Database Support for Matching: Limitations and Opportunities

Ameet M. Kini, Srinath Shankar, David J. Dewitt, and Jeffrey F. Naughton

Technical Report (TR 1545)
University of Wisconsin-Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706, USA
{akini, srinath, dewitt, naughton}@cs.wisc.edu

Abstract. A match join of R and S with predicate theta is 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 arise in many scenarios, including one that was our original motivation, assigning jobs to processors in the Condor distributed job scheduling system. We explore the use of RDBMS technology to compute match joins. We show that the simplest 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 beyond merely serving as passive storage for external 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 or enhance the query capabilities of the RDBMS to support these applications. Some recent examples of this include OPAC queries [8], preference queries [1,4], and top-k selection [7] and join queries [10,13,17]. 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 appear 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.

We first became interested in the matching problem in the context of the Condor distributed job scheduling system [16]. There, the RDBMS is used to store information on jobs to be run and machines that can (potentially) run the jobs. Then a matching operation can be done to assign jobs to machines. Instances of matching problems are ubiquitous across many industries, arising whenever it is necessary to allocate resources to consumers. In general, these matching problems place complex conditions on the desired match, and a great deal of research has been done on algorithms for computing such matches (the field of job-shop scheduling is an example of this). Our goal in this paper is not to subsume all of this research – our goal is much less ambitious: to take a first small 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 θ. 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 match over database-resident data would be to exploit the existing graph matching algorithms developed by the theory community over the years. This could be accomplished by first computing the θ-join (the usual relational algebraic join) of the two tables, with θ as the match predicate. This would materialize a bipartite graph that could be used as input to any graph 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* "matches" many rows in *S*, and the join will be a large fraction of the cross product).

Accordingly, in this paper we explore alternate optimal and approximate strategies of using an RDBMS to compute the *maximum cardinality matching* of relations *R* and *S* with match join predicate θ. If nothing is known about θ, 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 θ, faster algorithms are possible. We propose two such algorithms. The first, which we term MJMF (Match Join Max Flow), requires knowledge of which match join attributes form 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 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.

We have implemented all three algorithms in the Predator RDBMS [14] and report on experiments with the result. Our experience shows that these algorithms lend themselves well to a RDBMS implementation as they use existing DBMS 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 implementation in Predator and experimental results. Related work is discussed in Section 7. Finally, we conclude and discuss future work in Section 8.

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 θ. Here, the rows of *R* and *S* represent the nodes of the graph and the predicate θ is used to implicitly denote edges in the graph. The relational join $R \bowtie_\theta S$ then computes the complete edge set that would be the input to a classical graph matching algorithm.

Definition 1 (Match join) *Let $M = Match(R,S,\theta)$. Then M is a matching or a match join of R and S iff $M \subseteq R \bowtie_\theta S$ and 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 respectively.*

Definition 2 (Maximal Matching) *A matching $M'=Maximal\text{-}Match(R,S,\theta)$ if $\forall r \in R\text{-}M'(R), s \in S\text{-}M'(S), (r,s) \notin R \bowtie_\theta S$. Informally, M' cannot be expanded by just adding edges.*

Definition 3 (Maximum Matching) *Let M^* be the set of all matchings $M=Match(R,S,\theta)$. Then $MM=Maximum\text{-}Match(R,S,\theta)$ if $MM \in M^*$ of the largest cardinality.*

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. Approximate 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 $O(n^{2.5})$ [9], 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 [12] for a survey on the topic. However, they still require as input the entire graph. Specifically, [5] gives a $(2/3 - \varepsilon)$-approximation algorithm ($0 < \varepsilon < 1/3$) that makes multiple passes over the set of edges in the underlying graph. As a result of these drawbacks, the above approach will not be successful for large problem instances, and we need to search for better approaches.

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 an *(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 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 [2] 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 match returned by MJNL. Then, M is maximal.*

Proof: MJNL works by obtaining the first available matching node *s* for each and every node *r*. 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, or in other words, *M* is maximal □

Theorem 1 *Let MM = Maximum-Match(R,S,θ) where θ is an arbitrary match join predicate. 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 [2] 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. As such, 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

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

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

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

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 as a Max Flow problem

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 Condor application, 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 [2]:

Definition 4 (Max Flow Problem) *Consider a capacitated network G = (N,A) with a nonnegative capacity u_{ij} associated with each edge (i,j) $\in$ A. 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:

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

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 *Match(R,S,θ)*, we first construct a directed bipartite graph *G* = (*N1* $\cup$ *N2*, *E*) where a) nodes in *N1* (*N2*) represent tuples in *R* (*S*), b) all edges in *E* point from the nodes in *N1* to nodes in *N2*. We then introduce a source node *s* and a sink node *t*, with an edge connecting *s* to each node in *N1* and an edge connecting each node in *N2* 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 $m=|A|$ and $n=|N|$) [2]. 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* =

($N1 \cup N2$, E) such as the one shown in Figure 1(b). Now we construct a new network $G' = (N1' \cup N2', E')$ with source node s' and sink node t' as follows:

