Database Support for Weighted Match Joins

Ameet Kini, Jeffrey F. Naughton
Computer Sciences Department
1210 West Dayton Street, Madison, WI 53705
{akini, naughton} @ cs.wisc.edu

Abstract

As relational database management systems are applied to non-traditional domains such as scientific data management, there is an increasing need to support queries with semantics that differ from those appropriate for traditional RDBMS applications. Two interesting ideas currently being explored in the DBMS community are ranking query results (e.g., top-k computations) and, more recently, "match joins." In this paper we combine these two ideas and study weighted match joins, in which (a) like match joins, each tuple joins with at most one matching tuple, and (b) like top-k joins, the system attempts to provide a set of answer tuples that maximizes a weight function. We explore exact and approximate strategies for evaluating weighted match joins. Using a prototype implementation in PostgreSQL, we explore the performance characteristics of these strategies. Our results suggest that the DBMS optimization-based approach of providing several implementations of an operator and then choosing an appropriate one at run time can be useful in computing weighted match joins.

1. Introduction

In this paper, we focus on supporting "weighted matching" queries in an RDBMS. In mathematical terms, a weighted matching problem can be expressed as follows: given a bipartite graph G with weighted edge set E, find a maximally weighted subset of E, denoted M, such that for each $e = (u,v) \in M$, neither u nor v appears in any other edge in M.

Our work is motivated by two recent trends. The first trend is that instances of matching problems and weighted matching in particular have sprung up in diverse domains. These domains can be classified into two groups. In the first group, matching is used to allocate resources to consumers subject to certain quality metrics. These domains (with corresponding resources and consumers in parentheses) include grid job scheduling [10,12] (matching user jobs to available machines), online advertising [8] (matching search keywords to advertiser bids placed on the keywords), and gaming [11] (matching players based on skill level, etc.). Problems in the second group involve matching two sets of entities in an effort to find pairs of entities that are most similar. These domains include data cleaning [4,9] and protein matching in biological databases [14]. The second trend is that applications that differ substantially from traditional data processing have started to harness the power of RDBMSs. As more and more of these applications start using RDBMSs to store their data, a natural question to ask is whether we can leverage the query machinery in the RDBMS to support matching operations (instead of merely treating the RDBMS as an expensive file system by exporting/importing all the data to/from an external C/C++ function that implements the matching operation.) The premise of this paper is that by exposing the semantics of the matching operation to the RDBMS, one can efficiently support these matching operations by exploiting existing RDBMS query machinery.

One's initial reaction might be that the weighted matching problem has been well-studied by the algorithms community, and that the DBMS community is likely to have little to contribute. We think this is false for two reasons. First, there is the interesting question of how to exploit existing RDBMS capabilities to compute matchings. Second, weighted matching problems as they arise in RDBMSs differ in significant ways from the classical graph-theoretic matching problems.

For the second point, in the context of an RDBMS, matching occurs between two entity sets, one stored in a table, say R, and the other in another table, say S, that need to have their elements paired in a one-to-one fashion. Compared to classical graph theory, an interesting difference immediately arises: rather than storing the graph's complete edge set E, DBMS-resident applications will typically store only the nodes of the graph, representing the edge set E *implicitly* as a match join predicate θ and weight function F. That is, for any two tuples $r \in R$ and $s \in S$, $\theta(r,s)$ is true if and only if there is an edge from r to s in the graph; the weight on each such edge can be computed using the weight function $F(r,s)$.

Given such a problem formulation, one way to compute the weighted matching would be to materialize the implicit edge set by computing the standard relational join of R and S, with θ as the join predicate, then to feed the result to a classical matching algorithm. Unfortunately, this scheme is unlikely to be successful – often such a join will be very large (for example, when R and S are large and/or each row in R joins with many rows in S). Moreover, this approach ignores an important opportunity present due to the implicit graph aspect of the problem. In particular, it is possible to design faster matching algorithms for a specific problem instance by exploiting special properties of the match predicates, weight functions, and data characteristics of the problem at hand. This in turn fits in well with the classical relational query optimization paradigm: we implement several matching algorithms, and choose between them based upon the query and data instance at run time.

This work builds on our earlier work [7], in which we studied (non-weighted) match joins, where the quality of the join is the number of matching tuples rather than a sum of a weight function applied to the matches. While we showed that RDBMS technology can be brought to bear on such problems, there are many problems that do not fit into the limitations of a non-weighted matching problem.

For example, consider a simplified version of job scheduling in a grid. One can view this as a matching problem (matching jobs to processors). In this case the non-weighted matching problem corresponds to finding an assignment of jobs to machines such that the number of (job, machine) pairs is maximized. However, one can imagine that the quality of the matching may not be just a function of how many jobs get matched to machines but also a function of how well the jobs match the machines to which they are assigned. (For example, some users want a lot of memory, others want a fast CPU, some users have higher priority than others, and so forth.) This notion of "how well" translates to weights on the matches. Our goal in this paper is to begin to explore how well RDBMS technology can be applied to the more complex weighted matching problem.

2. Background

We formally describe the weighted match join problem, which takes as input two relations R and S, a predicate θ, and weight function F. Here, the rows of R and S represent the nodes of the graph, the predicate θ is used to implicitly denote edges in the graph, and each of these edges has a weight assigned by function F. The complete edge set can be materialized using the standard relational join $R \bowtie_\theta S$. We use $n = |R| + |S|$ and $m = |R \bowtie_\theta S|$ to refer to the number of nodes and edges, respectively, in the bipartite graph.

