

Resequencing and Clustering to Improve the Performance of Spatial Joins

David J. Abel, Volker Gaede, Robert A. Power, Xiaofang Zhou

CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, ACT 2601, Australia
{Dave.Abel,Volker.Gaede,Robert.Power,Xiaofang.Zhou}@cmis.csiro.au

Abstract. The filter-and-refine strategy is well-established as the basis for spatial join algorithms. In contrast to the filter step, the refinement step has received little attention, despite contributing significantly to the total cost of a join evaluation. Sorting candidate tuples as produced by the filter step has recently been shown to reduce the I/O cost of refinement. Our paper reports investigations of spatial join algorithms, with particular emphasis on interactions between choices of algorithms for the filter, sequencing and refinement steps and on the effects of clustered and unclustered organization of full spatial descriptions of objects. We extend existing techniques by caching spatial descriptions and a new sorting strategy, called *zig-zag*. Our experiments confirm that the choice of the sequencing strategy used is very important and that clustering has a significant influence on join performance. Clustering by z-ordered keys is shown to be superior to ordering by minimum bounding rectangles.

Keywords: Spatial joins, spatial access methods, advanced databases, caching.

1 Introduction

Spatial selection and *spatial join* are the basic query operations in spatial databases. Spatial selection deals with the evaluation of tuples from a single relation, satisfying a given spatial search predicate such as intersection and enclosure. Query 1, for example, identifies all land parcels stored in the `parcels` relation that intersect a given window with known coordinates.

Query 1

```
SELECT parcel.id
FROM   parcels
WHERE  intersects (parcel.extent, '100, 320, 115, 334');
```

Spatial joins generalize spatial selections by evaluating pairs of tuples coming from at least two different relations with respect to a given spatial predicate (neglecting the case of self-join). Query 2 shows an example of a spatial join that determines all houses adjacent to at least one hotel.

Query 2

```
SELECT houses.id
FROM   houses, hotels
WHERE  adjacent (house.extent, hotels.extent);
```

As with spatial selection, most reported spatial join algorithms follow a two stage process. The first stage is known as *filter step* and operates on approximations of the actual spatial descriptions of objects. This step aims at identifying inexpensively a set of candidate objects that are likely to satisfy the spatial predicate, while ruling out the major amount of data not relevant for answering the query. Due to the approximation, the candidate set almost inevitably contains *false drops*, that is, spatial objects not satisfying the actual spatial predicate. To identify these false drops, the

refinement step fetches the actual spatial description from disk and performs a precise test on the full descriptions.

Since spatial descriptions are typically extensive and range from hundreds of bytes for Land Information Systems (Abel 1993) to tens of kilobytes for Geographic Information Systems (Brinkhoff et al. 1993), the cost of fetching the descriptions can contribute significantly to the total cost of a join evaluation. A variant on the filter-and-refine strategy has been suggested by Patel and DeWitt (1996) who include an additional *housekeeping step* in which they perform sequencing of the candidate pairs before fetching them from disk. Their results show that such an additional step can reduce the cost of the refinement step considerably. Consequently, it is attractive to consider if further attention to the housekeeping step can yield performance gains.

This paper contributes to the development of spatial join algorithms by proposing and evaluating two additions to the refinement stage. These are aimed at minimizing the I/O cost by reducing the number of duplicate fetches of spatial descriptions from disk. The first is a novel sequencing algorithm, called *zig-zag*. The second is a cache of spatial descriptions that essentially complements the page-based system buffer.

In evaluating these techniques, we aim to establish their behavior in the context of the other components of a join algorithm and of the characteristics of the operand data sets. As we discuss further below, the design of a spatial join algorithm can be treated as the composition of component methods for the filter step, the sequencing strategy, and the refinement step. There are two motivations for taking such a whole algorithm approach to evaluation. Firstly, it provides information on the interactions between the various parameters for the filter and refinement steps, and so can be more meaningful than considering algorithms for the two steps separately. Secondly, these insights contribute to spatial query optimization. The query optimizer can examine the data dictionary entries of the operand data sets to determine the presence and type of indexes, the form of clustering, etc. It can then choose the best filter and sequencing algorithms from those available.

In Section 2, we outline the decomposition of a join algorithm into discrete steps. We then review the more common filter algorithms and note significant variations in the ordering of the sets of candidate pairs which they generate. In Section 3, we consider how caching can be applied and describe different strategies to maximize the benefits of a cache. An empirical assessment of these caching strategies is given in Section 4.

2 Background

2.1 Parameters of a Spatial Join

A spatial query optimizer has to take into account several factors when choosing a spatial join strategy. The selection of the algorithm for the filter step will be based on the presence of spatial indices (and the type of indices) and form of clustering of the data. In some situations, it might be efficient to build one or even two indices on the relations, to enable the use of very efficient join algorithms such as synchronized traversal. Depending on the join algorithm chosen for the filter step, it might be advantageous to introduce additional housekeeping to perform duplicate removal or load some additional information, which contributes to the overall cost. In addition, the query optimizer also might decide on a caching strategy to reduce the cost of the refinement step. All these parameters determine the performance of spatial join and their interaction is not clear. Figure 1 summarizes them.

To date, work in the area of spatial joins has to date focussed on algorithms for the filter step. The refinement step has not been considered in depth, despite its intuitively significant effect on the total cost and so its consequent significant optimization potential. For the optimizer to make an informed choice of processing strategy, it is necessary to have reliable heuristics or cost estimates of the refinement step.

This paper contributes to a better understanding of spatial join algorithms by showing that caching the spatial descriptions during the refinement step provides a significant reduction in the

Filter Step	Spatial Access Methods (none, one, two compatible or incompatible)	
	build-up index?	transformation step (eg., shift one map)
	Join Algorithm (nested-loops, index-supported, hash-based, sort-merge, synchronized traversal, etc.)	
	Housekeeping Step	duplicate removal, sorting, load additional information
Refinement Step	Computational Geometry Algorithm	
	Caching Algorithm (naive, sorting, zig-zag, prefetching, etc.)	
	Clustering (no clustering, row-wise clustering, sorted, etc.)	

Fig. 1. Optimization Parameters of a Spatial Query Optimizer

overall I/O cost of the operation. We consider two classes of refinement strategies: *immediate* and *deferred* processing. In the first class, candidate pairs are tested as they are generated by the filter step, ie., no sequencing takes place. In the second class, the full set of candidate pairs is assembled before applying the refinement test. The deferred strategy allows sorting of the set of candidate pairs. As we discuss in more detail below, however, the different classes of filter algorithms generate different sequences of candidate pairs and this has implications for the choice and effectiveness of the refinement step. We do not consider the choice of computational geometry algorithms to test satisfaction of predicates during refinement, since this does not influence I/O cost.

2.2 Related Work

Current work on spatial joins can be roughly classified into three categories:

1. Algorithms assuming the existence of spatial indices on both relations participating in the join operation (Günther 1993; Brinkhoff et al. 1993);
2. Algorithms assuming only one spatial index (Lo and Ravishankar 1994);
3. Algorithms assuming no index at all (Güting and Shilling 1987; Lo and Ravishankar 1995; Patel and DeWitt 1996; Lo and Ravishankar 1996).

