# Distributed Aggregation Strategies for Preference Queries*
## (Extended Abstract)

Ilaria Bartolini, Paolo Ciaccia, and Marco Patella

DEIS, University of Bologna, Italy
{ibartolini,pciaccia,mpatella}@deis.unibo.it

**Abstract.** Networks of cooperating peers are a new exciting paradigm for evaluating queries in a distributed environment. In this scenario, a query originated at a peer propagates through the network, and the overall result is obtained by aggregating those returned by the peers involved in the evaluation. In this paper we consider the relevant case of *preference queries*, in which the user is interested in obtaining all and only the "best" results. We highlight the fundamental difference between queries in which preferences define a weak order (wo) over objects and the more general ones for which a strict partial order (spo) has to be considered. While for wo queries a simple algorithm that minimizes the overall number of objects to be transmitted across the network can be easily derived, we show that this is not the case for spo queries. Then, we detail a set of basic issues whose solution is a key to the derivation of an efficient distributed algorithm.

## 1 Introduction

Efficiently processing queries in a peer-to-peer (P2P) environment is an important and challenging research area [12]. Specific approaches differ in how they consider the network topology (e.g., structured vs. unstructured), the type of supported queries (e.g., keyword-based vs. SQL-like), and the peers' query capabilities (see [2] for a survey). A relevant case of queries which has recently found its way in the database community are *preference queries*, in which, besides possibly stating a set of hard constraints that objects have to satisfy, the user can also specify in which sense an object is deemed to be better than another one, in which case only undominated objects are returned. Although several papers have addressed the problem of how to efficiently evaluate preference queries, see e.g., [3, 8], the few of them that have studied this for P2P environments lack a comprehensive view of the problem [1, 9, 13]. In this paper we partially fill this gap by first highlighting the fundamental difference, in terms of computational overhead, between queries in which preferences define a *weak order* (wo) over objects and the more general ones for which a *strict partial order* (spo) has to be considered. We show that, while for wo queries a simple algorithm that

* This work is supported by the WISDOM MIUR Project

minimizes the overall number of objects to be transmitted across the network can be easily derived, this is not the case for spo queries, for which performance deterioration can be arbitrarily large. Starting from this unpleasant finding we then suggest as a way to alleviate the problem a processing strategy based on the idea of making a peer aware of what other peers can contribute to the result. This opens a set of interesting and challenging problems.

## 2 The Model

Without loss of generality, we consider a "global" relation schema $R(A_1, \ldots, A_m)$, with attributes $A_i, i \in [1, m]$, whose current instance $r$ is distributed over a network of peers $\mathcal{P} = \{p_1, \ldots, p_j, \ldots, p_m\}$. The subset of $r$ managed by peer $p_j$ is denoted as $r_j$. We are interested in specifying *preferences* over the tuples of $r$. Intuitively, a query with preferences allows $r$ tuples to be *ranked*, so that only best-matching tuples are returned.

**Definition 1 (Preference Relation).** *Let $\mathcal{A}$ be a set of attributes. A* preference relation $\succ$ *over $\mathcal{A}$ is a binary relation over $dom(\mathcal{A}) \times dom(\mathcal{A})$ that is transitive and irreflexive, i.e., a* strict partial order *(spo). If $t_1$ and $t_2$ are $\mathcal{A}$-tuples and $(t_1, t_2) \in \succ$, written $t_1 \succ t_2$, we say that $t_1$* dominates *$t_2$. If neither $t_1 \succ t_2$ nor $t_2 \succ t_1$ hold, we say that $t_1$ and $t_2$ are* indifferent*, written as $t_1 \sim t_2$.* □

The query we consider are expressed as $\beta_{\succ_P}(r)$, where $P$ is a preference relation and $\beta$ is the *Best* operator,[1] which computes the set $B$ of undominated tuples in $r$. Let $L_j = \beta_{\succ_P}(r_j)$ be the set of "local" best tuples in $r_j$, and $B_j = L_j \cap B$ the subset of $L_j$ that is also globally undominated. We call $B_j$ the "contribution" of $r_j$ to $B$. Since $P$ is an spo, it is immediate to derive that $B = \bigcup_j B_j \subseteq \bigcup_j L_j$, i.e., it is not possible to have a globally undominated tuple that is not also a local best result. Also observe that, although $L_j \neq \emptyset \; \forall j$ (since $P$ is an spo), it might well be the case that $B_j = \emptyset$ for some $j$.

