Relaxing Join and Selection Queries

Nick Koudas[1] Chen Li[§2] Anthony K. H. Tung[3] Rares Vernica[§2]

[1]University of Toronto, Canada
[2]University of California, Irvine, USA
[3]National University of Singapore, Singapore

ABSTRACT

Database users can be frustrated by having an empty answer to a query. In this paper, we propose a framework to systematically relax queries involving joins and selections. When considering relaxing a query condition, intuitively one seeks the 'minimal' amount of relaxation that yields an answer. We first characterize the types of answers that we return to relaxed queries. We then propose a lattice based framework in order to aid query relaxation. Nodes in the lattice correspond to different ways to relax queries. We characterize the properties of relaxation at each node and present algorithms to compute the corresponding answer. We then discuss how to traverse this lattice in a way that a non-empty query answer is obtained with the minimum amount of query condition relaxation. We implemented this framework and we present our results of a thorough performance evaluation using real and synthetic data. Our results indicate the practical utility of our framework.

1. INTRODUCTION

Issuing complex queries against large databases is a relatively simple task provided one has knowledge of the suitable query conditions and constants to use. Commonly however, although one might have a clear idea about the parameters, the resulting query may return an empty answer. In such cases, users often find themselves in a position having to try different parameters hoping to get an answer. Essentially query formulation becomes a trial-and-error process. One has to adjust the parameters until an answer is obtained with which one is relatively comfortable. The process of parameter adjustment is not at all trivial. The more complex a query is, in terms of predicates, the more choices one has to conduct such an adjustment. A similar situation arises when one is unclear about the right parameters to use, so trying parameters in speculation seems a natural option.

[§]Chen Li and Rares Vernica are partially supported by NSF CAREER Award No. IIS-0238586.

VLDB '06, September 12-15, 2006, Seoul, Korea.

As an example, consider a recruitment company that has a database with two tables. The table `Jobs` has records of job postings, with information such as job ID (`JID`), category (`Category`), company name (`Company`), zip code (`Zipcode`), and annual salary (`Salary`). The table `Candidates` has records of applicants, with information such as candidate ID (`CID`), zip code (`Zipcode`), expected salary (`ExpSalary`), and number of years of working experience (`WorkYear`). Some sample data of the relations is shown in Tables 1 and 2.

A user issues the following query:

```
SELECT *
FROM Jobs J, Candidates C
WHERE J.Salary <= 95
  AND J.Zipcode = C.Zipcode
  AND C.WorkYear >= 5;
```

The query seeks for job-candidate pairs, such that the job and the candidate are in the same area (same zip code), the job's annual salary is at most 95K, and the candidate has at least 5 years of working experience. Suppose the answer to the query turns out to be empty for the database instance. As we can see from the tables, each record in one relation can join with a record in the other relation. However, no such pair of records satisfy both selection conditions. In this situation, one way to get results is to be more flexible about the jobs, in terms of job's annual salary. Moreover, we can also be more flexible about the candidates, in terms of years of experience. By relaxing both selection conditions on `Salary` and `WorkYear`, we can get a nonempty answer. We can also compute an nonempty answer by relaxing the join condition, i.e., by allowing a job and a candidate to have similar but not necessarily identical zip codes. There are other ways to relax the conditions as well.

From this example, two observations are in order. First, there are different ways to relax the conditions. The number of choices for adjusting the conditions is large (exponential to the number of the conditions). Second, how much to adjust each condition is not obvious. For instance, for a condition `Salary <= 95`, we could relax it to `Salary <= 100` or `Salary <= 120`. The former has a smaller adjustment than the latter, but the new query may still return an empty answer. Although the space of possible choices is very large, it is natural to expect that a user would be interested in the smallest amount of adjustment to the parameters in the query in order to compute a nonempty answer. Clearly the semantics of such adjustments have to be precisely defined. In our running example, would a larger adjustment to the join condition be favored over two smaller adjustments to the two selection conditions?

JID	Category	Company	Zipcode	Salary
r1	Sales	Broadcom	92047	80
r2	Hardware Engineer	Intel	93652	95
r3	Software Engineer	Microsoft	82632	120
r4	Project Manager	IBM	90391	130
...	...	...	...	...

Table 1: Relation R: `Jobs`

CID	Zipcode	ExpSalary	WorkYear
s1	93652	120	3
s2	92612	130	6
s3	82632	100	5
s4	90931	150	1
...	...	...	...

Table 2: Relation S: `Candidates`

In this paper we put such questions into perspective and formally reason about the process of adjusting the conditions in a query that returns an empty answer, in order to obtain nonempty query results. We refer to this process and *query relaxation*. We make the following contributions:

- We formally define the semantics of the query relaxation problem for queries involving numeric conditions in selection and join predicates.
- We propose a lattice-based framework to aid query relaxation while respecting relaxation semantics that aims to identify the relaxed version of the query that provides a nonempty answer, while being "close" to the original query formulated by the user.
- We propose algorithms for various versions of this problem that conduct query evaluation at each node of our lattice framework aiming to minimize query response time while obtaining an answer.
- We present the results of a thorough experimental evaluation, depicting the practical utility of our methodology.

This paper is organized as follows. Section 2 presents our overall framework. Section 3 presents our algorithms for relaxing selection conditions in equi-join queries. In Section 4 we study how to relax all conditions in a query. In Section 5 we show how to adapt our algorithms to variants of query relaxation. Section 6 contains the results of our experimental evaluation. Section 7 concludes the paper.

1.1 Related Work

A study related to our work presented herein is the work of Muslea et al. [23, 24]. In these papers they discuss how to obtain alternate forms of conjunctive expressions in a way that answers can be obtained. Their study however deals primarily with expressibility issues without paying attention to the data management issues involved. Another related piece of work is [1], where a method for automated ranking of query results is presented.

Efficient algorithms for computing skylines have been in the center of research attention for years. The basic idea of skyline queries came from some old research topics like contour problem [22], maximum vectors [20] and convex hull [28]. Recently there are studies on efficient algorithms for computing skylines, e.g., [4, 8, 31, 19, 25, 12, 26, 32, 27]. Unlike these works which aim to support answers of preference queries, our focus is on relaxing queries with selection conditions and *join conditions*. Consequently, unlike these studies which assume the attributes and ordering of the values are already pre-determined in a **single** table, our work require us to compute skyline dynamically for **a set of tables** which are to be joined and whose attribute values (i.e., the amount of relaxation) must also be determined on the fly. We are not aware of any work utilizing such structures for query relaxation, especially join-query relaxations.

Several papers have been devoted to the problem of answering top-k queries efficiently [21, 5, 6]. These work focus on finding k tuples in the database that are ranked the highest based on a scoring function. Users can assign weights to various attributes in the database so as to express their preference in the scoring function. Our study involves finding the skyline of **relaxations** for select and join conditions such that each set of relaxations is guaranteed to return at least 1 tuple in the result set. **Both** the selection and join conditions must be considered for relaxation in order for this to take place, unlike top-k queries which focus on only the selection conditions.

Some of our algorithms are related to similarity search in multiple-dimensional data, such as R-trees and multidimensional indexing structures [13, 29, 3], and nearest neighbor search and all pair nearest search [15, 30]. Several approaches have been proposed in the literature to relax queries. For example, Gaasterland [11] studied how to control relaxation using a set of heuristics based on semantic query-optimization techniques. Kadlag et al. [16] presented a query-relaxation algorithm that, given a user's initial range query and a desired cardinality for the answer set, produces a relaxed query that is expected to contain the required number of answers based on multi-dimensional histograms for query-size estimation. Finally, our work is also related to the work on preference queries [18, 7, 9, 17]

2. QUERY-RELAXATION FRAMEWORK

In this section, we define our framework of relaxing queries with joins and selections. For simplicity, we focus on the case in which a query joins two relations; our results are easily extendable to the case of multiple joins. Let R and S be two relations. We consider join queries that are associated with a set of selection conditions on R and S, and a set of join conditions. Each selection condition is a range condition on an attribute. A typical form of a range condition is "$A\ \theta\ v$", where A is an attribute, v is a constant value, and θ is a comparison operator such as $=, <, >, \leq$, or $\geq$. Examples are `Salary <= 95`, `WorkYear >= 5`, and `Age = 30`. Each join condition is in the form of "$R.A\ \theta\ S.B$", where A is an attribute of R, and B is an attribute of S.