1. (Build new node set) we add a node $n1' \in N1'$ for every group of nodes in $N1$ which have the same value on the match join attributes; similarly for $N2'$.
2. (Build new edge set) we add an edge between $n1'$ and $n2'$ if there was an edge between the original two groups which they represent.
3. (Connecting new nodes to source and sink) We add an edge between s' and $n1'$, and between $n2'$ and t'.
4. (Assign new edge capacities) For edges of the form (s', $n1'$) the capacity is set to the size of the group represented by $n1'$. Similarly, the capacity on ($n2'$, t') is set to the size of the group represented by $n2'$. Finally, the capacity on edges of the form ($n1'$, $n2'$) 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 shown 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 ($n1'$, $n2'$). Let this flow value be f. We can then match f members of $n1'$ to f members of $n2'$. Due to the capacity constraint on edge ($n1'$, $n2'$), we know that $f \leq$ the minimum of the sizes of the two groups represented by $n1'$ and $n2'$. 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: See [2] 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 M in G corresponds to a flow f' in G', and b) every flow f' in G' corresponds to a matching M in G. Here, by "corresponds to", we imply that the size of the matching and the value of the flow are equal. First, b) by the flow decomposition theorem [2], the total flow f' can be decomposed into a set of path flows 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'. Due to the capacity constraints, the flow on edge (i_1, i_2), say, φ = min(flow(s, i_1), flow(i_2, t)). As such, we can add φ edges of the form (i_1, i_2) to the final matching M in G. Since we do this for every edge of G' of the form (i_1, i_2) that is part of a path flow, the size of M corresponds to the value of flow f'. a) The correspondence between a matching in G and a flow f in a unit capacity network is shown in [2]. Going from f to f' on G' is simple. Take each edge of the form (s, i_1) in G'. Here, recall that i_1 is a node in G' and it represents a set of nodes in G; we refer to this set as the i_1 group and the members of the set as the members of the i_1 group. For each edge of the form (s, i_1) in G', set its flow to the number of members of the i_1 group that are matched in G. 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, flows on edges of the form (i_1, i_2) are guaranteed to be within their capacities. Now, since f' is the sum of the flows on edges of the form (s, i_1) in G', every matched edge of G contributes a unit to f'. As such, the value of f' represents the size of the matching in G.

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 θ-join of the two outputs of the group-by operators where θ 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). 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

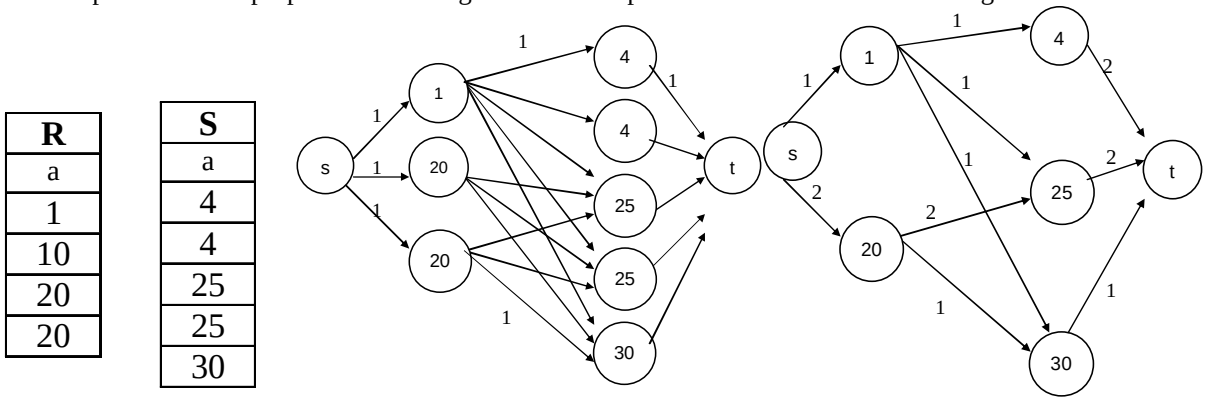

Fig 1: A 3-step transformation from (a) Base tables to (b) A unit capacity network to finally (c) A reduced network that is input to the max flow algorithm

$O(n^3)$ [6]. One such implementation can be encapsulated inside a UDF which first does the above transformation to a reduced graph, expressed in SQL as follows:

Tables: `R(a int, b int), S(a int, b int)`
Match Join Predicate: `θ(R.a, S.a, R.b, S.b)`
SQL for 3-step transformation to reduced graph:

```
SELECT *
FROM((SELECT count(*) AS group_size,
                  max(R.a) AS a1, max(R.b) AS b1
                  FROM R
                  GROUP BY R.a,R.b) AS T1,
                       (SELECT count(*) AS group_size,
                                max(S.a) AS a2, max(S.b) AS b2
                  FROM S
                  GROUP BY S.a,S.b) AS T2))
WHERE θ(T1.a, T2.a, T1.b, T2.b);
```

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 more efficient if there are many groups, although it requires more knowledge about the match predicates if it is to be optimal.

5. Match Join Sort-Merge

The intuition behind MJSM is that by exploiting the semantics of the match join predicate θ, 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 θ 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.)

Although MJSM always returns a match, as we later show (see Lemma 2 below), MJSM is only guaranteed to be optimal (returning a maximum match) if the match join predicate possesses certain properties. An example of a class of match predicates for which MJSM is optimal is when the predicate consists of the conjunction of zero or more equalities and at most two inequalities ('<' or '>'), and we focus on MJSM's behavior on this class of predicates for the remainder of this section.

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 θ. $R \bowtie_\theta S$ is, as usual, the relational θ join of *R* and *S*. In this section, unless otherwise specified, θ is a conjunction of *p* predicates of the form $R.a_1\ op_1\ S.a_1\ AND\ R.a_2\ op_2\ S.a_2\ AND, \ldots, AND\ R.a_{p-1}\ op_{p-1}\ S.a_{p-1}\ AND\ R.a_p\ op_p\ S.a_p$, where op_1 through op_{p-2} are equality predicates, and op_{p-1} and op_p are either equality or inequality predicates. Without loss of generality, let < be the only inequality operator. Finally, let *k* denote the number of equality predicates ($k \geq 0$).