Definition 1 (Weighted Match Join) *Let $M \subseteq R \bowtie_\theta S$. Then M is a weighted matching or a weighted match join of R and S with predicate θ and non-negative weight function $F(t)$, $t \in R \bowtie_\theta S$, iff each tuple of R and S appears in at most one tuple (r,s) in M. We use $M(R)$ and $M(S)$ to refer to the R and S tuples in M, and $w(M)$ to refer to the sum of weights of all tuples in M.*

Definition 2 (Maximum Weighted Matching) *Let M^* be the set of all weighted matchings of relations R and S with predicate θ and weight function F. Then MM is a maximum weighted matching iff $MM \in M^*$ and $\forall M' \in M^*$, $w(MM) \geq w(M')$.*

We frequently refer to a matching being maximal, which means that the size of the matching cannot be increased by simply adding edges.

Definition 3 (Maximal Weighted Matching) *A matching M' is a maximal weighted matching of relations R and S with predicate θ and weight function F if $\forall r \in R-M'(R)$, $s \in S-M'(S)$, $(r,s) \notin R \bowtie_\theta S$.*

Note that "maximal" here refers to the number of edges in the matching, not the weight of the matching. If a weighted matching M is not maximum, then it is an approximate weighted matching. We quantify its degree of approximation by an *optimality ratio*: $w(M)$ / $w(MM)$ ($w(MM) > 0$). For example, if $w(M) = 50$ whereas $w(MM) = 100$, then the optimality ratio = ½, in other words, M is ½-optimal.

By definition, the weighted match join is a weighted one-to-one subset of the relational θ-join of R and S. Recent work in the database community has addressed the closely related problem of computing the top-k join (also known as rank join) [6,13], which seeks an ordered set of the k highest weighted tuples in the relational join. A critical difference between the top-k join and the weighted match join is that the result of a top-k join need not satisfy the one-to-one matching constraint.

3. Weighted Match join Algorithms

3.1. Overview

As mentioned in the introduction, by exploiting the semantics of the match predicates and weight functions, we can explore ways to compute weighted matchings on these special graphs potentially more efficiently than general weighted matching algorithms. Our algorithms each exploit some combination of the following properties of the underlying graph: a) properties of match join predicates, b) properties of weight functions, and c) uniformity in the input relations.

Before we dive into a description of our weighted match join algorithms, we describe a simple approximate weighted matching algorithm [2] that can always be used to compute weighted match joins. This algorithm, referred to as GREEDY, computes and sorts the edges of the underlying graph in descending order of weight and iterates through this sorted list marking edges as "matched" while maintaining the one-to-one invariant. GREEDY can be easily implemented in procedural SQL: first, a relational θ-join can be used to compute the edges of the underlying bipartite graph, which can be then sorted in descending order of the weight function F using a SQL *order-by* clause. Finally, an iterator can be used to loop over the sorted results and construct a one-to-one matching. It can be shown that GREEDY returns a maximal matching that is at least ½-optimal [2]. It also has the benefit of working for any match predicate θ and weight function F. Unfortunately this generality comes at the price of performance as it requires building and sorting the edges of the underlying bipartite graph, an $O(m \log m)$ operation (note that $m \sim n^2$ in dense graphs). GREEDY is a "catch-all" algorithm that can be always applied; however, as the rest of our algorithms show, better performance can be gained by exploiting structural properties of the underlying graph.

3.2. A Nested Loops Based Technique

Our next algorithm W_MJNL (Weighted Match Join Nested Loops) exploits a property of weight functions, formalized here.

Property 1 (Decomposable Monotonic Weight Function) *A function $F(t)$, $t \in R \bowtie_\theta S$ is said to be decomposable monotone if F can be expressed as $F(f_1(\{\text{attributes of } R\}), f_2(\{\text{attributes of } S\})$ where F is monotone on its components f_1 and f_2. In other words, for any two tuples $t_1, t_2 \in R \bowtie_\theta S$ if $f_1(t_1) < f_1(t_2)$ and $f_2(t_1) < f_2(t_2)$ then $F(t_1) < F(t_2)$.*

An example of decomposable monotonic weight functions is a linear weighted sum of attributes of R and S, frequently referred to as a preference function in existing literature on top-k algorithms [5]. Consider such a preference function $F = (0.3 * R.a_1 + 0.4 * R.a_2 + 0.1 * S.a_1 + 0.2 * S.a_2)$. Then F can be expressed as $f_1 + f_2$ where $f_1 = (0.3 * R.a_1 + 0.4 * R.a_2)$ and $f_2 = (0.1 * S.a_1 + 0.2 * S.a_2)$. Another decomposable monotonic function is $F = max(R.a, S.b)$. Here, F can be expressed as $max(f_1, f_2)$ where $f_1 = R.a$ and $f_2 = S.b$. On the other hand, a function that is not decomposable monotone is the Euclidean distance function.

W_MJNL, illustrated in Figure 1, first sorts both R and S in descending order of f_1 and f_2 respectively. Matching tuples are then found by executing a nested loops algorithm over these sorted relations, marking each matched tuple so that it is not matched again. Note that unlike GREEDY, which computes and sorts the full relational join of R and S, W_MJNL never materializes the full edge relation and sorts R and S individually.