2.1 Relaxing Conditions

We focus on relaxing conditions on numeric attributes, whose relaxations can be quantified as value differences. Consider a query Q and a pair of records $\langle r, s\rangle$ in the two relations. For each selection condition

$$C : R.A\ \theta\ v$$

in R, the *relaxation* of r with respect to this condition is:

$$\mathsf{RELAX}(r, C) = \begin{cases} 0; & \text{if } r \text{ satisfies } C; \\ |r.A - v|; & \text{otherwise.} \end{cases}$$

Similarly, we can define the relaxation of record s with respect to a selection condition on S. The relaxation of the pair with respect to a join condition $J : R.A\ \theta\ S.B$ is:

$$\mathsf{RELAX}(r,s,J) = \begin{cases} 0; & \text{if } r,s \text{ satisfy } J; \\ |r.A - s.B|; & \text{otherwise.} \end{cases}$$

For instance, consider the query in our running example. Let C_R be the selection condition `J.Salary <= 95`. We have $\mathsf{RELAX}(r1, C_R) = 0$, since $r1$ satisfies this selection condition. In addition, $\mathsf{RELAX}(r3, R) = 25$, since record $r3$ does not satisfy the condition. Let J be the join condition, `J.Zipcode = C.Zipcode`. We have $\mathsf{RELAX}(r2, s1, J) = 0$, since the records $r2$ and $s1$ satisfy the join condition, while $\mathsf{RELAX}(r2, s2, J) = 1040$, since the records $r2$ and $s2$ do not satisfy this join condition.

Let the set of selection conditions for R in query Q be $C_{Q,R}$, for S be $C_{Q,S}$, and the set of join conditions be $C_{Q,J}$. Intuitively, every tuple $r \in R$ and every tuple $s \in S$ can produce an answer with respect to the query Q for some sufficiently large relaxation on the set of conditions $C_{Q,R} \cup C_{Q,S} \cup C_{Q,J}$. We denote this set of relaxations on different conditions as $\mathsf{RELAX}(r, s, Q)$. To separate out the relaxations for $C_{Q,R}$, $C_{Q,S}$, and $C_{Q,J}$, we will denote the relaxations for them as $\mathsf{RELAX}(r, C_{Q,R})$, $\mathsf{RELAX}(s, C_{Q,S})$, and $\mathsf{RELAX}(r, s, C_{Q,J})$, respectively.

2.2 Relaxation Skyline

Obviously, $\mathsf{RELAX}(r, s, Q)$ is different for different pairs of $\langle r, s\rangle$. Given two tuple pairs $\langle r_1, s_1\rangle$ and $\langle r_2, s_2\rangle$, it is possible that the first pair is "better" than the second in terms of their relaxations. To formulate such a relationship, we make use of the concept of "skyline" [4] to define a partial order among the relaxations for different tuple pairs.

DEFINITION 1. **(Dominate)** *We say* $\mathsf{RELAX}(r_1, s_1, Q)$ *dominates* $\mathsf{RELAX}(r_2, s_2, Q)$ *if the relaxations in* $\mathsf{RELAX}(r_1, s_1, Q)$ *are equal or smaller than the corresponding relaxations in* $\mathsf{RELAX}(r_2, s_2, Q)$ *for all the conditions and smaller in at least one condition.*

DEFINITION 2. **(Relaxation Skyline)** *The* relaxation skyline *of a query Q on two relations R and S, denoted by* $\mathsf{SKYLINE}(R, S, Q)$*, is the set of all the tuple pairs, $\langle r, s\rangle$, $r \in R$, $s \in S$, each of which has its relaxations with respect to Q not dominated by any other tuple pair $\langle r's'\rangle$, $r' \in R$, $s' \in S$.*

Computing the relaxation skyline for all the conditions of a query can ensure at least one relaxed answer being returned, it can sometimes return too many incomparable answers with large processing overhead due to the need to relax all the conditions.[1] Returning many results is not useful for users who just want a small set of answers. In addition, depending on the semantics of the query, often a user does not want to relax some conditions. For instance, in the job example, the user might not want to relax the join condition.

Therefore, we also consider the case where we do relaxations on a subset of the conditions, and compute the corresponding relaxation skyline with respect to these conditions. The tuple pairs on this relaxation skyline have to satisfy the conditions that cannot be relaxed. Interestingly, the various combinations of the options to relax these conditions form a lattice structure. For instance, consider the three conditions in our running example. Fig. 1 shows the lattice structure of these different combinations. In the lattice, for each node n, the conditions being relaxed at its descendants, are a superset of those being relaxed at this node. In such a case, it is natural that the set of tuple pairs in the relaxation skyline corresponding to this node n is a subset of those in the corresponding relaxation skyline for each descendant of n. By analyzing various factors that will result in an empty answer, we can try to identify the highest nodes in the lattice that can bring a user specified number of answers.

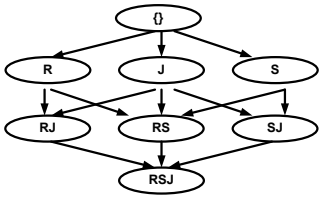


Figure 1: Lattice structure of various combinations of relaxations in the query in the jobs example. "R", "S", and "J" stand for relaxing the selection condition in `jobs`, the selection condition in `candidates`, and the join condition, respectively.

In our framework, a user can also assign a weight to each of the conditions in a query, and a value k. Then the system computes the k best answers using these weights. That is, for all the pairs in the relaxation skyline of the query, we return the k pairs that have the k smallest weighted summation of the relaxations on the conditions. In this way, the user can specify preference towards different ways to relax the conditions. In addition, computing the final answers could be more efficient. In Section 5.1 we show how to extend our algorithms for this variant.

3. ALGORITHMS FOR RELAXING SELECTION CONDITIONS

We study how to compute the relaxation skyline of a query with equi-join conditions, when we do not want to relax its join conditions, i.e., we only relax (possibly a subset of) its selection conditions. The motivation is that, many join conditions are specified on identifier attributes, such as employee ID, project ID, and movie ID. This case happens especially when we have a join between a foreign-key attribute and its referenced key attribute. Semantically it might not be meaningful to relax such a join attribute.

For instance, in our running example, we are allowed to relax the selection conditions C_R (`Salary <= 95`) and C_S (`WorkYear >= 5`), but we do not relax the join condition C_J (`Jobs.Zipcode = Candidates.Zipcode`). That is, each pair of records $\langle r, s\rangle$ of R and S in the relaxation skyline with respect to these two selection conditions should satisfy:

- $\mathsf{RELAX}(r, s, \mathsf{C_J}) = 0$, i.e., `r.Zipcode=s.Zipcode`.
- This pair cannot be dominated by any other joinable pair, i.e., there does not exist another pair $\langle r', s'\rangle$ of records such that:
  - `r'.Zipcode=s'.Zipcode`;
  - $\mathsf{RELAX}(r', \mathsf{C_R}) \leq \mathsf{RELAX}(r, \mathsf{C_R})$;
  - $\mathsf{RELAX}(s', \mathsf{C_S}) \leq \mathsf{RELAX}(s, \mathsf{C_S})$.

  One of the two inequalities should be strict.

[1]Technically a query has a final projection to return the values for some attributes. We assume that the main computational cost is to compute those pairs.