MJSM computes the match join of the two relations by first dividing up the relations into *groups* of candidate matching tuples and then computing a match join within each group. The groups used by MJSM are defined as follows:

Definition 5 (Groups) *A group $G \subseteq R \bowtie_\theta S$ such that:*

1. *$\forall r \in G(R), s \in G(S), r(a_1) = s(a_1), r(a_2) = s(a_2), \ldots, r(a_k) = s(a_k)$ thus satisfying the equality predicates on attributes a_1 through a_k. If k=p-1, then θ consists of at most one inequality predicate, $R.a_p < S.a_p$*
2. *However, if k=p-2, then both $R.a_{p-1} < S.a_{p-1}$ and $R.a_p < S.a_p$ are inequality predicates. Then:*
   a) $\forall r \in G(R), s \in G(S), r(a_{p-1}) < s(a_{p-1})$ thus satisfying the inequality predicate on attribute a_{p-1} and
   b) $\forall r \in G(R), s \in G'(S)$ where G' precedes G in sorted order, $r(a_p) \geq s(a_p)$ thus not satisfying the inequality predicate on attribute a_p .

We refer to G(R) (similarly, G(S)) to refer to the R-tuples (S-tuples) in G. Also, either G(R) or G(S) can be empty but not both. Figure 2 shows an example of how groups are constructed from underlying tables.

Note that groups here in the context of MJSM are not the same as the groups in the context of MJMF because of property 2 above.

Next we define something called a "zig-zag", which is useful in determining when MJSM returns a maximum matching.

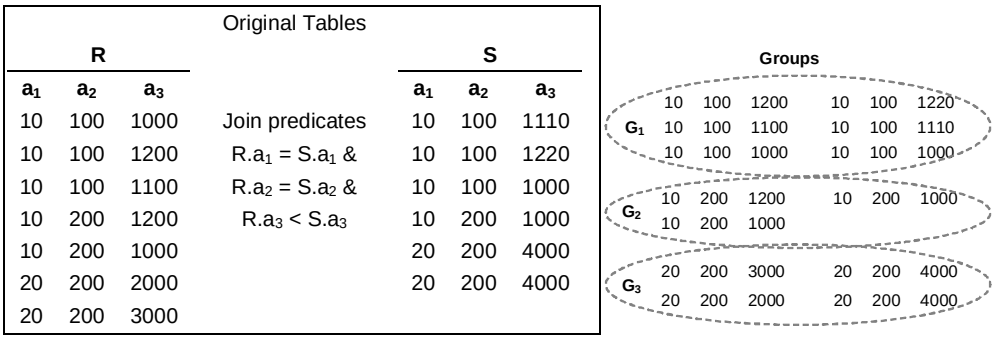

Fig 2 Construction of groups

Definition 6 (Zig-zags) *Consider the class of matching algorithms that work by enumerating (a subset of) the elements of the cross product of 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'.*

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.

Definition 7 (Spill-overs) *MJSM works by reading groups of tuples (as in Definition 5) and finding matches within each group. We say that a tuple $r \in G(R)$ is a spill-over if a match is not found for r in G(S) (either because no matching G(S) tuple exists or if the only matching tuples in G(S) are already matched with some other G(R) tuple) and there is a G', not yet read, such that G and G' match on all k equality predicates. In this case, r is carried over to G' for another round of matching.*

5.1 Algorithm Overview

Figure 3 shows the sketch of MJSM and its subroutine MatchJoinGroups. We describe the main steps of the algorithm:

1. Perform an external sort of both input relations on all attributes involved in θ.
2. Iterate through the relations and generate a group *G* (using GetNextGroup) of *R* and *S* tuples. *G* satisfies Definition 5, so all tuples in *G(R)* match with *G(S)* on all equality predicates, if any; further, if there are two inequality predicates, they all match on the first, and *G* is sorted in descending order of the second.
3. Call MatchJoinGroups to compute a maximum matching *MM* within *G*. Any *r* tuples within *G(R)* but not in *MM(R)* are spill-overs and are carried over to the next group.
4. *MM* is added to the global maximum match. Go to 2.

Figure 4 illustrates the operation of MJSM when the match join predicate is a conjunction of one equality and two inequalities. Matched tuples are indicated by solid arrows. GetNextGroup divides the original tables into groups which are sorted in descending order of the second inequality. Within a group, MatchJoinGroups runs down the two lists outputting matches as it finds them. Tuple <Intel, 1.5, 30> is a spill-over so it is carried over to the next group where it is matched.

As mentioned before, unless otherwise specified, in the description of our algorithm and in our proofs, we assume that the input predicates are a) a conjunction of k ($k \geq 0$) equalities and at most 2 inequalities. The rest of the predicates can be applied on the fly. Also, b) note that both inequality predicates are 'less-than' (i.e., $R.a_i < S.a_i$); the algorithm can be trivially extended to handle all combinations of < and > inequalities by switching operands and sort orders.

```
Algorithm MatchJoinSortMerge
Input: Tables R(a1,a2,…,ap,ap+1,…,am),
S(a1,a2,…,ap,ap+1,…,an) and a join predicate
consisting of k equalities R.a1 = S.a1, R.a2 =
S.a2,…, R.ak = S.ak and up to 2 inequalities
R.ap-1 < S.ap-1, R.ap < S.ap
Output: Match
Body:

Sort R and S in ascending order of <a1,a2,…,ap>;
Match = {};
curGroup = GetNextGroup({});
//keep reading groups and matching within them
while curGroup ≠ {}
  curMatch = MatchJoinGroups(curGroup, k, p);
  Match = Match U curMatch;
  nextGroup = GetNextGroup(curGroup);
  //either nextGroup is empty or curGroup and
  //nextGroup differ on equality predicates
  if nextGroup = {} OR (both groups differ on
  any a1,a2,…,ak)
    curGroup = nextGroup;
    continue;
  else
    // select R tuples that weren't matched
    spilloverRtuples = (curGroup(R) –
                        curMatch(R));
    // merge spillover R tuples with next group
    nextGroup(R) = Merge(spilloverRtuples,
                         nextGroup(R));
   curGroup = nextGroup;
  end if
end while
return Match
```

```
Subroutine MatchJoinGroups
Input: Group G, p = # of predicates
and k = # of equality predicates
Output: Match
Body:

Match = {};
//if there are no inequalities
if k = p
  r = next(G(R)); s = next(G(S));
  while neither r nor s are null do
    Match = Match U (r,s);
    r = next(G(R)); s = next(G(S));
  end while
  //else if there is at least one
  //inequality
else if k < p
  r = next(G(R)); s = next(G(S));
  //find tuples that satisfy
  //inequality predicate
  while neither r nor s are null do
    if r(ak+1) < s(ak+1)
      Match = Match U (r,s);
      r = next(G(R));
      s = next(G(S));
    else if r(ak+1) = s(ak+1)
      r = next(G(R));
    end if
  end while
end if
return Match
```

