Constraint-Based Algorithms for Computing Clique Intersection Joins

Nikos Mamoulis
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
mamoulis@cs.ust.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

1. ABSTRACT

Spatial joins constitute one of the most active research topics in spatial query processing. This paper deals with the processing of clique intersection joins using R-trees. A clique intersection join will retrieve all n-tuples of objects that pair-wise overlap. The corresponding MBR-based filter step retrieves n-tuples of rectangles that intersect at some common point. Here we modify three algorithms, first proposed in [13], for the specific problem and experimentally evaluate their performance using data sets of various densities.

1.1 Keywords

Spatial joins, spatial query processing, multi-way joins

2. INTRODUCTION

Spatial queries can be classified in two major categories [3]: the first one includes *single-scan* queries, which apply a selection condition over a spatial relation. A typical query in this category is the range query (e.g. find all cities within 300km distance from Hong Kong). The cost of single-scan queries is at most linear to the number of participating objects in the spatial relation. The second category includes *multiple-scan* queries that involve more than one spatial relations. Objects may have to be accessed several times and, in general, the execution time is superlinear to the size of participating relations. The most important representative of multiple-scan queries is the spatial join.

The spatial join operation selects from two spatial relations the pairs that satisfy some spatial predicate. A typical spatial join example is "find all cities that are *crossed by* a river". Here, two spatial relations, "Cities" and "Rivers", are joined using the spatial predicate *cross*. The main difficulty in processing spatial joins, is the fact that there does not exist a total ordering of objects in the multidimensional space that preserves spatial proximity. This characteristic does not permit the application of traditional relational join algorithms such as sort-merge join. As a consequence, several specialized methods have been developed for the computation of spatial joins. These methods can be classified in two categories.

The first category includes approaches (e.g., [3], [10]) which assume that the relations to be joined are indexed on the spatial attributes, an assumption which is true for most modern spatial databases, since spatial indexing facilitates fast execution of selection queries. Methods in the second category do not take under consideration an existing spatial index on the joined attributes, but instead they either use special built indices for spatial joins [14], or employ on-the-fly indexing mechanisms [11].

A complete query processor should be able to handle join of multiple (>2) inputs, without interrupting data between the join operators [5]. Unfortunately, the above techniques do not consider this issue; in other words, there is no systematic way to handle multi-way spatial joins. To the best of our knowledge, previous work on multi-way joins has concentrated mainly on the relational model. Query evaluation techniques for multiple relational joins include computation of optimal execution orderings [15], and parallel execution engines [6].

An interesting fact about multi-way joins is that they can be seen as constraint satisfaction problems (CSPs). A CSP [12] is defined as a set of variables, whose domain values are restricted by a number of constraints. A solution to a CSP is an assignment of a value to each variable, such that all constraints are satisfied. Although CSPs are in general NP-complete, there exist several systematic search algorithms that solve moderate size problems. One such algorithm, experimentally proven [1] to outperform the rest for a wide range of application domains, is *forward checking* [9].

In [13] we investigated the computation of multi-way joins that involve a set of spatial relations given in a specific order determined by some cost model [16]. Assuming that the spatial relations are indexed by *R-trees*, we presented three hybrid algorithms, which enhance forward checking

In Proceedings of ACM-GIS'98, Washington D.C., November 6-7, 1998

by taking advantage of spatial indexing. In this paper we adapt the algorithms to deal with *clique intersection joins* (the join condition is *overlap* between all pairs of spatial relations) and experimentally evaluate their performance. Figure 1 illustrates three objects that pair-wise overlap. Observe that if any set of objects satisfy the clique intersection property, their minimum bounding rectangles (MBRs) have a non-empty intersection (gray area).

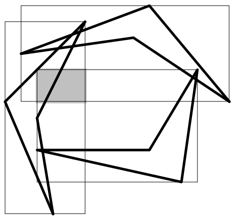

Figure 1: Clique intersections

The paper is organized as follows: Section 3 describes the common approach for computing pair-wise spatial joins using R-trees. The relation between multi-way spatial joins and CSPs, as well as a description of the general forward checking algorithm are given in section 4. In sections 5,6, and 7 we describe three systematic algorithms for computing clique intersection joins. Section 8 presents a performance comparison of the algorithms for a range of data sets and join sizes. Finally, section 9 concludes the paper.