The job-candidate pair $\langle r1, s1\rangle$ is not in the relaxation skyline since its join relaxation is not 0. The pair $\langle r4, s4\rangle$ is not in the answer since it is dominated by the pair $\langle r2, s1\rangle$. The relaxation skyline with respect to the two selection conditions should include two pairs $\langle r2, s1\rangle$ and $\langle r3, s3\rangle$. Both pairs respect the join condition, and neither of them is dominated by the other pairs. The first pair has the smallest relaxation on condition C_R, while the second has the smallest relaxation on condition C_S. In this section we develop algorithms for computing a relaxation skyline efficiently. In Section 4 we will study the general case where we want to relax join conditions as well.

3.1 Pitfalls

Let Q be a query with selection conditions and join conditions on relations R and S, and the query returns an empty answer set. To compute the relaxation skyline with respect to the selection conditions, one might be tempted to develop the following simple (but incorrect) algorithm. Compute the set K_R (resp. K_S) of the relaxation skyline points with respect to the selection conditions for relation R (resp. S). Then join the two sets K_R and K_S. For instance, in our running example, this algorithm computes the relaxation skyline of the relation `Jobs` with respect to the selection condition C_R: `J.Salary <= 95`. The result includes the jobs $r1$ and $r2$, whose salary values satisfy the selection condition C_R. Similarly, it also computes the relaxation skyline of relation `Candidates` with respect to the selection condition C_S: `C.WorkYear >= 5`, and the result has two records, $s2$ and $s3$, which satisfy the selection condition C_S. It then joins the points on the two relaxation skylines, and returns an empty answer.[2] The example shows the reason why this naive approach fails. Intuitively, the algorithm relaxes the selection conditions of each relation *locally*. However, our goal is to compute the pairs of tuples that are not dominated by any other pair of tuples with respect to *both* of the selection conditions, not just one selection condition of a relation. Trying to compute the dominating points in each relation and then joining them will lead to missing some points that might form tuples that would not be dominated.

3.2 Algorithm: JoinFirst (JF)

This algorithm, called JoinFirst, starts by computing a join of the two relations without using the selection conditions. It then computes a skyline of these resulting tuple pairs with respect to the relaxations on the selection conditions. Algorithm 1 describes the pseudo code of this algorithm.

Figure 1 JoinFirst

```
1: Compute tuple pairs respecting the join conditions, without
   considering the selection conditions;
2: Compute the skyline of these tuple pairs with respect to re-
   laxations on the selection conditions;
3: Return the pairs in the skyline (with necessary projection).
```

In our running example, the first step of the algorithm will compute the join of two relations with respect to the join condition `J.zip=C.zip`. In the second step, it computes the job-candidate pairs in this result that cannot be dominated by other pairs with respect to the relaxation on the C_R and C_S conditions. There are different ways to implement each step in the algorithm. In the join step, we can do a nested-loop join, a hash-based join, a sort-based join, or an index-based join. In the second step, we can use one of the skyline-computing algorithms in the literature, such as the block-nested-loops algorithm in [4]. One advantage of this algorithm is that it can use those existing algorithms (e.g., a hash-join operator inside a DBMS) as a black box without any modification. However, the algorithm may not be efficient if the join step returns a large number of pairs.

[2]There are examples showing that, even if this approach returns a nonempty answer set, the result is still not the corresponding relaxation skyline.

3.3 Algorithm: PruningJoin (PJ)

This algorithm tries to reduce the size of the results after the join step in the JoinFirst algorithm by computing the relaxation skyline *during* the join step. Algorithm 2 describes the pseudo code of this algorithm, assuming we are doing an index-based join using an index structure on the join attributes of S. The algorithm goes through all the records in relation R. For each one of them (say r), it uses an index structure on the join attribute of S to find those S records that can join with this record (say s). For each such record s, the algorithm calls a procedure "Update" by passing the pair $\langle r, s\rangle$ and the current skyline. This procedure checks if this pair is already dominated by a pair in the current relaxation skyline K. This dominance checking is based on Definition 1, assuming we can compute the relaxation of this record pair for each condition in the query.[3] We discard this pair if it is already dominated. Otherwise, we discard those pairs in K that are dominated by this new pair, before inserting this pair to K. The algorithm terminates when we have processed all the records in R.

Figure 2 PruningJoin (Index based)

```
1: Relaxation skyline K = empty;
2: for each tuple r in R do
3:     I = index-scan(S, r); // joinable records in S
4:     Call Update(⟨r, s⟩, K) for each tuple s in I;
5: end for
6: return K;
7: procedure UPDATE(element e, skyline K)
8:     if e is dominated by an element in K then
9:         discard e;
10:    else
11:        discard K's elements dominated by e;
12:        add e to K;
13:    end if
14: end procedure
```

The description can be easily modified for other possible physical implementations of the join. For instance, if we want to do a hash-based join, we first bucketize both relations. For each pair of buckets from the two relations, we consider each pair of records from these two buckets, and check if this tuple pair can be inserted into the current relaxation skyline, and potentially eliminate some existing record pairs. The algorithm terminates when all the pairs of buckets are processed. Extensions to other types of join methods (e.g., nested-loop or sort-based) are similar.

[3]Technically the dominance checking in the "Update" procedure relies on a set of query conditions. For simplicity, we assume the skyline K already includes these query conditions and the corresponding method to do the dominance checking, so that this procedure can be called by other algorithms.

One advantage of this algorithm (compared to the Join-First algorithm) is that it can reduce the number of pair records after the join (which might be stored in memory), since this algorithm conducts dominance checking on the fly. One disadvantage is that it needs to modify the implementations of different join methods.

3.4 Algorithm: PruningJoin$^+$ (PJ$^+$)

The algorithm modifies the PruningJoin algorithm by computing a "local relaxation skyline" for a set of records in one relation that join with a specific record in the other relation, and doing dominance checking within this local skyline. Algorithm 3 describes the algorithm, and it is based on an index-scan-based join implementation. For each record r in R, after computing the records in S that can join with r (stored in I in the description), the algorithm goes through these records to compute a local relaxation skyline L with respect to the selection conditions on S. Those locally dominated S records do not need to be considered in the computation of the global relaxation skyline. If both records s_1 and s_2 of S can join with record r, and s_1 dominates s_2 with respect to the selection conditions on S, then pair $\langle r, s_1\rangle$ also dominates $\langle r, s_2\rangle$ with respect to all the selection conditions in the query. Therefore the second pair cannot be in the global relaxation skyline. Extensions of the algorithm to other join implementation methods are straightforward.

Figure 3 PruningJoin$^+$ (Index based)

```
1: Relaxation skyline K = empty;
2: for each tuple r in R do
3:     I = index-scan(S, r); // joinable records in S
4:     Local relaxation skyline L = empty;
5:     Call Update(s, L) for each tuple s in I;
6:     Call Update(⟨r, s⟩, K) for each tuple s in L;
7: end for
8: return K;
```

EXAMPLE 3.1. *Consider the following query on two relations* `R(A, B, C)` *and* `S(C, D, E)`.

```
SELECT *
FROM R, S
WHERE R.A = 10 AND R.B = 30
  AND R.C = S.C
  AND S.D = 70 AND S.E = 90;
```

Fig. 2 shows an example. Currently there are two record pairs (p_1 and p_2) in the global relaxation skyline. For the given record r of relation R, $\langle 13, 34, 55\rangle$, there are four S records that join with record r. Among these four, record s_2 is locally dominated by record s_1, since the relaxations of s_2 on the two local selection conditions are both larger than those of record s_1. The local relaxation skyline of this record r will contain three records, s_1, s_3, and s_4. Among the three corresponding tuple pairs, $\langle r, s_1\rangle$ is dominated by the existing pair p_2. The two remaining pairs, $\langle r, s_3\rangle$ and $\langle r, s_4\rangle$, will be inserted into the global relaxation skyline.