Figure 3: The MJSM Algorithm

5.2 When does MJSM return Maximum-Match(R,S,θ)?

The general intuition behind MJSM is the following: If θ consists of only equality predicates, then matches can only be found within a group. A greedy pass through both lists (*G(R)* and *G(S)*) within a group retrieves the maximum match. As it turns out, the presence of one inequality can be dealt with a similar greedy single pass through both lists. The situation is more involved, however, when there are two inequalities present in the join predicates.

We now characterize the family of match join predicates θ for which MJSM can produce the maximum matching and outline a proof of the specific case when θ consists of k equality at most 2 inequality predicates. We first state the following lemma:

Lemma 2 *Let M be the result of a matching algorithm A, i.e, M=Match(R,S,θ) where θ consists of arbitrary join predicates. If M is maximal and A never encounters zig-zags, then M is also maximum.*

The proof uses a theorem due to Berge [3] that relates the size of a matching to the presence of an augmenting path, defined as follows:

Definition 8 (Augmenting path) *Given a matching M on graph G, an augmenting path through M in G 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 through 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 $r \in R\text{-}M(R)$, $s \in R\text{-}M(S)$ and edge $(r,s) \in R \bowtie_\theta S$, 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 φ consisting of nodes $r_{\varphi-1}, \ldots, r_1$ and $s_{\varphi-1}, \ldots, s_1$:

$$r_{\varphi-1} \rightarrow s_{\varphi-1} \rightarrow r_{\varphi-2} \rightarrow s_{\varphi-2} \rightarrow \ldots \rightarrow r_1 \rightarrow s_1$$

By definition of an augmenting path, both $r_{\varphi-1}$ 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 $(r_{\varphi-1},s_{\varphi-1})$, $(r_{\varphi-2},s_{\varphi-2})$, …, (r_2,s_2), (r_1,s_1) are not in M whereas edges $(s_{\varphi-1},r_{\varphi-2})$, $(s_{\varphi-2},s_{\varphi-3})$, …, (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 $r_{\varphi-1}$ can be matched with s_1. This is a contradiction, since we assumed that M is maximal.

Theorem 4 *Let $M=MJSM(R,S,\theta)$. Then, if θ is a conjunction of k equality predicates and up to 2 inequality predicates, M is maximum.*

Proof: Our proof is structured as follows: We first prove that M is maximal. Then we prove that MJSM avoids zig-zags, thus using Lemma 2 to finally prove that M is maximum.

Why is M maximal? An $r \in G(R)$, for some group G, is considered a spill-over only if it cannot find a match in $G(S)$. Hence, within a group, MatchJoinGroups guarantees a maximal match. At the end of MJSM, all unmatched R tuples are accumulated in the last group, and we have $\forall r \in R\text{-}M(R), s \in S\text{-}M(S), (r,s) \notin R \bowtie_\theta S$. As such, M is maximal.

Now, why does MJSM and its subroutine MatchJoinGroups avoid zig-zags? Let the input to MatchJoinGroups be group G. Now our join predicates can consist of i) zero of more equalities, and either ii) exactly one inequality or iii) exactly two inequalities. We show that in all three cases, MatchJoinGroups avoids zig-zags. First recall that within a group, any $G(R)$ tuple matches with any $G(S)$ tuple on any equality predicates by Definition 5. Also recall that in the presence of 2 inequalities each group is internally sorted on the second inequality a_p. We have then 3 cases:

case i) If there are only equalities, then all r match with all s. Trivially, MatchJoinGroups avoids zig-zags and will simply return $min(|G(R)|, |G(S)|) = |Maximum\text{-}Match(G(R), G(S), \theta)|$.

case ii) If, in addition to some equalities, there is exactly one inequality, and if $r \in G(R)$ can be matched with $s' \in G(S)$, then $r' \in G(R)$ after r can also be matched with s' since, due to the decreasing sort order on a_p, $r'(a_p) < r(a_p) < s'(a_p)$.

case iii) If in addition to some equalities, if there are two inequality predicates a_{p-1} and a_p, then $\forall r \in G(R), s \in G(S), r(a_{p-1}) < s(a_{p-1})$ by the second condition in Definition 5. So, all r tuples match with all s tuples on any equality predicates and the first inequality predicate. MatchJoinGroups avoids zig-zags here for the same reason as case ii) above.

So within a group, MatchJoinGroups does not encounter any zig-zags, and the iterator on R can be confidently moved as soon as a non-matching S tuple is encountered. In addition, we've already proven that MatchJoinGroups results in a maximal-match within G. Hence, by Lemma 2, MatchJoinGroups results in $Maximum\text{-}Match(G(R),G(S),\theta)$.