While the resulting weighted matching is not guaranteed to be optimal, the following theorem gives guarantees as to its quality. Due to space constraints, all proofs are omitted here.

Theorem 1. *Consider a weighted match join of R and S with predicate θ and a decomposable monotonic weight function F. Then W_MJNL on this weighted match join problem takes time $O(n^2)$ and produces a maximal matching that is at least ½-optimal.*

If W_MJNL is applied to a problem instance in which F is not decomposable monotonic, it will still produce a maximal matching, although it is not guaranteed to be ½ -optimal.

```
Procedure W_MJNL
Input  relations $R$ and $S$, match predicate $\theta(R,S)$, and
       decomposable monotonic weight function $F(f_1(R), f_2(S))$
Output  weighted match join $M \subseteq R \bowtie_\theta S$
Begin
1.  Sort $R$ in descending order of $f_1$
2.  for each tuple $r \in R$
3.      Let cand_s be the list of all tuples $s \in S$ such
        that $\theta(r,s)$= true and $s$ is not already matched
4.      Sort cand_s in descending order of $f_2$
5.      Retrieve the highest tuple $s'$ in this list
6.      Mark $s'$ as matched
7.      Add $<r,s'>$ tuple to $M$
    end for
End
```

Figure 1. W_MJNL pseudocode.

3.3. A Sort-Merge Based Technique

While W_MJNL exploits properties of weight functions, our second approach, W_MJSM (Weighted Match Join Sort Merge) exploits properties of match predicates. To see the insight behind it, consider the case when the match predicate is a conjunction of equality predicates. Also, for now, lets focus only on decomposable monotonic functions; we relax this assumption later in the section. Given this setup, we can sort the two relations on their join attributes and then execute a "merge" on the two relations, looking for matches, marking tuples as "matched" as we go. Since the join predicate consists of equalities, matches can be found *only* within the "bucket" consisting of *R* and *S* tuples that agree on all the join attributes. Moreover, each bucket is independent of the other – in other words, an *R* tuple from one bucket can never be matched with an *S* tuple from another bucket since they will not satisfy the join predicate. Now, within each bucket, all tuples of *R* and *S* satisfy the join predicate and are therefore eligible for matching. As such, we can simply sort the *R* and *S* parts of the bucket in descending order of f_1 and f_2 respectively, iterate down the two lists and match each pair of tuples as we go. Figure 2 illustrates W_MJSM on the equality predicate ($R.a_1 = S.a_1$).

Theorem 2 *Consider a weighted match join of R and S with decomposable monotonic weight function* $F(f_1(R), f_2(S))$ *and predicate* θ *consisting of a conjunction of k equality predicates (k > 0) of the form* $R.a_1 = S.a_1$ *AND* $R.a_2 = S.a_2$ *AND, …, AND* $R.a_k = S.a_k$. *Then W_MJSM on this weighted match join problem takes time O(n log n) and produces a maximum weighted matching.*

We now discuss extensions to the basic form of W_MJSM to handle various forms of match predicates and weight functions. We first show how W_MJSM can be extended to functions that are not decomposable monotone. To see how this works, recall that within a bucket, all pairs of *R* and *S* tuples satisfy the join predicate and are therefore eligible for matching. In graph theoretic terms, the tuples within a bucket form a *complete* bipartite sub-graph since every *r* tuple joins with every *s* tuple within the bucket. Here, we can use the idea from GREEDY – that is, compute the edges of this complete sub-graph by joining all tuples within the bucket, sort them in descending order by weight, and then iterate through this sorted list greedily picking edges as we go. Just as GREEDY, this extended version of W_MJSM can be applied to any weight function. However, unlike GREEDY, it is optimal, i.e., it returns the maximum weighted matching. In general, it can be shown that applying GREEDY on any complete bipartite graph results in the maximum weighted matching, a result that is validated in our experiments. However, in the interest of space, we do not pursue the details of this extension here.

Before proceeding to the second extension, recall that the main idea behind W_MJSM is that it uses the fact that the join predicate consists of equalities to par-

```
Procedure W_MJSM
Input   relations R and S, match predicate
        $\theta(R.a_1 = S.a_1)$, and decomposable monotonic
        function $F(f_1(R), f_2(S))$
Output  weighted match join $M \subseteq R \bowtie_\theta S$
Begin
1.   Sort R on ($a_1$ ascending, $f_1$ descending)
2.   Sort S on ($a_1$ ascending, $f_2$ descending)

     Do sort-merge style iteration
3.   open iterators on R and S
4.   while neither R nor S iterators are exhausted
5.       while $r.a_1 < s.a_1$ and R iterator is not exhausted
6.           r = get next tuple from R iterator;
         end while
7.       while $r.a_1 > s.a_1$ and S iterator is not exhausted
8.           s = get next tuple from S iterator;
         end while
9.       while $r.a_1 = s.a_1$
10.          add <r,s> tuple to M
11.          r = get next tuple from R iterator;
12.          s = get next tuple from S iterator;
         end while
     end while
End
```

Figure 2. W_MJSM pseudocode.