Notice that this algorithm does the local pruning using those local relaxation skylines, hoping that it can eliminate some S records locally. This local pruning is not always beneficial to performance, especially when the local pruning does not eliminate many S records. As our experiments have verified, whether the overhead of this local pruning is worth the performance gains depends on several factors, such as the number of conditions.

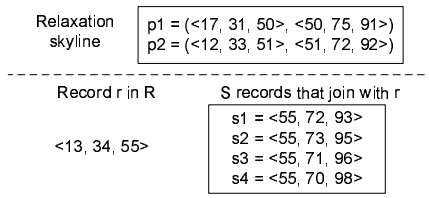


Figure 2: Example of algorithm PruningJoin$^+$.

3.5 Algorithm: SortedAccessJoin (SAJ)

This algorithm adopts the main idea in Fagin's algorithm, originally proposed to compute answers to top-k queries [10]. As shown in Algorithm 4, the algorithm first constructs a sorted list of tuple IDs for each selection condition in the given query Q, based on the relaxation of each record on that selection condition. Such a list can be obtained efficiently, e.g., when the corresponding table has an indexing structure such as B-tree. The algorithm goes through the lists in a round-robin fashion. For each of them L_i, it retrieves the next tuple ID (in an ascending order) and the corresponding tuple p. It then uses an available index structure on the other table to find records that can join with this record p, and stores them in I. For each such joinable tuple q, we form a tuple pair $\langle p, q\rangle$. We insert this pair into the set of candidate pairs P, if it is not in the set. The algorithm calls a function "CheckStopCondition()" to check if we can stop searching for tuple pairs. If so, we process all the candidate pairs in P to compute a relaxation skyline.

Figure 4 SortedAccessJoin

```
1: Let C_1, ..., ..., C_n be the selection conditions on R, and
   C_{n+1}, ..., C_{n+m} be the selection conditions on S;
2: Let L_i (i = 1, ..., n+m) be a sorted list of record IDs based
   on their relaxation on the selection condition C_i (ascending
   order);
3: set of candidate pairs P = empty;
4: StopSearching = false;
5: // produce a set of candidate pairs
6: while not StopSearching do
7:     Attribute j = round-robin(1, ..., n+m);
8:     Retrieve the next tuple ID k from list L_j;
9:     Retrieve the corresponding tuple p using k;
10:    I = index-scan(the other relation, p);
11:    for each-tuple q in I do
12:        if (⟨p, q⟩ not in P)
13:            insert ⟨p, q⟩ in P;
14:        StopSearching = CheckStopCondition();
15:    end for
16: end while
17: // compute Skyline
18: Relaxation skyline K = empty;
19: for each-tuple-pair ⟨r, s⟩ in P do
20:     Update(⟨r, s⟩), K);
21: end for
22: return K;
```

In the "CheckStopCondition()" function, we check if the current tuple pair $\langle p, q\rangle$ has a smaller relaxation than each of the current records on the lists, except the current list L_i. That is, the function returns true only if for each list L_j ($j \neq i$), the relaxation of this tuple pair on the condition C_j

is smaller than the relaxation of the current record on list L_j on this condition C_j. The stopping condition is based on the following observation. If a pair of records has a smaller relaxation than the current record on every of the sorted lists, than it is guaranteed that this pair will dominate all the pair of records formed by records starting on or bellow the current value on each of the lists. A similar approach was used for computing a skyline in a single relation [2].

EXAMPLE 3.2. *To understand the stopping condition for algorithm SAJ, consider again the query in Example 3.1. Fig. 3 shows the four sorted access lists $C_1, \ldots, C_4$ for the four selection conditions. The algorithm first starts processing record r_1 with its $r_1.A = 13$ based on the round robin. By performing the index scan at Line 10, it inserts four pairs, $\langle r_1, s_1\rangle$, $\langle r_1, s_2\rangle$, $\langle r_1, s_3\rangle$, and $\langle r_1, s_4\rangle$, into the result set I. These four joined tuple pairs are then subsequently inserted into the candidate set P at Line 13. The pointer for C_1 is then moved to the second record "$r_2.A = 6$" on the list. The round robin algorithm will next process record $r_2.B = 34$ on the second list. The process is repeated for $s_4.D = 70$ and $s_4.E = 92$ with all the relevant pointers being moved to the next tuple in the corresponding list.*

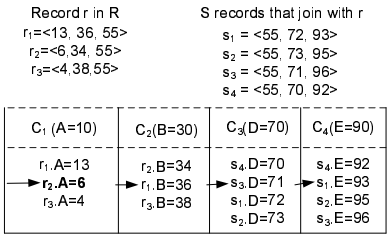


Figure 3: Example of algorithm SAJ.

When processing $r_2.A = 6$ (in bold), joined tuple pairs $\langle r_2, s_1\rangle$, $\langle r_2, s_2\rangle$, $\langle r_2, s_3\rangle$, and $\langle r_2, s_4\rangle$ are in I, and subsequently checked against the stopping condition at line 14. When checking the stopping condition, we see that the joined tuples $\langle r_2, s_4\rangle$ is found to have a smaller relaxation than all the tuples being pointed to in C_2 ($r_1.E$=93), C_3 ($s_3.D$=71), and C_4 ($r_1.B$=36). Since the stopping condition is satisfied, the while loop terminates. The skyline is then computed over all the tuples in P to produce the final result.

4. RELAXING ALL CONDITIONS

In the previous section we studied how to relax selection conditions in a equi-join query without relaxing its join conditions. There are cases where a join condition (not necessarily equi-join) can be relaxed to return meaningful answers. In our running example, the user may want to relax the join condition `jobs.zipcode = candidates.zipcode`, hoping to find candidates who are close to the job locations, even though their zip codes might not be identical. Developing algorithms for relaxing all the conditions, especially the join conditions, is more challenging, since record pairs that could be on the relaxation skyline may not agree on their join conditions. As a consequence, it is not straightforward to identify these tuple pairs.

In this section we present an algorithm for computing a relaxation skyline for a join query of two relations R and S, assuming we can allow the join conditions, and possibly other selection conditions to be relaxed. The algorithm assumes there is a multi-dimensional indexing structure on the selection attribute(s) and join attribute(s) of each relation. A typical structure is an R-tree [13]. We explain the algorithm assuming two R-trees on the two relations; the algorithm is extendable to other tree-based structures. We describe the algorithm for the case where we want to relax the selection conditions C_R on relation R, the selection conditions C_S on the relation S, and the join conditions C_J. The algorithm can be easily adapted to cover each variant of query relaxation (e.g., see Fig. 1).

4.1 Algorithm: MIDIR

The algorithm is called "MultI-Dimensional-Index-based Relaxation" (MIDIR). It uses the two R-trees and computes the relaxation skyline of the points (records). The algorithm traverses the two R-trees top-down, and constructs pairs of R-tree nodes or points that could potentially be in the relaxation skyline. During the traversal, the algorithm uses a queue to store the candidate pairs. It uses a list K to keep track of the skyline points seen so far during the traversal. At the beginning, the pair of the two roots is pushed to the queue. In each iteration, the algorithm pops a pair p from the queue. There are four cases about this pair: an object-object pair, an object-MBR (minimum bounding box) pair, an MBR-object pair, or an MBR-MBR pair. For the first case, the algorithm calls the function "Update" (defined in the PJ algorithm) to see if this pair can be inserted into skyline K, and eventually eliminate some pairs in K. For each of the other three cases, the algorithm considers the corresponding child pairs accordingly. The algorithm inserts each new pair to the queue only if no pair in the skyline can dominate this pair p.

