Efficient Join-Index-Based Spatial-Join Processing: A Clustering Approach

Shashi Shekhar, Chang-tien Lu, Sanjay Chawla
Computer Science Department, University of Minnesota
200 Union Street SE, Minneapolis, MN-55455
[*shekhar, ctlu, chawla*]@cs.umn.edu TEL:(612) 6248307 FAX:(612)6250572
http://www.cs.umn.edu/Research/shashi-group

Sivakumar Ravada
Spatial Data Product Division
One, Oracle Drive, Nashua, NH 03062
sravada@us.oracle.com

March 21, 2001

Abstract

A join-index is a data structure used for processing join queries in databases. Join-indices use pre-computation techniques to speed up online query processing and are useful for data-sets which are updated infrequently. The I/O cost of join computation using a join-index with limited buffer space depends primarily on the page-access sequence used to fetch the pages of the base relations. Given a join-index, we introduce a suite of methods based on clustering to compute the joins. We derive upper-bounds on the lengths of the page-access sequences. Experimental results with Sequoia 2000 data sets show that the clustering method outperforms existing methods based on sorting and online-clustering heuristics.

Acronym	Full form	Definition section/page
PCG	Page-Connectivity Graph	Section 1
OPAS-FB	Optimal Page-Access Sequence with Fixed Buffer	Section 1
AC	Asymmetric Clustering-based heuristic	Section 2
Sorting	Sorting-based heuristic	Section 2
SC	Symmetric Clustering based heuristic	Section 3
FP	Fotouhi and Pramanik's heuristic	Section 3
OM	Omiecinski's heuristic	Section 3
CO	Chan and Ooi's heuristic	Section 3
B-diagonal entry $M[i,j]$	$\|i-j\| \leq \lfloor \frac{B}{2} \rfloor$	Section 3

Table 1: Table of Acronyms

Keywords: Optimal Page Access Sequence, Join Index, Join Processing, Spatial Join.

1 Introduction

The join operation is a fundamental operation in relational databases, and substantial work has been done in optimizing join operations [15, 32]. A join-index [35, 41] is a special data structure that facilitates rapid join-query processing. For data sets which are updated infrequently, the join-index can be particularly useful.

The join-index is typically represented as a bipartite graph between the pages of encumbent relations or their surrogates. When the number of buffer pages is fixed, the join-computation problem is transformed into determining a page-access sequence such that the join can be computed with the minimum number of redundant page accesses. This problem has been shown to be NP-hard [31, 34], and consequently, it is unlikely that a polynomial time solution exists for this problem. Solutions in the literature use a clustering method that groups pages in one or both tables involved in the join to reduce total page accesses. Available heuristics either group the pages of a single table via sorting [41] or use incremental clustering methods [7, 11, 33].

Our Contribution: We introduce two new heuristics for this problem. One heuristic uses the clustering method to group the pages in one table, generalizing the sorting-based heuristic for joins. The other heuristic uses clustering for the pages of both tables. The former generalizes and outperforms the sorting heuristic, while the latter generalizes and outperforms the incremental clustering methods for joins. We provide a formal characterization of an upper bound on the number of redundant I/Os performed by our approaches. Experiments with the Sequoia 2000 [40] data-set show that both heuristics outperform other methods when the memory size is relatively small. The proposed approaches are useful for computing joins, given join-indices for large database, where the size of memory is small compared to the sizes of the individual data sets.

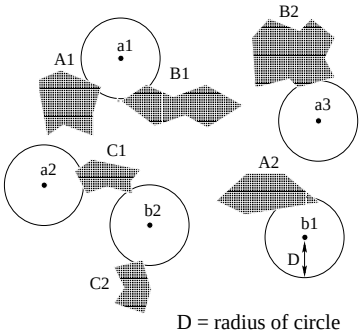
1.1 Join Index: Basic Concept

Consider a database with two relations, Facility and Forest Stand. Facility has a point attribute representing its location, and Forest Stand has a rectangle attribute that represents its extent by a bounding box. The polygon representing its extent may be stored separately. A point is represented by the x and y coordinates on the map. A rectangle is represented by points that represent the bottom left and top right corners.

In Figure 1(a), points $a1, a2, a3, b1, b2$ represent facility locations, and polygons $A1, A2, B1, B2, C1, C2$ are the bounding boxes that represent the limits of the forest stands. The circle around each location shows the area within distance D from a facility. The rectangle around each forest boundary represents the Minimal Orthogonal Bounding Rectangle(MOBR) for each forest stand. Figure 1(b) shows two relations, R and S, for this data set. Relation R represents facilities via the attributes of a unique identifier, $R.ID$, the location (x,y coordinates), and other non-spatial attributes. Relation S represents the forest stands via a unique identifier, $S.ID$, the MOBR and non-spatial attributes. MOBR$(X_{LL}, Y_{LL}, X_{UR}, Y_{UR})$, is represented via the coordinates of the lower-left corner point(X_{LL}, Y_{LL}) and the upper right corner point(X_{UR}, Y_{UR}). Now, consider the following query: **Q**: "Find all forest stands which are within a distance D from each facility." This query will require a join on the Facility and Forest Stand relations, based on their spatial attributes. A spatial join is more complex than an equi-join and is a special case of a Θ-join, where Θ is a spatial predicate,

e.g., touch, overlap, and cross. The query **Q** is an example of a spatial join.

A spatial join algorithm [2, 5, 6, 16, 30] may be used to find the pairs (Facility, Forest-stand) which satisfy query **Q**. Alternatively, a join-index may be used to materialize a subset of the result to speed up processing for the future occurrence of **Q**, if there are few updates to the spatial data. Figure 1(b) shows a join-index with two columns. Each tuple in the join-index represents a tuple in the table $JOIN(R, S, distance(R.Location, S.MOBR) < D)$. In general, the tuples in a join-index may also contain pointers to the pages of R and S where the relevant tuples of R and S reside. We omit the pointer information in this paper to simplify the diagrams.



(a) Spatial attribute of R and S

R Relation (Facility Location)

ID	Location (X,Y)	Non-Spatial Data
a1	(7.9,16.7)	(...)
a2	(3.4,11.4)	(...)
a3	(19.5,13.1)	(...)
b1	(18.7,6.4)	(...)
b2	(9.5,7.1)	(...)

S Relation (Forest-Stand Boundary)

ID	MOBR ($X_{LL}Y_{LL}X_{UR}Y_{UR}$)	Non-Spatial Data
A1	(3,12.2,6.6,16)	(...)
A2	(13.4,7.7,19.5,10)	(...)
B1	(7.6,12.7,15,5.2)	(...)
B2	(15.5,15,20.4,19)	(...)
C1	(5.1,8.9,9.1,10.9)	(...)
C2	(7.5,2,9.7,15.1)	(...)

Join-Index

R.ID	S.ID
a1	A1
a1	B1
a2	C1
a3	B2
b1	A2
b2	C1
b2	C2

(b) R and S relation table and join-index

Figure 1: Constructing a join-index from two relations

A join-index describes a relationship between the objects of two relations. Assume that each tuple of a relation has a surrogate (a system-defined identifier for tuples, pages, etc.) which uniquely identifies that tuple. A join-index is a sequence of pairs of surrogates in which each pair of surrogates identifies the result-tuple of a join. The tuples participating in the join result are given by their surrogates. Formally, let R and S be two relations. Then consider the join of R and S on attributes A of R and B of S. The join-index is thus an abstraction of the join of the relations. If F defines the join predicate, then the join-index is given by the set $JI = \{(r_i, s_j) | F(r_i.A, s_j.B)\ is\ true\ for\ r_i \in R\ and\ s_j \in S\}$, where r_i and s_j are surrogates of the ith tuple in R and the jth tuple in S, respectively. For example, consider the Facility and Forest Stand relational tables shown in Figure 1. The Facility relation is joined with the Forest Stand relation on the spatial attributes of each relation. The join-index for this join contains the tuple IDs which match the spatial join predicate.