A particular case of spo's are *weak orders* (wo), i.e., spo's whose indifference relation is transitive or, more intuitively, linear orders with ties. Relevant cases of wo's are those defined using numerical *scoring functions*. As an example, the following operators all produce a *base preference* that is a wo:

- $\mathsf{min}(E)$: prefers tuples minimizing the value of the numerical expression $E$ (e.g., $\mathsf{min}(price)$),
- $\mathsf{max}(E)$: prefers tuples maximizing the value of the numerical expression $E$ (e.g., $\mathsf{max}(rating)$),
- $\mathsf{pos}(E)$, prefers tuples for which the boolean expression $E$ is `true` (e.g., $\mathsf{pos}(price \in [30, 50])$).

Base preferences can be composed using either the Pareto rule, &, or prioritization, $\rhd$ [10], to create arbitrarily complex preferences. Intuitively, the Pareto rule considers all preferences equally important, thus $t_1 \succ t_2$ iff $t_2 \not\succ_{P_i} t_1$ on

[1] This is called *winnow* in [5, 6] and *preference selection* in [10].

all preferences $P_i$ and $t_1 \succ_{P_j} t_2$ on at least preference $P_j$, and produces spo (but not wo) preferences, whereas with prioritization base preferences are considered sequentially, thus $t_1 \succ t_2$ iff $t_1 \succ_{P_1} t_2$ on preference $P_1$ or $t_1 \sim_{P_1} t_2$ on preference $P_1$ and $t_1 \succ_{P_2} t_2$ on $P_2$, and so on.[2] Note that the Skyline operator proposed in [3] corresponds to the Pareto composition of a set of weak orders (e.g., $P = \mathsf{min}(price)$ & $\mathsf{max}(rating)$ & $\mathsf{pos}(cuisine = \text{“Indian”})$).

*Example 1.* Let $\mathcal{P}$ consist of three peers, $p_X$, $p_Y$, and $p_Z$, containing tuples of a relation $restaurants(\underline{name}, price, rating)$. Consider an user wanting to retrieve the best restaurants according to price and rating, i.e., the ones having a high rating and a low cost. The preference is thus, $P_{\mathsf{spo}} = \mathsf{min}(price)$ & $\mathsf{max}(rating)$ and defines an spo. If the local relations are:

$p_X$

| name | price | rating |
|---|---|---|
| $r_{X1}$ | 17 | 1 |
| $r_{X2}$ | 45 | 5 |
| $r_{X3}$ | 10 | 1 |
| $r_{X4}$ | 35 | 2 |
| $r_{X5}$ | 30 | 2 |
| $r_{X6}$ | 50 | 4 |

$p_Y$

| name | price | rating |
|---|---|---|
| $r_{Y1}$ | 12 | 0.5 |
| $r_{Y2}$ | 45 | 4 |
| $r_{Y3}$ | 42 | 4 |
| $r_{Y4}$ | 20 | 2 |
| $r_{Y5}$ | 25 | 2.5 |
| $r_{Y6}$ | 20 | 3 |

$p_Z$

| name | price | rating |
|---|---|---|
| $r_{Z1}$ | 40 | 5 |
| $r_{Z2}$ | 35 | 2.5 |
| $r_{Z3}$ | 38 | 2 |
| $r_{Z4}$ | 50 | 5 |
| $r_{Z5}$ | 15 | 1 |

then local results for the each peer are $L_X = \{r_{X2}, r_{X3}, r_{X5}\}$, $L_Y = \{r_{Y1}, r_{Y3}, r_{Y6}\}$, and $L_Z = \{r_{Z1}, r_{Z2}, r_{Z5}\}$, respectively. The global result is $B = \{r_{X3}, r_{Y6}, r_{Z1}\}$. For instance, $r_{X2} \in L_X$ but $r_{X2} \notin B$ since it is dominated by $r_{Z1}$ (same rating but lower price). On the other hand, if the user asks for higher-rated restaurants with price ranging in [30,50], $P_{\mathsf{wo}} = \mathsf{pos}(price \in [30, 50]) \rhd \mathsf{max}(rating)$ defines a wo. Local results are $L_X = \{r_{X2}\}$, $L_Y = \{r_{Y2}, r_{Y3}\}$, and $L_Z = \{r_{Z1}, r_{Z4}\}$, respectively, and now it is $B = \{r_{X2}, r_{Z1}, r_{Z4}\}$. □