4.2 Dominance Checking with MBRs

The algorithm needs to check the dominance from a pair p' in the skyline to the given pair p. Notice that these pairs could be object-MBR, MBR-object, or MBR-MBR pairs. In order to do the dominance checking for the relaxation of such a pair with respect to a condition, we need to compute a lower bound and an upper bound on this relaxation. For a pair t and a condition C_i, let $\mathsf{RELAX}_{MIN}(t, C_i)$ be a lower bound of its relaxation on this condition. That is, for every pair of tuples inside the MBR(s) mentioned in t, its relaxation with respect to this condition is no smaller than this lower bound. Similarly, let $\mathsf{RELAX}_{MAX}(t, C_i)$ be an upper bound of its relaxation on this condition. Clearly these two bounds can become the same when we have a pair of objects (tuples). We say the pair p' dominates the pair p if for each condition C_i in the query, we have

$$\mathsf{RELAX}_{MAX}(p', C_i) \leq \mathsf{RELAX}_{MIN}(p', C_i),$$

and the inequality is strict for at least one condition.

Now let us see how to compute a lower bound $\mathsf{RELAX}_{MIN}(t, C_i)$ and an upper bound $\mathsf{RELAX}_{MAX}(t, C_i)$ for a pair t with respect to a condition C_i. We focus on computing the lower bound, and the upper bound can be computed similarly. The main idea is to convert the problem to computing the minimum distance between two intervals of the attribute in the condition. If C_i is a selection condition, we convert it to an interval $I(C_i) = [v_{low}, v_{high}]$, which is the range of the values that can satisfy this condition. Here are a few examples of selection conditions and their corresponding intervals. Notice that for generality of this discussion, we

can represent a point as an interval.

Condition C	Corresponding Interval $I(C)$
`Salary <= 95`	$[0, 95]$
`WorkYear >=5`	$[5, +\infty)$
`age = 28`	$[28, 28]$

Let B be an MBR or a tuple, A be an attribute used in B. Let $I(B, A)$ denote the interval of this attribute in this MBR. We consider the following cases to compute $\mathsf{RELAX}_{MIN}(t, C_i)$:

- C_i is a selection condition. Let B be the MBR or tuple in t in which the attribute A_i of C_i appears. Then $\mathsf{RELAX}_{MIN}(t, C_i) = \text{MINDIST}(I(C_i), I(B, A_i))$, i.e., the minimum distance between the two intervals.
- C_i is a join condition. Let B_R be the MBR or tuple t (in relation R), in which the attribute A_i of C_i appears. Let B_S be the one for relation S. Then $\mathsf{RELAX}_{MIN}(t, C_i) = \text{MINDIST}(I(B_R, A_i), I(B_S, A_i))$.

The minimum distance between two intervals $I_a = [a_{low}, a_{high}]$ and $I_b = [b_{low}, b_{high}]$, represented as $\text{MINDIST}(I_a, I_b)$, can be computed as follows. If $a_{low} \in [b_{low}, b_{high}]$ or $a_{high} \in [b_{low}, b_{high}]$, then the minimum distance is 0. Otherwise, it is $min(|a_{high} - b_{low}|, |a_{low} - b_{high}|)$.

Fig. 4(a) shows an example of computing minimum relaxations and maximum relaxations on two selections on a relation R. The figure shows two selection conditions `X <= x0` and `Y <= y0` on two attributes X and Y. It does not show the attributes for the join conditions. It shows the minimum relaxation and maximum relaxation for the box $MBR1$ with respect to these two conditions. In addition, in Fig. 4(b), the point $(x1, y1)$ is an R record r in one pair of records in the current skyline computed so far. (Again, the values of the join attributes are not shown.) Consider the three MBRs for relation R. $MBR1$ is dominated by record r w.r.t. the selection conditions because its minimum relaxations on both attributes are larger than those of record r. $MBR2$ is not dominated by record r, since although its minimum relaxation on attribute Y is larger than that of record r, its minimum relaxation on attribute X is smaller than that of r. Similarly, $MBR3$ is not dominated by record r either, since its minimum relaxation on attribute Y is smaller than that of record r.

4.3 Extension to Other Relaxation Combinations

We can modify the MIDIR algorithm slightly to compute a relaxation skyline if the user does not want to relax some of the conditions in the query. Consider each condition C_i that the user does not want to relax. We make two changes to the algorithm.

- In each dominance checking, we do not consider relaxation on this condition C_i.
- Before inserting each pair into the queue, we verify if this pair could potentially satisfy this condition C_i, and discard those pairs that cannot satisfy C_i. This verification can be easily done for a pair of records. In the case where the pair has an MBR, we can also check if the records in this MBR could satisfy the condition C_i.

For instance, consider our running example. Suppose the user wants to relax the selection condition `J.Salary <= 95` and the join condition `J.zipcode = C.zipcode`, but not the selection condition `C.WordYear >= 5`. In this case, when we insert a pair of records into the queue, if the `candidates` record in the pair cannot satisfy this condition, we do not need to do the insertion. The same checking is done when inserting a pair with an MBR.

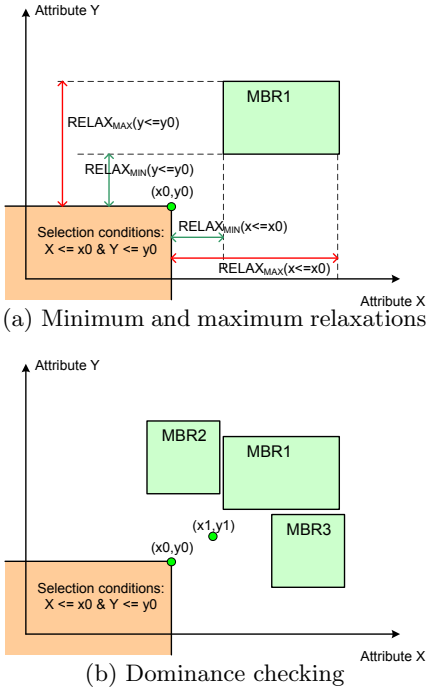


Figure 4: Pruning using minimum and maximum relaxations.

5. VARIANTS OF QUERY RELAXATION

In this section we show that our earlier algorithms can be extended to other variants of query relaxation.

5.1 Computing Top-k Answers on Relaxation Skylines

The algorithms we have developed so far treat the relaxations on different conditions in a query independently, assuming that they are equally important. Very often a user has different weights on these relaxations. In this case, one may want the k best answers *on the relaxation skyline* based on these weights, where k is a number specified by the user. That is, for all the points on the skyline, we return the k points that have the smallest weighted summations (scores) of the relaxations on those relaxable conditions. Computing top-k answers on the skyline is based on the following two assumptions. (1) The weight of a condition defines the importance of relaxing this condition; and (2) the user does not want to see answers that are already dominated by other answers. Other approaches to computing best answers also exist, e.g., the approach based on computing top-k answers (not necessarily on the skyline) [6].

Let Q be a join query of two relations R and S with selection conditions and join conditions. One specifies a weight W_i for each condition C_i in the query. For each pair $\langle r, s \rangle$ of records in the two relations, the overall relaxation with

respect to query Q is:

$$\mathsf{RELAX}(r, s, Q) = \sum_i w_i \times \mathsf{RELAX}(r, s, C_i).$$

For an integer k, the *top-k pairs* are the k pairs *on the relaxation skyline* that have the k *smallest* overall scores. Notice that this definition is different from a traditional definition of top-k pairs, which might not be on the skyline. The following example shows the difference. For instance, in the running example in Section 1, consider the query and the two pairs: $\langle r2, s1\rangle$ and $\langle r4, s4\rangle$. In a traditional top-k case, both of them could end up being in the top-2 pairs. In the case of top-k over skyline, the second pair will never be in the result set, since it is dominated by the first one.