3. BACKGROUND ON SPATIAL JOINS

The R-tree [7] is a multi-dimensional extension of the B-tree, used in many commercial GIS and DBMS. The MBRs of the actual data objects are stored in the leaf nodes of the tree and intermediate nodes are built by grouping rectangles at the lower level. Each node of the tree corresponds to a disk page and is associated with some rectangle which *encloses* all rectangles that correspond to lower level nodes. The processing of a traditional overlap query in R-trees involves the following procedure: Starting from the top node, exclude the nodes that are disjoint with the query window, and recursively search the remaining ones. Among the entries of the leaf nodes retrieved, select the ones that overlap the query window.

When two MBRs are disjoint we can conclude that the objects that they represent are also disjoint. If the MBRs however share common points, no conclusion can be drawn about the spatial relation between the objects. For this reason, spatial queries involve the following two step strategy: first a *filter* step uses the tree to rapidly eliminate objects that could not possibly satisfy the query. The result is a set of candidates, which includes all the results and possibly some false hits. Then during a *refinement* step each candidate is examined (by using computational geometry techniques) and false hits are detected and eliminated. Several variations of the original R-trees have been proposed to increase efficiency. In our implementation we use R*-trees [2] which employ the same data structure but a different insertion algorithm in order to minimize the overlapping area of R-tree nodes.

The most influential approach for efficiently computing pair-wise, intersection joins using R-trees is presented in [3]. The algorithm is based on the *enclosure property*: if two intermediate nodes of the R-trees to be joined do not intersect, there can be no MBRs below them that intersect. The algorithm first joins the high level nodes of the trees and then follows the links in order to find qualifying pairs below them:

```
SpatialJoin(R,S: R_Node) /* R,S have equal height */
  FOR (all E_S ∈ S) DO /* for all entries in S */
     FOR (all E_R ∈ R with E_R.rect ∩ E_S.rect ≠ ∅) DO
        IF (R is a leaf page)
        THEN  output (E_R, E_S)
        ELSE
          ReadPage(E_R.ref); ReadPage(E_S.ref);
          SpatialJoin(E_R.ref, E_S.ref)
```

Two CPU time optimization techniques, namely *search space restriction* and *plane sweep*, are used to improve the CPU speed of the above algorithm. In addition, [3] applies I/O time optimization methods to determine an optimal page fetching policy during spatial joins. [10] extends *SpatialJoin* by introducing an on-the-fly indexing mechanism to optimize the execution order of matches at intermediate levels of the joined trees, while [4] exploits parallel execution of pair-wise spatial joins. In the rest of the paper we apply CSP search techniques to process clique intersection joins.

4. MULTIWAY SPATIAL JOINS AS CSPs

A constraint satisfaction problem [12] is defined over a finite set of variables, each with a finite domain of potential values. Formally, a binary CSP consists of:

- A set of *n* variables, $V_1, V_2, \dots, V_n$
- For each variable V_i a finite domain $D_i = \{u_1, \dots, u_N\}$ of potential values
- For each pair of variables V_i, V_j a binary constraint C_{ij} which is simply a subset of $D_i \times D_j$. If $(u_k, u_l) \in C_{ij}$ then the assignment $\{V_i \leftarrow u_k, V_j \leftarrow u_l\}$ is *consistent*.

A *solution* is an assignment $\{V_1 \leftarrow u_p, \dots, V_n \leftarrow u_r\}$, such that for all i,j: $\{V_i \leftarrow u_k, V_j \leftarrow u_l\}$ is consistent. A spatial join can be mapped to a CSP as follows:

- Each joined attribute is a variable, e.g., the query “find all cities that are *crossed by* a river” contains two variables, V_1 and V_2, for rivers and cities respectively.
- The domain of each variable V_i is the corresponding spatial relation R_i (e.g., R_1 is the set of cities). V_i can only take as a value a rectangle r_{ik} from relation R_i.
- Each join predicate (e.g. "crossed by") corresponds to a binary spatial constraint. In clique intersection joins the constraint is *overlap* between all pairs of variables.