The model we consider for query propagation and execution follows standard approaches in P2P proposals. A query $q : \beta_{\succ_P}(r)$ is issued at a peer, denoted $p_{init}$, which, besides computing its local result, will forward $q$ to a set of other peers in the network. These will recursively propagate $q$ to other peers in the network, so that a query tree for $q$, $T(q)$, rooted at $p_{init}$ originates. Each peer $p_j$ (but $p_{init}$) has a unique parent peer, $par_j$, and, if not a leaf, a set of children, $ch_j$. The sub-tree of $T(q)$ rooted at $p_j$ is denoted as $T_j$. Issues related to how $T(q)$ is actually built and on how many peers are involved in the evaluation of $q$ are orthogonal to the problem we address here and are not considered at all. We only require that the topology of $T(q)$ does not change during the evaluation of $q$. Further, we do not enter into details of how peers actually compute their local result, since this might depend on specific peer's algorithms [8].

As to peers' interface, we assume that each peer exports standard methods for incrementally delivering its results. Besides `Open`() and `Close`() methods,

[2] The definition of the composition operators is deliberately simplified here. Indeed, when composing spo preferences, care is needed to guarantee that the resulting preference is still an spo [11].

respectively needed for initializing internal structures and for terminating the execution, the interface exports a `GetNext()` method, which returns the next best result for the query under evaluation; if no more results can be delivered, an EndOfStream (EOS) message is returned.

In the above-sketched model, all strategies for evaluating $B = \beta_{\succ_P}(r)$ need to propagate partial results through the tree until they reach $p_{init}$, which will deliver the final result to the user. How this can be done by minimizing the network overhead, i.e., the number of tuples that flow through the network, is the problem we address in the following. The overall logic implemented by the peer $p_{init}$ at which a query $q$ originates is however common to all the algorithms we describe, and is summarized by Algorithm 1.

**Algorithm 1** Main @ peer $p_{init}$

1: $B \leftarrow L_{init}$ ▷ Initialize the global result with the local one
2: **for all** peers $p_i$ in $ch_{init}$ **do** ▷ This can be done in parallel
3: $id \leftarrow p_i.\texttt{Open}(q)$ ▷ $p_i$ assigns the ID $id$ to the query
4: **while** not $p_i.\texttt{EOS}(id)$ **do** $B \leftarrow \beta_{\succ_P}(B \cup \{p_i.\texttt{GetNext}(id)\})$ ▷ Updates $B$

## 3 Query Evaluation

The simplest (naïve) way of computing the result of preference queries is to have each peer sending all its local best results to $p_{init}$, which will eventually make all the necessary comparisons (Algorithm 2). This works since $B \subseteq \bigcup_j L_j$:

**Algorithm 2** Naïve `GetNext`($id$) @ peer $p_j$

1: **if** $first$ **then** compute $L_j$, $first \leftarrow$ **false** ▷ 1st invocation of `GetNext`($id$)
2: **if** $L_j \neq \emptyset$ **then** remove the first tuple from $L_j$ and return it
3: **else** ▷ All tuples in local result have been returned
4: **for all** peers $p_i$ in $ch_j$ **do**
5: **if** not $p_i.\texttt{EOS}(id_i)$ **then** return $p_i.\texttt{GetNext}(id_i)$
6: return $EOS$

Above algorithm is inefficient for two reasons: First, if the cardinality of the result, $|B|$, is much less than $\sum_j |L_j|$, many objects are needlessly sent up to the tree root; second, all the computation concerning comparison of local results is performed at $p_{init}$, thus not exploiting the parallelism offered by the network.