Most attention has been paid to the first case, where the two operand sets have a compatible spatial index and the join is to be performed on the full relation. The second and the third cases are, however, also very important. In a complex query, for example, one operand set might be an interim result from previous operations. A fourth case arises when the set of candidates presented to a refinement stage is the result of non-spatial operations. For example, if we extend Query 2 to consider only houses with a value of greater than \$500,000 and hotels of five-star rating, and the query execution planner determines that there are few houses with a value of \$500,00 and few five-star hotels, a suitable query execution plan might be to evaluate a join of all houses with value greater than \$500,000 and all five-star hotels and then test each tuple for satisfaction of the

spatial predicate. Essentially the early steps of the query execution plan act as the filter step in generating a stream of candidate pairs.

Except for the work by Patel and DeWitt (1996), none of the above proposals addresses the refinement step, even though the I/O cost of the refinement step can dominate the overall performance of the join. This is particularly true if the spatial descriptions are extensive and there is a high likelihood that descriptions from at least one of the sets will be fetched more than once. (Patel and DeWitt 1996) address this issue by deferring the refinement step until all candidate pairs are available, sorting the list by object identifiers of one operand data set as the primary key and object identifiers of the other as the secondary key, and then traversing the list. However, even after this optimization, the I/O cost of the refinement step is approximately 30 to 60 percent of the total I/O cost, depending on the system buffer size. This suggests that investigation of the refinement step might yield further performance improvements of the spatial join. In this paper, we are particularly concerned with minimizing the I/O cost of the refinement step.

3 Spatial Access Methods and Join Algorithms

In the paper, we restrict our attention to arguably the two most popular access methods, namely *z-ordering* and *R-trees*. We briefly review the most important traits of these access methods and their join algorithms to establish characteristics relevant to the design of algorithms for the refinement step.

3.1 Z-Ordering

The *z-ordering* technique is based on space filling curves and has been discovered by several researchers independently under different names (Gaede and Günther 1995). Z-ordering approximates a given object's shape by recursively decomposing the embedding data space into smaller subspaces known as *Peano cells*. The *z-ordering* decomposition works as follows: starting with the complete data space, we form smaller and smaller subspaces (the Peano cells) by splitting the respective data space along a hyperplane that is parallel to the coordinate axes, eg., parallel to the $x - y$ -plane for two dimensions. For each subspace, we visit these planes cyclically until a termination criterion is met. After every split we proceed with the subspace where the object or parts of it are to be found. If the hyperplane intersects the object's shape, we split it along the hyperplane and proceed recursively with the two resulting sub-shapes and their corresponding subspaces until a termination criterion is met (Gaede 1995b). Each of these Peano cells can be canonically identified by the splitting sequence required to generate it. Depending on the location of the given (sub-) shape relative to the splitting hyperplane, we either attach a zero ('0') or a one ('1') to the splitting sequence known as *z-value*. As a net result of this decomposition, we approximate the extended object's shape with numerous Peano cells of possibly different size or, to put it differently, we represent it by a set of *z-values* of possibly different length. Figure 2 shows an object's shape and its approximation by numerous Peano cells.

From a mathematical viewpoint, this decomposition is a transformation of a two (or higher) dimensional object into a set of one-dimensional points (the *z-values*) which can be represented as numbers. These numbers can in turn be maintained by a ubiquitous one-dimensional access method such as B^+ -tree. The resulting structure is known as *zkdb⁺-tree* (Orenstein 1989).

In addition to its simplicity, *z-ordering* several good features. For example, it can be easily integrated into today's commercial database systems (Abel 1989; Gaede and Riekert 1994); it allows the application to control redundancy (Gaede 1995b) identify hits during the housekeeping step (Gaede 1995a); and it can be easily extended to higher dimensions.

For *z-ordering*, Orenstein and Manola (1988) adapted the well-known sort-merge join algorithm for two or more sets to support the *z-value*-based filter step. The logic of the intersection join using *z-values* is different from the regular sort-merge join, because the "numbers" (ie., *z-values*) now represent multidimensional intervals in native space. Thus, *z-values* can exhibit containment

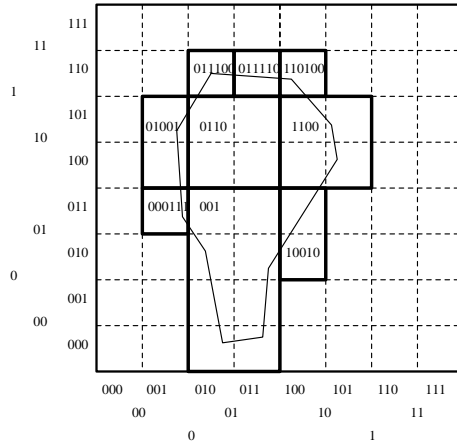


Fig. 2. Decomposition of a given object using the z-value approximation.

or enclosure. We illustrate the merge join using the example depicted in Figure 3. We indicate the spatial extent of the corresponding objects by rectangles. For simplicity, we assume each z-value is associated with a different object and that the spatial predicate is intersection. In our example, two input streams X and Y are given, each of which are sorted by their z-values. Z-values having the same prefix are ordered by their length.

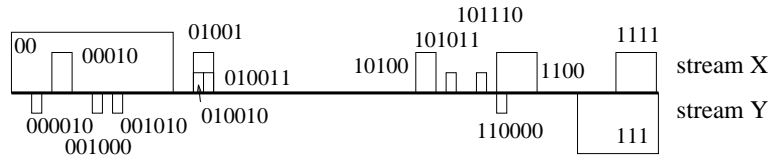


Fig. 3. Example of an intersection join of two sorted streams X and Y .

The first inspected pair is $(00, 000010)$ and because the Peano cell 00 contains 000010 , we place the pair in the candidate set. Since 00 may also contain other Peano cells we have to sweep over the Y stream up to the Peano cell 001111 (the last possibly enclosed z-value) to find them. As a result, we find the candidate pairs $(00, 001000)$ and $(00, 001010)$. Next, we have to advance to the next pair of possibly matching elements. The way we advance is critical for the overall performance of the merge, since it is possible to do better than linear performance by skipping several elements (Orenstein and Manola 1988; Aref and Samet 1994). In our example, we can skip $(00010, 001000)$ and $(01001, 001010)$. $(01001, 110000)$ does not qualify, because the cells have different prefixes. By repeating this procedure and paying particular attention to skipping, we eventually determine our set of candidate pairs.

3.2 R-Trees

An *R-tree* (Guttman 1984) represents a hierarchy of nested rectangles, each of which correspond to a disk page. If a disk page is an interior node, then the intervals corresponding to the descendants of the node are entirely contained in the interior node's interval. Intervals at the same tree level may overlap. If it is a leaf node, the interval represents the minimum bounding rectangle (MBR) of the objects stored in the node. For each spatial object in turn, the page only stores its MBR and a reference to the complete object description. Similar to B-trees, R-trees are height-balanced

and guarantee a minimum storage utilization by imposing a limit on the minimum number of entries. The most notable difference to B-trees is that even for exact match queries we may have to follow multiple paths from the root, because of the overlapping internal intervals. Thus, there is no non-trivial worst-case bound for the number of pages we have to visit. Insertion follows only a single path using some heuristic for choosing a subtree. As a result of the insertion, we may have to split nodes. Guttman (1984) discusses various policies to minimize the overlap during node splitting. Later work by other researchers led to the development of more sophisticated policies. For example, the split algorithm of the R*-tree (Beckmann, Kriegel, Schneider, and Seeger 1990) also optimizes for minimal overlap between intervals at the same level.