A join-index can be described by a bipartite graph $G = (V_1, V_2, E)$, where V_1 contains the tuple IDs of relation R, and V_2 contains the tuple IDs of relation S. Edge set E contains an edge (v_r, v_s) for $v_r \in R$ and $v_s \in S$, if there is a tuple corresponding to (v_r, v_s) in the join-index. The bipartite graph models all of the related tuples as connected vertices in the graph. In a graph, the edges connected to a node are called the incident edges of that node, and the number of edges incident on a node is called the degree of that node.

We use Figure 2 to illustrate one of the major differences between tuple-level adjacency matrices of linear-key equi-join and spatial join. Figure 2(a) shows the adjacency matrix for an equi-join. The

horizontal-coordinate shows distinct values of tuple-ids from one relation; the vertical-coordinate shows distinct values of tuple-ids from the other relation. Shaded areas are collections of dots representing tuple-pairs satisfying the equi-join predicate from the set of all tuple-pairs in the cross-product of two relations. White space designates the tuple-pairs which do not satisfy the equi-join predicate. Figure 2(b) presents the same information as Figure 2(a), where tuples in each relation are sorted by the join attribute. Note that the shaded areas come close the diagonal for linearly ordered join attributes. Join-processing algorithms (e.g., sort-merge) can take advantage of this property. Figure 2(c) shows adjacency matrix for a Θ-join, e.g., a spatial join. Note that the join attribute (e.g., spatial location) may not be linear in general and may not have a natural sort-order. However, one may reorder the rows and columns of the adjacency matrix to bring in as many dots (object-id pairs satisfying the spatial-join predicate) near the diagonal as possible. We found that the result of such efforts often yields an adjacency matrix similar to the one shown in Figure 2(d), where a substantial number of shaded areas remain away from the diagonal. Join-processing algorithms for spatial-join need to account for the off-diagonal shaded areas.

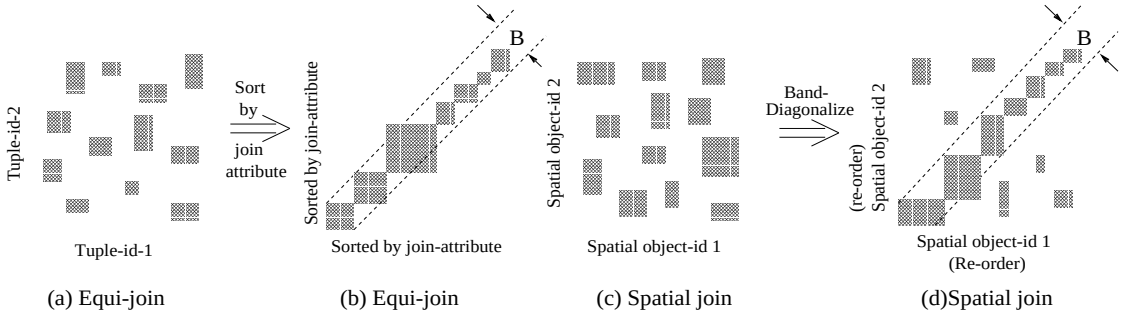


Figure 2: Comparison of tuple-level adjacency matrices for equi-join and spatial join

1.2 Page-Connectivity Graph, Page-Access Sequence

When the join relationship between two relations is described at the page level, we get a page-connectivity graph. A Page-Connectivity Graph (PCG) [31] $B_G = (V_1, V_2, E)$ is a bipartite graph in which vertex set V_1 represents the pages from the first relation, and vertex set V_2 represents the pages from the second relation. The set of edges is constructed as follows: an edge is added between page (node) v_1^i in V_1 and page (node) v_2^j in V_2, if and only if there is at least one pair of objects (r_i, s_j) in the join-index such that $r_i \in v_1^i$ and $s_j \in v_2^j$. Figure 3 shows a page-connectivity graph for the join-index from Figure 1(b). Nodes (a, b) represent the pages of relation R, and nodes (A, B, C) represent the pages of relation S. A *min-cut* node partition [17, 27] of graph $B_G = (V_1, V_2, E)$ partitions the nodes in V into disjoint subsets while minimizing the number of edges whose incident nodes are in two different partitions. The cut-set of a min-cut partition is the set of edges whose incident nodes are in two different partitions. Fast and efficient heuristic algorithms [24, 21] for this problem have become available in recent years. They can be used to cluster pages in a PCG.

A join-index helps speed up join processing, since it keeps track of all the pairs of tuples which satisfy the join predicate. Given a join-index JI, one can use the derived PCG to schedule an efficient

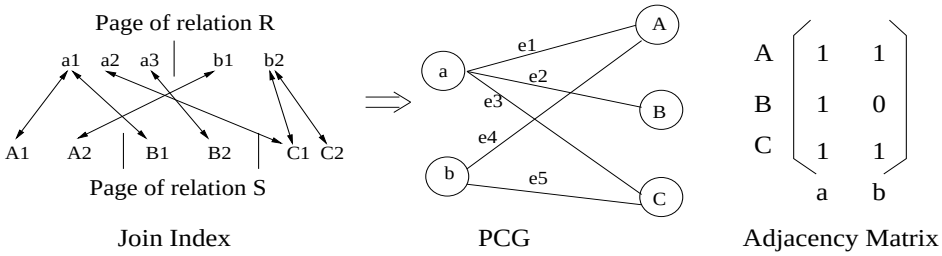


Figure 3: Construction of a Page-Connectivity Graph(PCG) from a Join Index.

page-access sequence to fetch the data pages. The CPU cost is fixed, since there is a fixed cost associated with joining each pair of tuples, and the number of tuples to be joined is fixed. I/O cost, on the other hand, depends on the sequence of pages accessed. When there is limited buffer space in the memory, some of the pages may have to be read multiple times from the disk. The page-access sequence(and in turn the join-index clustering and the clustering of the base relation) determines the I/O cost.

Example: We illustrate the dependency between the I/O cost of a join and the order in which the data pages are accessed with the help of an example using the page-connectivity graph shown in Figure 3. Assume that the buffer space is limited to allow at most two pages of the relations in memory, after caching the whole page-connectivity graph in memory. Consider the two-page access sequences: (i) (a, A, b, B, a, C, b) and (ii) (a, A, b, C, a, B). Each sequence allows the computation of join results using a limited buffer of two pages. However, in the first case, there are a total of seven page accesses, and in the second case there are a total of six page accesses. Note that the lower bound on the number of page accesses is five, since there are five distinct pages in the PCG. However, with two buffer spaces, there is no page-access sequence which will result in five page accesses. This is because the cycle (a, A, b, C, a) requires that at least three pages be in memory to avoid redundant page accesses. With three buffer spaces, (a, B, A, C, b) is a page-access sequence which results in five page accesses.