One of the most effective algorithms for solving CSPs is forward checking (FC) [9]. FC works as follows: when a variable V_i is assigned a value u_k, the domain of each *future* (un-instantiated) variable V_j is pruned according to u_k and the constraint C_{ij} (all values that are inconsistent with C_{ij} and u_k are removed from D_j); the values of variables $V_1,\dots,V_i$ will constitute a *consistent partial solution*, and the domains of the future variables V_j ($i<j\le n$) will be consistent with all constraints C_{hj}, where $h\le i$.

The above procedure is called *check forward*. If, after a check forward, the domain of a future variable is exhausted, FC un-assigns the current variable's value, and restores the corresponding eliminated values of future variables. When the domain of the current variable is exhausted the algorithm *backtracks* to the previous variable and assigns a new value to it. FC outputs a solution whenever the last variable is given a value, and terminates when it backtracks from the first variable.

```
FC()
  i := 1; /*index to the current variable*/
  WHILE (TRUE) {
    new_value := getNextValue(i);
    IF new_value = NULL /*empty domain*/
      THEN  IF i=1  /*first variable*/
              THEN RETURN;  /*termination of the algorithm*/
              ELSE i:=i-1; CONTINUE;  /*backtrack*/
      ELSE  /*non-empty domain*/
        instantiations[i] := new_value; /*store instantiation*/
        IF i = n  /*last variable instantiated*/
          THEN  output_solution(instantiations);
          ELSE /*intermediate variable instantiated*/
            IF check_forward(i)  /*successful instantiation*/
              THEN  i := i+1; /*proceed to the next variable*/
              ELSE restore_eliminations(i); /*restore eliminated values*/
 } /*end WHILE*/
BOOLEAN check_forward(int i)
  FOR j = i+1 TO n DO /*for all uninstantiated variables*/
    FOR all not already eliminated values ul ∈ Dj
      IF inconsistent (instantiations[i], ul)
        THEN eliminate (ul); /*var. Vj cannot take value ul */
    IF Dj = ∅   /*the whole domain of Vj has been eliminated */
      THEN RETURN FALSE;
 RETURN TRUE;
```

The application of FC for computing multi-way spatial joins is illustrated though the following example. Consider the multi-way intersection join of the spatial relations R_1, R_2, R_3, R_4, as shown in Figure 2. The problem is to find all 4-tuples (r_{1i}, r_{2j}, r_{3k}, r_{4l}), $r_{1i} \in R_1$, $r_{2j} \in R_2$, $r_{3k} \in R_3$, $r_{4l} \in R_4$, such that r_{1i}, r_{2j}, r_{3k}, r_{4l} share some common point.

Initially, $V_1=r_{11}$. *Check forward* reduces the domain of V_2 to $\{r_{21}, r_{22}\}$, as these are the only rectangles that intersect the current value of V_1, i.e., r_{11}. Moving one step further, all values from the domain of V_3 are eliminated. At this point, there is no reason for proceeding to variable V_4, because all assignments having $V_1 = r_{11}$ are inconsistent with R_3. FC now sets $V_1 = r_{12}$. After checking forward, the domains of future variables V_2, V_3, V_4 become $\{r_{21}, r_{22}\}$, $\{r_{33}, r_{34}\}$, $\{r_{42}\}$, respectively. No future variable has been eliminated, thus we can proceed to the next instantiation level. After setting $V_2 = r_{21}$, r_{33} is eliminated from R_3. FC goes forward twice, and outputs the first valid assignment (r_{12}, r_{21}, r_{34}, r_{42}). As r_{34} and r_{42} were the only valid values for variables V_3 and V_4, respectively, the algorithm backtracks twice to V_2 to change its value to r_{22}. Notice that when V_2 is un-assigned the value r_{21}, the domain of V_3 becomes again $\{r_{33}, r_{34}\}$, because the eliminated value r_{33} due to $V_2 = r_{21}$ is restored. FC continues and completes the solution set $\{(r_{12}, r_{21}, r_{34}, r_{42}), (r_{12}, r_{22}, r_{33}, r_{42}), (r_{12}, r_{22}, r_{34}, r_{42})\}$.

