

# Database Support for Matching: Limitations and Opportunities

Ameet Kini   Srinath Shankar   Jeffrey F. Naughton   David J. Dewitt

Department of Computer Sciences  
University of Wisconsin – Madison  
1210 W. Dayton Street, Madison, WI 53706  
{akini, srinath, naughton, dewitt}@cs.wisc.edu

## ABSTRACT

We define a match join of  $R$  and  $S$  with predicate  $\theta$  to be a subset of the  $\theta$ -join of  $R$  and  $S$  such that each tuple of  $R$  and  $S$  contributes to at most one result tuple. Match joins and their generalizations belong to a broad class of matching problems that have attracted a great deal of attention in disciplines including operations research and theoretical computer science. Instances of these problems arise in practice in resource allocation scenarios. To the best of our knowledge no one uses an RDBMS as a tool to help solve these problems; our goal in this paper is to explore whether or not this needs to be the case. We show that the simple approach of computing the full  $\theta$ -join and then applying standard graph-matching algorithms to the result is ineffective for all but the smallest of problem instances. By contrast, a closer study shows that the DBMS primitives of grouping, sorting, and joining can be exploited to yield efficient match join operations. This suggests that RDBMSs can play a role in matching related problems beyond merely serving as expensive file systems exporting data sets to external user programs.

## 1. INTRODUCTION

As more and more diverse applications seek to use RDBMSs as their primary storage, the question frequently arises as to whether we can exploit the query capabilities of the RDBMS to support these applications. Some recent examples of this include OPAC queries [9], preference queries [2, 5], and top- $k$  selection [8] and join queries [12, 20]. Here we consider the problem of supporting “matching” operations. In mathematical terms, a matching problem can be expressed as follows: given a bipartite graph  $G$  with edge set  $E$ , find a subset of  $E$ , denoted  $E'$ , such that for each  $e = (u, v) \in E'$ , neither  $u$  nor  $v$  appears in any other edge in  $E'$ . Intuitively, this says that each node in the graph is matched with at most one other node in the graph. Many versions of this problem can be defined by requiring different properties of the chosen subset – perhaps the most simple is the one we explore in this paper, where we want to find a subset of maximum cardinality.

Instances of matching problems are ubiquitous across many industries, arising whenever it is necessary to allocate resources to its consumers; [3] contains references to many real-world matching problems, some of which are personnel assignment, matching moving objects, warehouse inventory management, and job scheduling. [18] argues that the problem of matchmaking players in online gaming [21] can be effectively modeled as a matching problem. Our goal in this paper is not to subsume all of this research – our goal is much less ambitious: to take a first step in investigating whether DBMS technology has anything to offer even in a simple version of these problems.

In an RDBMS, matching arises when there are two entity sets, one stored in a table  $R$ , the other in a table  $S$ , that need to have their elements paired in a matching. Compared to classical graph theory, an interesting and complicating difference immediately arises: rather than storing the complete edge graph  $E$ , we simply store the nodes of the graph, and represent the edge set  $E$  implicitly as a match join predicate  $\theta$ . 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.

Perhaps the most obvious way to compute a matching over database-resident data would be to exploit the existing graph matching algorithms developed by the theory community over the years. Because these algorithms require the fully materialized bipartite graph as input, this could be accomplished by first computing the  $\theta$ -join (the usual relational algebraic join) of the two tables, with  $\theta$  as the match predicate. 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  $R$  “matches” many rows in  $S$ ).

Accordingly, in this paper we explore alternate exact and approximate strategies of using an RDBMS to compute the *maximum cardinality matching* of relations  $R$  and  $S$  with match join predicate  $\theta$ . If nothing is known about  $\theta$ , we propose a nested-loops based algorithm, which we term MJNL (Match Join Nested Loops). This will always produce a matching, although it is not guaranteed to be a maximum matching.

If we know more about the match join predicate  $\theta$ , faster algorithms are possible. We propose two such algorithms. The first, which we term MJMF (Match Join Max Flow), requires knowledge of which attributes serve as inputs to the match join predicate. It works by first “compressing” the input relations with a group-by operation, then feeding the result to a max flow algorithm. We show that this always generates the maximum matching, and is efficient if the compression is effective. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.  
Copyright 2006 ACM 1-59593-256-9/06/0006...\$5.00.

second, which we term MJSM (Match Join Sort Merge), requires more detailed knowledge of the match join predicate. We characterize a family of match join predicates over which MJSM yields maximum matches.

Our algorithms are implemented using vanilla SQL and user defined functions (UDFs) in the Predator RDBMS [16] and we report their performance. Our results show that these algorithms lend themselves well to a RDBMS-based implementation as they make good use of existing RDBMS primitives such as scanning, grouping, sorting and merging. A road map of this paper is as follows: We start by formally defining the problem statement in Section 2. We then move on to the description of the three different match join algorithms MJNL, MJMF, and MJSM in Sections 3, 4, and 5 respectively. Section 6 contains a discussion of our experiments with Predator. Section 7 defines and describes a generalization of the match join and discusses future work. Related work is presented in Section 8. Finally, we conclude in Section 9.

## 2. PROBLEM STATEMENT

Before describing our algorithms, we first formally describe the match join problem. We begin with relations  $R$  and  $S$  and a predicate  $\theta$ . Here, the rows of  $R$  and  $S$  represent the nodes of the graph and the predicate  $\theta$  is used to implicitly denote edges in the graph. The relational join  $R \bowtie_{\theta} S$  then computes the complete edge set that serves as input to a classical matching algorithm.

**Definition 1 (Match join)** Let  $M \subseteq R \bowtie_{\theta} S$ . Then  $M$  is a matching or a match join of  $R$  and  $S$  with predicate  $\theta$  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  $M$ .

**Definition 2 (Maximal Matching)** A matching  $M'$  is a maximal matching of relations  $R$  and  $S$  with predicate  $\theta$  if  $\forall r \in R - M'(R), s \in S - M'(S), (r,s) \notin R \bowtie_{\theta} S$ . Informally,  $M'$  cannot be expanded by just adding edges.

**Definition 3 (Maximum Matching)** Let  $M^*$  be the set of all matchings of relations  $R$  and  $S$  with predicate  $\theta$ . Then  $MM$  is a maximum matching iff  $MM \in M^*$  and  $\forall M' \in M^*, |MM| \geq |M'|$ .

Note that just as there can be more than one matching, there can also be more than one maximal and maximum matching. Also note that every maximum matching is also a maximal matching but not vice-versa.

## 3. MATCH JOIN USING NESTED LOOPS

Assuming that the data is DBMS-resident, a simple way to compute the matching is to materialize the entire graph using a relational join operator, and then feed this to an external graph matching algorithm. While this approach is straightforward and makes good use of existing graph matching algorithms, it suffers two main drawbacks:

- Materializing the entire graph is a time/space intensive process;
- The best known maximum matching algorithm for bipartite graphs is  $O(n^{2.5})$  [11], which can be too slow even for reasonably sized input tables.

Recent work in the theoretical community has led to algorithms that give fast approximate solutions to the maximum matching problem, thus addressing the second issue above; see [14] for a survey on the topic. Specifically, [6] gives a  $(2/3 - \epsilon)$ -approximation algorithm ( $0 < \epsilon < 1/3$ ) that makes multiple passes over the set of edges in the underlying graph. However, since both the exact and the approximate algorithms require the entire set of edges as input, the full relational join has to be materialized. As a result, these approaches have their performance bounded below by the time to compute a full relational join, thus making them unlikely to be successful for large problem instances.