-tition the relations into disjoint buckets. If the join predicate consists of equalities and other predicates (e.g., $R.a_1 = S.a_1$ and $R.a_2 < S.a_2$), one can still apply this basic idea with a minor extension. In particular, now within each bucket, there may be tuples that do not match each other – even if they agree on the equality attributes ($R.a_1$, $S.a_1$), they may not satisfy the additional predicates ($R.a_2 < S.a_2$) and hence fail to satisfy the overall join predicate. In general, matching tuples can be scattered anywhere within the bucket. Assuming for now that the weight function is decomposable monotone (we will relax this assumption later), the main observation is that within a bucket, we can effectively apply a technique like W_MJNL, that is, first sort *R* and *S* within a bucket in descending order of f_1 and f_2 respectively, then iterate through both sorted lists in nested loops fashion finding *r* and *s* tuples that satisfy the join predicate and marking them as "matched" so that they are not matched again.

Note that this extension combines ideas from both W_MJSM and W_MJNL. By sorting on the equality attributes, it partitions the relation into disjoint buckets, like the basic W_MJSM discussed earlier in the section; however, unlike W_MJSM, it uses a nested loops style evaluation to find matching tuples within each bucket.

Theorem 3 *Consider a weighted match join of R and S with decomposable monotonic weight function $F(f_1(R), f_2(S))$ and predicate θ consisting of a conjunction of k equality predicates (k > 0) and other arbitrary predicates of the form $(R.a_1 = S.a_1$ AND $R.a_2 = S.a_2$ AND, ..., AND $R.a_k = S.a_k.)$ AND (other predicates). Then W_MJSM on this weighted match join problem takes time $O(n^2)$ and produces a maximal matching that is at least ½-optimal.*

In the discussion following Theorem 2, we described how W_MJSM can be extended to work on arbitrary weight functions when the join predicate consists of only equalities, using the GREEDY technique to find matching tuples within each bucket. This extension can be applied unmodified when the join predicate consists of equalities and other predicates. The worst case running time and quality of resulting matching for this extension is the same as that of GREEDY, i.e., $O(m \log m)$ and a maximal matching that is at least ½-optimal, respectively.

3.4. A Network Optimization Based Technique

Our previous algorithms exploit properties of match join predicates and weight functions. In this section, we present an approach that exploits data uniformity to efficiently compute weighted match joins on arbitrary join predicates and weight functions.

The motivation and basic ideas for this algorithm, henceforth referred to as W_MJMC (Weighted Match Join Min-Cost), come from the observation that in many problem instances it is possible to partition the input relations into groups such that tuples within a group are identical with respect to the match (that is, either all tuples of a given group will join with a given tuple from the other table, or none will). For example, when scheduling jobs on grids, most clusters have relatively few unique machine configurations and similarly many users submit large number of jobs with almost identical resource requirements. In such situations, it is possible to aggregate the input relations to the match join, as was done for MJMF [7], using standard relational grouping techniques.

The basic idea of our approach is to perform a relational group-by operation on the attributes in the join predicate and the weight function. Using one representative of each group along with its weight and the total size of the group, we transform the weighted matching problem into a *minimum cost circulation* problem, or simply *min-cost circulation* [1].

Definition 4 (Minimum Cost Circulation) *Consider a directed network G = (N, E) with a cost c_{ij} and nonnegative flow capacity u_{ij} associated with each edge $(i,j) \in E$. There are two special nodes in the network G: a start node u and an end node v. The minimum cost circulation problem can be stated formally as:*

Minimize $z(x) = \sum_{(i,j)\in E} c_{ij}x_{ij}$

subject to:

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = 0 \text{ for all } i \in N,$$

$$0 \le x_{ij} \le u_{ij} \text{ for all } (i,j) \in E.$$

Here, we refer to the vector $x = \{x_{ij}\}$ satisfying the constraints as a flow.

Figures 3 and 4 illustrate the construction of network $G = (N_1 \cup N_2, E)$ with start node u and end node v (line 1) that is input to the min-cost circulation algorithm.

The following main steps describe how to build network G (parenthesized line numbers refer to Figure 3):

1. (Build node set) add a node $n_1 \in N_1$ for every group of tuples in R which have the same value on the attributes used in the match join predicate and weight function (line 5); similarly for N_2 (line 10).

2. (Build edge set) add an edge between n_1 and n_2 if the groups of tuples that they represent satisfy the match predicate (line 14).

3. (Connect nodes to start and end) add an edge between u and n_1, and between n_2 and v (lines 6 and 11).

4. (Assign edge capacities) For edges of the form (u, n_1) the capacity is set to the size of the group represented by n_1 (line 7). Similarly, the capacity on (n_2,v) is set to the size of the group represented by n_2 (line 12). Finally, the capacity on edges of the form (n_1, n_2) is set to the minimum of the two group sizes (line 15).

5. (Assign edge weights) Here, recall that what we eventually want is a *maximum* weighted matching when in fact this is a *min*-cost problem. This duality is achieved by setting weights on edges of the form (n_1, n_2) to the additive inverse (negative) of the weight of the underlying join tuple (line 16). For edges of the form (u, n_1) and (n_2,v), the weights are to set -1 (lines 8 and 13).