Figure 4 shows a set of spatial objects on the left, and an R-tree for them on the right.

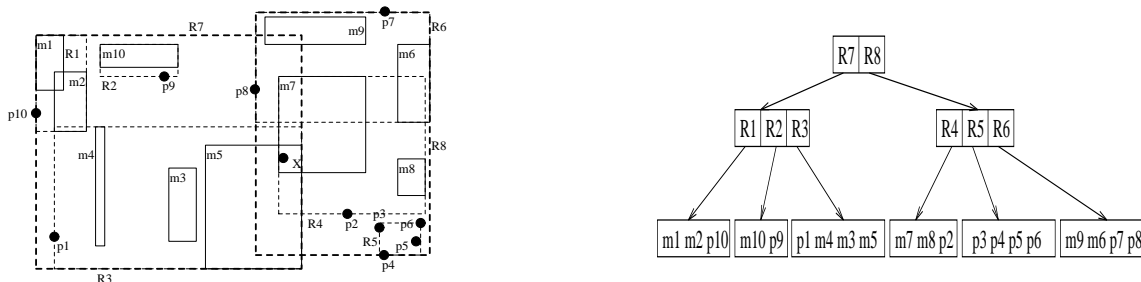


Fig. 4. R-Tree

For R-trees a spatial join algorithm known as *synchronized tree traversal* has been proposed independently by Günther (1993) and Brinkhoff et al. (1993). As the name suggests, the main idea of this join algorithm is to traverse both trees in a synchronized way using, for each level of the tree, the node information of one tree to restrict the search space of the other tree and vice versa. The algorithm works as follows: starting from the root nodes, the entries of the two nodes are compared with respect to the given spatial predicate so as to identify matching entries (subtrees). By recursively repeating this procedure for nodes of matching subtrees, we eventually reach the leaf level. As a result, we are left with a set of qualifying candidate pairs that are subject to the refinement step.

4 Sequencing Strategies

The sequence in which candidates are generated by the filter step clearly depends on the filter algorithm which in turn reflects the basic organisation of the spatial index. The I/O cost of the refinement and housekeeping steps will be influenced by the sequence of candidates. Intuitively, if the sequence of candidates has widely-scattered references to a particular object, and the candidates are processed in the sequence generated by the filter step, then several fetches of that object's description would be incurred. A *good* sequence would require few duplicated fetches. As we consider in more detail below, the presence of a *system buffer* which allows some fetches to be made from the buffer rather than from physical disk requires more careful analysis. More accurately, the *best* sequence of processing during refinement (under our objective of minimising I/O cost) will require fetching the fewest pages from disk to the system buffer.

4.1 Buffering and Caching

Architecture The effectiveness of the system buffer depends significantly on the match between the members of the operand sets (denoted by R and S , the i 'th member of R denoted by r_i and

similarly s_i) in the list of candidate pairs and the order in which they appear on disk. Essentially, if r_i and r_{i+1} are close in the ordered list of candidate pairs, and are also stored on the same disk page, it is highly likely that a fetch of r_{i+1} is satisfied from the system buffer.

In addition to the system buffer, we consider an *application cache* for storing spatial descriptions once fetched. Essentially, this is a private buffer, able to be specified and managed to suit the refinement process alone. It differs from the page-based system buffer by being an array of spatial descriptions organized by their corresponding object identifiers. We assume a least-recently-used (LRU) policy for managing the cache. We note that, while the system buffer is page-centered, the application cache is object-centered and so has a much finer granularity.

Clustering The organization of spatial descriptions on disk – known as *clustering* – has been considered in the literature (Brinkhoff and Kriegel 1994). Essentially, clustering attempts to ensure that objects which are close in native space will be close on disk. Clustering in this case inherently assumes that a linear ordering of objects is applied. The linear keys of the z-ordering techniques is one suitable strategy. It is also plausible that, for some data sets, the lexicographical sequence for object identifiers represents a contiguous sequence. For example, if the object identifier for a house is the concatenation of suburb name, street name and street number, then houses along a street, will be loosely clustered spatially if stored in sorted order by object identifier.

The argument for clustering, however, is based on the assumption that the object descriptions should be organized physically for maximal efficiency of spatial search. In our experience, this is not always desirable. In an extended relational database system, for example, the spatial description can be treated as just another attribute, so that spatial and other attributes are held in a single relation. The *best* logical and physical organization of the relation will then require consideration of all uses of the relation and it is possible that spatial operations will not be dominant. Spatial clustering of the relation would then not be appropriate.

Sequencing The order of the candidate pair list is determined by the filter algorithm. In the case of z-ordering, the order of candidate pairs will itself conform approximately to a z-order sequence. For indexes based on a containment hierarchy such as R-trees, the sequence will depend on the precise join algorithm and on the scan of a pair of leaf pages. The algorithm by Brinkhoff et al. (1993), for example, generates the candidate pairs in sequence by the minimum x -coordinate of objects from one relation and, within that, the minimum x -coordinate of the objects from the other relation. For cases in which the candidate pairs have been generated arbitrarily, the order of candidate pairs will not be known *a priori*.

The refinement test can be performed either immediately after a candidate pair is identified or can be deferred to a separate phase once all candidate pairs are produced (Patel and DeWitt 1996). Filtering algorithms based on access methods allowing one object to correspond to multiple index entries require duplicate removal during the housekeeping step. One obvious advantage of a separate phase in these cases is that duplicate candidate pairs can be removed by sorting the complete candidate pair list. For the immediate strategy to recognize duplicate candidate pairs, all the candidate pairs processed must be remembered. Though the computational overhead incurred by sorting and maintaining the candidate pairs in the deferred strategy might not pay off in all cases, in general it is worth the effort.

In the remainder of this section, we shall discuss the naive method as an example of an immediate strategy, and three examples – the *simple*, the *segmented* and the *zig-zag* – of deferred strategies.

4.2 Naive Strategy

The *naive* method processes candidate pairs in the order generated by the filter algorithm, ie., no additional sequencing is performed in the housekeeping step. If the objects referenced by a given candidate pair are not in the application cache, they are fetched from disk. Once the cache is full, objects are replaced using a least-recently-used policy: regardless of the set to which they belong,

the least recently used objects are removed from the cache and new objects are loaded. To see how this strategy works, let us consider the following example.

Example 1. Suppose that we have a cache size of at most four objects and the candidate pairs (r_i, s_j) are returned in the sequence shown in Figure 5 by the filter or housekeeping step. Applying the naive strategy to this sequence and assuming an initially empty application cache, we observe the behaviour shown in Figure 6. Here we assume that the application cache is split evenly between objects $r_i \in R$ and $s_j \in S$. The total number of objects loaded is 20, which is clearly not optimal.