Our first approach is based on the nested loops join algorithm. Specifically, consider a variant of the nested-loops join algorithm that works as follows: Whenever it encounters a matching  $(r,s)$  pair, it adds it to the result and then marks  $r$  and  $s$  as “matched” so that they are not matched again. We refer to this algorithm as MJNL; it has the advantage of computing match joins on arbitrary match join predicates. In addition, one can show that it always results in a maximal matching, although it may not be a maximum matching (see Lemma 1 below). It is shown in [3] that maximal matching algorithms return at least 1/2 the size of the maximum matching, which implies that MJNL always returns a matching with at least half as many tuples as the maximum matching. We can also bound the size of the matching produced by MJNL relative to the percentage of matching  $R$  and  $S$  tuples. These two bounds on the quality of matches produced by MJNL are summarized in the following theorem:

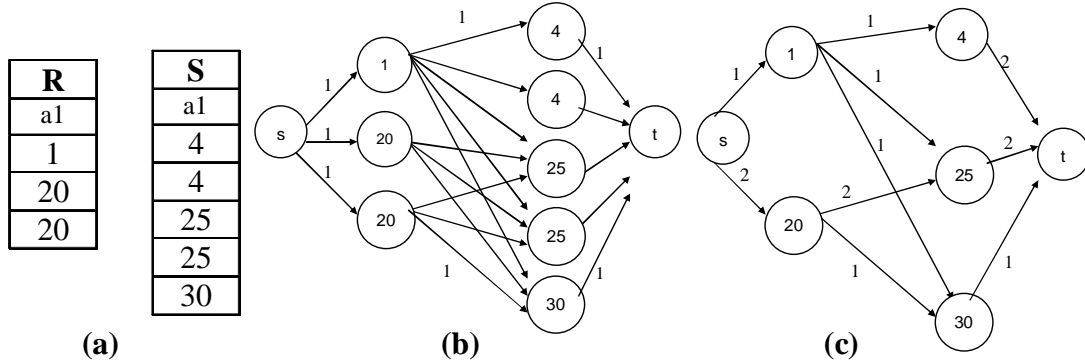
**Lemma 1** Let  $M$  be the matching returned by MJNL. Then,  $M$  is maximal.

**Proof:** MJNL works by searching through the entire set of matching  $s$  nodes for each and every node  $r$ , and picking the first one available. Once entered, an edge never leaves  $M$ . As such, if a certain edge  $(r,s) \notin M$  where  $M$  is the final match returned by MJNL, it is because either  $r$  or  $s$  or both are already matched with other nodes, or because both  $r$  and  $s$  cannot be matched with any node. In either case,  $M$  cannot be expanded by adding  $(r,s)$   $\square$

**Theorem 1** Let  $MM$  be the maximum matching of relations  $R$  and  $S$ . Let  $M$  be the match returned by MJNL. Then,  $|M| \geq \lceil 0.5 * |MM| \rceil$ . Furthermore, if  $p_r$  percentage of  $R$  tuples match at least  $p_s$  percentage of  $S$  tuples, then  $|M| \geq \min(p_r * |R|, p_s * |S|)$ . As such,  $|M| \geq \max(\lceil 0.5 * |MM| \rceil, \min(p_r * |R|, p_s * |S|))$ .

**Proof:** By Lemma 1,  $M$  is maximal. It is shown in [3] that for a maximal matching  $M$ ,  $|M| \geq \lceil 0.5 * |MM| \rceil$ . We now prove the second bound, namely that  $|M| \geq \min(p_r * |R|, p_s * |S|)$  for the case when  $p_s * |S| \leq p_r * |R|$ . The proof for the reverse is similar.

By contradiction, assume  $|M| < p_s * |S|$ , say,  $|M| = p_s * |S| - k$  for some  $k > 0$ . Now, looking at the  $R$  tuples in  $M$ , MJNL returned only  $p_s * |S| - k$  of them, because for the other  $r' = |R| - |M|$  tuples, it either saw that their only matches are already in  $M$  or that they did not have a match at all, since  $M$  is maximal. Therefore, each of these  $r'$  tuples match with less than  $p_s * |S|$  tuples. By assumption, since  $p_r$  percentage of  $|R|$  tuples match with at least  $p_s * |S|$  tuples, the percentage of  $R$  tuples that match with less than  $p_s * |S|$  tuples are at most  $1 - p_r$ . So  $r' / |R| \leq 1 - p_r$ . Since  $r' = |R| - (p_s * |S| - k)$ , we have



**Figure 1. A 3-step transformation from (a) Base tables to (b) A unit capacity network to (c) A reduced network that is input to the max flow algorithm**

$$(|R| - (p_s^*/|S| - k)) / |R| < 1 - p_r$$

$$\rightarrow |R| - p_s^*/|S| + k < |R| - p_r^*/|R|$$

$\rightarrow k < p_s^*/|S| - p_r^*/|R|$ , which is a contradiction since  $k > 0$  and  $p_s^*/|S| - p_r^*/|R| \leq 0$   $\square$

Note that the difference between the two lower bounds can be substantial; so the combined guarantee on size is stronger than either bound in isolation. The above results guarantee that in the presence of arbitrary join predicates, MJNL results in the maximum of the two lower bounds.

Of course, the shortcoming of MJNL is its performance. We view MJNL as a “catch all” algorithm that is guaranteed to always work, much as the usual nested loops join algorithm is included in relational systems despite its poor performance because it always applies. We now turn to consider other approaches that have superior performance when they apply.

## 4. MATCH JOIN USING MAX FLOW

In this section, we show our second approach of solving the match join problem for arbitrary join predicates. The insight here is that in many problem instances, the input relations to the match join can be partitioned into groups such that the tuples in a group are identical with respect to the match (that is, either all members of the group will join with a given tuple of the other table, or none will.) For example, in the context of job scheduling on a grid, most clusters consist of only a few different kinds of machines; similarly, many users submit thousands of jobs with identical resource requirements.

The basic idea of our approach is to perform a relational group-by operation on attributes that are inputs to the match join predicate. We keep one representative of each group, and a count of the number of tuples in each group, and feed the result to a max-flow UDF. As we will see, the maximum matching problem can be reduced to a max flow problem. Note that for this approach to be applicable and effective, (1) we need to know the input attributes to the match join predicate, and (2) the relations cannot have “too many” groups. MJNL did not have either of those limitations.

## 4.1 Max Flow

The max flow problem is one of the oldest and most celebrated problems in the area of network optimization. Informally, given a graph (or network) with some nodes and edges where each edge has a numerical flow capacity, we wish to send as much flow as possible between two special nodes, a source node  $s$  and a sink node  $t$ , without exceeding the capacity of any edge. Here is a definition of the problem from [3]:

**Definition 4 (Max Flow Problem)** Consider a capacitated network  $G = (N, E)$  with a nonnegative capacity  $u_{ij}$  associated with each edge  $(i, j) \in E$ . There are two special nodes in the network  $G$ : a source node  $s$  and a sink node  $t$ . The max flow problem can be stated formally as:

Maximize  $v$  subject to:

$$\begin{aligned} v & \quad \text{for } i = s, \\ \sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} &= 0 \quad \text{for all } i \in N - \{s \text{ and } t\} \\ -v & \quad \text{for } i = t \end{aligned}$$

Here, we refer to the vector  $x = \{x_{ij}\}$  satisfying the constraints as a flow and the corresponding value of the scalar  $v$  as the value of the flow.

We first describe a standard technique for transforming a matching problem to a max flow problem. We then show a novel transformation of that max flow problem into an equivalent one on a smaller network. Given a match join problem on relations  $R$  and  $S$ , we first construct a directed bipartite graph  $G = (N_1 \cup N_2, E)$  where a) nodes in  $N_1$  ( $N_2$ ) represent tuples in  $R$  ( $S$ ), b) all edges in  $E$  point from the nodes in  $N_1$  to nodes in  $N_2$ . We then introduce a source node  $s$  and a sink node  $t$ , with an edge connecting  $s$  to each node in  $N_1$  and an edge connecting each node in  $N_2$  to  $t$ . We set the capacity of each edge in the network to 1. Such a network where every edge has flow capacity 1 is known as a *unit capacity network* on which there exists max flow algorithms that run in  $O(m\sqrt{n})$  (where  $m=|E|$  and  $n=|N|$ ) [3]. Figure 1(b) shows this construction from the data in Figure 1(a).

