburg, Germany.

Kießling W., Hafenrichter B., Fischer S., Holland S. (2001): Preference XPATH: A Query Language for E-Commerce. *Proceedings of the 5th Internationale Konferenz für Wirtschaftsinformatik*, Augsburg, Germany, 425 - 440.

Kießling W., Köstler G. (2002): Preference SQL – Design, Implementation, Experiences. *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 990 - 1001.

Köstler G., Kießling W., Thöne H., Güntzer U. (1995): Fixpoint Iteration with Subsumption in Deductive Databases. *Journal of Intelligent Information Systems*, 4(2): 123 - 148.

Kossmann D., Ramsak F., Rost S. (2002): Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 275 - 286.

Lacroix M., Lavency P. (1987): Preferences : Putting More Knowledge into Queries. *Proceedings of 13th International Conference on Very Large Data Bases*, Brighton, UK, 217 - 225.

Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., Price T.G. (1979): Access Path Selection in a Relational Database Management System. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, USA, 23 - 34.

Tan K.-L., Eng P.-K., Ooi B. C. (2001): Efficient Progressive Skyline Computation. Proceedings of 27the International Conference on Very Large Data Bases, Rome, Italy, 301 - 310.

Ullman J. (1989): Principles of Database and Knowledge-Base Systems. Vol. 1, Comp. Science Press.