If, at the end of MatchJoinGroups, a tuple r' turns out to be a spill-over, we cannot discard it as it may match with a $s' \in G'(S)$ for a not-yet read group G' as $r'(a_{p-1}) < s'(a_{p-1})$. MJSM would then insert r in G'. Now, running MatchJoinGroups on G' before insertion of r would not have resulted any zig-zags, as proven above for G. After inserting r, G' is still sorted in decreasing order of the last inequality predicate a_p. So, by above reasoning for G, running MatchJoinGroups on G' after inserting r would not result in zig-zags either. Hence, by Lemma 2, MJSM results in $Maximum\text{-}Match(R,S,\theta)$ □

Note that according to Lemma 2, MJSM's optimality can encompass 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 two inequalities, MJSM uses sorting to obtain its groups and avoid zig-zags, a technique that does not generalize to arbitrary predicates. We illustrate the case when θ consists of three 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 <1,105,47> is matched with <10,111,50>, <9,110,42> is a spill-over and is copied 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 descending 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. As such, 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, as such, both and interesting and challenging area of future research.

5.3 Complexity and an optimization

We now present a cost-based analysis of the performance of MJSM. As we will see, the running time depends strongly on the percentage of spill-over tuples from each group. We first present the CPU and I/O costs under the assumption that a constant percentage *P* of each group spill-over, i.e., the number of tuples carried over into each group from the previous is constant. Then we revise the analysis after relaxing this assumption. Finally, we describe ways to improve the running time for a large subset of join predicates.

Let *M* and *N* be total # of pages of *R* and *S* respectively. Also, let $m = |R|$ and $n = |S|$ and assume that $m > n$. We first analyze the running time in terms of the CPU utilization and then measure the I/O usage.

Let the # of groups be *k* and a group, on average, be of size *m/k*. First, both *R* and *S* are sorted in increasing order of all join attributes. The cost of this operation is $O(m \log m)$. Then, as groups are read in, they are first sorted in descending order and then merged together. This is an in-memory sort plus traversal of both lists once, which is $O(m/k * \log (m/k))$. The running time of MJSM over *k* groups is then

$$\sum_{i=1}^{k}(1+P)*\frac{m}{k}*\log\left((1+P)*\frac{m}{k}\right) = O(m*\log m)$$

If the number of spill-over tuples is not constant, however, then the running time of the algorithm varies significantly. In the worse case, every group completely spills over to the next; the size of the last group then becomes $|R|$. So, instead of $(1+P)$ above, the multiplicative factor increases linearly. However, we can exploit the fact that each group of spill-over tuples is itself sorted, and thus, instead of resorting it in the next group, we simply merge the two sorted groups together in linear time. The running time is now

$$\sum_{i=1}^{k}\left((i-1)*\frac{m}{k}+\frac{m}{k}\right)+\sum_{i=1}^{k}\left(\frac{m}{k}\log\left(\frac{m}{k}\right)\right)$$

$$=\sum_{i=1}^{k}(i-1)*\frac{m}{k}+\sum_{i=1}^{k}\frac{m}{k}+O(m\log m)$$

$$=\left(k*\frac{k-1}{2}\right)+m+O(m*\log m)$$

$$=O(k^2+m\log m)$$

The left summation represents the cost of merging the spill-over tuples with the new group. Since we assume that every group spills over to the next, by the time we reach the *k*th group, we would be merging two groups of size $(k-1)*m/k$ and m/k. The right summation represents the cost of sorting the new group.

In the best case, when $k = 1$, we have only 1 group, and the running time is $O(m \log m)$. This corresponds to the case of constant spill-over tuples with $P = 0$. In the worst case, when $k = m$, each group consists of just one tuple, and every group spills onto the next, effectively the running time of MJSM is $O(m^2)$. The analysis above only takes into account the operations performed on *R*. However, the work performed on *S* is dominated by sorting its groups as *S* tuples do not spill over.

The I/O costs of MJSM depend on the number of spill-over tuples. If each group, after the addition of any spill-over tuples, fits into memory, the I/O costs for MJSM, not including the cost of writing out the result, is $(2M+2N) + (M+N) = 3M + 3N$. Here the sum inside the left pair of parentheses corresponds to the cost of externally sorting *R* and *S*, whereas the second sum corresponds to the cost of reading each

group once; MJSM then returns the maximum match by doing three passes through both relations. However, if, after spill-overs, a group cannot fit in memory, then we would need to externally merge the spill-over tuples with the new group, similar to merging in the sort-merge algorithm.

The above CPU and I/O costs for MJSM can be significantly improved if the join predicates consist of *k* equality predicates *(k ≥ 0)* and at most one inequality predicate. This is because in the presence of at most one inequality, it suffices to sort the two relations in either ascending or descending order and go down the two lists, retrieving matches in a greedy fashion. Since we already sort the relations at the beginning of the algorithm, GetNextGroup can avoid re-sorting the group again in descending order. In addition to reduced CPU time, avoiding the second sort also saves considerable I/O in the case when groups do not fit in memory. Thus, the I/O cost for *k* equality predicates and at most one inequality is *3M+3N*. On the other hand, the CPU cost is similar to the best-case analysis of a sort-merge join = *O(max(m log m, n log n))*. Table 1 summarizes the CPU and I/O costs of MJSM under different conditions.