Such a unit capacity network can be “compressed” using the following idea: If we can somehow gather the nodes of the unit capacity network into groups such that every node in a group is connected to the same set of nodes, we can then run a max flow algorithm on the smaller network in which each node represents

a group in the original unit capacity network. To see this, consider a unit capacity network  $G = (N_1 \cup N_2, E)$  such as the one shown in Figure 1(b). Now we construct a new network  $G' = (N_1' \cup N_2', E')$  with source node  $s'$  and sink node  $t'$  as follows:

1. (Build new node set) add a node  $n_i' \in N_1'$  for every group of nodes in  $N_1$  which have the same value on the match join attributes; similarly for  $N_2'$ .
2. (Build new edge set) add an edge between  $n_1'$  and  $n_2'$  if there was an edge between the original two groups which they represent.
3. (Connecting new nodes to source and sink) add an edge between  $s'$  and  $n_1'$ , and between  $n_2'$  and  $t'$ .
4. (Assign new edge capacities) For edges of the form  $(s', n_1')$  the capacity is set to the size of the group represented by  $n_1'$ . Similarly, the capacity on  $(n_2', t')$  is set to the size of the group represented by  $n_2'$ . Finally, the capacity on edges of the form  $(n_1', n_2')$  is set to the minimum of the two group sizes.

Figure 1(c) shows the above steps applied to the unit capacity network in Figure 1(b).

Finally, the solution to the above reduced max flow problem can be used to retrieve the maximum matching from the original graph, as stated below. The underlying idea is that by solving the max flow problem subject to the above capacity constraints, we obtain a flow value on every edge of the form  $(n_1', n_2')$ . Let this flow value 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.

**Theorem 2** *A solution to the reduced max flow problem in the transformed network  $G'$  constructed using steps 1-4 above corresponds to a maximum matching on the original bipartite graph  $G$ .*

**Proof (Sketch):** See [3] for a proof of the first transformation (between matching in  $G$  and max flow on a unit capacity network). Our proof follows a similar structure by showing a) every matching in  $G$  corresponds to a flow in  $G'$ , and b) every flow in  $G'$  corresponds to a matching in  $G$ . b) By the flow decomposition theorem [3], every path flow must be of the form  $s \rightarrow i_1 \rightarrow i_2 \rightarrow t$  where  $s, t$  are the source, sink and  $i_1, i_2$  are the aggregated nodes in  $G'$ . Moreover, due to the capacity constraints, the flow on edge  $(i_1, i_2)$ , say,  $\varphi = \min(\text{flow}(s, i_1), \text{flow}(i_2, t))$ . Thus, we can add  $\varphi$  edges of the form  $(i_1, i_2)$  to the final matching. a) The correspondence between a matching in  $G$  and a flow  $f$  in a unit capacity network is shown in [3]. Going from  $f$  to  $f'$  on  $G'$  is simple. For an edge of the form  $(s, i_1)$  in  $G'$ , set its flow to the number of members of the  $i_1$  group that got matched. This is within the flow capacity of  $(s, i_1)$ . Do the same for edges of the form  $(i_2, t)$ . Since  $f$  corresponds to a matching, edges of the form  $(i_1, i_2)$  are guaranteed to be within their capacities  $\square$

## 4.2 Implementation of MJMF

We now discuss issues related to implementing the above transformation in a relational database system.

The complete transformation from a matching problem to a max flow problem can be divided into three phases, namely, that of grouping nodes together, building the reduced graph, and invoking the max flow algorithm. The first stage of grouping involves finding tuples in the underlying relation that have the same value on the join columns. Here, we use the relational group-by operator on the join columns and eliminate all but a representative from each group (using, say the min or the max function). Additionally, we also compute the size of each group using the count() function. This count will be used to set the capacities on the edges as was discussed in Step 4 of Section 4.1. Once we have “compressed” both input relations, we are ready to build the input graph to max flow. Here, the tuples in the compressed relations are the nodes of the new graph. The edges, on the other hand, can be materialized by performing a relational  $\theta$ -join of the two outputs of the group-by operators where  $\theta$  is the match join predicate. Note that this join is smaller than the join of the original relations when groups are fairly large (in other words, when there are few groups). We illustrate the SQL for this transformation on the following example schema:

Tables:  $R(a_1, \dots, a_m), S(b_1, \dots, b_n)$

Match Join Predicate:  $\theta(R.a_1, \dots, R.a_m, S.b_1, \dots, S.b_n)$

SQL for 3-step transformation to reduced graph:

```
SELECT *
FROM((SELECT COUNT(*) AS group_size,
      MAX(R.a1) AS a1, ..., MAX(R.am) AS am
      FROM R
      GROUP BY R.a1, ..., R.am) AS T1,
      (SELECT COUNT(*) AS group_size,
      MAX(S.b1) AS b1, ..., MAX(S.bn) AS bn
      FROM S
      GROUP BY S.b1, ..., S.bn) AS T2)
WHERE  $\theta(T_1.a_1, \dots, T_1.a_m, T_2.b_1, \dots, T_2.b_n)$ ;
```

Finally, the resulting graph can now be fed to a max flow algorithm. Due to its prominence in the area of network optimization, there have been many different algorithms and freely available implementations proposed for solving the max flow problem with best known running time of  $O(n^3)$  [7]. One such implementation can be encapsulated inside a UDF which first issues the above SQL to obtain the reduced graph before invoking the max flow algorithm on this graph.

In summary, MJMF always gives a maximum matching, and requires only that we know the input attributes to the match join predicate. However, for efficiency it relies heavily on the premise that there are not too many groups in the input. In the next section, we consider an approach that is efficient even in the presence of a large number of groups, although it requires more knowledge about the match predicates if it is to return the maximum matching.

Original Tables						
R				S		
a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>		a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>
10	100	1000	Join predicates	10	100	1110
10	100	1200	R.a <sub>1</sub> = S.a <sub>1</sub> &	10	100	1220
10	100	1100	R.a <sub>2</sub> = S.a <sub>2</sub> &	10	100	1000
10	200	1200	R.a <sub>3</sub> < S.a <sub>3</sub>	10	200	1000
10	200	1000		20	200	4000
20	200	2000		20	200	4000
20	200	3000				

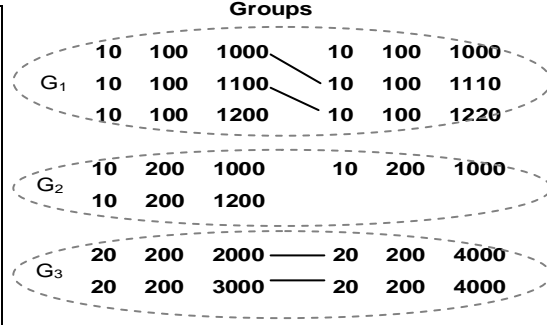


Figure 2. Illustration of MJSM

## 5. MATCH JOIN USING SORT MERGE

### 5.1 The algorithm

The intuition behind MJSM is that by exploiting the semantics of the match join predicate  $\theta$ , we can sometimes efficiently compute the maximum matching without resorting to general graph matching algorithms. To see the insight for this, consider the case when  $\theta$  consists of only equality predicates. Here, we can use a simple variant of sort-merge join: like sort-merge join, we first sort the input tables on their match join attributes. Then we “merge” the two tables, except that when a tuple  $r$  in  $R$  matches a tuple  $s$  in  $S$ , we output  $(r,s)$  and advance the iterators on both  $R$  and  $S$  (so that these tuples are not matched again.) In this subsection, we describe this algorithm and prove conditions under which it returns a *maximum matching*. Although this algorithm always returns a *matching*, as we later show, it is guaranteed to return a *maximum matching* if the match join predicate possesses certain properties.

Before describing the algorithm and proving its correctness, we introduce some notation and definitions used in its description. First, recall that the input to a match join consists of relations  $R$  and  $S$ , and a predicate  $\theta$ .  $R \bowtie_{\theta} S$  is, as usual, the relational  $\theta$  join of  $R$  and  $S$ . For now, assume that  $\theta$  is a conjunction of the form  $R.a_1 \text{ op}_1 S.a_1 \text{ AND } R.a_2 \text{ op}_2 S.a_2 \text{ AND, ..., AND } R.a_{p-1} \text{ op}_{p-1} S.a_{p-1} \text{ AND } R.a_p \text{ op}_p S.a_p$ , where  $\text{op}_1$  through  $\text{op}_p$  are relational operators ( $=, <, >$ , etc.); we will relax some of these assumptions later.