1.3 Problem Definition, Scope, Outline

Given that the I/O cost depends on the page-access sequence, the following optimization problem characterizes the problem of designing efficient algorithms for processing joins, given a join-index and a fixed buffer size. This problem, called the Optimal Page-Access Sequence with a Fixed Buffer (OPAS-FB) problem [31], is formally defined as follows:

OPAS-FB Problem

Given: A page-connectivity graph $PCG = (V, E)$, representing the join-index, and a buffer of size $B \leq |V|$.

Find: A page-access sequence.

Objective: To minimize the number of page accesses.

Constraint: Such that the number of pages in the buffer is never more than B.

For example, the optimal page-access sequence for the PCG in Figure 3 for $B = 2$ is (a, A, b, C, a, B), which results in six page accesses.

The OPAS-FB problem[¶] is known to be NP-hard [31, 34], and heuristic solutions have been proposed in the literature for solving this problem. These heuristics can be broadly divided into two groups, namely asymmetric single-table clustering and symmetric two-table on-line clustering. We describe the relevant literature and our contribution in sections 2 and 3.

Scope: In this paper, we focus on the OPAS-FB problem. We do not address the update problems associated with managing a join-index. Base-relation clustering and tuple-level join-index optimization are also beyond the scope of this paper. Readers interested in the update problem of join-indices or in a comparison of join-indices with other join-strategies are referred to [41]. Our focus is on join processing algorithms given a join-index.

Outline: The rest of the paper is organized as follows. In Section 2, we discuss asymmetric single-table clustering methods, propose our first approach, Asymmetric Clustering(AC), and evaluate its performance with that of the Sorting-based heuristic. Section 3 reviews the literature on two-table on-line clustering and provides an illustrative example. We propose Symmetric Clustering (SC) in Section 4. Section 5 compares the proposed methods, AC and SC, with traditional algorithms for the OPAS-FB problem. We summarize our work and discuss future research directions in Section 6.

2 Asymmetric Methods

The main approach in asymmetric single-table clustering is based on sorting the join-index on one of the join keys. In the following discussion, let R and S be the two relations, with JI being the join-index. The Sorting-based asymmetric heuristic presented in [41] reads as much as possible of the join-index (JI) and one relevant relation (R semi-join JI) into memory. Here JI is assumed to be sorted on relation $R.ID$. To reduce redundant accesses to S, access to S is clustered by sorting the list of all the surrogates from S that are related to the subset of the join-index in memory. This heuristic ensures that no redundant accesses are performed on relation R, but it may incur redundant accesses to the second relation. The Sorting-based heuristic is most suited to applications that have totally ordered join-keys. Rigorously speaking, the sorting-based heuristic sorts the surrogates(e.g., system-defined identifiers for pages) rather than the join-key attributes. If tables are sorted by the respective join-keys, then surrogates for the pages in a table may be ordered by the lowest key-value for any tuple in the page. This reduces redundant page I/O in computing joins using a join-index for join-keys with totally ordered domains. Since multi-dimensional domains, such as spatial data types, do not have natural total-order, sorting surrogates may not be as effective for computing spatial-joins using join-indices. We propose Asymmetric Clustering(AC) to address this problem. Asymmetric clustering uses the entries in a join-index for grouping pages of one relation, say R, based on their interaction with the pages in the other relation, say S. If the join-index (see Figure 1(b)) represents the summary of a spatial join, then the pages of R are clustered using their spatial relationship with the pages of S, and the proposed method is called Asymmetric Spatial Clustering.

[¶]The OPAS-FB problem is a special case of buffer pool management problems for databases. Prior work [36, 8, 39] has examined access patterns for index traversal, scan, and nested-loop join. They have not explored buffer management for join computation given a join-index, which is the focus of various solutions to OPAS-FB problem.

2.1 Basic Idea Behind Asymmetric Clustering(AC)

The example in Figure 4 highlights the different approaches of AC and the Sorting-based heuristic. Figure 4(a) shows a 49-node PCG, with numbers 1-24 and letters A-Y corresponding to the surrogates of the two page-level relations. The figure can represent a spatial-join computation between two layers of geographic data consisting of small polygons. The pages from each relation may overlap pages from the other. Consider a memory with seven pages for buffering the pages of R and S relations. There may be additional buffering pages for managing the result, index, etc. AC uses a different clustering of

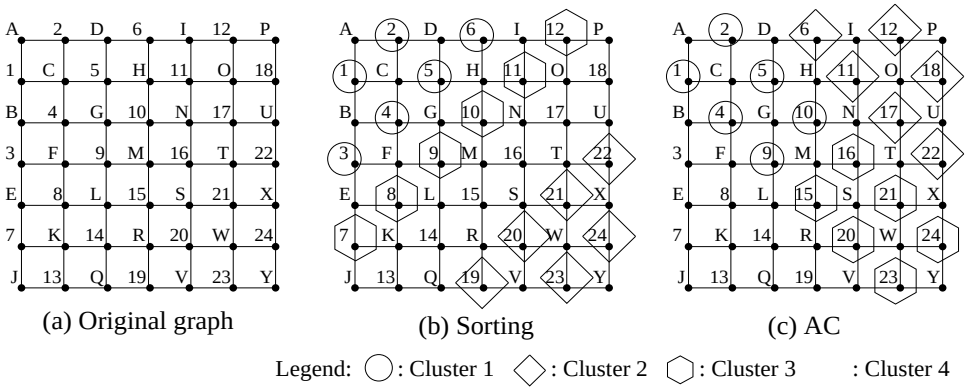


Figure 4: Example of the AC and Sorting heuristic

pages of the first relation, compared to the Sorting-based approach. Figures 4(b) and 4(c) show the clusters of pages of the first relation that are used by AC and the sorting-based method. The pages of R are numbered 1-24, and within a cluster are annotated by a common symbol. For example, nodes 1,2,3,4,5,6 in Figure 4(b) are all circled, denoting that these are loaded together by sorting. Similarly, nodes 7,8,9,10,11,12 are annotated with a hexagon, and so on. Visually, one can verify that the clusters used by AC are spatially cohesive. The pages of the second relation which have edges to multiple clusters of pages in the first relation yield redundant I/Os. One may consider using space-filling curves, such as Z-order or Hilbert, to improve the performance of the sorting-based method. However, min-cut graph partitioning, the method used by AC, outperforms space-filling curves in the clustering of non-uniformly distributed spatial data, as shown in our previous work [37, 38].

The Sorting-based heuristic clusters the nodes of the first page-level relation and then loads the pages in the sorted order. The loading sequence for the example in Figure 4 is shown in Table 2. With a buffer size of seven, the sorting heuristic will load pages $\{1, 2, 3, 4, 5, 6\}$ from the first relation in the first six buffers and then will load one page at a time from the set of pages $\{A, B, C, D, E, F, G, H, I\}$ in the 7th buffer. For the next round, it loads $\{7, 8, 9, 10, 11, 12\}$ in the first six buffers, and then $\{E, F, G, H, I, J, K, L, M, N, O, P\}$ one at a time in the 7th buffer, and so on, as shown in Table 2. The sorting heuristic results in 17 redundant I/Os, with a total of 66 I/Os.