1. (r_6, s_3)
2. (r_7, s_1)
3. (r_2, s_1)
4. (r_1, s_2)
5. (r_2, s_6)
6. (r_1, s_3)
7. (r_2, s_7)
8. (r_3, s_1)
9. (r_4, s_1)
10. (r_5, s_2)
11. (r_2, s_5)
12. (r_1, s_1)
13. (r_2, s_4)

Candidate pair	Cache content	No. of objects loaded
(r_6, s_3)	(r_6, s_3)	2
(r_7, s_1)	(r_6, r_7, s_3, s_1)	2
(r_2, s_1)	(r_7, r_2, s_3, s_1)	1
(r_1, s_2)	(r_2, r_1, s_1, s_2)	2
(r_2, s_6)	(r_1, r_2, s_2, s_6)	1
(r_1, s_3)	(r_2, r_1, s_6, s_3)	1
(r_2, s_7)	(r_1, r_2, s_3, s_7)	1
(r_3, s_1)	(r_2, r_3, s_7, s_1)	2
(r_4, s_1)	(r_3, r_4, s_7, s_1)	1
(r_5, s_2)	(r_4, r_5, s_1, s_2)	2
(r_2, s_5)	(r_5, r_2, s_2, s_5)	2
(r_1, s_1)	(r_2, r_1, s_5, s_1)	2
(r_2, s_4)	(r_1, r_2, s_1, s_4)	1

Fig. 5. Candidate pairs as returned by the filter or housekeeping step.

Fig. 6. Order of execution for the naive strategy.

4.3 Simple Strategy

A simple but effective approach is to defer the refinement step until all candidate pairs are available, to sort the list of candidate pairs by their object identifiers $r_i \in R$, say, and then to work sequentially down the list. Such a policy ensures that there are no replicated fetches on the members of R , but there can be disk I/O cost associated with the sort if the number of candidate pairs is very large. Clearly, some S objects still have to be fetched multiple times.

Example 2 shows the order and performance of the simple strategy for the sequence given in Figure 5. The simple technique outperforms the naive, since the total number of objects loaded is 17.

Example 2. This example shows the execution trace for the simple strategy, which sorts the candidate pairs (r_i, s_j) by $r_i \in R$, paying no particular attention to s_j . In this example, we assume that only one object $r_i \in R$ is kept in the application cache and the remaining space is used by s_j objects. The total number of objects loaded is 17.

Candidate pair	Cache content	No. of objects loaded
(r_1, s_1)	(r_1, s_1)	2
(r_1, s_2)	(r_1, s_1, s_2)	1
(r_1, s_3)	(r_1, s_1, s_2, s_3)	1
(r_2, s_1)	(r_2, s_2, s_3, s_1)	1
(r_2, s_4)	(r_2, s_3, s_1, s_4)	1
(r_2, s_5)	(r_2, s_1, s_4, s_5)	1
(r_2, s_6)	(r_2, s_4, s_5, s_6)	1
(r_2, s_7)	(r_2, s_5, s_6, s_7)	1
(r_3, s_1)	(r_3, s_6, s_7, s_1)	2
(r_4, s_1)	(r_4, s_6, s_7, s_1)	1
(r_5, s_2)	(r_5, s_7, s_1, s_2)	2
(r_6, s_3)	(r_6, s_1, s_2, s_3)	2
(r_7, s_1)	(r_7, s_2, s_3, s_1)	1

4.4 Segmented Strategy

Valduriez (1987) proposed an efficient sequencing method for join processing using join indices, which has been recently adapted by Patel and DeWitt (1996) to minimize the cost of the refinement step for spatial joins. We refer to this method as *segmented* sequencing. A segment is defined dynamically by the maximum number of R objects that can be loaded into the application cache such that there is enough space left for at least one spatial object of S . The details of the segmented algorithm are described in Algorithm 4.4.

Algorithm 1 Segmented Sequencing

Given a list of candidate pairs (r_i, s_j) , return pairs of objects satisfying the given spatial predicate after performing a segmented sort.

- S1.** [Sort by r_i] Sort the candidate list by r_i .
 - S2.** [Read segment] Read a subset R_s of R into memory such that there is enough space left for at least one object from S .
 - S3.** [Sort R_s] Let S_r denote the set of different object identifiers of members of S which appear in the tuples of R_s . Sort R_s by $s_j \in S_r$.
 - S4.** [Refinement step] Fetch the different objects $s_j \in S_r$ one by one and pass them to the refinement step.
 - S5.** [Repetition] If no candidate pairs are left, return result, otherwise go to **S2**.
-

Segmented sequencing guarantees that no object $r_i \in R$ within a segment is loaded twice. However, objects in S appearing in several segments are likely to be fetched multiple times. Example 3 shows the trace for segmented sequencing, corresponding to the input sequence of Figure 5. For our example, the segmented strategy performs worse than the simple strategy but better than naive, loading 18 objects.

Example 3. This example shows the behaviour of the segmented strategy. The total number of objects loaded is 18.

Candidate pair	Cache content	No. of objects loaded
(r_1, s_1)	(r_1, r_2, r_3, s_1)	4
(r_2, s_1)	(r_1, r_2, r_3, s_2)	0
(r_3, s_1)	(r_1, r_2, r_3, s_3)	0
(r_1, s_2)	(r_1, r_2, r_3, s_2)	1
(r_1, s_3)	(r_1, r_2, r_3, s_3)	1
(r_2, s_4)	(r_1, r_2, r_3, s_4)	1
(r_2, s_5)	(r_1, r_2, r_3, s_5)	1
(r_2, s_6)	(r_1, r_2, r_3, s_6)	1
(r_3, s_7)	(r_1, r_2, r_3, s_7)	1

Load next segment

(r_4, s_1)	(r_4, r_5, r_6, s_1)	4
(r_5, s_2)	(r_4, r_5, r_6, s_2)	1
(r_6, s_3)	(r_4, r_5, r_6, s_3)	1

Load next segment

(r_7, s_1)	(r_4, r_5, r_7, s_1)	2
--------------	------------------------	---

From the above discussion it becomes clear that both the simple and segmented strategies favor one set of objects over the other. This policy is advantageous if one set is much larger than the other or if there is high skew in one data set. However, where the query optimizer does not know the resulting distribution *a priori*, a more balanced approach is attractive.

In the next section, we describe a new algorithm that dynamically balances the two sets. The new strategy is called *zig-zag*.

4.5 Zig-Zag Strategy

The zig-zag algorithm is aimed at maximising the effectiveness of the application cache by guiding the traversal of the list of candidates by the contents of the application cache. The approach adopted assumes that objects of one operand set which are close in space are likely to have similar sets of intersecting objects from the other set. Informally, if two members of R are close spatially and appear in candidate pairs, those candidate pairs are likely to have common members of S , and similarly for two close members of S . This suggests that an efficient traversal sequence is to proceed through the candidate list by those pairs, alternately, with close members of S and with close members of R . More precisely, the zig-zag strategy is intended to perform best when the following condition holds:

If an object $r_i \in R$ is (spatially) close to objects $\{s_1, s_2, \dots, s_n\} \subseteq S$ and s_1 is close to $r_j \in R$ ($i \neq j$), then it is likely that r_j is also close to s_2, \dots, s_n .

The thrust of the zig-zag algorithm is to put an emphasis on one of the two sets, say S , as long as there are S objects in the cache which have not yet been processed completely with respect to the candidate list. If all cache objects of one set are processed, the algorithm switches to the respective other set and processes all cache objects of this set. By zig-zagging between both sets we alternate the emphasis on one of the two relations, yielding a much more balanced execution than the other two deferred strategies.

The zig-zag method uses two lists F and F' . Each entry $e \in F$ stores a candidate pair (r_i, s_j) along with a flag indicating whether this pair has already been processed or not. This flag is necessary, since in the course of zig-zagging, we may skip certain pairs which have to be processed later. We also maintain a pair of cursors, one of which points to the element $e \in F$ that has been processed before we zig-zagged and the other, pointing to the next element to be processed. The F' list consists of entries having the following form (s_j, p_k) , with p_k being a pointer to an entry $e_k \in F(r_i, s_j)$ for some i . The details of the zig-zag algorithm are described in Algorithm 2.