All our algorithms can be extended to compute top-k skyline answers efficiently. The main idea is, instead of keeping a relaxation skyline in each algorithm, we keep a buffer to store the k best skyline answers. Every time we have a new candidate pair, in order to decide whether it is in the top-k, we have to compare this pair with those in the buffer. We add this new pair into the buffer only if its weighted relaxation is smaller than the k-th answer in the buffer (as a consequence, it cannot be dominated by other answers in the buffer). We can implement this change, e.g., by modifying the function "Update()" defined in the PJ algorithm.

We can further improve the MIDIR algorithm by implementing its queue as a priority queue, sorted based on the weighted summation of the relaxations of its pairs in an ascending order. If an MBR is in a pair, we do not know all its records. However, for each attribute of the MBR, we can still compute a lower bound and an upper bound for the relaxation for its condition (Section 4.2). We then use the weights to compute a lower bound and an upper bound on the total relaxation for this pair. We sort all the pairs in the queue based on their lower bounds on each attribute. The algorithm can terminate when the queue has produced k pairs of tuples (not MBRs). Due to the order of the queue, we can guarantee that these k pairs are top-k answers on the relaxation skyline. To see the reason, if there was another pair of records that had a smaller overall relaxation than these k reported pairs, then the lower bound of the overall relaxation for the pairs of MBRs (or its variants) of the latter pair should also be smaller than the overall relaxations of these k pairs. Then this pair should been popped earlier.

5.2 Queries with Multiple Joins

All our algorithms can be easily extended to the case where we want to relax a query with multiple joins (not necessarily joining two relations). For instance, if the user does not want to relax the join conditions in such a query, we can modify algorithm JF that computes the join result without using the selection conditions, then compute a skyline using the join result. Similarly, extending the SAJ algorithm, we can first obtain a sorted list of records for each selection condition based on the relaxation of each record on the corresponding condition. When we access the records along the list, we use available indexing structures to access other records that can join with the current record. Since the query could have multiple joins, we need to access multiple index structures to access other records that could join with earlier joinable records. The rest of the algorithm is the same as before. For the MIDIR algorithm, we can also traverse the available R-trees on multiple relations, and use a queue to maintain possible "vectors" of records or MBRs when searching for skyline answers.

5.3 Relaxing Conditions on Nonnumeric Attributes

Often queries have conditions on nonnumeric attributes, whose relaxations cannot be quantified as value differences, as assumed in Section 2.1. The main reason is that the concept of "relaxation" of a condition depends on the semantics of the attribute. For instance, if a user is looking for cars with a `Gold` color, we could relax this condition to find cars with a `Beige` color. Our developed algorithms assume we can check the dominance between two records with respect to a condition (selection condition). Consider a "blackbox" function *RELAX(r,C)* that can do the following. Given a record (or a record pair, depending on the condition) r and a condition C, this function can return a value to quantify the minimum amount of relaxation for the condition so that r can satisfy the condition. Our algorithms can be modified to deal with cases of relaxing such conditions, as long as such functions are available for the conditions. If we want to use the pruning step in the MIDIR algorithm, we need to have a function that can compute a lower bound and an upper bound between two pairs of "MBRs" for the attributes in the conditions. The semantics of the MBRs should depend on the attributes. To use the SAJ algorithm, we need to have lists of records sorted based on relaxation of records on each condition.

5.4 Lattice Traversal

Having developed algorithms for query relaxation on different combination of conditions, a natural question to ask is how we can pick the "good" set of conditions to relax, if it is not specified by the user. One requirement of such set of conditions is that it should give the user at least one answer. Take the lattice structure in Fig. 1 as an example. We wish to do relaxation as high in the lattice as possible while ensuring that the relaxation can return at least one result. For the non-weighted case, this could be sufficient for the user, while for the weighted relaxation, this tuple could be used as a candidate answer to prune off searches for nodes at a lower level. On the other hand, we also want to avoid starting at a level that still would not return results. The crucial question to ask here is what causes an empty result for the initial query. This can be monitored during processing of the queries and can be categorized into the following.

Case (1): There are tuples in both R and S that satisfy conditions C_R and C_S, respectively, but C_J is too strict and none of these tuples can be joined pairwise. A quick way to get the answer is to perform a relaxation on C_J using the tuples returned from R and S. Note that since there can be multiple conditions in C_J, there can still be different ways of relaxing the join conditions. To return at least one result through the relaxation, the processing here is sufficient. However, if there is a need to find top-k weighted results, then C_S and C_R might still need to be relaxed if C_J is assigned a very high weight. The results returned here can then be used as candidates to prune off searches during relaxation at lower levels of the nodes.

Case (2): Either C_S or C_R is too strict. Only one of the relations has tuples that satisfy the selection conditions. Since one relation does not return any result, the obvious strategy is to first visit the lattice node that just relaxes

the conditions for the problematic relation. This does not guarantee to return results as C_J can still be too strict. Once this is found to be true, then a lattice node which relaxes C_J together with either C_S and C_R will be visited to guarantee that there is at least one answer. For the weighted top-k case, more relaxation with candidate pruning can follow.

Case (3): Both C_S and C_R are too strict. Tuples from both relations do not satisfy their corresponding condition. Since C_R and C_S are both too strict, visiting just the lattice nodes S and R alone will not be useful at all. The first node to visit in this case will be RS. Should the node fail to return result, node J must be visited as it means that the C_J conditions are too strict. If visiting node J still does not return a nonempty result, node RSJ must be visited. Like before, once a result is obtained in any of the relaxations, it can be used as a candidate to prune off the search for the weighted top-k case.

6. EXPERIMENTS

In this section we present our extensive experimental evaluation of the algorithms.

6.1 Experimental Setting

We used two real datasets and three synthetic datasets with different distributions. The first real data set is from the Internet Movie Database (IMDB).[4] We used two constructed tables. The `Movies` table has 120,000 records with the information about movies, including their ID, title, release year, runtime, IMDB rank, and number of votes. The `ActorsInMovies` table has 1.2 million records with information about actors and their movies, including date of birth (day, month, year), movie ID, and position in the cast list of the movie. The `ActorsInMovies` table has a foreign key constraint on the ID attribute of the `Movies` table. The second real data set is the Census-Income dataset from the UCI KDD Archive.[5] It contains 199,523 census records from two cities for the years 1970, 1980, and 1990. We use three numerical attributes: age, wage per hour, and dividends from stocks.

In addition, we adopted the approach in [4] to generate three datasets with different types of correlations, called "Independent," "Correlated," and "Anti-correlated." Because we are dealing with queries with joins, we generated two tables for each type of correlation. We generated the tables such that, the values respect the correlation property after the join of the two tables. We chose the size of the synthetic tables to be similar to those of the real datasets. We also simulated a foreign key constraint from the first table to the second table. The selection attributes have values in the range [0, 1), and join attributes have integer values.

We implemented the algorithms in gnu C++. For algorithms PJ, PJ$^+$, and SAJ, we used a hash index on the second table. For algorithm SAJ, we assumed that indexing structures such as B-tree are available to produce sorted relaxation lists. For the case where we relax all the conditions, we used the R-tree implementation in Spatial Index Library [14]. We also implemented the corresponding algorithms to compute top-k answers on relaxation skyline. All the experiments were run on a Linux system with an AMD Opteron 240 processor and with 1GB RAM.

[4]http://www.imdb.com/interfaces#plain

[5]http://www.ics.uci.edu/∼kdd/

6.2 Relaxing Selection Conditions

We implemented the algorithms for the case where we relax only selection conditions, namely JoinFirst (JF), PruningJoin (PJ), PruningJoin$^+$ (PJ$^+$), and SortedAccessJoin (SAJ). We used the IMDB data set and the three synthetic datasets. For IMDB, we considered queries that ask for movies and actors that played in these movies, that is, the join condition is (`Movies.ID = ActorsInMovies.movieID`). For the `Movies` table, the selection conditions were defined on the attributes of release year, runtime, IMDB rank, and number of votes. For the `ActorsInMovies` table, the selection conditions were on the attributes of date of birth (day, month, year) and the position in the cast list of a movie. For the synthetic datasets, we considered queries asking for values that match on the join attribute and have a 0 value on all the selection attributes. For the real dataset, we used from one to four different values for each selection condition. We used 16 to 18 queries in each run of the experiments. We ran each query 10 times and report their average performance values. We considered different factors that can affect the performance of the algorithms: size of the dataset, cardinality of the join, and number of selection conditions.