### 3.1 Evaluation of wo Queries

Let us first assume that the preference $P$ included in the query $q$ induces a weak order. In this case, it is possible to substantially improve over the Naïve `GetNext()` by deriving an algorithm, $\text{LocalBest}_{wo}$ `GetNext()` (Algorithm 3), that

is optimal as to the amount of data flowing through the network. The idea is twofold. First, each peer $p_j$, rather than sending to $p_{init}$ its local result $L_j$, delivers back to its parent $par_j$ only the best results, denoted as $L_{T_j}$, for its sub-tree $T_j$. This increases the level of concurrency and filters tuples not contributing to the final result earlier. The second idea is that, for computing $L_{T_j}$, $p_j$ does not need to retrieve all the results from (the sub-trees of) its children. The key observation here is that, when $P$ is a weak order, then if the first tuple in $L_{T_i}$ is dominated by the first tuple in $L_{T_k}$, where both $p_i$ and $p_k$ are children of $p_j$, then *all* the tuples of $L_{T_i}$ are dominated as well.

This is exploited in line 3 of Algorithm 3, where the set of *active* children of $p_j$ is determined by just fetching a *single* tuple from every child peer. Thus only peers producing undominated tuples are kept active, whereas transactions for non-active children can be immediately closed.

**Algorithm 3** $\mathsf{LocalBest}_{wo}$ $\texttt{GetNext}(id)$ @ peer $p_j$

1: **if** $first$ **then** $L_{T_j} \leftarrow L_j$, $first \leftarrow$ **false** ▷ 1st invocation of $\texttt{GetNext}(id)$
2: **for all** peers $p_i$ in $ch_j$ **do** $L_{T_j} \leftarrow \beta_{\succ_P}(L_{T_j} \cup \{p_i.\texttt{GetNext}(id_i)\})$
3: $active \leftarrow$ peers $p_i$ in $ch_j$ that provided tuples which are still in $L_{T_j}$
4: remove the first tuple from $L_{T_j}$ and return it
5: **else** ▷ Subsequent invocations of $\texttt{GetNext}(id)$
6: **for all** peers $p_i$ in $active$ **do**
7: **while** not $p_i.\texttt{EOS}(id_i)$ **do** $L_{T_j} \leftarrow L_{T_j} \cup \{p_i.\texttt{GetNext}(id_i)\}$
8: **if** $L_{T_j} \neq \emptyset$ **then** remove the first tuple from $L_{T_j}$ and return it
9: **else** return $EOS$ ▷ All result tuples have been returned

*Example 2.* Consider the scenario described in Example 1, where the preference is $P_{\mathsf{wo}} = \mathsf{pos}(price \in [30, 50]) \rhd \mathsf{max}(rating)$ and the query tree $T(q)$ has $p_{init} \equiv p_X$ and leaf nodes $p_Y$ and $p_Z$. Both $p_Y$ and $p_Z$ only compute their local results, $L_Y = \{r_{Y2}, r_{Y3}\}$ and $L_Z = \{r_{Z1}, r_{Z4}\}$, respectively. When $p_X$ probes $p_Y$ and $p_Z$, it first receives, say, $r_{Y2}$ and $r_{Z1}$ and compares them with its local result $L_X = \{r_{X2}\}$. Since $r_{X2} \sim_{P_{\mathsf{wo}}} r_{Z1}$ but $r_{X2} \succ_{P_{\mathsf{wo}}} r_{Y2}$, only $p_Z$ needs to be further accessed, and the global result is then correctly obtained as $B = \{r_{X2}, r_{Z1}, r_{Z4}\}$. □

### 3.2 Evaluation of spo Queries

When the preference $P$ is not a weak order, Algorithm 3 is not correct. Since in a strict partial order the indifference relation is not necessarily transitive, one cannot just look at the first result of a peer to decide that such peer cannot contribute to the final result. Therefore, local results for children nodes have to be collected at each parent node $p_j$, the result for the sub-tree rooted at $p_j$ is computed and it is forwarded up to $par_j$. This is formalized in Algorithm 4, where the $\mathsf{LocalBest}_{spo}$ strategy is shown.