Note that the objects $s_j \in S$ are loaded in the order in which they appear in F and no further sequencing is performed. Applying Algorithm 2 to the sequence given in Figure 5 results in the

Algorithm 2 Zig-Zag

Given a list of candidate pairs F , return all pairs of objects satisfying the given spatial predicate using the zig-zag strategy.

- Z1.** [Sort F] Sort F by r_i .
- Z2.** [Sort F'] Initialize F' such that each $(r_i, s_j) \in F$ there is a corresponding entry $(s_j, p_k) \in F'$ with p_k pointing to $e_k \in F$. Next sort F' by s_j .
- Z3.** [Load (r_i, s_j)] Unless all entries in F are processed, load the next unprocessed object pair $(r_i, s_j) \in F$ into the cache and pass them to the refinement step.
- Z4.** [Load s_j] For each object $r_i \in R$ which is in the application cache, load the corresponding unprocessed object $s_j \in S$ as given by the entries in $e_k \in F$ if and only if s_j is not already present in the cache. Pass each pair (r_i, s_j) to the refinement step. Mark e_k as being processed. Repeat this step until there is no unprocessed r_i in the buffer.
- Z5.** [Load r_i] If there is an object $s_j \in S$ in the application cache, for which at least one $e_j \in F'$ points to an unprocessed candidate pair in $e_k \in F(r_i, s_j)$ then load the corresponding $r_i \in R$ and pass (r_i, s_j) to the refinement step. After they have been processed mark e_k as being processed. Repeat this until there is no unprocessed s_j in the application cache. If there are any unprocessed r_i 's in the application cache go to **Z4**, else go to **Z3**.
-

trace shown in Example 4. The zig-zag strategy performs as well as the simple and better than the segmented strategy.

Example 4. This example shows the performance of the zig-zag technique. The total number of objects loaded is 17.

Candidate pair	Cache content	No. of objects loaded
(r_1, s_1)	(r_1, s_1)	2
(r_1, s_2)	(s_1, r_1, s_2)	1
(r_1, s_3)	(s_1, s_2, r_1, s_3)	1
Change to S		
(r_2, s_1)	(s_2, s_3, s_1, r_2)	1
(r_3, s_1)	(s_3, r_2, s_1, r_3)	1
(r_4, s_1)	(r_2, r_3, s_1, r_4)	1
(r_7, s_1)	(r_3, r_4, s_1, r_7)	1
Go back to R		
(r_2, s_4)	(r_4, r_7, r_2, s_4)	2
(r_2, s_5)	(r_7, s_4, r_2, s_5)	1
(r_2, s_6)	(s_4, s_5, r_2, s_6)	1
(r_2, s_7)	(s_5, s_6, r_2, s_7)	1
Go to S		
(r_5, s_2)	(s_6, s_7, s_2, r_5)	2
(r_6, s_3)	(s_2, r_5, s_3, r_6)	2

5 Performance Evaluation

In order to investigate the effect of the different caching strategies, we ran several experiments on the SEQUOIA data set (Stonebraker, Frew, and Dozier 1993). Clearly, the performance figures given in this section depend on the characteristics of the data, but it is likely that the trends

observed will hold for most spatial data. To get a better understanding of the different factors governing the performance, we followed a factorial experimental design to assess the influence of such aspects as varying data characteristics.

5.1 Data Sets

Polygons for the 37 land uses from the SEQUOIA data set were joined with the islands polygon set to determine the land use polygons enclosing the island polygons. A small number of land use polygons (270) and island polygons (66) were discarded because they failed the geometric integrity of the SIRO-DBMS (Abel 1989) used to implement the z-order filter, typically because of self-intersecting boundaries. In addition to its spatial extent, we assigned to each spatial object a unique identifier (assigned serially within each land use), a land use code, and a random number. By examining a display of the objects generated in order by identifier, we found that, within each land use, polygons with consecutive identifiers were typically close spatially. Summary characteristics of the test data sets are reported in Table 1. The join problem considered has 80059 candidate pairs from the filter step.

Data Set	No. of polygons	No. of vertices			Size in MBytes
		Avg.	Min	Max	
Land use	58316	50.1	4	5538	88.5
Islands	20955	31.4	4	2461	17.5

Table 1. Parameters of the data sets used.

5.2 Experimental Setup

In our experiments, we assumed that the application cache and the system buffer are specified independently and are managed solely by the join operation. (That is, there is no contention from other users for the system buffer.) The application cache size was varied from 0.1% (79 objects, 28 Kbytes) to 2.5% (1981 objects, 696 Kbytes) of the combined size of the operand data sets. The system buffer size was varied from 10 pages (80Kbytes) to 510 pages (4.08 Mbytes).

We tested combinations of these parameters with six filter algorithms: the R-tree (Guttman 1984) and R*-tree (Beckmann et al. 1990), each with and without plane sweep ordering of the traversal of trees (Brinkhoff et al. 1993), z-ordering with one key and with at most four keys. Four sequencing methods were evaluated: the naive, the simple, the segmented and the zig-zag. In terms of the I/O cost for refinement, we found no significant differences between the R-tree and the R*-tree, with or without plane sweep. We also confirmed that z-ordering with at most 4 keys always outperforms z-ordering with a single key. We therefore restrict our reporting of the observed performance to the cases of the R*-tree with plane sweep and z-ordering with at most four keys as the filter algorithms.

Two forms of clustering were simulated For z-order clustering, the objects were sorted by their z-order (single) locational key and object identifiers assigned serially to the sorted list. For unclustered data, the object identifiers were stored in sequence by a randomly-assigned integer.

Note that the R*-tree and z-order filters produce identical sets of candidates, although in different sequences. The deferred methods (the simple, the segmented and the zig-zag) each re-sort the list of candidates. We report, then, our results for the direct and naive methods for candidates from the R*-tree and z-order filters (separately), and for the simple, segmented and zig-zag methods (where the results are independent from the choice of filter algorithm).

5.3 Baseline Performance

Two baseline performance figures are of interest. The first is performance for an immediate strategy, without an application cache- the *direct* method. This corresponds to algorithms which process candidates immediately on generation without attention to minimising the refinement cost. It allows comparison of the techniques presented in this paper with most algorithms reported to date. The second is the floor cost for refinement which allows assessment of the absolute effectiveness of the techniques.

The direct method was tested for all combinations of filter algorithm, clustering and system buffer size.

The floor cost was estimated as the minimum number of disk accesses needed to fetch all objects included in the candidate tuples. This was measured simply as the number of pages containing one or more objects in the candidate tuples. The floor costs were 3604 accesses for the unclustered data set and 3593 for z-order clustering.

5.4 Results

Varying Buffers, Unclustered Data Our first tests dealt with unclustered data. Note that the R*-tree and z-order filters produce identical sets of candidates, although in different sequences. The deferred methods (the simple, the segmented and the zig-zag) each re-sort the list of candidates. We report, then, our results for the direct and naive methods for candidates from the R*-tree and z-order filters (separately), and for the simple, segmented and zig-zag methods (where the results are independent from the choice of filter algorithm). Note also that, in the figures reporting the dependence of performance of a method on application cache size, the curves are labelled by the maximum number of objects able to be held in the cache.