AC clusters the nodes of the first page-level relation according to their connections with the second relation. As shown in Figure 4(c), clustering with a buffer size of seven provides four clusters. Note that these clusters are different from the page-clusters used in sorting. AC loads pages $\{1, 2, 4, 5, 9, 10\}$

	Sorting Heuristic		AC	
Round	First Six Buffers	The 7th buffer	First Six Buffers	The 7th buffer
1	1,2,3,4,5,6	A,B,C,D,E,F,G,H,I	1,2,4,5,9,10	A,B,C,D,F,G,H,L,M,N
2	7,8,9,10,11,12	E,F,G,H,I,J,K,L,M,N,O,P	3,7,8,13,14,19	B,E,F,J,K,L,Q,R,V
3	13,14,15,16,17,18	J,K,L,M,N,O,P,Q,R,S,T,U	6,11,12,17,18,22	D,H,I,N,O,P,T,U,X
4	19,20,21,22,23,24	Q,R,S,T,U,V,W,X,Y	15,16,20,21,23,24	L,M,N,R,S,T,V,W,X,Y
Total I/O	**66**		**62**	

Table 2: Loading Sequence of Sorting and AC for Figure 4

of the first relation in the first six buffers, and then, one by one, loads $\{A, B, C, D, F, G, H, L, M, N\}$ in the 7th buffer. In the next round, it loads $\{3, 7, 8, 13, 14, 19\}$ together into the first six buffers, and then $\{B, E, F, J, K, L, Q, R, V\}$ one at a time into the 7th buffer. Table 2 shows the total loading sequence. AC results in 13 redundant I/Os, with a total of 62 I/Os. The difference of four I/Os out of 66 in this example may not look large. However, the relative difference in I/Os using the sorting and clustering methods will increase as the data set size increases. This linear characteristic of sorting yields poor clustering and limits the savings in redundant I/Os.

2.2 Description of the Asymmetric Clustering Method

The goal of asymmetric clustering methods is to cluster the pages of one relation, given the join-index or its PCG. This can be formalized as a min-cut hypergraph-partitioning problem*. The pages of a relation will form the nodes of the hypergraph. Each page p of the other relation will form a hyperedge, covering all pages of the first relation connected to p in the PCG. Partitioning the nodes in this hypergraph will form a group of pages of the first relation that can be loaded together. Minimizing cut hyperedges during partitioning reduces the number of pages of the second relation that will need to be loaded into memory multiple times.

Consider the example spatial-join problem depicted in Figures 5(a) and 5(b) with two point data sets, (a,b,c,d) and (A,B,C,D). To simplify the example, we assume a unit blocking factor, i.e., one point object per disk page. The PCG of the join-index for $Distance(i, j) < \frac{L}{\sqrt{2}}$ is shown in Figure 5(c) using overlay and distance buffer information†. The nodes of the hypergraph shown in Figure 5(d) consist of the nodes of relation R, i.e., (a,b,c,d). The hyperedges represent nodes (A,B,C,D) of S. The hyperedge corresponding to A connects a and c, since (A,a) and (A,c) satisfy the join predicate. The partition ((a,c),(b,d)) has no cut hyperedges, and using it to perform the join results in no redundant I/O with loading sequence (a,c,A,C,b,d,B,D) if three buffers are available to hold the pages of the two relations. In contrast, the partition ((a,b),(c,d)) cuts all four hyperedges, and computing this join will yield four redundant I/Os with loading sequence (a,b,A,C,B,D,c,d,A,C,B,D), if only three buffers are available to hold the pages of the two relations.

We formally describe AC now via the following pseudo-code.

*A hypergraph $G = (V, E)$ is defined as a set of vertices V and a set of hyperedges E, where each hyperedge is a subset of the vertex set [4]. The min-cut hypergraph-partitioning problem is to partition the vertices of a hypergraph into k roughly equal parts, such that the number of hyperedges spanning different partitions is minimized.

†If we extend the join distance in Figure 5(c) with the maximum distance, the PCG will form a full-connected bipartite graph, which is not interesting for join-indices. Join-indices are not useful for joins with selectivity = 1.

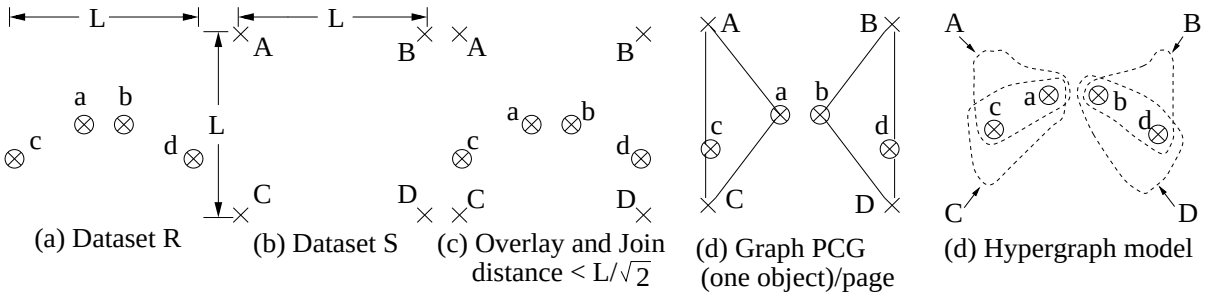


Figure 5: Construction of a one-sided hypergraph from the data set

AC Algorithm

Input: $G = (V_r, V_s, E)$ is a page connectivity graph

Output: $S =< P_1, P_2, ..., P_r >$ is a page access sequence with $r \geq |V_r| + |V_s|$.(P_is need not be distinct)

assert($|V_r| < |V_s|$);
assert($B \geq 2$); /* *number of buffers* */
$HG_r(V_r, HE_r)$ = DeriveHypergraph(G); /* HG_r *is a hypergraph,* $|HE_r| = |V_s|$ */
/* *For each node in* $|V_s|$*, build a hyperedge to encompass all of its corresponding nodes in* V_r */
$PSet_r$ = hMetis-Partition($HG_r, B-1$) /* $PSet_r$ *is the set of partitions* */
i=0;
while ((P_{i_r}=SelectUnprossedPartition($PSet_r$))!=NULL) /* *Select the un-processed partition* */
{
    AddPageSequence(S, P_{i_r});/* *Add all the nodes in* P_{i_r} *into the loading sequence* */
    P_{i_s} = Sort-Eliminate-Dup(G, P_{i_r});
    /* *Sort and eliminate the duplicated nodes in* V_s *of* G *which connect to nodes in* P_{i_r} */
    AddPageSequence(S, P_{i_s});/* *Add all the nodes in* P_{i_s} *into the loading sequence* */
    P_{i_r}.flag = "processed"; /* *Mark this partition as "processed"* */
    i++;
}

The first step of the AC algorithm, i.e., DeriveHypergraph(G), creates a hypergraph from a given page connectivity graph G. Nodes of the first relation R form the nodes of the hypergraph. For each node v of the second relation, AC builds a hyperedge to encompass a set of nodes on the first relation(R) that are connected to v in G. Next, AC partitions this hyper-graph using the min-cut hyper-graph partitioning algorithm, hMetis [21, 22, 23], which is a multi-level hypergraph-partitioning algorithm that has been shown to produce high quality bi-sections on a wide range of problems that arise in scientific and VLSI applications. hMetis minimizes the (weighted) hyper-cut and thus tends to create partitions in which connectivity among the vertices in each partition is high, resulting in good clusters. Finally, AC loads each partition in the primary relation and its connected nodes in the second relation, one by one, to compute the join. The I/O cost of AC can be characterized via the following lemma:

Lemma 1 *Given a partition* $\{V_{r_1}, V_{r_2}, \ldots, V_{r_p}\}$ *of* V_r*, i.e., pages of relation* R*, from the page-*

connectivity graph $PCG = (V_r, V_s, E)$*, there is a page-access sequence of length* $K = |V_r| + \sum_{v \in V_s} f(v)$ *to process the spatial join, where* $f(v)$ *denotes the number of partitions of* V_r *that have an edge to node* v *in* V_s*.*

Proof: A node v in V_s is connected to $f(v)$ partitions of V_r. Therefore, the node v in V_s has to be loaded $f(v)$ times into the buffer to compute the spatial join. The total number of redundant I/Os is $\sum_{v \in V_s}(f(v) - 1)$. The total I/O cost $= |V_r| + |V_s| + \sum_{v \in V_s}(f(v) - 1) = |V_r| + |V_s| + \sum_{v \in V_s} f(v) - |V_s| = |V_r| + \sum_{v \in V_s} f(v)$ ■

The computational complexity of AC is $O(|E|) + O(|V_r| * log(|V_r|) + |V_s| * log(|V_s|)) + O(|E| * log(\frac{|E|}{P})$, for a given bipartite page-connectivity graph $G = (V_r, V_s, E)$ and a buffer size B. The DeriveHypergraph has computational complexity $O(|E|)$. The hMetis-Partition software we used has computation complexity $O(|v| * log(|v|) + |e| * log(|e|))$ where $|v|$ is the number of vertices and $|e|$ is the total number of hyperedges. Since AC builds a hyperedge for each node in V_s to encompass all of its corresponding nodes in V_r, $|v| = |V_r|$ and $|e| = |V_s|$, the actual computational complexity for hMetis-Partition is $O(|V_r| * log(|V_r|) + |V_s| * log(|V_s|))$. The while procedure executes P times, where P is the number of partitions and $P = |V_r|/(B-1)$. The Sort-Eliminate-Dup step, with computational complexity $O(\frac{|E|}{P} log \frac{|E|}{P})$, is the dominant cost inside the while loop. The computational complexity of AC is: $O(|E|) + O(|V_r| * log(|V_r|) + |V_s| * log(|V_s|)) + O(|E| * log(\frac{|E|}{P}))$.

2.2.1 Experiment Design

We now compare the performance of Sorting and AC, using a join-index derived from the Sequoia 2000 [40] data set. The *Point* table contains 62,584 California place names with their associated locations(Longitude and Latitude), extracted from the US Geological Survey's Geographic Names Information System(GNIS). The *Polygon* table contains 4,388 records, representing Cropland and Pasture land use in California. Throughout Sections 2.3 and 5, the *Point* and *Polygon* tables will be referenced as R and S, respectively. We plot the point and polygon data sets of California records as in Figure 6.

Readers may note our use of real spatial data sets from the Sequoia 2000 [40] benchmark due to the interest in spatial join-indices in contrast to the use of synthetic data sets, such as randomly generated graphs, in much related work [11, 7, 33]. We plan to use additional data sets, both real and synthetic, to expand the experimental evaluation in future work.

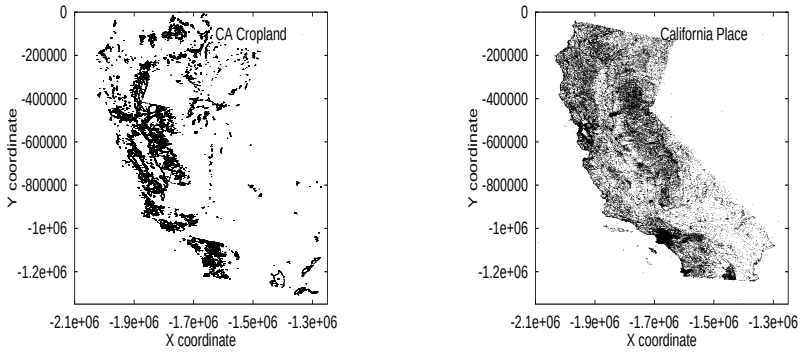
Now, consider the following queries:

Q.A. "For each place in the *Point* table, find **N** nearest croplands from the *Polygon* table".

Q.B. "For each place in the *Point* table, find all croplands which are within a distance **D**".

The spatial join of these two queries produces sets of join-indices and such join-indices are of interest in spatial data mining for neighborhood indexing [10]. The value of **N** and **D** can be increased/decreased for adjusting the edge ratio [7]. Give a join graph $G = (V_R, V_S, E)$, the edge ratio of G, denoted by Θ, is defined as the ratio of the total number of edges in G to the maximum possible number of edges in G if it is a fully connected graph; i.e., $\Theta = \frac{|E|}{|V_R||V_S|}$. The edge ratio provides a measure of the page-level join selectivity.

The variable parameters are buffer size, page size and edge ratio. In future work we plan to include additional parameters, such as the size of join-indices, as well as performance measures, such as the



(a) Polygons for Landuse

(b) GNIS Points

Figure 6: Two example of the Sequoia 2000 data sets

overhead of OPAS-FB algorithms. The metric of evaluation is the number of page accesses required by each algorithm to implement the join. The edge ratio is controlled by **N** and **D**. The number of nearest neighbors, N, is varied from 1 to 5, yielding an edge-ratio of 0.002 to 0.005. The side-size of range queries, D, is varied from 400 to 4800 units where the extent of California is almost $1.2 * 10^6$ units (North-South) x $0.8 * 10^6$ units (East-West). This yields an edge-ratio of 0.002 to 0.003.

Page size represents the size of disk blocks and memory pages. Different values of page sizes include 2, 4, 8, 16, 32, and 64 Kbytes. The size of the records in the point table is 64 bytes. The blocking factor for the Point table is the ratio of page size and record size. Point records are spatially clustered in the pages of the point table. The records in the Polygon table are of variable size. The size of a record in the Polygon table is 16+ 32*(# of Points in the Polygon) bytes. The number of points in a polygon can vary from a dozen to a few thousand, and a large polygon may span multiple pages.

The buffer size represents the ratio of available memory size as a fraction of the size of the Point table, which is the smaller of the two tables. Memory buffer size varies from 4% to 20% of the size of the smaller table.

Figure 7 shows various steps in the experiment. From the base point and polygon tables, we derived families of join-indices for queries **Q.A.** and **Q.B.** for different values of edge ratio. Next, we generated page-connectivity graphs(PCGs) from join-indices, given different values of page size. The page-connectivity graphs were input into a "page-Access-Sequence Generator," which simulated the behavior of the OPAS-FB algorithms (i.e. Sorting and AC) for a given buffer size. The page-access sequences and total page I/Os were tracked for each combination of join algorithm, page size, buffer size, and edge ratio.

Note that Queries **Q.A.** and **Q.B.** have parameters **N** and **D**. It may not be realistic to use join-index for parametric queries due to the requirement of the existence of many spatial join-indices. A join-index is more likely to be used for overlap, adjacent, cross, and other non-parametric spatial predicates. We do not advocate the use of a spatial join-index for parametric queries. Instead, parametric queries were used to generate a set of join-indices to study the effect of different values of edge-ratio parameter on relative performance of OPAS-FB strategies.