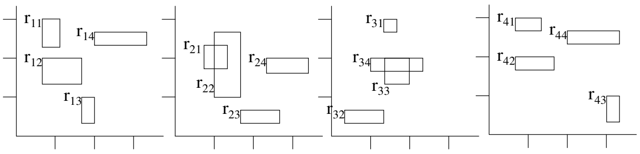

Figure 2: Example of FC

During the backtracking phase, values of future variables excluded due to an inconsistent instantiation should be restored. In order to keep track of the consistent values for each variable at every instantiation level, FC uses a *domain table* of size $O(n^2N)$, where n is the number of the variables (usually < 10) and N the size of the domain. If the size of the domain is prohibitively large, as is the case for most spatial applications, the algorithm is inapplicable. Another drawback of plain FC is the fact that it does not utilize possibly existing spatial indices. In the next sections we propose some methods that combine the basic idea of FC with spatial indexing and can effectively compute clique intersection joins for large spatial databases.

5. THE ADAPTED WR ALGORITHM

In this section, we present and analyze a version of the *Window Reduction* (WR) algorithm, proposed in [13], adapted for the clique intersection join problem. This algorithm, assuming that the joined spatial relations are indexed by R-trees, considers the domains of the variables as windows and takes advantage of spatial indexing to avoid main memory limitations and accelerate the retrieval of qualifying tuples.

The original WR algorithm incrementally assigns values to variables, just like plain FC does. After an assignment, instead of updating the domain of the future variables, it calculates the *domain windows,* which contain all the potential values for each future variable. For instance after V_i is instantiated to u_k, the domain window dw_{ij}, of (subsequent) variable V_j contains all the possible values for V_j, which are consistent with u_k and the input constraints. WR stores the domain windows of the variables in a n×n array (instead of n×n×N for simple forward checking), analogous to the domain table of FC, in order to facilitate restoration of the domain windows after backtracking. When WR reaches at instantiation level i (for variable V_i), it performs a window query on relation R_i, using the domain window dw_{ii}. After choosing a value for V_i the algorithm updates the domain windows of the future variables. Hence, while moving forward, the domains of the last variables become gradually smaller. If the domain window of a variable becomes empty, the variable has been eliminated and the algorithm backtracks.

The adapted version of WR for clique intersection joins (WR-I), has the following differences from the original WR:

- The domain window of all future variables is the same and equal to the intersection area of the instantiations so far (because of the non-empty intersection of MBRs as in Figure 1). Thus, WR-I does not need to keep the domain windows in a n×n array, rather they are kept in a one dimensional array as opposed to the original WR. When backtracking, restoring is done using the previous level's domain window.
- As a result of the previous property, WR-I can be thought of as type of forward checking with depth 1. If *new_value* is the rectangle assigned to currently instantiated variable V_i, the new domain window becomes $dw_{i+1}=dw_i \cap new\_value$ for all future variables.

```
WR-I(RTree rt[])
  i := 1; /* index to the current variable */
  domainWindow[1] = U; /* the first variable has universal domain */
  WHILE (TRUE) {
    new_value := getNextValue(i, rt[i], domainWindow[i]);
    IF new_value = NULL /*empy domain*/
      THEN IF i = 1 /*first variable*/
        THEN RETURN;  /*termination of the algorithm*/
        ELSE i := i-1; CONTINUE; /*backtrack*/
      ELSE /*non-empy domain*/
        instantiations[i] := new_value; /*store instantiation*/
        IF i= n /*last variable has been instantiated*/
          THEN output_solution(instantiations);
          ELSE /*update the window for all next variables */
            domainWindow[i+1] = domainWindow[i] ∩ new_value;
            i=i+1; /*proceed to the next variable*/
} /*end WHILE*/
```