6.2.1 Dataset Size

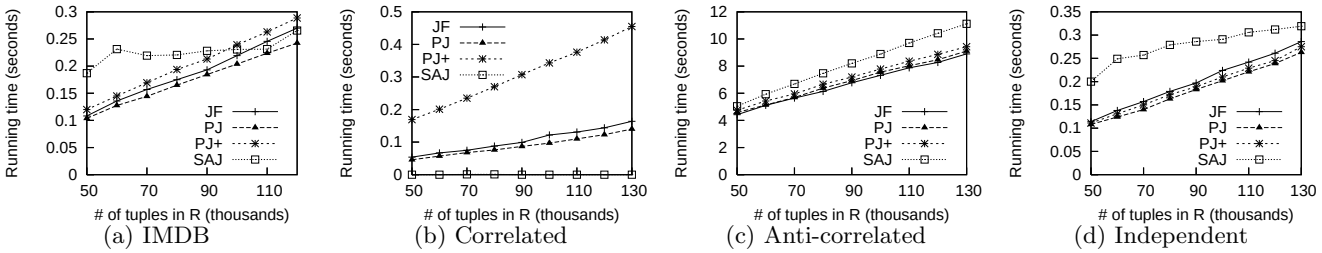
In this set of experiments, we show how the algorithms are affected by the size of the data set. For each data set, we kept the size of the second relation ("S") to be about 1.2 million records. We varied the size of the first (smaller) table ("R") from 50,000 to 120,000 records. Fig. 5 shows the results of the four algorithms on the four datasets. On each query we used four selection conditions (two on each dataset) and the cardinality of the join was 10, i.e., each record in R joins with 10 S records on average.

For the IMDB dataset, the running time of the four algorithms increased as the dataset size increased. The dataset size did not affect the performance of algorithm SAJ very much. This algorithm SAJ has the best performance for the correlated dataset, with a running time less than $1ms$. The reason is that its stopping condition was satisfied very early when traversing the sorted lists of the selection conditions. On the other hand, algorithm PJ$^+$ has the longest running time, mainly because computing the local skyline on the correlated dataset result in additional overhead without pruning many tuples. For the anti-correlated and independent datasets, the SAJ algorithm took more time to run than in the case of the correlated dataset, because the stopping condition is satisfied much later.

6.2.2 Join Cardinality

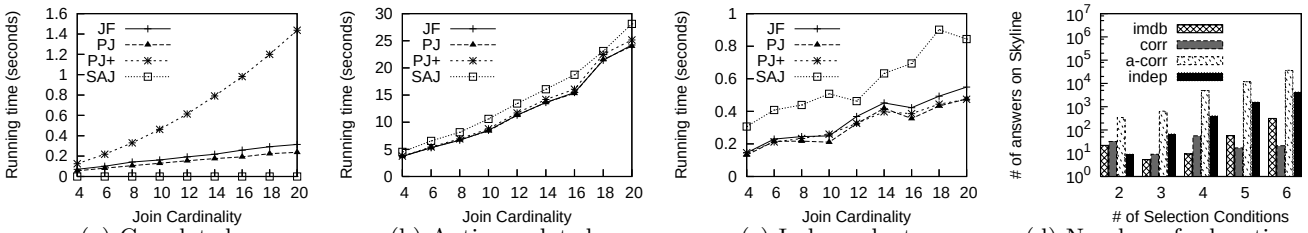
In this experiment we evaluated the running time of our algorithms while changing the cardinality of the join condition in each query. We modified the second relation S in each synthetic dataset, by changing the number of tuples that join with a tuple from the first relation R. If a join cardinality is n, then the number of S records that join with an R record is uniformly distributed between 0 and $2n$. By changing this average join cardinality on the second relation, the size of the second relation changed too. We varied the join cardinality between 4 and 20. The size of the first relation was 130,000 records. The size of the second relation varied between 520,000 and 2.6 million. On each query we used two selection conditions on each relation.

Fig. 6 shows the running time for different join cardinali-



(a) IMDB (b) Correlated (c) Anti-correlated (d) Independent

Figure 5: Running time for different dataset sizes



(a) Correlated (b) Anti-correlated (c) Independent (d) Number of relaxation answers.

Figure 6: Running time for different join cardinalities

ties on the three synthetic datasets. (The join cardinality of the IMDB dataset could not be changed.) For the correlated dataset, the algorithm SAJ performed very efficiently, and its running time was less than $1ms$. The reason is that the algorithm stopped very early, and the join cardinality did not significantly affect the performance of the algorithm. On the other hand, the PJ$^+$ algorithm is very much affected by the join cardinality. The reason is that the size of the record set on which it has to compute a local skyline increases as the join cardinality increases. In this case, the local pruning introduced additional computational overhead. For the anti-correlated dataset and the independent dataset, the algorithms took longer time than the case of the correlated dataset, and they were all affected by the join cardinality.

6.2.3 Number of Selection Conditions

This set of experiments evaluate the performance of the algorithms on different numbers of selection conditions. We changed the total number of selection conditions on the two relations. The number of selection conditions on the second relation is either the same or one more than the number of selection conditions of the first relation. Fig. 7 shows the running time of the algorithms, for the IMDB dataset and the three synthetic datasets. We varied the total number of selection conditions between 2 and 6. The sizes of the two relations were 45,000 and 1.1 million records, respectively. The average join cardinality of the join was 10.

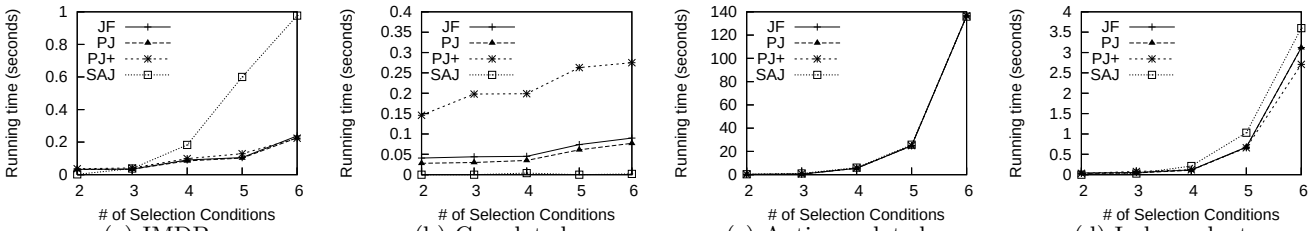
For the IMDB dataset, SAJ was very much affected by the number of selection conditions. The reason is that, as this number increases, this algorithm has to manage more sorted lists, and a stopping condition becomes harder to meet. The other three algorithms were also affected by this number, but they running time grew more slowly.

On the correlated dataset case, SAJ became the fastest algorithm, and its running time was always within $4ms$. PJ$^+$ is the most expensive algorithm, again due to its local-pruning overhead. For the anti-correlated and independent datasets, the algorithms took much more time when we increased the number of selection conditions. There is an important observation for the independent dataset: the PJ$^+$ algorithm became the fastest of all the algorithms. It is because by increasing the number of selection conditions, the local pruning in the algorithm becomes more beneficial.

Fig. 6(d) shows the number of answers on the relaxation skyline, for different number of selection conditions. (This number of answers is independent from the algorithms.) It shows that the anti-correlated data set has the largest number of skyline answers. As the number of selection conditions increased, the number of skyline answers also increased. The independent dataset becomes the dataset with the second largest number of skyline answers.