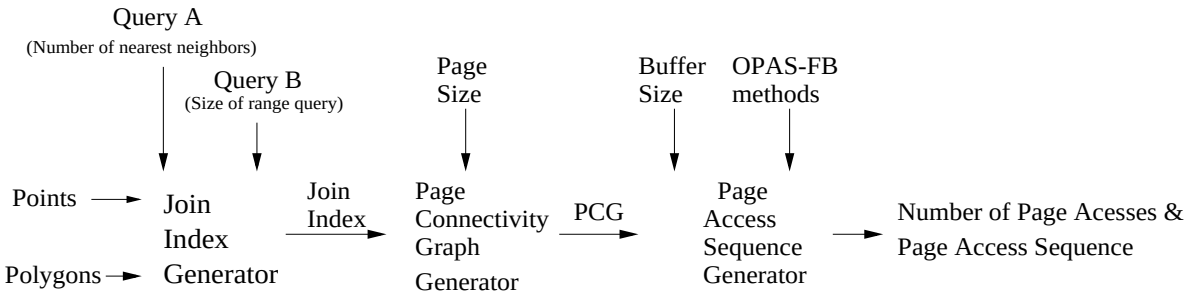


Figure 7: Experiment setup and design

2.2.2 Experiment Results

Figures 8 and 9 show the comparison between AC and Sorting for range-query join-indices and N-nearest-neighbor join-indices, respectively. AC performed uniformly better than Sorting.

Figures 8(a) and 9(a) show the impact of page size, which we varied from 2 Kbytes to 64 Kbytes. The page size and the number of page accesses are shown in logarithm scale(base two). As the page size increased, the number of pages decreased, and clustering efficiency improved for all methods, reducing the performance gap between the two methods.

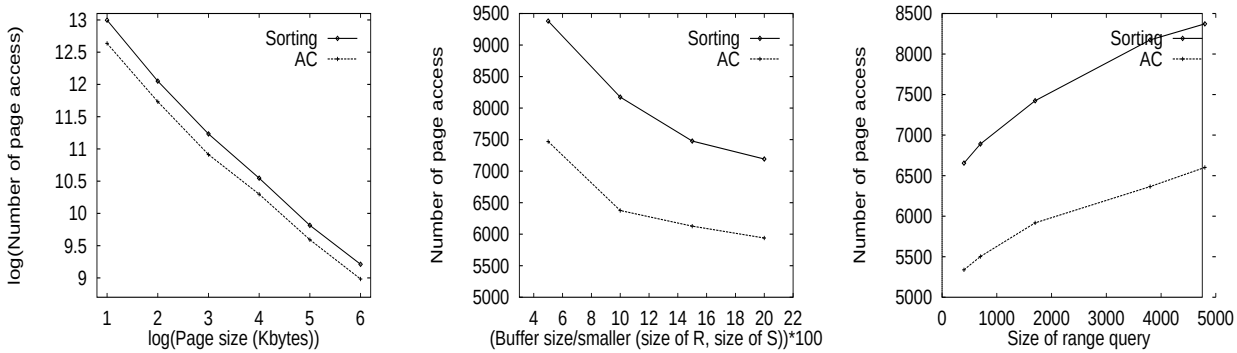
Figures 8(b) and 9(b) show the effect of buffer size (as a fraction of the size of the smaller relation) on the I/O performance of AC and Sorting. As long as the buffer size was smaller than the smaller of the two relations involved in the join, both AC and Sorting used most of the buffers to load the pages of only one relation. The difference in performance came from the difference in their clustering abilities: AC generated a lower I/O cost than Sorting. Note that in Figure 9(b), the increase of buffer size from 15% to 20% of the smaller relation did not reduce the number of page accesses for Sorting, an non-intuitive result similar to Belady's anomaly [3] for some page-replacement algorithms used in managing virtual memory. Often, we assume that allocating more buffer to load disk pages would reduce the I/O cost. Our experiment result showed that this assumption is not always true for Sorting.

Figures 8(c) and 9(c) show the effect of the edge ratio. AC uniformly outperformed Sorting. The gap between the performance of the two methods did not show any trend.

We note that the min-cut hypergraph-partitioning method, hMetis, used in AC minimizes the number of hyperedges connecting nodes across clusters. This does not directly minimize the redundant I/O cost ,$\sum_{v \in V_s}(f(v)-1)$, as defined in Lemma 1, since it does not distinguish between a hyperedge spanning four clusters and another spanning two clusters. While AC outperforms the sorting-based heuristic already, the performance of AC will improve when better algorithms for hypergraph partitioning are available which minimize the total number of cuts on cut-hyperedges.

3 Symmetric Methods

While AC is an improvement over Sorting for spatial joins, it has a few drawbacks. For example, its buffer utilization can be poor, since it gives almost the entire buffer space to one relation. We illustrate this with the help of the spatial join problem shown in Figure 10. Figure 10(a) shows a polygon set with



(a) Page size: 2K - 64K, Buffer size: 10% of the smaller relation, Range query size = 3800

(b) Buffer size: 5% - 20% of the smaller relation, Page size: 2K, Range query size = 3800

(c) Range query size = 400 - 4800(Θ=0.002-0.003), Buffer size: 10% of the smaller relation, Page size: 2K

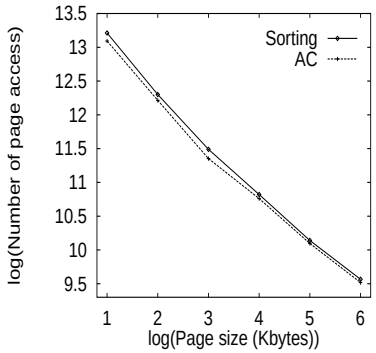
Figure 8: **Range Query Join-index**: The effect of page size, buffer size, and number of nearest neighbors on AC and Sorting

6 polygons, R0..R5, and a point data set with 6 points. The adjacency matrix M_{PCG} representation of a join-index is shown in Figure 10(b), along with the page access sequence for the sorting-based algorithm with three memory buffers. Sorting requires 16 I/Os, including 4 redundant I/Os on S1, S2, S3, and S4, using a page-access sequence of $R0, R1, S0, S1, S2, R2, R3, S1, S2, S3, S4, R4, R5, S3, S4, S5$.

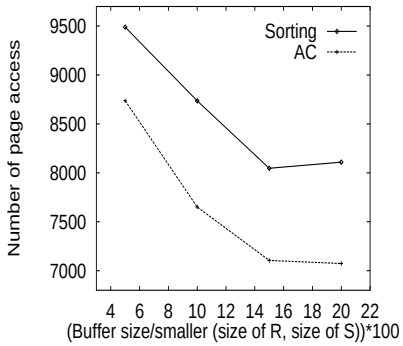
A symmetric method may alternate between the pages of the two relations, as shown in Figure 10(c), to compute the join with 12 I/Os (i.e., no redundant I/O) using a page-access sequence of $R0, S0, S1, R1, S2, R2, S3, R3, S4, R4, S5, R5$. This property can be generalized to other adjacency matrices with only B-diagonal entries, where $\{M_{PCG}[i,j] = 1\} \Rightarrow \{|i-j| \leq \lfloor B/2 \rfloor\}$, and B is the number of buffers available for pages of R and S. The indices i and j refer to the row-indices and column-indices. The symmetric method can process the B-diagonal entries of an adjacency matrix with no redundant I/Os, given B buffers for R and S.