**Algorithm 4** $\mathsf{LocalBest}_{spo}$ `GetNext`($id$) @ peer $p_j$

```
1: if first then L_{T_j} ← L_j, first ← false                ▷ 1st invocation of GetNext(id)
2:     for all peers p_i in ch_j do                          ▷ Retrieve all results from children nodes
3:         while not p_i.EOS(id_i) do L_{T_j} ← β_{≻_P}(L_{T_j} ∪ {p_i.GetNext(id_i)})
4:     remove the first tuple from L_{T_j} and return it
5: else                                                      ▷ Subsequent invocations of GetNext(id)
6:     if L_{T_j} ≠ ∅ then remove the first tuple from L_{T_j} and return it
7:     else return EOS                                       ▷ All result tuples have been returned
```

*Example 3.* Consider the case depicted in Example 2, where the preference is now expressed as $P_{\mathsf{spo}} = \mathsf{min}(price)$ & $\mathsf{max}(rating)$ and the query tree $T(q)$ is $p_{init} \equiv p_X - p_Y - p_Z$. The evaluation starts at $p_Z$, that as soon as its local result, $L_Z = \{r_{Z1}, r_{Z2}, r_{Z5}\}$, has been computed, sends the 3 tuples to $p_Y$. $p_Y$, in turn, computes its local result $L_Y = \{r_{Y1}, r_{Y3}, r_{Y6}\}$ and waits to receive $L_{T_Z}$ to compute $L_{T_Y} = \beta_{\succ_{P_{\mathsf{spo}}}}(L_Y \cup L_{T_Z}) = \{r_{Y1}, r_{Y6}, r_{Z1}, r_{Z5}\}$, which is sent to $p_X$. When $p_X$ receives $L_{T_Y}$, it can compute the global result $B$ as $\beta_{\succ_{P_{\mathsf{spo}}}}(L_X \cup L_{T_Y})$, correctly obtaining $B = \{r_{X3}, r_{Y6}, r_{Z1}\}$. □

Unlike $\mathsf{LocalBest}_{wo}$, $\mathsf{LocalBest}_{spo}$ performs all the computation in the first call of `GetNext`(), where local results of children nodes are collected and compared to build the result for the local tree, $L_{T_j}$. With respect to the Naïve algorithm, $\mathsf{LocalBest}_{spo}$ reduces the number of tuples that are sent through the network, since dominated tuples are trumped earlier.

### 3.3 Cost Analysis

We analyze the performance of above-presented algorithms according to the amount of traffic flowing through the network, by only counting the tuples that are sent through individual peers, e.g., from a peer $p_j$ to its parent node $par_j$ in $T(q)$. Here the aim is not to provide a detailed cost model for distributed preference queries, rather to highlight ways for possible improvements.

When using the Naïve algorithm, the local result of each peer $p_j$ is sent up to the root of $T(q)$, thus the cost paid for tuples obtained from $p_j$ is $|L_j| \cdot \lambda_j$, where $\lambda_j$ is the level of $p_j$ in $T(q)$ ($\lambda_{init} = 0$). The overall shipping cost is therefore:

$$cost(\mathsf{Naïve}) = \sum_j |L_j| \ \lambda_j \tag{1}$$

Should one be able to freely configure $T(q)$, above equation suggests to place peers providing largest local results close to the tree root. Clearly, this requires of being able to estimate $|L_j|$, a problem which is not completely solved yet [4].

For the $\mathsf{LocalBest}_{wo}$ algorithm, the overall cost depends on where active peers (i.e., the ones contributing to the result) are located. The cost is now:

$$cost(\mathsf{LocalBest}_{wo}) = \sum_{j \in active} |L_j|\lambda_j + \sum_{j \notin active} 1 = \sum_{j \in active} |B_j|\lambda_j + \sum_{j \notin active} 1 \tag{2}$$

Note that this cost is indeed optimal, in that, except for the result tuples (this cost has to be paid anyway), only a single tuple for each peer has to be transmitted. Clearly, it is impossible to pay a lower cost, since this would require to ignore a peer that may however actually provide results for the query.

The cost of $\mathsf{LocalBest}_{spo}$ can be derived by focussing on sub-trees, rather than on individual peers, and it is:

$$cost(\mathsf{LocalBest}_{spo}) = \sum_{j \neq init} |L_{T_j}| \tag{3}$$

Assume now that each peer somehow "magically" knows which tuples of its local result $L_j$ will contribute to the global result, i.e., $p_j$ knows $B_j$. Under this ideal assumption, the best one could obtain is:

$$cost(\mathsf{ideal}) = \sum_{j} |B_j| \lambda_j \tag{4}$$

One might wonder if the cost of $\mathsf{LocalBest}_{spo}$ can be bounded from above by a polynomial function of the size of global result, $|B|$. This would provide us with the guarantee that performance of $\mathsf{LocalBest}_{spo}$ never degenerates to the Naïve cost. Unfortunately, this is not the case, as the following example demonstrates.