The performance of the direct method operating on candidates from z-order and R*-tree filters (Figures 7 and 8) the expected influence of the system buffer. A buffer of 510 pages leads to a refinement cost by 40% to 50% lower than a very small buffer. Processing candidates from a z-order filter is also cheaper than processing R*-tree filter candidates. The explanation is that the z-order filter localises references to objects in the list of candidates more strongly than the R*-tree filter.

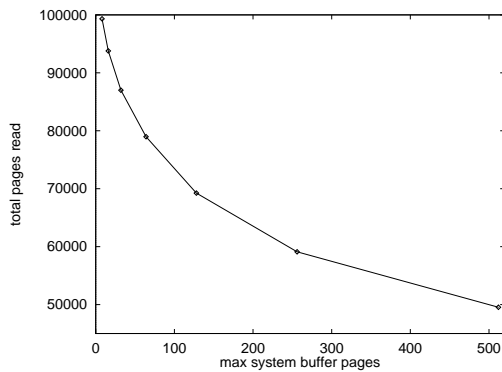


Fig. 7. Direct method with z-order filter and unclustered data.

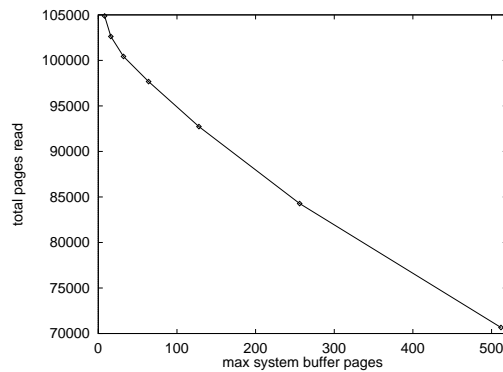


Fig. 8. Direct method with R*-tree filter and unclustered data

The performance of the naive method (Figures 9 and 10) shows the effect of adding an application cache to the direct method. The cache is useful for both the z-order and R*-tree filter. For candidates from both filters, the cost reduction is sharply asymptotic with respect to the cache size, and the nett effectiveness of the cache falls with increasing system buffer size. It is better to use main memory to increase the application cache rather than the system buffer. The difference

in costs for processing candidates from the two filters, however, remains over the range of system buffer sizes tested.

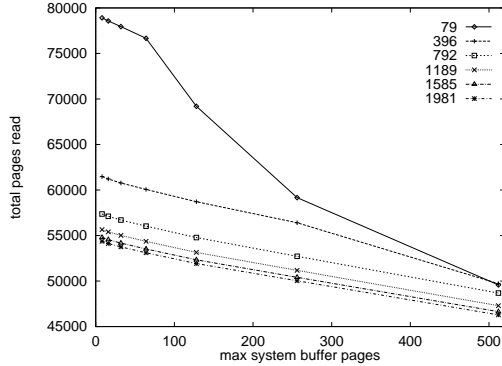


Fig. 9. Naive method with z-order filter and unclustered data.

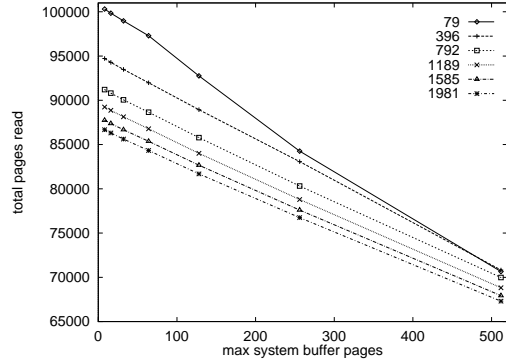


Fig. 10. Naive method with R*-tree filter and unclustered data.

The simple and segmented methods (Figures 11 and 12) resequence the candidates to traverse the candidates by the identifiers for objects from one of the operand data sets. The segmented method performs more efficiently for all combinations of applications cache and systems buffer sizes. Performance is near identical for small cache and buffer sizes, but the segmented method is more sensitive to increases in both. For a large cache and a large system buffer, the segmented method has a refinement cost lower by approximately 15%. Compared to the naive method, the segmented method is superior only when the candidates have been generated by an R*-tree filter and a large system buffer (of greater than 350 pages) is available.

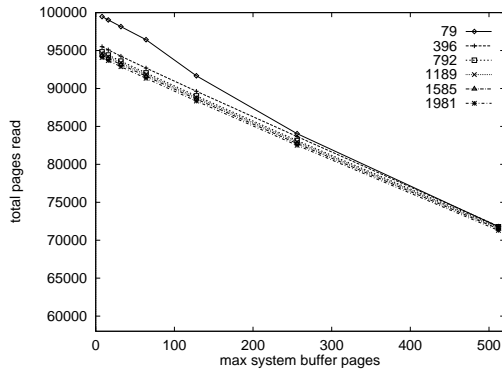


Fig. 11. Simple method with unclustered data.

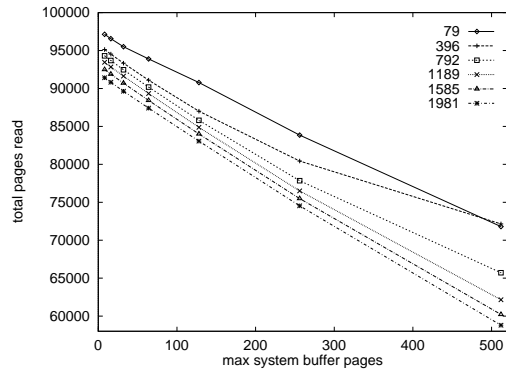


Fig. 12. Segmented method with unclustered data.

The zig-zag method (Figure 13) is essentially a modification of the simple method which seeks to balance traversal of the candidates. Its observed behaviour on unclustered data, however, is closely similar to that of the naive method when operating on candidates from a z-order filter. It is also strongly asymptotic with respect to the size of the application cache and is less sensitive than the other deferred methods to the size of the system buffer. It has slightly better performance than the naive method, provided a large application cache is available, for both cases of filter algorithms. It has markedly superior performance when processing candidates from an R*-tree filter.

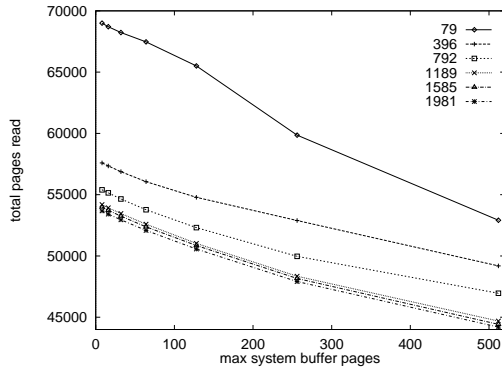


Fig. 13. Zig-zag method with unclustered data.

Varying Buffers, Clustered Data We repeated the tests with the data sets clustered by z-order keys.