MJSM computes the match join of the two relations by first dividing up the relations into *groups* of candidate matching tuples of  $R$  and  $S$  and then computing a match join within each group. Groups are constructed in such a manner that in each group  $G$ , all tuples of  $G(R)$ , (i.e., the  $R$  tuples in  $G$ ) match with all tuples of  $G(S)$  (i.e., the  $S$  tuples in  $G$ ) on all equality predicates (e.g.,  $R.a_1 = S.a_1 \text{ AND } R.a_2 = S.a_2$ ), if there are any.

The main steps of the algorithm are as follows:

1. Perform an external sort of both input relations on all attributes involved in  $\theta$ .
2. Iterate through the relations and generate the next group  $G$  of  $R$  and  $S$  tuples.
3. Within  $G$ , merge the two subsets of  $R$  and  $S$  tuples, just as in merge-join, except that iterators on both tables can be advanced as soon as matches are found.

4. Add the matching tuples to the final result. Go to 2.

Figure 2 illustrates the operation of MJSM when the match join predicate is a conjunction of two equalities and one inequality. The original tables are divided into groups. Within a group, MJSM runs down the two lists outputting matches as it finds them. Note that the groups are sorted in (increasing) order of all attributes that appear in the match join predicate. Matched tuples are indicated by solid arrows.

In its worst case, the running time of a conventional sort-merge join is proportional to the product of the sizes of its input relations (e.g. when the size of the join is equal to the size of the cross product). The cost of MJSM, however, is simply that of sorting (Step 1 above) and scanning once (Steps 2 and 3 above) of both relations. This is because in MJSM, iterators are never “backed up” as they are in the conventional sort-merge join.

### 5.2 When does MJSM find the maximum match?

The general intuition behind MJSM is the following: If  $\theta$  consists of only equality predicates, then matches can only be found within a group. A greedy pass through both tables within a group can then retrieve the maximum match<sup>1</sup>. As it turns out, the presence of one inequality can be dealt with a similar greedy single pass through both relations.

We now characterize the family of match join predicates  $\theta$  for which MJSM can produce the maximum matching. First, we define something called a “zig-zag”, which is useful in determining when MJSM returns a maximum matching.

**Definition 5 (Zig-zags)** Consider the class of matching algorithms that work by enumerating (a subset of) the elements of the cross product of relations  $R$  and  $S$ , and outputting them if they match (MJSM is in this class). We say that a matching algorithm in this class encounters a zig-zag if at the point it picks a tuple  $(r,s)$   $r \in R$  and  $s \in S$  as a match, there exists tuples  $r' \in R-M(R)$  and  $s' \in S-M(S)$  such that  $r'$  could have been matched with  $s$  but not  $s'$  whereas  $r$  could also match  $s'$ .

<sup>1</sup> Due to this property, a simple extension of the hash join algorithm can also be used to compute match joins on equality predicates.

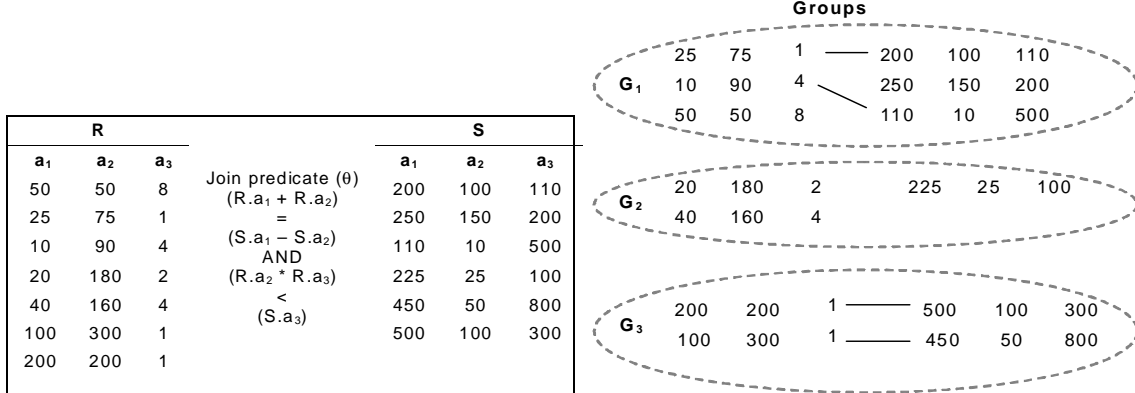


Figure 3. Extending MJSM to accept predicates that contain functions

Note that  $r'$  and  $s'$  could be in the match at the end of the algorithm; the definition of zig-zags only require them to not be in the matched set at the point when  $(r,s)$  is chosen. As we later show, zig-zags are hints that an algorithm chose a 'wrong' match, and avoiding zig-zags is part of a sufficient condition for proving that the resulting match of an algorithm is indeed maximum.

**Lemma 2** *Let  $M$  be the result of a matching algorithm  $A$ , i.e.  $M$  is a match join of relations  $R$  and  $S$  with predicate  $\theta$ . If  $M$  is maximal and  $A$  never encounters zig-zags, then  $M$  is also maximum.*

The proof uses a theorem due to Berge [4] that relates the size of a matching to the presence of an augmenting path, defined as follows:

**Definition 6 (Augmenting Path)** *Given a matching  $M$  on graph  $G$ , an augmenting path through  $M$  is a path in  $G$  that starts and ends at free (unmatched) nodes and whose edges are alternately in  $M$  and  $E-M$ .*

**Theorem 3 (Berge)** *A matching  $M$  is maximum if and only if there is no augmenting path with respect to  $M$ .*

**Proof of Lemma 2:** Assume that an augmenting path indeed exists. We show that the presence of this augmenting path necessitates the existence of two nodes  $r \in R-M(R)$ ,  $s \in R-M(S)$  and edge  $(r,s) \in R \not\bowtie_{\theta} S$ , thus leading to a contradiction since  $M$  was assumed to be maximal.

Now, every augmenting path is of odd length. Without loss of generality, consider the following augmenting path of size  $2k-1$  consisting of nodes  $r_k, \dots, r_1$  and  $s_k, \dots, s_1$ :

$$r_k \rightarrow s_k \rightarrow r_{k-1} \rightarrow s_{k-1} \rightarrow \dots \rightarrow r_1 \rightarrow s_1$$

By definition of an augmenting path, both  $r_k$  and  $s_1$  are free, i.e., they are not matched with any node. Further, no other nodes are free, since the edges in an augmenting path alternate between those in  $M$  and those not in  $M$ . Also, edges  $(r_k, s_k), (r_{k-1}, s_{k-1}), \dots, (r_2, s_2), (r_1, s_1)$  are not in  $M$  whereas edges  $(s_k, r_{k-1}), (s_{k-1}, s_{k-2}), \dots, (s_3, r_2), (s_2, r_1)$  are in  $M$ . Now, consider the edge  $(r_1, s_1)$ . Here,  $s_1$  is free and  $r_2$  can be matched with  $s_2$ .

Since  $(s_2, r_1)$  is in  $M$  and, by assumption,  $A$  does not encounter zig-zags,  $r_2$  can be matched with  $s_1$ . Now consider the edge  $(r_2, s_1)$ . Here again,  $s_1$  is free and  $r_3$  can be matched with  $s_3$ . Since

$(s_3, r_2)$  is in  $M$  and  $A$  does not encounter zig-zags,  $r_3$  can be matched with  $s_1$ . Following the same line of reasoning along the entire augmenting path, it can be shown that  $r_k$  can be matched with  $s_1$ . This is a contradiction, since we assumed that  $M$  is maximal  $\square$

Lemma 2 gives a useful sufficient condition which we use as a tool in the rest of the subsection to prove the circumstances under which MJSM returns maximum matches.