To comprehend the functionality of WR-I consider the example of Figure 3. Suppose that we want to join the three images which are indexed by the respective R-trees. The domain window of the first variable is the whole data space; WR-I cannot avoid assigning all possible values to the first variable. After V_1 gets value a_1, dw_2 becomes $U \cap a_1 = a_1$; all candidate values for V_2 should intersect a_1. A window search in R_2 retrieves b_1, as candidate value for V_2 and dw_3 becomes $dw_2 \cap b_1$, i.e. $a_1 \cap b_1$. Next, the algorithm searches in R_3 for rectangles that intersect dw_3 and gets c_1. After finding solution (a_1,b_1,c_1), the algorithm backtracks from V_3 to V_2 and subsequently backtracks from V_2, because no other value in R_2 intersects a_1. Similarly, WR-I finds the second solution (a_2,b_2,c_3) and finishes, after it fails to find values in R_2 that intersect a_3, a_4 and a_5.

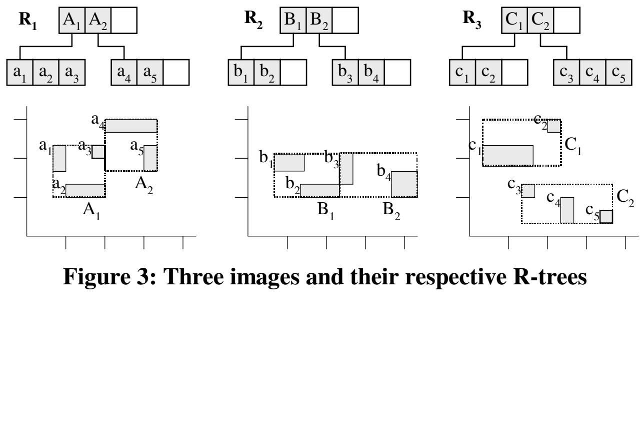

Figure 3: Three images and their respective R-trees

6. THE ADAPTED JWR ALGORITHM

Although WR-I works fast once the first variable has been instantiated, it seems to suffer from the large number of values V_1 has to take, most of which may be redundant. To address this issue, we implemented a modified version of the *Join Window Reduction* (JWR) [13] for the specific problem. The following JWR-I algorithm applies *SpatialJoin* [3] to first compute the pair-wise overlap join of the first two variables. It then employs the window reduction technique to compute the desired multi-way join.

```
JWR-I(Rtree rt[])
  i := 2; /*values for the first 2 variables come as pairs*/
  WHILE (TRUE) {
    IF i = 2  /*values of first two variables*/
      THEN IF getNextPair(rt[1], rt[2], instantiations) = NULL
        THEN RETURN;/*termination-backtrack from first 2 variables*/
        ELSE domainWindow[3] := instantiations[1] ∩ instantiations[2];
      ELSE  /*values of third and subsequent variables*/
        new_value := getNextValue(i, rt[i], domainWindow[i]);
        IF new_value = NULL /*empy domain for 3rd or later variable*/
          THEN i := i-1; CONTINUE; /*backtrack*/
          ELSE  /*non-empy domain for 3rd or later variable*/
            instantiations[i] := new_value;
            IF i = n /*last variable has been instantiated*/
              THEN output_solution(instantiations);
              ELSE /*update the window for all next variables */
                domainWindow[i+1] = domainWindow[i] ∩ new_value;
                i = i+1;
  } /* end WHILE */
```

The main difference between JWR-I and WR-I is the introduction of *getNextPair()*, which returns a joined pair of values for the first two variables. This function is the *SpatialJoin* algorithm (with the two CPU-time optimization techniques [3]). In the 3-way join example of Figure 3 JWR-I retrieves one by one the pairs (a_1,b_1) and (a_2,b_2) using *SpatialJoin* and then applies the window reduction technique to complete the solutions.