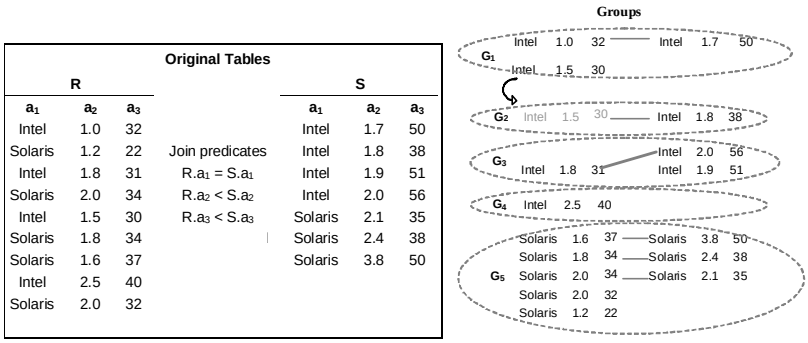

Fig 4: Illustration of MJSM on sample input

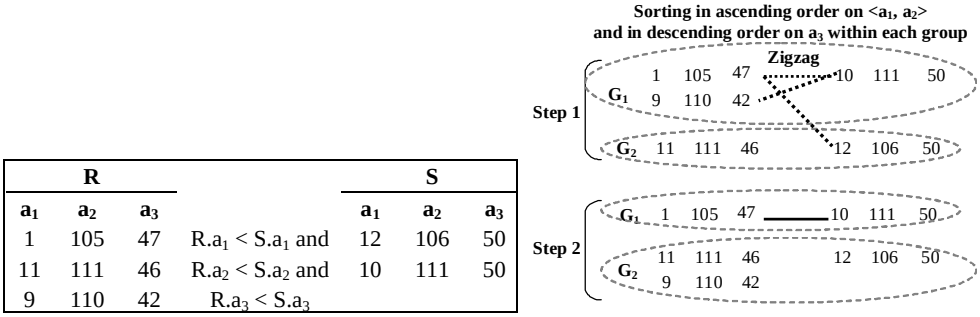

Fig 5: MJSM on 3 inequalities - prone to zig-zags

Table 1: CPU and I/O costs of MJSM

	Groups fit in-memory	Groups do not fit in memory
k equalities + ≤ 1 inequality	*O(m log m)* / *3M + 3N*	O(*m* log *m*) / *3M + 3N*
k equalities + 2 inequalities	*O(m²)* / *3M + 3N*	*O(m²)* / *> 3M + 3N*

6. Experiments

We implemented our algorithms in the open source object relational database Predator [14], 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.

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 join in the RDBMS, then feed the result to any well-known bipartite matching algorithm, such as the one presented in [9]. For this reason, the time to compute the full join serves as a lower bound on the time for this approach. 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 system.

In order to carefully control various data characteristics such as selectivity and group size, we generated synthetic datasets using two separate techniques. For the experiments where we study the effects of data sizes or join selectivities (those in Sections 6.1 and 6.3 below), the data was generated using the following technique: Values for all attributes which appear in the match-join predicate were independently selected using a uniform distribution from a range selected to yield the desired selectivity. First we explain the case for equality predicates ($R.a = S.a$ and $R.b = S.b$). Given any two discrete uniformly distributed random variables in the range $[1\ldots n]$, the chances that they are equal is $1/n$. Thus, choosing $R.a$ and $S.a$ from $[1\ldots n]$ and $R.b$ and $S.b$ from $[1\ldots m]$ gives a combined selectivity of $1/(n*m)$. For the inequality predicates ($R.a > S.a$ and $R.b > S.b$), both attributes in S were chosen uniformly from [1…1000] and $R.a$ and $R.b$ were chosen uniformly from $[1\ldots(2000*\sigma_a)]$ and $[1\ldots(2000*\sigma_b)]$ respectively for a combined selectivity of $\sigma_1*\sigma_2$. Data for the experiments in Section 6.2 where we study the effects of the group size on MJMF was generated using a different technique, and we defer its discussion to Section 6.2 below.

Unless explicitly mentioned, the schema of the input tables are *R(a int, b int, c int)* and *S(a int, b int, c int)*; each tuple is of 150 bytes. For sake of conciseness, the particular join predicates (equality, one inequality, etc.) and other parameters that vary in the experiments are reported in the figures themselves.

The full relational join would return *all* pairs of joining tuples, while the match join algorithms only return a subset. Obviously, 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 regular join, but as the results show, the match joins algorithms are still significantly superior.

6.1 Validating MJNL

Here, we show the performance of MJNL, comparing it to the full join for 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 maximal 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 MJNL does take longer (albeit it grows much slower than the full join) with larger selectivities. This is due to the construction time of the resulting matching which is larger as the selectivity increased.

6.2 Validating MJMF

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, $|G| = f^*((|Table_{left}| * |Table_{right}|)/g)$. As such, for a fixed selectivity, the larger the group size, the smaller the graph. Similarly, for a fixed group size, a low selectivity results in a small graph.

Since $f^*((|Table_{left}| * |Table_{right}|)/g) = (f^*(|Table_{left}| / g))^* |Table_{right}|$, we generate a dataset with the required variables f and g by making $(f^*(|Table_{left}| / g))$ groups of tuples from $Table_{left}$ join with *all* groups from $Table_{right}$ and the rest of the groups from $Table_{left}$ join with none in $Table_{right}$. Note that this "all-or-none" technique, while contrived for simplicity in data generation, does not, by itself, influence the behavior of MJMF which, as we mention above, is only dependent on $|G|$ and not in the manner G was generated.

Accordingly, using synthetic datasets, we conducted two 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. Group sizes range from 10 (low compression) to 5000 (high compression). Accordingly, $|G|$ ranges from 500000 to 2.