**Lemma 3** *Let  $M$  be the match returned by  $MJSM(R,S,\theta)$ . Then  $M$  is maximum if  $\theta$  is a conjunction of  $k$  equality predicates.*

**Proof:** Let  $\theta$  be of the form  $R.a_1 = S.a_1$  AND  $R.a_2 = S.a_2$  AND, ..., AND  $R.a_k = S.a_k$ . When  $\theta$  consists of only equalities, within each group  $G$ , all  $R$  and  $S$  tuples match each other. The number of matches found by MJSM within each group =  $\min(|G(R)|, |G(S)|) = |maximum\ matching\ of\ G(R)\ and\ G(S)|$ . As a result, within each group, MJSM is maximal and avoids zig-zags. Since tuples across groups do not match, MJSM is maximal and avoids zig-zags across groups  $\square$

**Theorem 4:** *Let  $M$  be the match returned by  $MJSM(R,S,\theta)$ . Then  $M$  is maximum if  $\theta$  is a conjunction of  $k$  equality predicates and up to 1 inequality predicate.*

**Proof:** First, note that the case where  $\theta$  consists of only equality predicates is covered by Lemma 3. So let's consider the case where in addition to equalities, there is also exactly 1 inequality predicate. Without loss of generality, let  $\theta$  be 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  $R.a_{k+1} < S.a_{k+1}$ . Now within each group  $G$ , all  $R$  and  $S$  tuples match each other on the  $k$  equality predicates; tuples across groups do not match. Due to the way in which iterators are moved, each tuple in  $G(R)$  is matched with the first unmatched  $G(S)$  tuple starting from the current position of the  $G(S)$  iterator. Also, unlike the conventional sort-merge join, in MJSM, iterators are never backed up. So, if at the end of MJSM, a tuple  $r \in G(R)$  is not matched with any  $G(S)$  tuple, it is because one is not available. As a result,  $M$  is maximal. Furthermore, if  $r \in G(R)$  can be matched with  $s, s' \in G(S)$  where  $s'$  comes after  $s$  in the sort order, and if another tuple  $r' \in G(R)$  after  $r$  can also be matched with  $s$ , then  $r'$  can also be matched with  $s'$  since, due to the increasing sort order on  $a_{k+1}$ ,  $r'(a_{k+1}) < s(a_{k+1}) < s'(a_{k+1})$ . Therefore, MJSM avoids zig-zags; by Lemma 2, the resulting match is maximum  $\square$

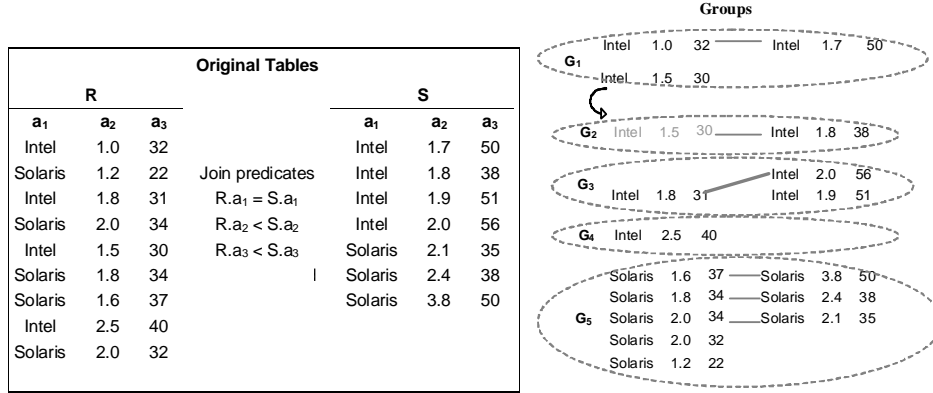


Figure 4. Extending MJSM to accept predicates that contain at most two inequalities

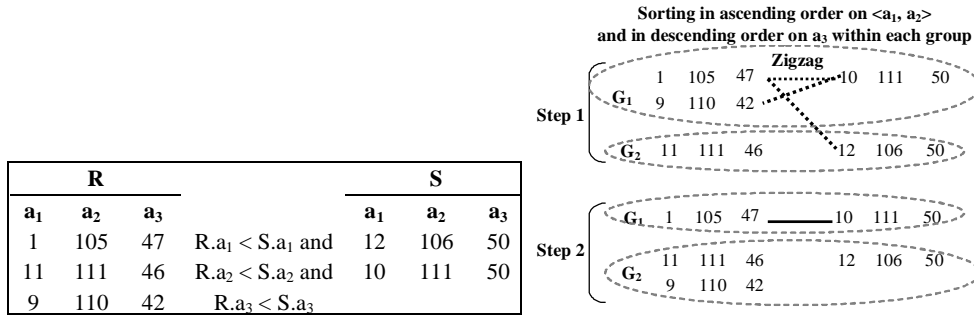


Figure 5. MJSM on 3 inequalities - prone to zig-zags

### 5.3 Extensions to MJSM

According to Lemma 2, MJSM returns maximum matches on arbitrary match join predicates provided that the combined sufficient condition of maximality and avoidance of zig-zags is met. In the case of equalities and at most one inequality, MJSM uses sorting to obtain its groups and avoid zig-zags. This simple technique can be extended to compute maximum matchings on a broader class of predicates. The first natural extension is the following: Instead of serving the attributes of the relations as operands to the equality and inequality operators, we can serve as operands, any function of those attributes. For example,  $\theta = ((R.a_1 + R.a_2) = (S.a_1 - S.a_2)) \text{ AND } ((R.a_2 * R.a_3) < S.a_3)$ . As long as the groups are constructed in such a way that all  $R$  and  $S$  tuples within the group match each other on the equality predicate and the groups are in sorted order of all attributes in the match join predicate, MJSM will return the maximum matching. In general, if  $\theta = ((f_1() = f_2()) \text{ AND } (f_3() = f_4()) \text{ AND } \dots \text{ AND } (f_{k-1}() = f_k()) \text{ AND } (f_{k+1}() < f_{k+2}()))$  where  $f_1, f_3, f_5, \dots, f_{k-1}, f_{k+1}$  are functions of attributes of  $R$ , and  $f_2, f_4, f_6, \dots, f_k, f_{k+2}$  are functions of attributes of  $S$ , then the groups can be constructed by sorting  $R$  on  $f_1(), f_3(), f_5(), \dots, f_{k-1}(), f_{k+1}()$ , and  $S$  on  $f_2(), f_4(), f_6(), \dots, f_k(), f_{k+2}()$ . In the above example, this amounts to sorting  $R$  on  $(R.a_1 + R.a_2)$ ,  $(R.a_2 * R.a_3)$  and  $S$  on  $(S.a_1 - S.a_2)$ ,  $S.a_3$ . Figure 3 illustrates how this is done.

Another extension is allowing  $\theta$  to contain at most two inequalities instead of at most one as discussed in Section 5.2. At first glance, this may seem like a simple extension. As it turns

out, however, the addition of another inequality creates opportunities for zig-zags with tuples in groups that are not yet read. The extension to MJSM then involves, among other steps, carrying over unmatched  $R$  tuples from the current group to the next. In the worst case, all  $R$  tuples from all groups keep getting carried over, and this makes the worst case complexity of this extension quadratic in the size of the larger relation; recall that the basic MJSM algorithm is  $O(n \log n)$  where  $n$  is the size of the larger relation. Figure 4 illustrates how this is done.

Note that the groups are sorted in descending order of the second inequality attribute – this is also part of the extension to the basic MJSM algorithm. It can be shown that these extensions indeed enable MJSM to compute maximum matchings when  $\theta$  contains up to two inequalities. Unfortunately, the proof is tedious and we omit it due to lack of space.

These techniques, however, do not generalize to arbitrary predicates. We illustrate the case when  $\theta$  consists of 3 inequalities in Figure 5. Here, MJSM is unable to return a maximum match due to the zig-zag identified in Step 1 of the algorithm. Once tuple  $\langle 1, 105, 47 \rangle$  is matched with  $\langle 10, 111, 50 \rangle$ ,  $\langle 9, 110, 42 \rangle$  is carried over to  $G_2$  where it finds no matches. This is because, within a group, unless there is a total order on all inequality attributes, sorting in order on one may disturb the sort order on another, thus making the algorithm vulnerable to zig-zags. However, even in such cases when MJSM does not produce the maximum match, it still produces a maximal match; thus, the lower bounds from Theorem 1 also apply for MJSM. Discovering techniques to avoid zig-zags while retaining maximality of

MJSM on other predicates is, therefore, both an interesting and challenging area of future research.