The direct method (Figure 14), when processing candidates from the z-order filter, now has a cost (for system buffers of greater than 150 pages) which is very close to the floor cost. This is explained by the match between the sequence of candidates generated by the z-order filter and the stored order on disk of the spatial descriptions. The match is not present for candidates from the R^* -tree filter (Figure 15), and the cost is higher by a factor of four. The cost falls with increasing system buffer size over the range tested. However, comparison of Figures 8 and 15 shows that the costs of processing R^* -tree candidates are lower when the data is clustered by z-order keys and that the reduction increases with increasing system buffer size. The explanation is that, with clustered data, a page loaded from disk has a higher likelihood (compared to the unclustered case) of containing descriptions of objects to be encountered later. (Quite simply, if r_1 and r_2 are close spatially, their descriptions are likely to be stored on the same page. If r_1 is found in a candidate tuple with s_1 , it is likely that s_2 will later be found in a candidate tuple with s_2 .) There is then a higher likelihood of the first fetch of an object's description being satisfied from pages already in the system buffer, rather than needing a fetch from disk (Figure 16). There is a flow-on effect in that a new page will not be added to the system buffer, so that a page will stay in the system buffer longer. Second and subsequent fetches of descriptions are also then more likely to be satisfied without disk accesses (Figure 17).

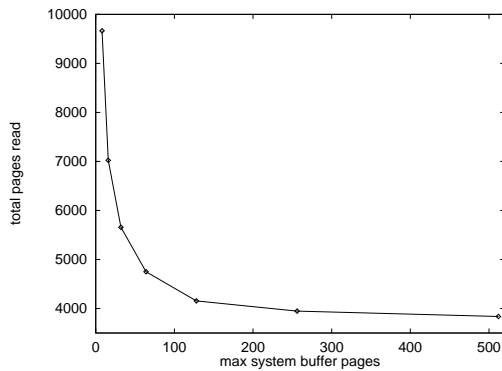


Fig. 14. Direct method with z-order filter and z-order clustering

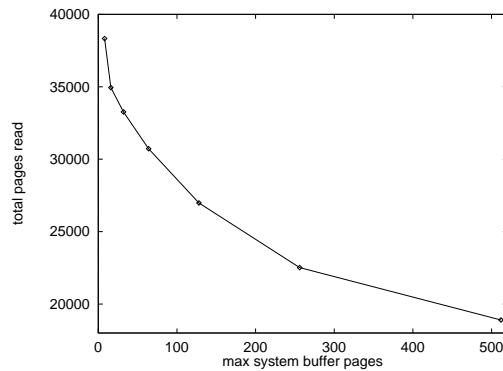


Fig. 15. Direct method with R^* -tree filter and z-order clustering

As the direct method has near-optimum performance for candidates from the z-order filter and

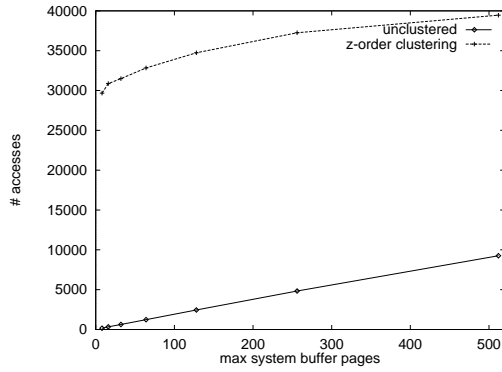


Fig. 16. First hits on system buffer

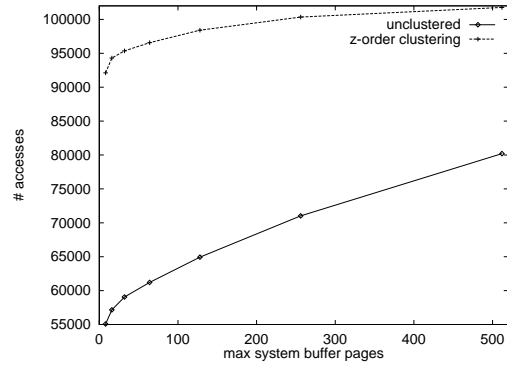


Fig. 17. Subsequent hits on system buffer

for medium-sized system buffers, the only opportunity for the naive method (Figure 18) with its application cache to provide improvements is when the system buffer is small. This is confirmed by Figure 18. There is an improvement for very small system buffers, but the floor cost is still achieved at about 150 pages. When the candidates are generated by an R^* -tree filter (Figure 19), the naive method behaves very similarly to the direct method, with slight advantages for small to medium system buffers.

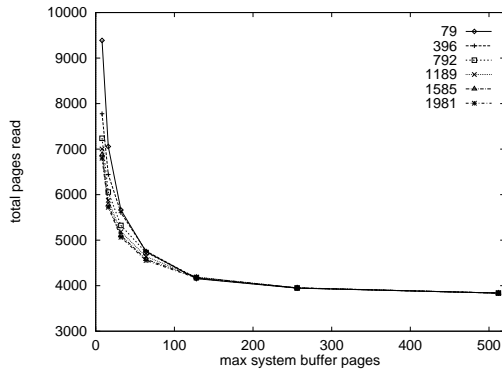


Fig. 18. Naive method with z-order filter and z-order clustering.

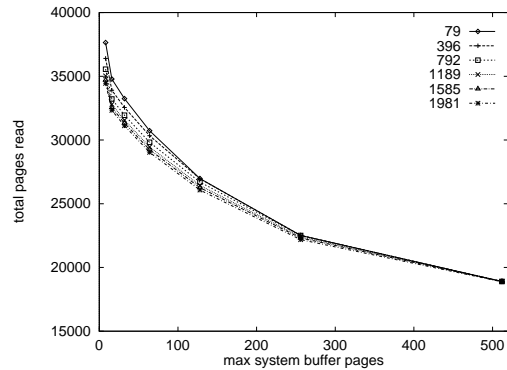


Fig. 19. Naive method with R^* -tree filter and z-order clustering.

The simple and segmented methods (Figures 20 and 21) also benefit from z-order clustering, with refinement costs lower by approximately 50%. Their behaviour is strongly asymptotic with respect to the system buffer size: there is little difference in refinement costs with system buffers in the range from 150 to 510 pages. The segmented method continues to have a slight performance advantage over the simple method, but is unable to match the direct method. The zig-zag method (Figure 22) has intermediate performance. It also has asymptotic performance with respect to the size of the system buffer, but continues to have refinement costs falling up to the greatest system buffer size examined. With a system buffer size of 510 pages, however, it does not match the performance of the direct method.

The zig-zag method shows unexpected crossover in its cost curves for some combinations of application cache and system buffer sizes. The explanation is as follows. If the data set contains large objects, zig-zag tends to “drift away” from its initial location in space by loading possible matches for these large objects and in turn possible matches for the just loaded objects and so on. This keeps going on until the application cache is exhausted and step **Z3** of Algorithm 2 is

executed again. At this point, however, most initially adjacent objects have already been flushed out of the buffer and we have to reload them at the expense of additional page accesses.

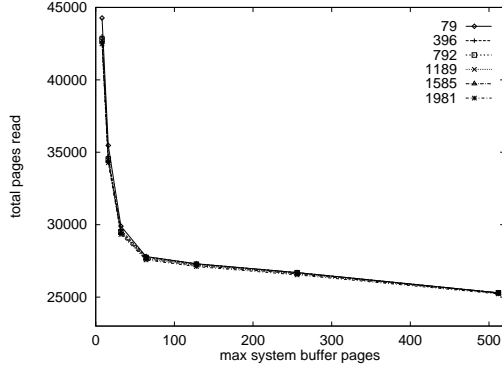


Fig. 20. Simple method with z-order clustering.