The main approaches in symmetric two-table clustering are based on either the Traveling Salesman Problem heuristic or on incremental clustering. The Traveling Salesman-based heuristic [14] uses a complete graph constructed by taking the nodes of one relation as the nodes of the graph. The weight on an edge between nodes a and b denotes the number of page-accesses required to fetch all of the neighbors of b, given that all of the neighbors of a are in memory. This method requires a large amount of memory, since the complete graph grows quadratically with the number of nodes in the smaller of the relations. Incremental clustering is based on selecting the next page or the next set of pages to be fetched into memory, given the pages in the buffer and the remaining edges to be processed in the bipartite page-connectivity graph. The selection is often based on the number of neighbors in the memory buffers and the number of neighbors on the disk. Details of the actual heuristics follow.

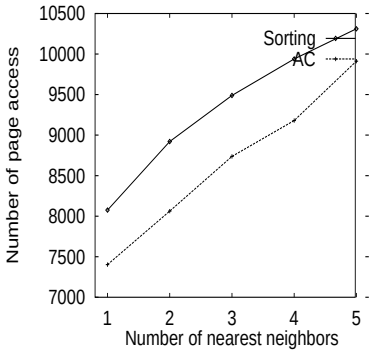
Symmetric Heuristic: FP was proposed by Fotouhi and Pramanik [11]. The buffer is initialized with a node which has the smallest degree in the page-connectivity graph. The memory buffer is added



(a) Page size: 2K - 64K, Buffer size: 5% of the smaller relation, No of nearest neighbors=3

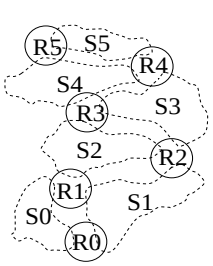


(b) Buffer size: 5% - 20% of the smaller relation, Page size: 2K, No of nearest neighbors=3

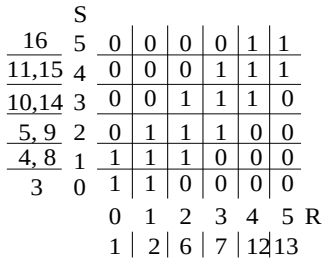


(c) No of nearest neighbors=1-5(Θ=0.002-0.005), Buffer size: 5% of the smaller relation, Page size: 2K

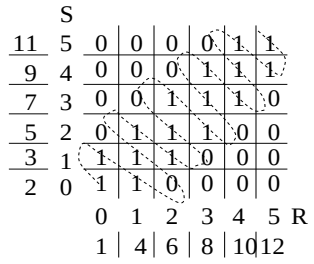
Figure 9: **Nearest Neighbor Join-index**: The effect of page size, buffer size, and number of nearest neighbors on AC and Sorting



(a) Spatial Join



(b)Sorting/AC



(c)Symmetric method

Figure 10: Comparison of symmetric and asymmetric methods

with the largest resident-degree node. The resident degree of node A is the number of nodes which are connected to A and are in memory buffers. If more than one node has the largest resident degree, the algorithm chooses the one with the smallest non-resident degree. The non-resident degree of a node A is equal to $total\_degree(A) - resident\_degree(A)$. When the buffer is full, the node that has the smallest number of edges with the nodes on the disk can be swapped out. (Time complexity: $O(|V|^2)$, where $|V|$ is the number of vertices in the page-connectivity graph).

Symmetric Heuristic: OM, developed by Omiecinski [33], is designed specifically for bipartite join graphs $G = (V_r, V_s, E)$. Initially, a pair of nodes (r_i, s_j) is loaded, where $r_i \in V_r$ and $s_j \in V_s$, from the page-connectivity graph in the memory buffers, such that (a)(r_i, s_j) is connected and (b)the sum of the degree of r_i and s_j is minimal. In each iteration, an out-of-memory least-non-residential-degree neighbor p of an in-memory lowest-non-residential-degree node q is selected to be swapped in. If the memory buffers are full, then the lowest-non-residential degree node r which is not connected to p in PCG is swapped out. Node r may come back to memory within the next few iterations. (Time complexity is $O(|V_r|^2 * |V_s|^2)$, where $|V_r|$ and $|V_s|$ denote the number of vertices for V_r and V_s, respectively).

Symmetric Heuristic: CO [7] generalizes the OM heuristic by swapping in a set of nodes, i.e. segments, in each iteration. The set of nodes selected for an iteration are the unprocessed on-disk neighbors of the lowest-non-residential degree node n, either in memory or on disk. Node n can be processed and swapped out of the memory buffers at the end of the current iteration. If the memory buffers do not have enough empty space, then the page with the lowest number of a non-residential degree is swapped out. (Time complexity: $O(|V|^2)$, where $|V|$ is the number of vertices in the page-connectivity graph.)

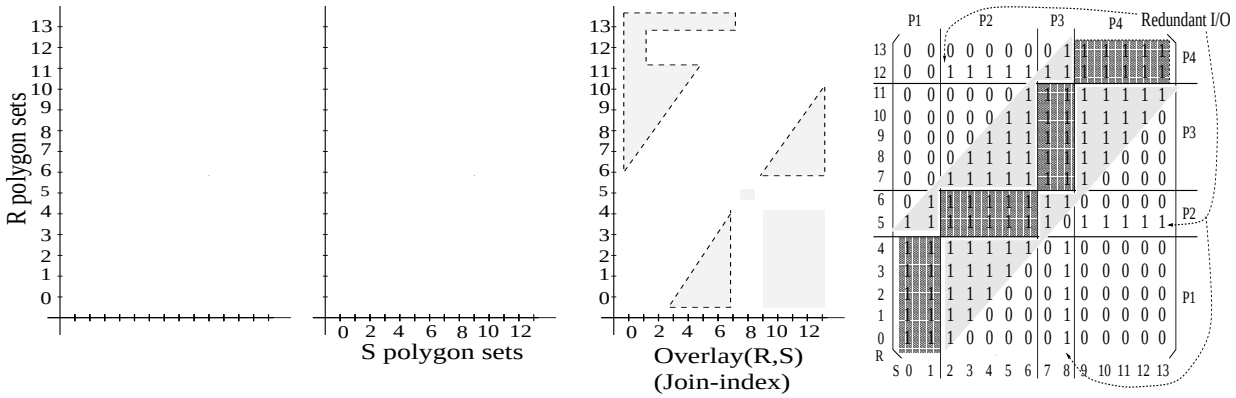
Most symmetric methods proposed in the literature are incremental, considering local information in the PCG. In Section 4, we propose the Symmetric Clustering(SC) method which exploits global information across the entire PCG. The global information used in SC optimizes the page access sequence in one step. While incremental methods, such as FP, OM, and CO, iteratively use local information and greedy strategies to reduce the number of page access sequences. Global methods like Sorting, SC, and AC have higher initial costs, particularly for join-indices with very high join-selectivities. On the positive side, however, global methods yield a re-clustering of the join-index in addition to the page-access-sequence. A reclustered join-index may be reused across queries if updates are rare, allowing amortization of the initial cost over multiple queries. This may result in lower amortized costs for global methods in environments with low update rates.

3.1 An Example

We use an example to illustrate the differences between the various heuristics for computing joins, given a join-index. Readers are warned that this example is a bit detailed to bring out the differences between the various methods. On the first reading, one may gloss over the details of Section 3.1 and simply look at Table 3 to get the summary. We have tried hard to find an example which could explain our method, symmetric clustering, and also explain the differences between symmetric clustering and other methods. We have not been able to find anything simpler. Figure 11(a) shows the polygon-clusters in the R and S relations with their overlays. Some polygon-clusters have one polygon; others have two polygons, for example, $R0, R1, \ldots, R5, S1, S2, \ldots, S5$, and $S8, S9, \ldots, S13$. Polygon clusters are natural in geographic data as well. For example, the boundary of the United States will be represented by a collection of polygons representing the Mainland, Alaska, Hawaii, etc.