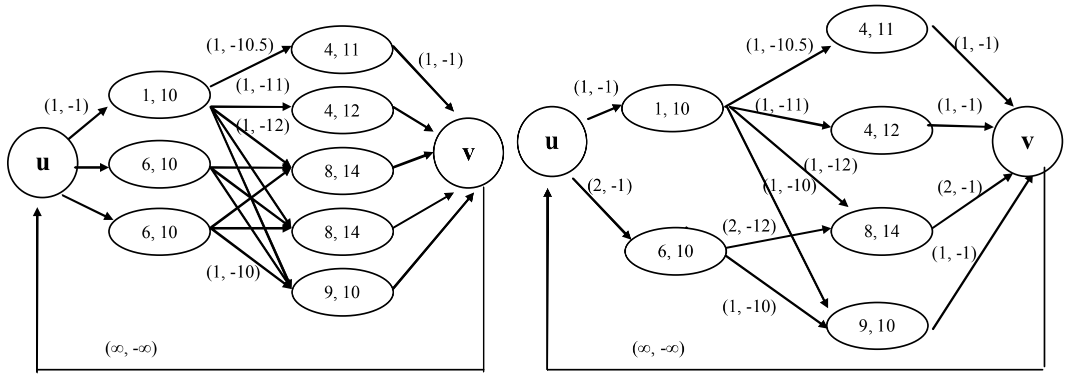
6. Finally, we add a "reverse" edge (v,u) with infinitely high capacity and infinitely low weight (lines 17-19).

```
Procedure W_MJMC
Input     relations R and S,
          match predicate $\theta$(R.a, R.b, S.a, S.b), and weight
          function F(R.score,S.score)
Output    weighted match join $M \subseteq R \bowtie_\theta S$
Begin
     Build the reduced directed network G
1.   First, add start node u and end node v to G
     The rest of the nodes and edges can be obtained using
     following SQL
2.   SELECT * FROM
          (SELECT count(*) as r_count,
                  a as r_a, b as r_b, score as r_score
          FROM R
          GROUP BY a, b, score) as R_grouped,
          (SELECT count(*) as s_count,
                  a as s_a, b as s_b, score as s_score
          FROM S
          GROUP BY a, b, score) as S_grouped
     WHERE $\theta$(R.a, R.b, S.a, S.b) is true
     Iterate over result tuples t of the form
     <r_count, r_a, r_b, r_score, s_count, s_a, s_b,
     s_score>
3.   for each tuple t {
4.        if we are seeing this r node for the first time
5.             add this r node to G
6.             add an edge connecting r to u
7.             set the edge flow capacity to t.count_r
8.             set the edge weight to -1
          end if
9.        if we are seeing this s node for the first time
10.            add this s node to G
11.            add an edge connecting s to v
12.            set the edge flow capacity to t.count_s
13.            set the edge weight to -1
          end if
14.       add an edge connecting r and s
15.       set the flow capacity to min(t.count_r, t.count_s)
16.       set the edge weight to (-1 * F(R.score,S.score))
     end for
17.  add the reverse edge connecting v to u
18.  set the edge flow capacity to infinity
19.  set the edge weight to negative infinity
20.  invoke the GOBLIN min-cost flow algorithm on G
          .
          .
     reconstruct matching from obtained flow
          .
          .
End
```

Figure 3. W_MJMC pseudocode. For illustrative purposes, the weighted match join query on relations *R* and *S* has a match predicate *θ* on attributes *(a, b)* and weight function *F* on attribute *score* from both relations.

R	
a_1	Score
1	10
6	10
6	10

S	
a_1	Score
4	11
4	12
8	14
8	14
9	10



(a) (b) (c)

Figure 4. A 3-step transformation from (a) Base tables to (b) A directed weighted network to (c) A reduced network that is input to the min-cost algorithm. $\theta = R.a_1 < S.a_1$, and $F = 0.5 * R.score + 0.5 * S.score$.

After converting the matching problem to a min-cost problem, we can use any standard min-cost solver (e.g., GOBLIN [3]) on the transformed problem (line 20), and then translate the solution to that problem back to a solution to the original weighted matching problem. While omitted from Figure 3, this phase is exactly the same as that in the MJMF algorithm [7] – by solving the min-cost problem subject to the capacity constraints, we obtain a flow value on every edge in the network. Consider an edge of the form (n_1, n_2). Let the flow value on this edge be f. We can then match f members of n_1 to f members of n_2. Due to the capacity constraint on edge (n_1, n_2), we know that $f \leq$ the minimum of the sizes of the two groups represented by n_1 and n_2. Similarly, we can take the flows on every edge and transform them to a matching in the original graph. It can be shown that this matching is an optimal solution to the original weighted matching problem; due to space constraints, we omit the proof, which is elaborate and follows a similar argument as that given in [7] for MJMF.

W_MJMC yields a maximum weighted matching on arbitrary match join predicates and weight functions. For efficiency, it relies heavily on the premise that the initial grouping before transforming the matching problem to a min-cost problem results in a small number of groups, as the min-cost algorithm is ultimately $O(n^3)$, where n is proportional to the number of groups. Our next and last algorithm combines the benefits of grouping without relying on an $O(n^3)$ algorithm at the cost of obtaining the maximum weighted matching.

3.5. Combining Grouping and Nested Loops

Our previous algorithm W_MJMC was based on two distinct ideas: compressing the two input relations using grouping, and then transforming the matching problem to a min-cost circulation problem. In this section, we explore the idea of compressing by grouping but then using a variant of W_MJNL to work on such compressed relations. Since this approach does not invoke the $O(n^3)$ min-cost algorithm we expect it to be more robust in the face of less compressible inputs. We call this algorithm W_MJGNL (Weighted Match Join Grouped Nested Loops) and just like W_MJNL, it yields an approximate weighted match join over decomposable monotonic weight functions and arbitrary match join predicates.