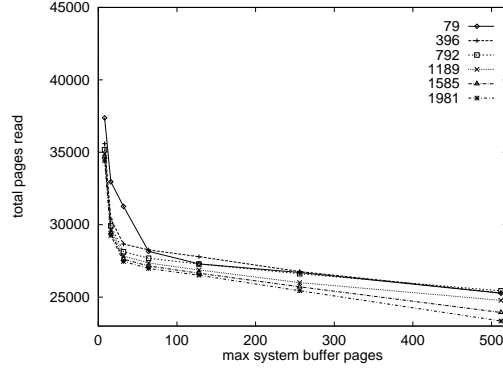


Fig. 21. Segmented method with z-order clustering.

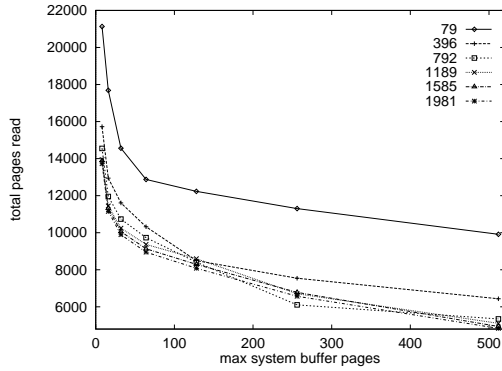


Fig. 22. Zig-zag method with z-order clustering.

6 Conclusions and Future Work

This paper has contributed to the investigation of spatial join algorithms through consideration of the refinement stage. The usual filter-and-refine strategy has been extended, following the work of Patel and DeWitt (1966), to a filter, sequence and refine strategy, where the sequence step reorders the candidates to minimise I/O cost during refinement. A new sequencing algorithm, called the zig-zag, has been described. Assessment has included testing the effects of the choice of filter algorithm, of system buffer size and of the application cache. Clearly, care must be exercised in drawing generalized conclusions from tests on a single data set. Some very influential effects appear to be present.

Clustering has demonstrated as highly influential on refinement performance. In the case where the sequence of candidates from the filter step matches the stored order of spatial descriptions on disk, costs very close to the floor were achieved. When no such match is possible (i.e. the data is unclustered), the refinement costs were approximately ten times the floor cost. The match

between sequence of candidates and the stored order of candidates also makes the system buffer size surprisingly insignificant: a system buffer of approximately 150 pages (1.2 Mbytes) allows the minimum refinement cost to be reached. This result is significant. Our tests have taken the system buffer to be not subject to contention by other users. In practice, the system buffer is a resource shared by many users and the length of time for which a disk page remains in the buffer depends on the disk activity by all users. A low sensitivity to system buffer size is useful as performance of the join algorithm will not degrade markedly as a result of increasing contention for the system buffer in multi-user environments.

The empirical assessment suggests that there are simple rules to guide selection of strategy and of sequencing method for combinations of the given environmental parameters of clustering, filter algorithm and system buffer size. The zig-zag method has superior performance in all but one case. That case is when the spatial descriptions are clustered by z-order keys and a z-order filter has been used to generate the candidates. Our diagnosis of a weakness in the zig-zag method's behaviour suggests that further development of the algorithm might be productive.

We have investigated clustering in terms only of ordering the stored data by z-order keys. Other forms are possible, including ordering by the coordinates of the minimum bounding rectangles for objects and ordering by a regular grid. It is plausible that there exists an ordering which is compatible with R*-tree filters in the same way that a z-order filter is compatible with clustering by z-order keys. As the R*-tree is used widely as a spatial access method, discovery of such an ordering would be practically significant.

Acknowledgments

We thank our colleagues, Dean Kuo and Kerry Taylor, for their insightful and careful comments on earlier forms of this paper.

References

- Abel, D. (1993). Some evolutionary paths for spatial databases. In *Int. Symp. on Next Generation Databases and their Applications NDA '93*, Fukuoka, pp. 1–10.
- Abel, D. J. (1989). SIRO-DBMS: A database toolkit for geographical information systems. *Int. J. Geographical Information Systems* 4(3), 443 – 464.
- Aref, W. G. and H. Samet (1994). The spatial filter revisited. In *Proc. 6th Int. Symp. on Spatial Data Handling*, pp. 190–208.
- Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger (1990). The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 322–331.
- Brinkhoff, T., H. Horn, H.-P. Kriegel, and R. Schneider (1993). A storage and access architecture for efficient query processing in spatial database systems. In *Advances in Spatial Databases*, Number 692 in LNCS, Berlin/Heidelberg/New York, pp. 357–376. Springer-Verlag.
- Brinkhoff, T. and H.-P. Kriegel (1994). The impact of global clustering on spatial database systems. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pp. 168–179.
- Gaede, V. (1995a). Geometric information makes spatial query processing more efficient. In *Proc. 3rd ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, Baltimore, Maryland, USA, pp. 45–52.
- Gaede, V. (1995b). Optimal redundancy in spatial database systems. In M. J. Egenhofer and J. R. Herring (Eds.), *Proc. 4th Int. Symp. on Spatial Databases (SSD'95)*, Number 951 in LNCS, Berlin/Heidelberg/New York, pp. 96–116. Springer-Verlag.

- Gaede, V. and O. Günther (1995). Survey on multidimensional access methods. Technical Report ISS-15, Humboldt-Universität zu Berlin, Germany. submitted.
- Gaede, V. and W.-F. Riekert (1994). Spatial access methods and query processing in the object-oriented GIS GODOT. In *Proc. of the AGDM'94 Workshop*, Delft, The Netherlands, pp. 40–52. Netherlands Geodetic Commission.
- Gambosi, G., M. Scholl, and H.-W. Six (Eds.) (1991). *Geographic Database Management Systems*, Berlin/Heidelberg/New York. Springer.
- Günther, O. (1993). Efficient computation of spatial joins. In *Proc. 9th IEEE Int. Conf. on Data Eng.*, pp. 50–59.
- Gütting, R. H. and W. Shilling (1987). A practical divide and conquer algorithm for the rectangle intersection problem. *Information Science* 42, 95–112.
- Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 47–54.
- Lo, M. and C. Ravishankar (1994). Spatial joins using seeded trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 209–220.
- Lo, M. and C. Ravishankar (1995). Generating seeded trees from data sets. In M. J. Egenhofer and J. R. Herring (Eds.), *Proc. 4th Int. Symposium on Advances in Spatial Databases (SSD'95)*, Volume 951 of LNCS, pp. 328–347. Springer.
- Lo, M. and C. Ravishankar (1996). Spatial hash-join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, pp. 247–258.
- Orenstein, J. (1989). Strategies for optimizing the use of redundancy in spatial databases. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 115–134. Springer-Verlag.
- Orenstein, J. and F. A. Manola (1988). Probe spatial data modeling and query processing in an image database application. *IEEE Trans. Software Eng.* 14(5), 611–629.
- Patel, J. M. and D. J. DeWitt (1996). Partition based spatial-merge join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, pp. 259 – 270.
- Stonebraker, M. (Ed.) (1994). *Readings in Database Systems*, San Mateo. Morgan Kaufmann.
- Stonebraker, M., J. Frew, and J. Dozier (1993). The SEQUOIA 2000 project. In *Advances in Spatial Databases*, Number 692 in LNCS, Berlin/Heidelberg/New York, pp. 397–412. Springer-Verlag.
- Valduriez, P. (1987). Join indices. *ACM Trans. Database Systems* 12(2), 219 – 246.