## 6. EXPERIMENTS

Our overall experimental objective was to measure the performance of our algorithms and evaluate their sensitivity to various data characteristics. We start from the most general algorithm MJNL, then consider MJMF and finally MJSM. First, recall that an alternative approach to computing the matching is to compute the full relational join in the RDBMS, then feed the result to any well-known bipartite matching algorithm, such as the ones presented in [6, 11]. As such, these approaches have their performance bounded below by the time to compute a full relational join, and henceforth, we use the latter as a basis for comparison with our algorithms; note that this underestimates the improvements offered by our algorithms as the full join, however expensive, forms only a portion of the total time in many problem instances.

We start out by comparing the performance of MJNL to the full join and show that MJNL is faster in all cases, hence its running time always dominates approaches exploiting existing graph algorithms by first computing the full join. The second set of experiments measure the performance of MJMF relative to our other match join algorithms while varying the parameter to which it is most sensitive: the size of the input graph to the max flow algorithm. We then compare MJSM to the full join for various table sizes and join selectivities. Finally, we validate our algorithms on a real-world dataset consisting of jobs and machines in the Condor job scheduling system [19].

Our algorithms were built on top of the object relational database Predator [16], which uses SHORE as its underlying storage system. All queries were run “cold” on an Intel Pentium 4 CPU clocked at 2.4GHz. The buffer pool was set at 32 MB.

In order to carefully control various data characteristics such as selectivity and group size, the first set of experiments were conducted on synthetic data; the two tables in this dataset were each ten columns wide (columns named  $a, b, c, \dots, i, j$ ), and all columns were of integer type. The particular join predicates (equality, one inequality, etc.) and other parameters that vary in the experiments are reported in the figures themselves.

Note that the size of the result produced by the full join is never smaller and may be much larger than that produced by match join algorithms. To avoid including the time to output such a large answer, we suppressed output display for all our queries. This unfairly *improves* the relative performance of the full join, but as the results show, the match joins algorithms are still significantly superior.

### 6.1 Validation on synthetic datasets

We begin by showing the performance of MJNL, comparing it to the full join on various join selectivities. With a join predicate consisting of 10 inequalities (both  $R$  and  $S$  are 10 columns wide here), grouping does not compress the data much, and MJSM will not return maximum matches. As seen in Figure 6, MJNL outperforms the full join (for which the Predator optimizer chose page nested loops, since sort-merge, hash join, and index-nested loops do not apply) in all cases. This is expected as MJNL

generates only a subset of the full join. Since the size of the full join increases with selectivity, the difference between the two algorithms also increases accordingly. Note that in its worst case, (e.g. when none of the tuples of  $R$  and  $S$  match each other), the performance of MJNL would be similar to that of the full join, thus still outperforming the overall alternative approach.

We now evaluate the performance of MJMF on varying group sizes and selectivities. Recall that MJMF works by performing a group-by on the match join attributes, followed by a full join, thus building a graph which is then fed to the max flow algorithm. Due to the  $O(n^3)$  running time of the max flow algorithm, the size of the graph  $|G|$  (or, number of edges) plays a major role in the overall performance of MJMF.  $|G|$  is a function of two variables: the average group size  $g$  and the join selectivity  $f$ . More precisely,  $|G| = f * (|Table_{left}| * |Table_{right}|) / g$ . For a fixed selectivity then, the larger the group size, the smaller the graph. Similarly, for a fixed group size a low selectivity results in a small graph. Accordingly, using synthetic datasets, we conducted 2 experiments that measured the effect of those variables on the performance of MJMF. Figure 7 shows the running times of MJMF on a join predicate consisting of 3 inequalities, joining relations of size 10000.  $f$  was kept at a constant 0.5 and  $g$  ranges from 10 (low compression) to 5000 (high compression). Accordingly,  $|G|$  ranges from 500000 to 2.

First, observe that when compression is high, MJMF consistently outperforms MJNL by almost two orders of magnitude. Additionally, MJMF has similar running times to MJSM which does not return the maximum matching for these queries. However, MJMF’s response time grows quickly as groups get smaller ( $g \leq 25$ ) and  $G$  gets larger; eventually the performance of MJMF approaches that of MJNL. (Note: the full relational join query took over 2 minutes in all the cases so we did not include it in the figure.)

In Figure 8, we report measured times spent by MJMF in its three stages: grouping, joining, and applying max flow which are labeled GBY, PNL (page nested loops) and Flow respectively in the figure. Here, we varied  $f$  keeping  $g$  at a constant 10. As  $f$  increases from 0.1 to 1,  $|G|$  ranges from around 150000 to 1.5 million, and the performance of MJMF degrades in a manner similar to Figure 7. Note that the last bar is scaled down by an order of magnitude in order to fit into the graph. Since the table sizes are kept constant at 10000, the time taken by group-by is also constant (and unnoticeable!) at 0.16 seconds. For graph sizes up to around 1 million, the max flow algorithm takes a fraction of the overall time and is dominated by the join operation. However, beyond that cross-over point, the graph was too large to be held in main memory; this caused severe thrashing and drastically slowed down the max flow algorithm. This shows that when grouping ceases to be effective, MJMF is not an effective algorithm.

As shown above, on some data sets MJSM outperforms both of the other algorithms, sometimes by an order of magnitude. Here, we take a closer look at its behavior on queries where it *does* return the maximum matching.

First we report the running times on a query consisting of two equalities in Figure 9. The sizes of the two tables were 200,000, 1 million and 5 million, and the selectivity was kept at  $10^{-6}$ .

MJSM clearly outperforms the regular sort-merge join, and the difference is more marked as table sizes increase. The algorithms differ only in the merge phase, and it is not hard to see why MJSM dominates. When two input groups of size  $n$  each are read into the buffer pool during merging, the regular sort merge examines each tuple in the right group once for each tuple in the left group, resulting in  $n^2$  comparisons, while MJSM examines each tuple at most once. For a fixed selectivity, the size of a group increases in proportion to the size of the relation, so the differences are more marked for larger tables. While not shown here, we observed similar trends in the reverse scenario in which the table sizes are fixed but selectivities are varied, as MJSM examines each tuple only once in the merge phase and is unaffected by selectivity; the performance of regular sort-merge join degrades as the selectivity increases, as it has to merge larger groups.

We now report on the performance of MJSM on inequality predicates (for sake of brevity, the extension to MJSM to handle two inequality predicates is referred to as “MJSM on 2 inequalities”). Recall from Section 5.2 that in the case of one inequality ( $R.a < S.a$ ), the merge phase of MJSM performs only a single pass through both tables. On two inequalities, tuples are carried over across groups, which can affect performance. Comparing MJSM on one vs. two inequalities on various table sizes (Figure 10) we notice the performance of MJSM on inequality joins scales well with size. In fact the performance on inequality joins is comparable to equality joins, as can be seen from the similarity of the trends in Figures 9 and 10. Another noteworthy aspect of the graph is that the difference in performance between single and double inequalities is insignificant. This is indeed the average performance of MJSM on two inequalities where not many tuples are carried over; a more in-depth performance study of MJSM on two inequalities is warranted and we leave it for future work.

We summarize with the following observations:

- MJMF outperforms MJNL (and the full-join) for all but the smallest of group sizes. In cases when the input graph to max flow is large (e.g., 500000), the performance of MJMF degrades to that of the full-join.
- MJMF can be applied to any match join predicate so it can be used as a general match join algorithm to compute the maximum matching.
- MJSM is faster than the other algorithms, so it is always a good option for match predicates over which it can be guaranteed to produce maximum matches, or in cases where an approximate match (that is, a non-maximum match) is acceptable.

## 6.2 Validation on a Grid dataset

Here we apply our three match join algorithms to a real world dataset obtained from the Condor job scheduling system [19]. Condor currently runs on 1009 machines in the UW-Madison Computer Science pool, and at the time we gathered data, there were 4739 outstanding jobs (submitted but not completed). Every job submitted in this system goes through a resource allocation process, which occurs at least once every five minutes. In each allocation cycle, the requirements of a job are matched to the specifications of an available machine. A machine can run at

most one job and a job is run on at most one machine, so what we desire is a matching.