7. THE ADAPTED MFC ALGORITHM

An alternative multi-way spatial join algorithm is *Multilevel Forward Checking* (MFC) [13]. MFC is an extension of *SpatialJoin*, that takes advantage of the enclosure property of the high level R-tree nodes to prune out the search space. Here we present MFC-I, an adapted MFC for the clique intersection joins problem. The key idea behind MFC-I is the fact that if the intermediate nodes at the high levels of the R-trees do not pair-wise intersect, there can be no rectangles under these nodes that may pair-wise intersect. Following this observation, we can apply FC to the intermediate nodes of the trees and follow the links from solutions at the high levels to find solutions at the lower levels of the trees.

To clarify the above idea, consider again the three images of Figure 3. Observe that since (A_2,B_2,C_2) do not formulate a solution to the problem (they do not pair-wise intersect), there can be no combination of values (a_i,b_j,c_k), $a_i \in A_2$, $b_j \in B_2$, $c_k \in C_2$, such that (a_i,b_j,c_k) is a solution. This property is very useful especially when it excludes combinations of nodes at high-levels of the indices, as the

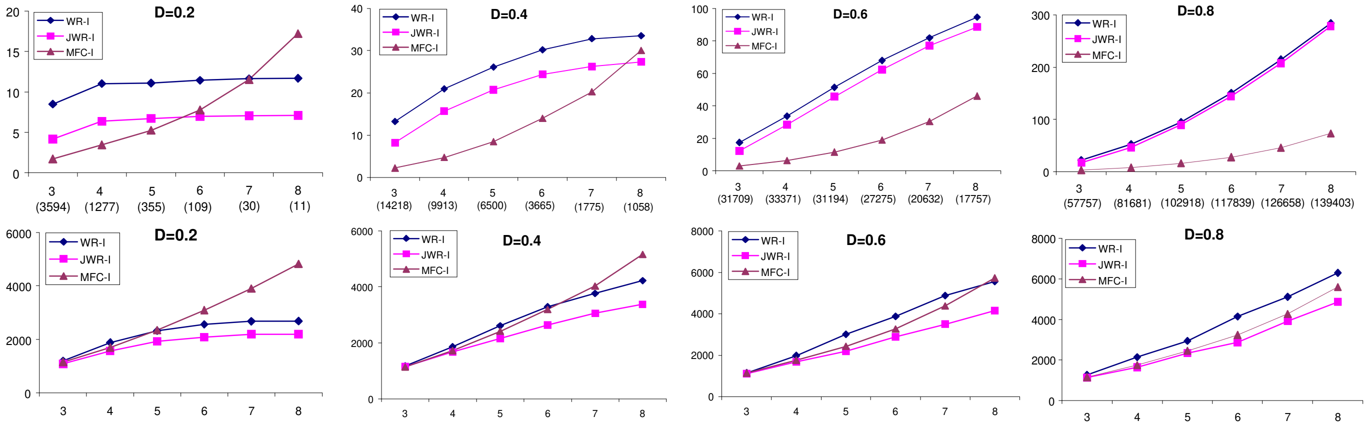

Figure 4: Performance of the algorithms in terms of CPU seconds (first row) and disk accesses (second row) for various densities of data sets. The values on the x-axis correspond to the number of relations and the parentheses enclose the number of solutions.

search space can be dramatically reduced. A pseudo-code for MFC-I is given below:

```
MFC-I(RTreeNode[] nodes)
  i := 1; /*index to the current variable at this level*/
  WHILE (TRUE) {
    new_value := getNextValue(i); /*as in plain FC*/
    IF new_value = NULL /*empty  domain*/
       THEN IF i=1 THEN RETURN; /*return to the upper tree level*/
       ELSE i:=i-1; CONTINUE;  /*backtrack (at the current level)*/
    ELSE  /*non-empty  domain*/
       instantiations[i]:=new_value;
       IF i = n /*last variable instantiated*/
          THEN  IF level>0 /*intermediate level*/
                THEN MFC-I(instantiations, n); /*go to lower level*/
                 ELSE output (instantiations); /*solution found*/
          ELSE  /*intermediate variable instantiated*/
                 IF check_forward(nodes, i);  /*succesful instantiation*/
                     THEN    i = i+1; /*proceed to the next variable*/
                     ELSE /*unsuccessful instantiation*/
                        restore_eliminations(i);
  } /*end WHILE*/

BOOLEAN check_forward(Node[] nodes, int i)
  FOR j = i+1 TO n DO /*for all uninstantiated variables*/
    FOR all not eliminated entries Ejl ∈ nodes[j]
      IF Ejl ∩ instantiations[i] = ∅
         THEN eliminate (Ejl); /*var. Vj cannot take value Ejl*/
    IF Dj = ∅  /*the whole domain of Vj has been eliminated */
         THEN RETURN FALSE;
  RETURN TRUE;
```