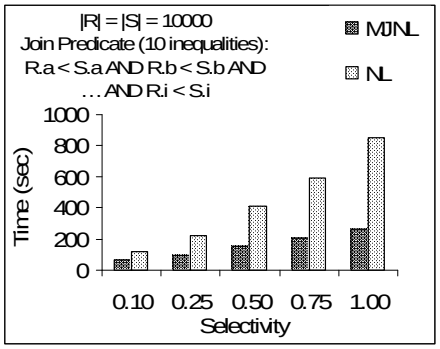

Figure 6: MJNL on varying join selectivities

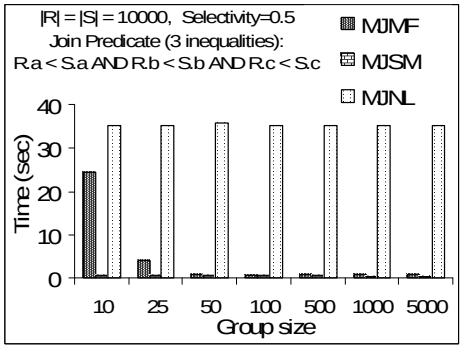

Figure 7: MJMF on varying group sizes.

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 gets higher as groups get smaller ($g \leq 25$) and G gets larger ($|G| \geq 80000$); eventually the performance of MJMF approaches that of MJNL. As an aside, the full relational join query took over 2 minutes in all the cases so we did not include its timings in the figure.

We also measured the times spent by MJMF in its three stages (Figure 8). Here, we varied f keeping g at a constant 8. As f increases from 0.1 to eventually 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 CPU bound max flow takes a fraction of the overall time and is dominated by the join operation (page nested loops, in this case). 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. We summarize with the following observations:

- MJMF outperforms MJNL (and the full-join) for all but the smallest of group sizes (Figure 7). When the input graph to max flow is large (e.g. 500000), MJMF's performance 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.

6.3 Validating MJSM

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 (recall that it always does so for match predicates with at most two inequalities.)

First we report the running times on a query consisting of two equalities (Figure 9). The sizes of the two tables were 200,000, 1 million and 5 million, and the selectivity was kept at 10^{-6}. We see that MJSM clearly outperforms the regular join, and the difference is more marked as table size increases. 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.

The difference between the two is more pronounced when we vary the selectivity for a fixed table size (Figure 10). The sizes of the input tables were fixed at 1 million tuples while the selectivities were 10^{-7}, 10^{-6} and 10^{-5}. Note that the performance of MJSM was largely unaffected by varying selectivity (similar to MJNL), since as mentioned above, it examines each tuple only once in the merge phase. The performance of regular sort-merge join worsened considerably as *f* increased, as this meant that individual group sizes in the merge phase increased.

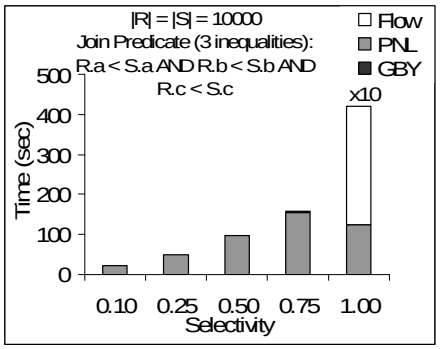

Figure 8: Various stages of MJMF

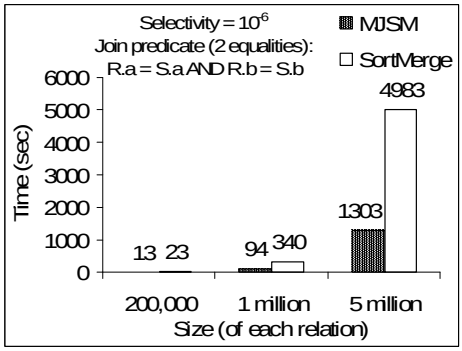

Figure 9: Equality predicates, varying table sizes

We now report on the performance of MJSM on inequality predicates. Recall from Section 5.3 that in the case of one inequality (R.a > S.a), the merge phase of MJSM is optimized and performs a single pass through both tables without causing any spill-overs. Here we first report on the time taken by MJSM on one inequality predicate (Figure 11) for selectivities of 0.01, 0.001 and 0.0001 at a fixed table size of 10000 tuples. Due to the inequality join predicate, the optimizer chose page nested loops for the regular join, and as such both algorithms were unaffected by selectivity. MJSM consistently outperforms the regular join by two orders of magnitude.

Comparing MJSM on one vs. two inequalities on various table sizes (Figure 12) 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 in the similarity of the trends in Figure 12 and Figure 9. This validates the average case analysis of $O(n \log n)$ in all cases. Another noteworthy aspect of the graph is that the difference in performance between single and double inequalities is insignificant. This is indeed the case when the number of spill-over tuples from each group is fairly constant as was shown in the cost analysis in Section 5.3. As an aside, while this is not in the graph, page nested loops took over an hour for the case when the input tables were each of size 1 million!

To summarize, 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.