Machines and jobs in Condor have a large number of attributes and can be added dynamically. We chose a representative subset of those in our schema:

Jobs( wantopsys varchar, wantarch varchar, diskusage int, imagesize int)	Machines( ophys varchar, arch varchar, disk int, memory int)
---	---

The queries we ran on the dataset contained match predicates consisting of i) 2 equalities, ii) 1 equality + 1 inequality, and iii) 2 inequalities. The corresponding queries were:

i) Match predicate consists of two equalities:

```
SELECT *
FROM Jobs J, Machines M
WHERE J.wantopsys = M.opsys AND
      J.wantarch = M.arch
```

ii) Match predicate consists of one equality and one inequality:

```
SELECT *
FROM Jobs J, Machines M
WHERE J.wantopsys = M.opsys AND
      M.disk > J.diskusage
```

iii) Match predicate consists of two inequalities:

```
SELECT *
FROM Jobs J, Machines M
WHERE M.memory > J.imagesize AND
      M.disk > J.diskusage
```

We present the time taken to compute the full join for comparison - for computing the full join, Predator’s optimizer chose sort-merge for the first two queries and page nested loops for the third.

Figure 11 shows the results of the experiment. Firstly, note that all three match join algorithms outperform the full join by factors of 10 to 20; MJSM and MJMF take less than a second in all cases. Also, the response time of the match join algorithms stay fairly constant across all queries. In the case of MJSM, this is consistent with its behavior observed on the synthetic datasets. MJMF’s fast response times can be explained by the fact that group sizes for machines are quite large; in fact, for all the queries, the number of groups in the machines table was no more than 30 and frequently under 10. This is expected since there are relatively few distinct machine configurations. In addition, both MJMF and MJSM result in maximum matches for all queries; MJNL, on the other hand, is an approximate but more general algorithm that takes longer than the other two but still fares better than the full join. This shows that a match join is indeed a favorable alternative to computing the full join in many cases. This will become even more important in the future as Condor is expected to be deployed in configurations up to two orders of magnitude larger than the one from which we gathered data. Condor currently does not store its data in a DBMS, although the Condor team is exploring that option for future versions of the system.

## 7. RPJs: MATCH JOIN IN CONTEXT

As we mentioned in the introduction, the match join we consider in this paper is a simple example of a broad class of problems in

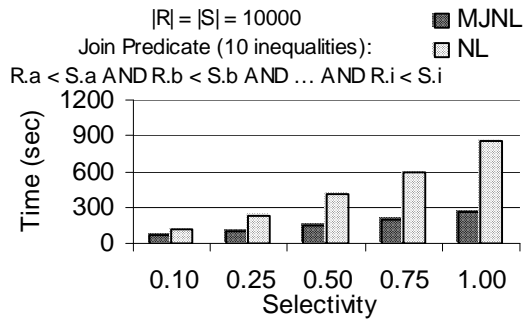


Figure 6. MJNL on varying join selectivity.

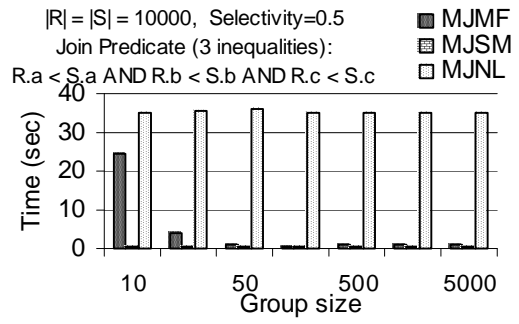


Figure 7. MJMF on varying group sizes.

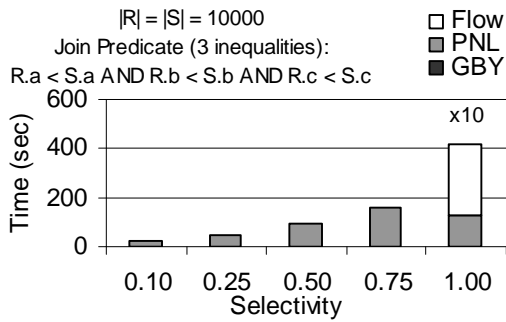


Figure 8. Various stages of MJMF.

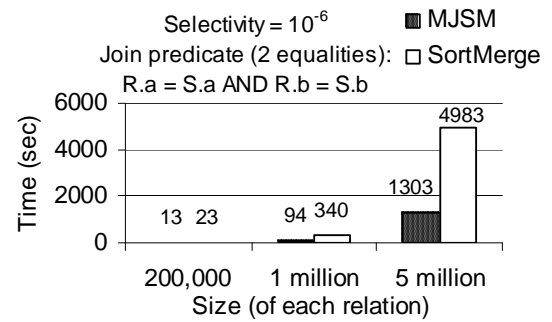


Figure 9. MJSM on equalities.

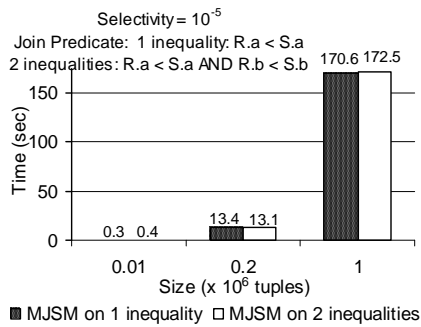


Figure 10. MJSM on 1 vs. 2 inequalities.

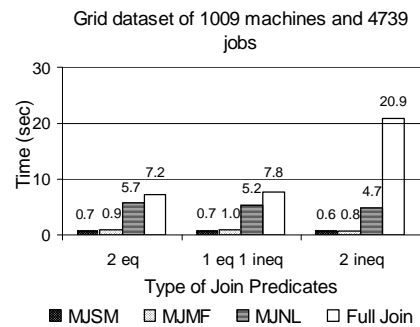


Figure 11. Validation on Condor dataset.

which we want a subset of a join, and the desired subset is a global property of the subset. In this section we put the match join into context by considering a generalization of the match join, which we term the Ranked Partial Join.

**Definition 8 (Ranked Partial Join)** *A ranked partial join on relations  $R$  and  $S$  using join predicate  $\theta$  and weight function  $F$ , denoted by  $RPJ(R,S,\theta,F,b) \subseteq R \bowtie_{\theta} S$  satisfies the following two criteria:*

a) **Degree criterion:** *Each tuple  $r \in R$  can be matched with at most  $b(r)$  tuples from  $S$ . Similarly, each tuple  $s \in S$  can be matched with at most  $b(s)$  tuples from  $R$ .*

b) **Quality criterion:** *Of the exponentially many subsets of  $R \bowtie_{\theta} S$ ,  $RPJ(R,S,\theta,F,b)$  is the one with the “best” quality, where the quality of a subset is defined by the total sum of the weights (as defined by  $F$ ) of all the edges in the subset.*

Depending on the application, “best” can be the maximization or minimization of the sum of edge weights. Typically  $F$  is monotonic [8, 12, 20] and is computed using attributes of  $R$  and  $S$  [10, 12, 20]. Instances of RPJ have been studied extensively in the operations research and theory community. Specifically, RPJ is an implicit version of the weighted b-matching problem. [3] contains references to several real-world examples of weighted b-matching. By “implicit version” here we mean that, as was the case for match join, in the RPJ the input graph is represented by a pair of relations and a predicate that says for each pair of tuples  $r \in R$  and  $s \in S$  if  $(r,s)$  is an edge in the graph.

At least two special cases of the RPJ have been studied in the database community. Firstly, if we relax the degree constraint above, thus allowing tuples of  $R$  to match with any number of tuples from  $S$  and vice versa (i.e.  $b(r)$  and  $b(s) = \infty$  for all  $r \in R$ ,  $s \in S$ ), the top- $k$  relational join [12, 20] is an instance of RPJ with the added restriction that the desired result is of size  $k$  and is ordered by a monotonic function  $F$ . Another instance of RPJ arises in the context of data cleaning [10] in which the authors show how to merge the results of approximate match operations using a variant of a weighted matching problem. An admittedly small instance of the RPJ occurs in assigning papers to reviewers for a conference program committee – each reviewer should have perhaps about 15 papers to review, each paper should be reviewed by 3 reviewers, papers cannot be assigned to reviewers for which there are conflicts, and you should maximize the sum of the bids in the resulting collection of (reviewer, paper, bid) tuples.