The parameters of MFC-I at the first call are the roots of the R-trees of the relations to be joined. When a solution is retrieved at the intermediate levels of the trees, MFC-I is recursively called, taking as parameter the references to the underlying nodes. Solutions at the leaf level correspond to joined tuples and are output.

Consider again the example in Figure 3. Initially, MFC-I is applied for the root level of the trees. The first solution at this level is (A_1,B_1,C_1). After following the links and applying forward checking at the succeeding level, MFC-I outputs the solution (a_1,b_1,c_1). Next, the algorithm returns to the root level and identifies the tuple (A_1,B_1,C_2). Going down one level again, we obtain (a_2,b_2,c_3). The last root-level solution (A_2,B_2,C_1) does not lead to any leaf-level solution, and after going up from the root, the algorithm terminates.

8. EXPERIMENTS

In order to compare the performance of WR-I, JWR-I, and MFC-I, we implemented and tested the algorithms under several conditions. The implementation language was C++, and all experiments were run on a SUN UltraSparc2 (200MHz) workstation with 256 MB of RAM.

For our experiments we created a number of synthetic data sets each consisting of 10,000 uniformly distributed rectangles. As we are interested on the behaviour of the algorithms under several density conditions[1], we created 4 classes of data sets of densities 0.2, 0.4, 0.6, and 0.8, respectively. Each class consists of 8 data sets with the same density. Every data set was stored in an R*-tree [2] of page size 1KB. We also implemented a cache with an LRU buffer that fits 128 blocks (a typical value), in order to measure the performance of the algorithms by means of I/O page accesses.

The experiments focus on the performance of the algorithms when computing clique intersection joins of data sets of the same density. Figure 4 shows the performance for all 4 classes of data sets. From the results we observe the following:

- JWR-I outperforms WR-I in all the cases by a constant factor. Since the only difference of the algorithms is the calculation of the first instantiation pair, we expected this constant factor improvement.
- JWR-I behaves totally differently from MFC-I. As an overall conclusion, we can say that JWR-I is better than MFC-I only when the joined data sets are sparse (density 0.2, 0.4) and the number of joined relations is large (>5). In this case, MFC-I does a lot of redundant

[1] Density is defined as the sum of areas of all rectangles divided by global space [16].

work at the high levels of the trees, being unable to detect early the small fraction of successful tuples. In most other cases, MFC-I outperforms JWR-I.

- JWR-I is almost always better than MFC-I by means of I/O page accesses, even for large density files. The CPU-time overhead makes the overall performance of JWR-I worse (we typically charge 10ms for each disk access [10]), but it is interesting to notice that the window reduction policy saves disk accesses.
- MFC-I presents a similar behaviour under all data densities; the running cost grows superlinearly to the number of joined relations, in both CPU-time and I/O page faults, independently of the problem conditions. The running time of JWR-I almost stabilises when the number of solutions shrinks with the number of joined relations.

Ordering	WR-I		JWR-I		MFC-I	
D_1, D_2, D_3, D_4	17.24	1808	12.71	1494	4.65	1679
D_4, D_3, D_2, D_1	32.62	2065	27.31	1728	5.55	1763

Table 1: Performance of the algorithms (sec/page faults) for two different orders of the same data sets.