This hybrid algorithm begins by grouping the two input relations R and S and writing this to temporary relations *Grouped_R* and *Grouped_S* respectively. Like W_MJMC, each tuple in this compressed relation represents a group in the original relation; it consists of the attributes that were grouped over, i.e., the match join and weight function attributes, and a count that represents the size of the group. After materializing the temporary relations, it sorts them in descending order of f_1 and f_2 respectively. Finally, it iterates over them in nested loops fashion finding r and s tuples that satisfy the join predicate. However, when it finds tuples r and s that satisfy the match join predicate, instead of just marking the s tuple as "matched" as was done in W_MJNL, it outputs as many matching r and s tuples as is the smaller of the two counts and decrements the counts accordingly. As in the case of W_MJNL, certain guarantees can be made about the quality of the matching computed by W_MJGNL.

Theorem 5 *Consider a weighted match join of R and S with predicate θ and a decomposable monotonic weight function $F(f_1(R), f_2(S))$. Then W_MJGNL on this weighted match join problem takes time $O(n^2)$ and produces a maximal matching that is at least ½-optimal.*

4. Experiments

In this section, we describe the implementation of our weighted match join algorithms and report experimental results exploring their performance.

4.1. Implementation

Our algorithms are implemented as user defined functions (UDFs) in PostgreSQL version 8.1. Most of the implementation was done in PL/pgSQL, PostgreSQL's procedural interface; only part of W_MJMC was written as a C UDF since it required integration with the GOBLIN C++ network optimization library. At a little under 1000 lines, the code-base is fairly small. This includes not only the algorithms themselves but scripts used to run the experiments. We credit this simplicity to two factors: first, the algorithms were built on top of the database system using already supported primitives such as grouping, sorting, among others. Second, the high level interface offered by PL/pgSQL either encapsulates or simplifies common tasks such as creating cursors and looping over query results, thereby allowing the application programmer to focus on the algorithm itself.

There is, however, an inherent performance tradeoff by choosing a UDF-based implementation versus implementing these algorithms natively inside the database. To get a rough estimate of this overhead, we implemented the regular sort-merge algorithm as a PL/pgSQL UDF and compared its running time to that of the natively implemented sort-merge join algorithm; we observed that the UDF based implementation consistently performed 20-25% slower than the native implementation over a range of equijoin queries. As such, we expect that our algorithms could be sped up by a similar percentage if they were implemented natively.

4.2. Experimental Setup

All experiments were run on an Intel Pentium 4 dual-processor machine with 1 GB physical memory running CentOS 4. The database buffer pool size was set to 256 MB. Each experiment was run 4-5 times, flushing the buffer pool and system memory between runs. The numbers were fairly consistent across runs, and hence we report their average across all the runs.

In order to control various data characteristics such as selectivity and group size, the experiments were conducted on synthetic data; the two tables being matched, *R* and *S*, each had the same schema: (*id*: integer, *a*: integer, *b*: integer, *score*: float), where *a* and *b* were used as match join attributes, and *score* was used as an input to a weight function. Unless explicitly mentioned, *R* and *S* each contained 10000 tuples. All times reported include the time to output the weighted match join result, which was written to a temporary table.

Experiment 1

The first experiment measures the performance of GREEDY, W_MJNL, W_MJSM, W_MJMC, and W_MJGNL on a weighted match join query whose predicate consists of two equalities: (*R.a* = *S.a* AND *R.b* = *S.b*) and whose weight function *F* is the decomposable monotonic function (0.5 * *R.score* + 0.5 * *S.score*).

On the x-axis, we vary a parameter called group size, which controls the number of tuples in each group as determined by the combination of the match join and weight function attributes; in this case, those attributes are *(a, b, score)*. Thus this experiment explores how well the grouping algorithms are able to exploit the compression provided by their grouping.

Recall that W_MJMC works by performing a group-by on the combined match join and weight function attributes, followed by a full join, thus building a graph which is then fed to the min-cost algorithm. The size of the graph |*G*| plays a major role in the overall performance of the O(n^3) min-cost algorithm, and |*G*| is a function of two variables: the average group size *g* and the full join selectivity μ. More precisely, |*G*| ~ μ *((|*R*| * |*S*|) / *g*). For a fixed selectivity then, the larger the group size, the smaller the graph. As such, for the dataset in this experiment, we vary the group size from small (*g* = 5) to large (*g* = 500) and keep the join selectivity constant (μ = 0.1). Accordingly, |*G*| ranges from approximately 400000 to 40.