It is not our contention that DBMSs should implement special purpose algorithms for RPJs or even instances of RPJs. Rather, as was our claim for match joins, we think it will be useful to come up with a “bag of tricks” that can exploit DBMS functionality to efficiently solve RPJ-like problems over RDBMS-resident data. This is an interesting area for future work. Our preliminary impression at this point is that the grouping-based approach exploited by our MJMF algorithm has counterparts for more general RPJ problems. Furthermore, an adaptation of the MJNL algorithm will always be applicable as a “catch all” algorithm that enumerates some subset of the join that respects the cardinality constraints on the tuples (the  $b(r)$  and  $b(s)$  constraints), although the quality of the subset in general cannot be guaranteed.

Obviously a great deal of scope for future work remains for both the ranked partial join and its instances, including the match join. For example, an interesting twist to these problems which we have not considered in this paper arises in scenarios where, instead of specifying a common match predicate for all entities, each entity specifies its own match predicate. Yet another interesting problem to consider is how these various match join algorithms including the ones we have proposed should be integrated in an RDBMS. The simplest approach would be to implement these algorithms as user defined functions without modifying the RDBMS engine – this approach makes the most sense if each variant of matching is only useful to a small subset of RDBMS users. If a commonly accepted abstraction of matching becomes accepted as a generally useful DBMS primitive, it may make more sense to implement this abstraction as a new operator, in which case it would be “exposed” to the system and could participate in query optimization. In this case interesting problems would arise with respect to what statistics are needed to choose among match join related algorithm alternatives.

## 8. RELATED WORK

Bipartite maximum cardinality matching is one of the oldest studied problems in graph theory [3, 4, 11]. Over the last decade, researchers have studied many variants of the original problem that work in parallel [7], approximate [6], and online settings [13], the latter being the case when the input(s) come in streaming order. Reference [3] contains many references to theoretical work in the area. Matching has also been used in the context of online search engines [15] to assign keywords to bids placed by various companies who wish to display ads relevant to the keyword.

On the other hand, researchers have explored using databases to solve well-studied combinatorial graph algorithms such as shortest-path without adding new operators to the query engine [1]. Our work, to the best of our knowledge, is the first to marry the two by exploring the use of database technology to compute matchings in general, and to do so without explicitly materializing the entire bipartite graph.

Match join is just one of many recently proposed novel operations seeking to enhance the functionality of relational data sources. Recently proposed query types include preference queries [2, 5], top- $k$  queries [8, 12, 20] and OPAC queries [9]. Both match joins and top- $k$  joins seek to compute a subset of the full-join without enumerating the full-join, but match joins differ from top- $k$  in that the quality of the result is a property of the entire subset, not of its constituent tuples. OPAC queries, on the other hand, are selections over single tables that are expressed as parameterized queries with linear programming constraints. Our work is similar to the OPAC work in that both involve modeling an optimization problem as a relational query and using RDBMS infrastructure to compute the answer, although the classes of queries considered and approaches employed are very different.

In the Condor system, jobs and machines are matched using classified advertisements represented in a semi-structured language [17]. In the case of multiple matches, both entities specify a rank function which is used to determine the best match out of the lot. In addition, jobs are matched in order of a system-

assigned priority, and the resulting matching need not be of maximum cardinality.

## 9. CONCLUSION

It is clear from our experiments that our proposed match join algorithms perform much better than performing a full join and then using the result as input to an existing graph matching algorithm. As more and more graph applications store their data sets in RDBMSs, and as these data sets grow in size, supporting some kind of matching over RDBMS-resident data will become increasingly attractive.

Clearly, however, most applications involving a match operation, as one can envision, are more complex than the simple maximum matching problem that we consider in this paper. We have focused on this problem because it is a simple example of a class of problems that “look like” a join but, to the best of our knowledge, have not yet been explored in the context of relational database systems. These problems are interesting because they require the computation of a subset of a full join and the “quality” of the subset returned is a global property of the subset rather than a property of the individual tuples in the subset. Our results show that at least in the restricted case we consider, relational database technology can effectively be applied to such problems. This is encouraging, because if this were not the case, when faced with such problems, relational database systems would be relegated to serving only as heavy-weight file systems, storing data that is input to other programs without exploiting any of the query machinery built in to the system.

## 10. ACKNOWLEDGEMENTS

This work was supported in part by Department of Energy Award DE-FC02-01ER25458 and National Science Foundation Award SCI-0515491.

We are grateful to Robert Meyer and three anonymous reviewers for helpful feedback on various drafts of this paper.

## 11. REFERENCES

- [1] R. Agrawal and H.V. Jagadish. “Materialization and Incremental Update of Path Information”, *Proceedings of ICDE 1989*, p.374-383.
- [2] R. Agrawal and E. Wimmers. “A Framework for Expressing and Combining Preferences”, *Proceedings of ACM SIGMOD 2000*, p. 297-306.
- [3] R. K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.
- [4] C. Berge, “Two Theorems in Graph Theory” *Proceedings of the National Academy of Science USA*, 1957, p. 842-844.
- [5] Y. Chang et al. “The Onion Technique: Indexing for Linear Optimization Queries”, *Proceedings of ACM SIGMOD 2000*, p. 391-402.
- [6] J. Feigenbaum et al., “On Graph Problems in a Semi-Streaming Model”, *Proceedings of ICALP 2004*, p. 531-543
- [7] A.V. Goldberg and R. E. Tarjan. “A new approach to the maximum-flow problem”, *Journal of the ACM (JACM)*, v.35 n.4, Oct 1988, p. 921-940.
- [8] L. Gravano and S. Chaudhuri. “Evaluating Top-k Selection Queries”, *Proceedings of VLDB 1999*, p. 397-410
- [9] S. Guha et al. “Efficient Approximation of Optimization Queries Under Parametric Aggregation Constraints”, *Proceedings of VLDB 2003*, p. 778-789
- [10] S. Guha et al. “Merging the Results of Approximate Match Operations”, *Proceedings of VLDB 2004*, p. 636-647.
- [11] J. Hopcroft and R. Karp. “An  $n^{5/2}$  Algorithm for Maximum Matching in Bipartite Graphs”, *SIAM Journal of Computing*, 1975, p. 225-231.
- [12] I. Ilyas et al. “Supporting Top-k Join Queries in Relational Databases”, *VLDB Journal*, v.13 n.3, p. 207-221.
- [13] R. Karp, U.V. Vazirani, and V.V. Vazirani. “An optimal algorithm for online bipartite matching”, *Proceedings of ACM STOC 1990*, p. 352-358.
- [14] J. Magun. “Greedy Matching Algorithms: An experimental study”, *Proceedings of the 1<sup>st</sup> Workshop on Algorithm Engineering*, p. 22-31, 1997.
- [15] A. Mehta et al. “AdWords and Generalized Online Matching”, *Proceedings of IEEE FOCS 2005*, p. 264-273.
- [16] Predator Project. <http://www.distlab.dk/predator/>
- [17] R. Raman et al. “Matchmaking: Distributed Resource Management for High Throughput Computing”, *Proceedings of IEEE HPDC 1998*, p. 140-146.
- [18] C. Schlup. “Automatic Game Matching”, [http://dcs.ethz.ch/theses/ws0203/OnlineMatching\\_abstract.pdf](http://dcs.ethz.ch/theses/ws0203/OnlineMatching_abstract.pdf)
- [19] T. Tannenbaum et al. “Condor – A Distributed Job Scheduler”, *Beowulf Cluster Computing with Linux*, The MIT Press, 2002.
- [20] P. Tsaparas et al. “Ranked Join Indices”, *Proceedings of IEEE ICDE 2003*, p. 277-288.
- [21] <http://www.research.microsoft.com/mlp/trueskill/default.aspx>