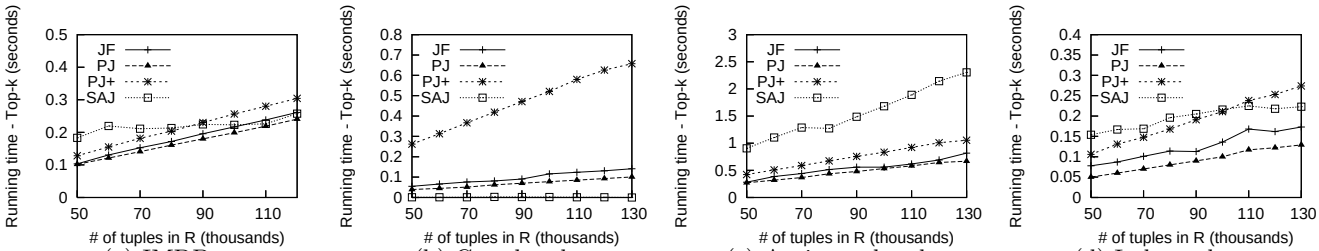
6.2.4 Top-k Relaxation Answers

We adapted the algorithms to compute top-k answers on a relaxation skyline, as described in Section 5.1. For each run we used an equi-join query with four selection conditions (two selection conditions on each relation). The queries asked for top-10 points on the skyline, based on the following weights: 0.2 and 0.4 for the selection conditions on R, and 0.3 and 0.1 for the selection conditions on S. We evaluated the performance of the algorithms on the IMDB and the three synthetic datasets, with different dataset sizes. Similar to the setting of computing a relaxation skyline, we varied the size of the R relation from 50,000 records to 120,000 records. The size of dataset S was 1.2 million records. The average join cardinality was 10. Fig. 8 shows the running time of these algorithms on the datasets with different sizes. For the IMDB dataset, PJ was the best. At the beginning, SAJ performed the worst. As the size of relation R increased, the performance of PJ$^+$ became the worst.



(a) IMDB (b) Correlated (c) Anti-correlated (d) Independent

Figure 7: Running time for different numbers of selection conditions



(a) IMDB (b) Correlated (c) Anti-correlated (d) Independent

Figure 8: Running time for different dataset sizes (Top-k)

6.3 Relaxing All Conditions

In this set of experiments, we evaluated our algorithms for the case where we want to relax all the conditions in a query, especially including the join condition. We used the Census-Income dataset in these experiments. We implemented the MIDIR algorithm described in Section 4, as well as its corresponding algorithm to compute top-k answers on the relaxation skyline.

We evaluated the algorithms with 3 queries (Q1, Q2, and Q3). Each query had two selection conditions (on attributes of wage per hour and dividends from stocks) and one join condition (self join on the age attribute). Q1 used two frequent values for both selection condition selections. Q2 we used the same frequent value for the first condition, but an infrequent value for the second condition. Q3 used two infrequent values for both conditions. For the top-k case, we asked for top-10 best points on the skyline based on the following weights, 0.15 for the first selection attribute, 0.15 on the second selection attribute, and 0.15 on the join attribute. We experimented with three relaxation cases: relaxing only the join condition (called "case RELAX(J)"), relaxing the selection condition on R and the join condition (called "case RELAX(RJ)"), and relaxing the selection condition on S and the join condition (called "case RELAX(SJ)"). We varied the size of the dataset between 4,000 - 20,000 records.

Fig. 9 shows the running time for computing a relaxation skyline and running time for computing top-10 answers on the skyline, for various cases and queries. Take Fig. 9(a) as an example, where we want to compute the relaxation skylines for the three queries. As we increased the size of the relation (the queries used self-joins), the algorithm took more time for all three queries. Q3 was the most expensive query, while Q2 was the most efficient one. Fig. 9(b) shows the results of different relaxation cases for the same query Q2. It shows that RELAX(RJ) is the most expensive case, while RELAX(J) is fastest relaxation case.

Summary: We have the following observations from the experiments.

JF and PJ: For the case where we relax only selection conditions, these two algorithms have very similar running times. The result is not surprising, since these two algorithms have almost the same set of operations on the skyline, possibly in different orders. Their main difference is that JF has to keep in memory the entire set of tuple pairs after the join step, while PJ can do the pruning on the fly, thus it can reduce the size of the intermediate result. Notice that JF has the advantage of not modifying implementations of existing join operators.

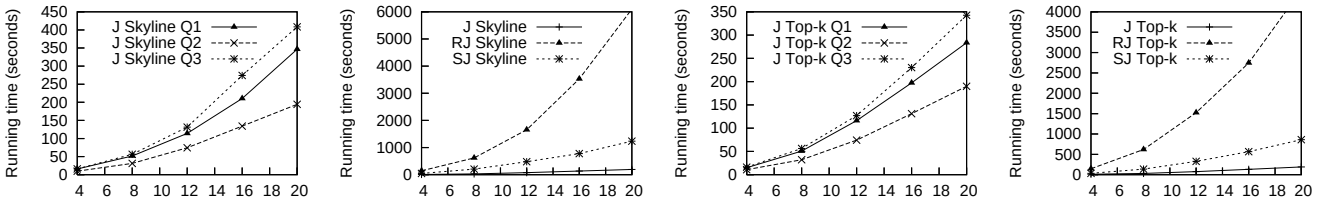
PJ and PJ$^+$: In most cases the PJ algorithm performs faster. This is because the computation of the local skyline does not prune many records, and it introduces additional overhead. The performance of the PJ algorithm is very much affected by the join cardinality; the performance will decrease as the cardinality increases. As the number of conditions increases, the computation of local skyline improves the running time of the algorithm. This is because the size of the global skyline increases and so, checking for dominance becomes expensive in the global skyline.

SAJ: This algorithm performs efficiently when the data and query conditions are correlated.

Compared to relaxing selection conditions, it is computationally more expensive to relax join conditions. The main reason is that we do not know how much we want to relax the join conditions, thus we have to keep a large number of candidate pairs.

7. CONCLUSIONS

In this paper we presented a framework for relaxing relational queries in order to compute nonempty answers. We formally define the semantics of relaxing query conditions in this framework. We developed algorithms for the case where we can only relax selection conditions, and algorithms for



(a) Skyline (case RELAX(J)) (b) Skyline (Query Q2) (c) Top-k (case RELAX(J)) (d) Top-k (Query Q2)

Figure 9: Running time for different relaxation cases and dataset sizes (relaxing all conditions)

the general case where we can relax all conditions, including join conditions. We also show that these algorithms are extendable to variants of the problem. We presented results from an experimental validation of our techniques demonstrating their practical utility. Our future work in this area will address extensions of our framework to other types of attributes such as categorical attributes.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[2] W. Balke, U. Güntzer, and X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, 2004.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, 2001.

[5] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

[6] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.

[7] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

[9] P. Eng, B. Ooi, H. Sim, and K. Tan. Preference-driven query processing. In *ICDE*, Bangalore, India, March 2003.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems*, 2001.

[11] T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):48–59, 1997.

[12] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[14] M. Hadjieleftheriou. Spatial index library. http://u-foria.org/marioh/spatialindex/.

[15] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[16] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Supporting exploratory queries in databases. In *DASFAA*, pages 594–605, 2004.

[17] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[18] W. Kießling. Preference queries with sv-semantics. In *COMAD*, pages 15–26, 2005.

[19] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *VLDB*, 2002.

[20] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4), 1975.

[21] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[22] D. H. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4), November 1974.

[23] I. Muslea. Machine learning for online query relaxation. In *KDD*, pages 246–255, 2004.

[24] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

[25] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.

[26] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[27] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[28] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[29] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[30] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD Conference*, pages 343–354, 2000.

[31] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[32] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.