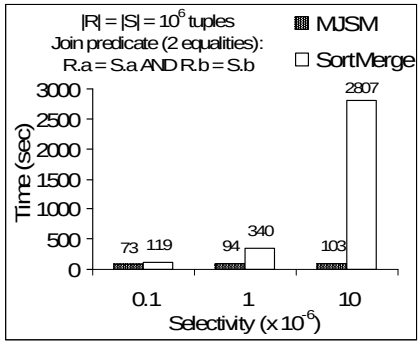

Figure 10: Equality predicates, varying selectivity

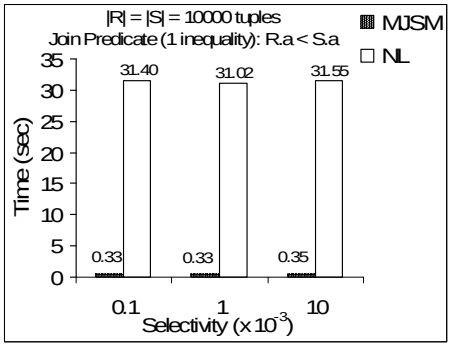

Figure 11: 1 inequality, varying selectivity

6.4 Validation on the Condor dataset

Here we apply our 3 match join algorithms to our motivating application, the Condor distributed computing system. Condor currently runs on 1009 machines in the UW-Madison Computer Science pool, and at the time we gathered data (October 2004), there were 4739 outstanding jobs (submitted but not completed). Every job submitted to Condor goes through a "matchmaking" process. Matchmaking in Condor occurs at least once every five minutes. In each matchmaking 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 can 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(opsys varchar, arch varchar, disk int, memory int)
```

The match queries we ran on the Condor dataset contained match predicates consisting of i) 2 equalities, ii) 1 equality + 1 inequality, and iii) 2 inequalities. We present the times for full joins 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 13 shows the results of the experiment. Firstly, note that all 3 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 were 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 ones from which we gathered data. Currently, for matching Condor uses a handcrafted algorithm external to the DBMS that runs in two minutes.

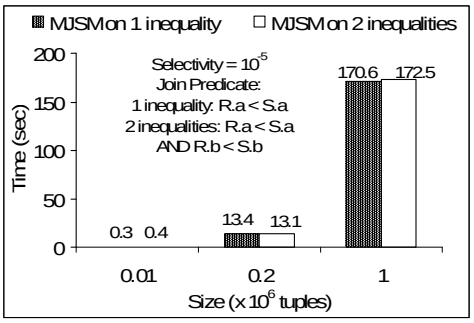

Figure 12: 1 vs. 2 inequalities, varying table sizes

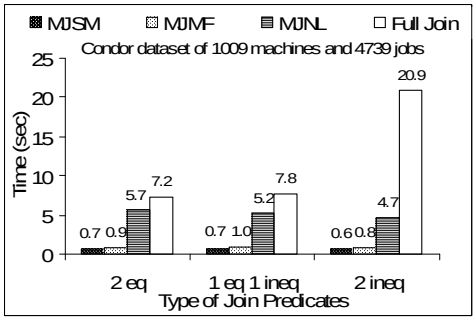

Figure 13: Performance on the Condor dataset

7. Related Work

Bipartite maximum matching is one of the oldest studied problems in graph theory [2,3,9]. Over the last decade, researchers have studied many variants of the original problem that work in parallel [6], approximate [5], and online settings [11], the latter being the case when the input(s) come in streaming order. Reference [2] contains many references to theoretical work in the area. Our work, to the best of our knowledge, is the first to address the problem of computing matchings efficiently by using database technology.

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 [1,4], top-k queries [7,10,13,17] and OPAC queries [8]. 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.

8. Conclusions and Future Work

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 within an RDBMS 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 we consider in this paper. We have focused on this maximum matching problem because it is a simple example of a class of problem that "looks like" a join but, to the best of our knowledge, has not yet been explored in the context of relational database systems. The maximum match join is interesting because it requires 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 or their generalizations 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.

Obviously a great deal of scope for future work remains. One interesting direction would be to investigate generalizations of the maximum match problem. Indeed, maximum cardinality matching is just one of many different ways to evaluate the quality of a match. Another variant of the match join

problem arises in scenarios where, instead of specifying a common match join predicate for all entities, each entity specifies its own match predicate. Yet another interesting problem to consider is how match join algorithms or their variants should be implemented within 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 the 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.

References

1. R. Agrawal and E. Wimmers. "A Framework for Expressing and Combining Preferences", *Proceedings of ACM SIGMOD 2000*, p. 297-306.
2. R. K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.
3. C. Berge, "Two Theorems in Graph Theory" *Proc. Nat. Acad. Sci. USA*, 1957, p. 842-844.
4. Y. Chang et al. "The Onion Technique: Indexing for Linear Optimization Queries", *Proceedings of ACM SIGMOD 2000*, p. 391-402.
5. J. Feigenbaum et al., "On Graph Problems in a Semi-Streaming Model", *Proceedings of ICALP 2004*, p. 531-543
6. 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.
7. L Gravano and S. Chaudhuri. "Evaluating Top-k Selection Queries", *Proceedings of VLDB 1999*, p. 397-410
8. S. Guha et al. "Efficient Approximation of Optimization Queries Under Parametric Aggregation Constraints", *Proceedings of VLDB 2003*, p. 778-789
9. J. Hopcroft, R. Karp. "An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs", *SIAM Journal of Computing*, 1975, p. 225-231.
10. I. Ilyas et al. "Supporting Top-k Join Queries in Relational Databases", *VLDB Journal*, v.13 n.3, p. 207-221
11. R. Karp, U.V. Vazirani, and V.V. Vazirani. "An optimal algorithm for online bipartite matching", *Proceedings of 22nd Annual ACM Symposium on Theory of Computing*, 1990.
12. J. Magun. "Greedy Matching Algorithms: An experimental study", *Proceedings of the 1st Workshop on Algorithm Engineering*, p. 22-31, 1997.
13. A. Natsev et al. "Supporting Incremental Join Queries on Ranked Inputs", *Proceedings of VLDB 2001*, p. 281-290.
14. Predator Project. http://www.distlab.dk/predator/
15. R. Raman et al. "Matchmaking: Distributed Resource Management for High Throughput Computing", *Proceedings of IEEE HPDC 1998*.
16. T. Tannenbaum et al., "Condor - A Distributed Job Scheduler", *Beowulf Cluster Computing with Linux*, The MIT Press, 2002
17. P. Tsaparas et al. "Ranked Join Indices", *Proceedings of IEEE ICDE 2003*, p. 277-288.