Table 1 shows the performance of the algorithms when computing the clique intersection join of 4 data sets D_1, D_2, D_3, D_4, with density 0.2, 0.4, 0.6, 0.8, respectively, in two different orders. Observe that the order of the data sets is crucial, especially for the window reduction algorithms. The optimal order for the specific problem is D_1, D_2, D_3, D_4. Clearly, as the data sets consist of the same number of rectangles, the density is the only remaining factor that determines the cost of pair-wise joins [16]. The side-effects of ordering are smaller in MFC-I because false instantiations of the former variables are detected early at the high levels of the trees.

9. CONCLUSIONS

A multi-way spatial join can be defined as follows: Given a set of spatial relations $\{R_1, R_2, \ldots, R_n\}$ and a set of binary spatial predicates $\{C_{ij} \mid i \neq j, 1 \leq i,j \leq n\}$ find all tuples $\{u_1, \ldots, u_n \mid u_i \in R_i, 1 \leq i \leq n\}$, such that for each i,j, $i \neq j$, $1 \leq i,j \leq n$, $u_i\, C_{ij}\, u_j$.

In this paper we dealt with a specific instance of the above problem where for each i,j the spatial predicate C_{ij} is *overlap*. We adapted the three general multi-way spatial join algorithms proposed in [13] for this problem and tested their performance with several spatial data sets of various densities. The results show that if the density (and consequently the number of expected solution tuples is large), MFC-I outperforms the other algorithms. Notice that in the experimental evaluation of [13] with real data sets of density around 0.2 and various types of spatial constraints, JWR ouperformed MFC by orders of magnitude, implying that the properties of the data and the types of constraints have a serious effect on performance.

In our future work, we are interested in investigating page fetching policies in order to improve the performance of MFC-I in terms of I/O page accesses. Especially for large number of joins, this number grows at prohibitively high levels. We believe that a good I/O optimization policy will make this method applicable for real spatial database systems.

10. REFERENCES

[1] Bacchus, F., van Run, P. "Dynamic Variable Ordering in CSPs", International Conference on Principles and Practice of Constraint Programming, 1995.

[2] Beckmann, N., Kriegel, H.P. Schneider, R., Seeger, B. "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles". ACM SIGMOD, 1990.

[3] Brinkhoff, T., H.-P. Kriegel, B. Seeger "Efficient processing of spatial joins using R-trees". ACM SIGMOD, 1993.

[4] Brinkhoff T., Kriegel H.-P., Seeger B. "Parallel Processing of Spatial Joins Using R-trees". Proc. of the 12th International Conference on Data Engineering, 1996.

[5] Graefe G., "Query Evaluation Techniques for Large Databases". ACM Computing Surveys, vol. 25, no. 2, pp. 73-170, 1993.

[6] Graefe G., "Volcano, An Extensible and Parallel Dataflow Query Processing System". IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, pp. 120-135, 1994.

[7] Guttman, A. "R-trees: A Dynamic Index Structure for Spatial Searching". ACM SIGMOD, 1984.

[8] Güting, R-H. "An Intoduction to Spatial Database Systems". VLDB Journal 3(4), 357-399, 1994.

[9] Haralick R.M., Elliot G.L., "Increasing tree search efficiency for constraint satisfaction problems". Artificial Intelligence, vol 14, pp.263-313, 1980.

[10] Huang, Y-W, Jing, N, Rundensteiner, E. "Spatial Joins using R-trees: Breadth First Travesral with Global Optimizations". VLDB, 1997.

[11] Koudas N., Sevcik K., "Size Separation Spatial Join". ACM SIGMOD, 1997.

[12] Mackworth A.K. "Constraint Satisfaction". In S.C. Shapiro editor, Encyclopedia of Artificial Intelligence". John Wiley and Sons, New York, 1987.

[13] Papadias, D., Mamoulis, N., Delis, B. "Algorithms for Querying by Spatial Structure". VLDB, 1998.

[14] Rotem, D. "Spatial Join Indices". IEEE International Conference on Data Engineering, 1991.

[15] Swami A., Gupta A., "Optimization of Large Join Queries". ACM SIGMOD, 1988.

[16] Theodoridis Y., Stefanakis E., Sellis T., "Cost Models for Join Queries in Spatial Databases", Proc. of the 14th International. Conference on Data Engineering, 1998.