*Example 4.* Consider a query $q : \beta_{\succ_P}(r)$, where $P$ is an spo, and the query tree $T(q)$ with $p_{init} \equiv p_X$ and leaf nodes $p_Y$ and $p_Z$. Assume that $L_Y = \{t\}$, $L_Z = \{t_1, \ldots, t_n\}$, and that $t \succ_P t_i, i \in [1, n]$. Then, we have $|B| = 1$, but $cost(\mathsf{LocalBest}_{spo}) = n + 1$, thus the cost cannot be upper-bounded. □

## 4 Reducing Costs for spo Queries

As we just saw, the $\mathsf{LocalBest}_{spo}$ algorithm is not able to provide adequate performance guarantees. In the following, we briefly sketch some basic strategies that could be exploited in order to reduce its cost. The common idea shared by such strategies is to make a peer somewhat aware of what other peers can contribute to the final result.

**Pushing-down tuples:** This is similar to semi-join strategies in distributed databases: when a join involves two sites, a possibility to reduce the amount of transmitted data is to have one site sending join values to the other, so that only tuples that satisfy the join condition are sent back. In our scenario, this strategy would push down the query tree some tuples which are currently in $B$, so as to filter out dominated tuples from local results.

**Sampling local results:** This strategy complements the previous one by addressing the problem of getting as soon as possible tuples which: (1) are likely to be in the final $B$ and (2) are likely to trump many other tuples. The idea is to have each peer in the query tree getting a representative sample of its children' results, and then push-down the sample obtained from a child to its siblings.

**Maintaining peers' synopses:** The third strategy we envision to reduce the amount of transmitted tuples consists in extending the peer network with a distributed directory, in which synopses of peers' contents are maintained. Such *content summaries*, which are typically used in P2P networks to intelligently drive the search towards peers relevant to a query [7], could provide a query-independent view of what each peer can make available to other peers. In this sense they could be exploited both for creating effective query trees and for driving the sampling process, e.g., by implementing a *biased* sampling strategy.

## 5 Conclusions

In this paper we have considered the problem of evaluating preference queries in P2P environments. While for weak order preferences the problem can be optimally solved, for generic strict partial order preferences we have shown that this is not the case. The set of strategies we have suggested to overcome this limitation will be the subject of future investigation. This will also include the analysis of more complex scenarios where preferences are expressed over two or more partitioned relations.

## References


1. W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *ICDE 2005*, pages 174–185, Tokyo, Japan, Apr. 2005.
2. I. Bartolini, P. Ciaccia, A. Linari, and M. Patella. Critical analysis of query processing techniques for heterogeneous environments. Tech. Rep. D3.R2, WISDOM - Italian MIUR Project, 2005. Available at URL `http://dbgroup.unimo.it/wisdom/`.
3. S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *ICDE 2001*, pages 421–430, Heidelberg, Germany, Apr. 2001.
4. S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE 2006*, Atlanta, GA, Apr. 2006.
5. J. Chomicki. Querying with intrinsic preferences. In *EDBT 2002*, pages 34–51, Prague, Czech Republic, Mar. 2002.
6. J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.
7. A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS 2002*, pages 23–32, Vienna, Austria, July 2002.
8. P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB 2005*, pages 229–240, Trondheim, Norway, Aug. 2005.
9. Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in MANETs. In *ICDE 2006*, Atlanta, GA, Apr. 2006.
10. W. Kießling. Foundations of preferences in database systems. In *VLDB 2002*, pages 311–322, Hong Kong, China, Aug. 2002.
11. W. Kießling. Preference queries with SV-semantics. In *COMAD 2005*, pages 15–26, Goa, India, Jan. 2005.
12. I. Tatarinov and A. Y. Halevy. Efficient query reformulation in peer data management systems. In *SIGMOD 2004*, pages 539–550, Paris, France, June 2004.
13. P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT 2006*, pages 112–130, Munich, Germany, Mar. 2006.