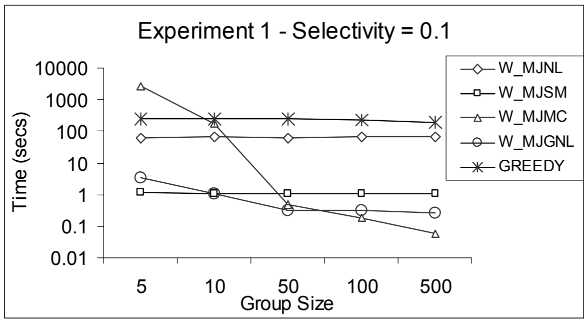


Figure 5. Experiment 1

To see how *g* is varied while keeping μ fixed, recall that grouping is done on the composite (match-join-attributes, weight-function-attributes), in this case, (*a*, *b*, *score*), whereas the join selectivity is solely dependent on the match join attributes (*a*, *b*). As such, by changing the value of only the weight function attribute, while at the same time, keeping the match join attribute value fixed, we can control the two parameters *g* and μ accordingly. Putting some numbers in, if *g* = 5 and μ = 0.1, then values for attributes (*a*, *b*) of both relations *R* and *S* are fixed for batches of 0.1 * 10000 = 1000 tuples, and no two tuples from two different batches are assigned the same value on (*a*, *b*); since the join predicate in this experiment consists of only equalities, this ensures that μ = 0.1. Now the desired *g* value of 5 is obtained by ensuring that within the batch of 1000 tuples, attribute *score* is assigned 200 distinct values, each appearing 5 times. Similarly, if *g* were 500, then *score* would be assigned 2 distinct values, each appearing 500 times.

Figure 5 shows the execution times for the five algorithms. Due to the wide range of values taken by the various algorithms (from 60 ms to 45 minutes), the y-axis is displayed on a log-scale. The figure highlights a number of features of the algorithms. First, we comment on the behavior of each of the algorithms individually and then comment on their relative performance. Here, as expected, the performance of W_MJMC and W_MJGNL are directly dependent on the degree of compression achieved by grouping. On the other hand, W_MJNL and W_MJSM are unaffected by grouping; their performance is only dependent on the sizes of the input relations. This observation was validated by another experiment where we varied the selectivity and kept group size constant, but since the trends are similar, we omit those figures.

Finally, GREEDY is also unaffected by varying group sizes; since it materializes the full relational join, its performance is dependent only on the sizes of the input relations and the selectivity, both of which are fixed.

Of the algorithms, W_MJMC has the most varied performance, ranging from being the slowest to the fastest depending on input graph size. The figure also shows that W_MJGNL consistently outperforms W_MJNL by over two orders of magnitude for even the smallest group sizes. W_MJSM stays around the one second mark for all cases and dominates the other algorithms when there are many groups. Finally, GREEDY is outperformed by at least one and usually all the other algorithms by a wide margin.

Experiment 1 also highlights an interesting property of the quality of the matchings obtained by our approximate algorithms. Recall that W_MJMC is always optimal, i.e., it returns the maximum weighted matching, as is W_MJSM on equality predicates. On the other hand, GREEDY, W_MJNL, and W_MJGNL return, in general, approximate weighted matchings. It turns out that when the match predicate consists of only equalities and the weight function is decomposable monotone - as is the case in this experiment - all the algorithms are optimal. This is because, as mentioned in the discussion immediately following Theorem 2 in Section 3.3, when the match predicate consists of only equalities, one can divide the relations into buckets of *R* and *S* tuples such that each bucket represents a complete bipartite sub-graph. It can indeed be shown that on complete graphs, GREEDY is optimal regardless of the weight function, whereas W_MJNL and W_MJGNL are optimal when the weight function is decomposable monotone; the basic idea behind this is that on complete bipartite graphs, one can build a maximum weighted matching by incrementally adding to the result set the highest weighted (*r*,*s*) tuple such that neither *r* nor *s* are previously matched. GREEDY achieves this by computing and sorting the complete edge set in descending order of weight, whereas W_MJNL and W_MJGNL exploit the property of decomposable monotone weight functions to sort both relations by their weight function components before iterating over them in nested loops fashion.

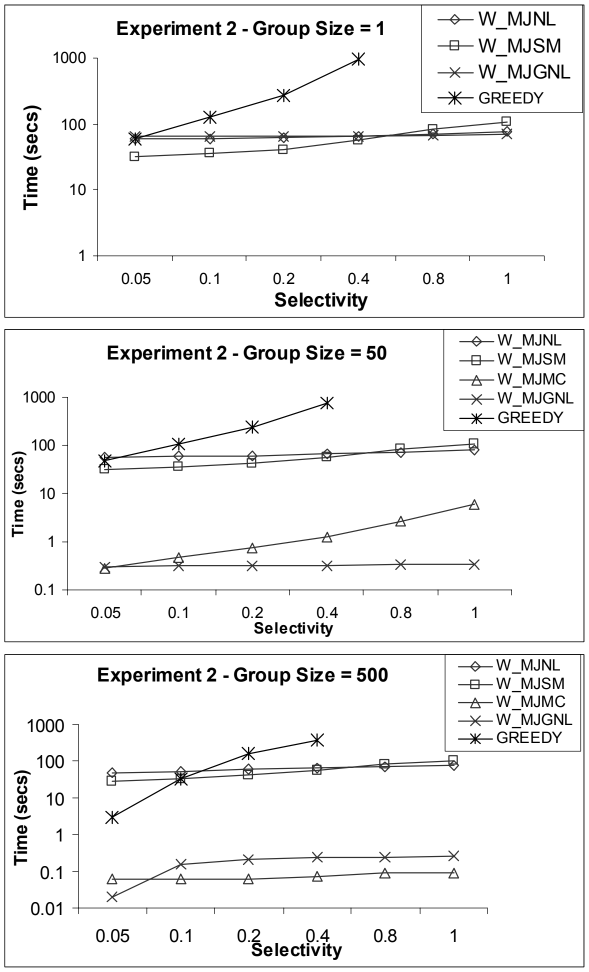
Experiment 2

Our second experiment measures the performance of all five algorithms on a weighted match join query with predicate (*R.a* = *S.a* AND *R.b* < *S.b*) and whose weight function *F* is again (0.5 * *R.score* + 0.5 * *S.score*).

Recall from Section 3.3 that W_MJSM can be extended to work on match predicates consisting of equalities and other predicates. Since, in such cases, it relies on the equality predicate to limit the range of tuples over which it performs nested loops, the selectivity of the equality predicate plays a major role in its performance. On the other hand, as was shown in the previous experiment, the performance of W_MJMC and W_MJGNL depends on the degree of compression due to grouping on the combination of the match join and weight function attributes (*a*, *b*, *score*). Accordingly, we report the performance of the five algorithms on three different group sizes: small, medium and large (g = 1, 50, and 500 respectively). Within each group size, we vary the selectivity of only the equality predicate from 0.05 to 1.

The results are shown in Figures 6 a, b, and c respectively. W_MJNL is, as expected, unaffected by either the group size or selectivity and consistently takes around a minute. The sensitivity of GREEDY to the selectivity, which in turn controls the size of the full relational join is also evident from the rise in each of the curves; for large selectivities ($\mu \geq 0.8$), GREEDY took longer than an hour, at which point we aborted it. The take-away message of this experiment is that it shows instances where the grouping-based algorithms outperform the sorting-based W_MJSM, and vice versa. In particular, when there is ample compression due to grouping (Figures 6b and 6c), both W_MJMC and W_MJGNL clearly outperform W_MJSM. On the other hand, when grouping does not help at all (g = 1), W_MJMC took longer than 12 hours for the smallest of these graphs and we aborted it at that point; as such, the times for W_MJMC are not included in Figure 6a. In such cases one can still use W_MJSM when the selectivity of the equality predicate is low (the left half of Figure 6a). Finally, when neither grouping nor the selectivity of the equality predicate help (the right half of Figure 6a), W_MJNL becomes the algorithm of choice.

Now we comment on the quality of the matchings found by the various algorithms. First recall that W_MJMC always obtains the maximum weighted matching whereas the other four can yield approximate results. While they are each guaranteed to produce a ½-optimal maximal matching, their actual results were, on average, 0.7-optimal and in about 25% of the cases, greater than 0.8-optimal.



Figures 6 a, b, and c. Experiment 2

5. Conclusion

It is clear from our results in both Experiment 1 and Experiment 2 that of the algorithms we consider, no single algorithm is applicable and superior in all cases. The choice of algorithm for a particular problem instance rests on two factors: 1) finding algorithms that are applicable to this problem instance, and 2) of all applicable algorithms, picking one that promises to work best. The first question can be answered by just inspecting the match predicate and weight function (that is, W_MJNL and W_MJGNL work only on decomposable monotone functions, the basic W_MJSM works only on equality predicates, and so forth.) However, the second question is more involved, since the "best" choice depends on query and data characteristics. For the various algorithms we propose, this decision frequently boils down to choosing between the grouping-based and the sorting-based algorithms, and ultimately rests on data characteristics such as the amount of compression achieved by grouping and selectivity of the underlying match predicates. Since commercial query optimizers already make such decisions when optimizing joins with aggregates, we think this is a natural fit for an RDBMS.

6. References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] D. Avis. "A Survey of Heuristics for the Weighted Matching Problem". Journal of the ACM (JACM), Vol. 35, No. 4, p. 769-776, 1983.

[3] "GOBLIN: A Graph Object Library for Network Programming Problems", http://www.math.uni-augsburg.de/~fremuth/goblin.html

[4] S. Guha et al. "Merging the Results of Approximate Match Operations". In *VLDB 2004*, p. 636-647.

[5] V. Hristidis, N. Koudas, and Y. Papakonstantinou. "PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries". In *ACM SIGMOD 2000*, p. 259-270.

[6] I. Ilyas, W. Aref, and A. Elmagarmid. "Supporting Top-k Join Queries in Relational Databases". *VLDB Journal*, v.13 n.3, p. 207-221, 2004.

[7] A. Kini, S. Shankar, J. F. Naughton, and D. J. Dewitt. "Database Support For Matching: Limitations and Opportunities". In *ACM SIGMOD 2006*, p. 85-96.

[8] A. Mehta et al. "Adwords and Generalized Online Matching". In *IEEE FOCS 2005*, p. 264-273.

[9] B.-W. On et al. "Group Linkage". To appear in *ICDE 2007*.

[10] R. Raman et al. "Matchmaking: Distributed Resource Management for High Throughput Computing". In *IEEE HPDC 1998*, p. 140-146.

[11] C. Schlup. "Automatic Game Matching", http://dcg.ethz.ch/theses/ws0203/OnlineMatching_abstract.pdf

[12] T. Tannenbaum et al. "Condor – A Distributed Job Scheduler". *Beowulf Cluster Computing with Linux*, The MIT Press, 2002.

[13] P. Tsaparas et al. "Ranked Join Indices". In *IEEE ICDE 2003*, p. 277-288.

[14] Y. Wang et al. "A Bipartite Graph Matching Framework for Finding Correspondences between Structural Elements in Two Proteins". In *IEEE EMBS 2004*, p. 2972-2975.