The spatial-join relationship shown uses a geometric as well as a topological representation. Figure 11(a) shows an overlay of the two data sets to provide a visual representation of the join relationships. Shaded areas are provided for realism. One may imagine a beautiful city with many parks and lakes(shaded areas) which break the continuity of the streets (objects in R) and avenues (objects in S). Figure 11(b) provides an adjacency matrix representation of the page-connectivity graph with one object per page. To simplify the example, we assume one unit blocking factor, that is, one polygon-cluster or road per disk page.

A summary of the behavior of the alternative methods is shown in Figure 12, which is divided into five different parts. The adjacency matrix representation of the join-index is reproduced from Figure 11(b) to facilitate understanding. The IDs of the polygon clusters in R and S appear immediately to the left of and below the matrix. The two vectors to the left of and below the node-identifiers list the degree of each node in PCG. For example, the degree of R13 is 6. The nodes with the highest degree (13) are R5 and S8, followed by R12, which has a degree of 12. Note that the available buffer size in this example is



(a) Overlay of Two Relations

(b) Adjacency matrix of PCG for Join-index of overlay(R,S). Dark region=edges within a partition. Lightly shaded region=edges within B-diagonal

Figure 11: Connectivity graph of two relations

eight, i.e., memory buffers can hold at most eight disk pages at a time.

The remaining two parts of Figure 12 present the summary behavior of various algorithms in terms of page-access sequence. These tables show the rank-order of the polygon-clusters (i.e., pages assuming one unit blocking factor) from R and S in their respective page-access sequences. For example, the page access sequence for OM is R0,S0,S1,S2,S8,R1,S3,R2,S4, R3,S5,R4,R5,R6, and so on. Multiple ranks for a node indicate redundant I/Os. For example, OM loads R4,R6,R7,R8, and S7 twice in memory with rankings of (12,30),(14,31),(15,24),(17,32), and (25,33). This information is further summarized in Table 3, which shows total I/Os, nodes with redundant I/Os, and min/max/average degree of nodes with redundant I/Os. More details are available in Appendix A.

Method		AC	FP	OM	CO	SC
Total I/O		40	45	33	35	31
Nodes with redundant I/O		S2-S13	R1-R4, R6, R7, R9, R10, S0-S5, S7, S8	R4,R6,R7, R8,S7	S0-S6	R12, R5, S13
Degree of nodes with redundant I/O	Avg	8	8.3	8	8.3	12.6
	Min	4	5	8	6	12
	Max	13	13	8	9	13

Table 3: Summary of page accesses for different algorithms

FP prefers to swap-in the nodes with the highest residential degree and swap out the nodes with the lowest non-residential degree. In this example, these policies tend to favor high degree nodes for buffering,

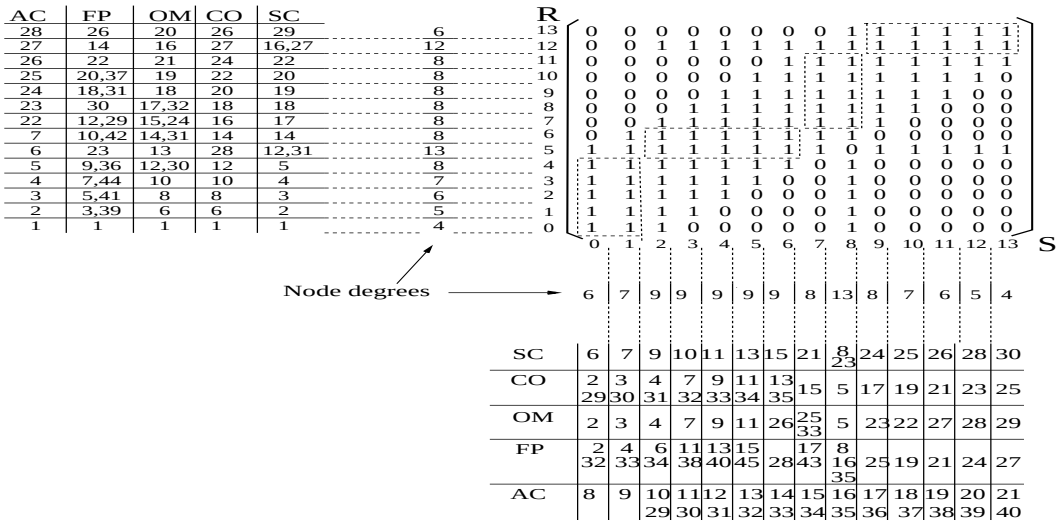


Figure 12: Comparison of different methods

leading to redundant I/Os on low-to-medium degree nodes which have been loaded into memory already. Note that nodes R5 and R12 have no redundant I/Os despite having a very high degree, while R1, R2, and S0 have redundant I/Os despite having a very low degree.

OM performs well overall with only five redundant I/Os. However, the redundant I/Os occur with nodes of medium degree, e.g., R4,R6,R7,R8, and S7(degree 8.) The nodes with a higher degree have no redundant I/O. This is due to the swap-in and swap-out policies. The swap-in policy favors low-residential degree neighbors of low non-residential degree in-memory nodes. The policy has the unintended effect of pulling all of the neighbors (S0,S1,S2,S8) in order of degree of the in-memory node with the lowest non-residential degree, e.g. R0. If one of these neighbors (S8) has a very high degree, it may stick around for a long time due to the swap-out policy based on the lowest non-residential degree[‡]. This favored treatment of high non-residential degree nodes in memory leads to increased redundant I/Os, in this example.

The CO method is a logical refinement of the OM method. It tries to bring in all of the neighbors of the node N with the lowest non-residential degree. This allows the complete processing of all edges incident on N. The hope is to reduce redundant I/O; however, the CO method leads to an unintended side effect for spatial joins. After R0 comes into the buffer, its neighbors(S0,S1,S2,S8) are brought in to process it. After processing R0, the next node with the lowest non-residential degree is R1, which enters into memory with its remaining neighbor S3. After R1 is processed, R2 is selected, and soon the nodes of R are favored to be processed first, since the first node to be loaded was from R. This leads to excessive redundant loading for nodes of S in this example.

[‡]It is non-intuitive. See the explanation from [33]: "If a node in the buffer is going to be swapped out, then in the worst case it could be brought in one additional time for each page it is connected to outside the buffer, assuming it were to be swapped out each time." Thus a swap-out kicks out the node with the lowest non-residential degree.

4 Proposed Symmetric Clustering Method

In contrast to the incremental approaches of FP, OM, and CO, Symmetric Clustering(SC) uses a global approach based on band-diagonalization of the adjacency matrix representation of PCG. The number of redundant I/Os depends only on the edges outside of the B-diagonal and can often be reduced via identifying a vertex cover[§].

Recall that pure B-diagonal entries for a square matrix are defined by $\{M_{PCG}[i,j] = 1\} \Rightarrow \{|i-j| \leq \lfloor B/2 \rfloor\}$, where B is the number of buffers available for pages R and S. band-diagonalization of a matrix rearranges the rows and columns of the matrix to bring in as many non-zero entries as possible within the B-diagonals. Thus, a matrix with only B-diagonal elements is already band-diagonalized. However, a band-diagonalized matrix may have a few entries outside the B-diagonal.

SC Algorithm

Input: $G = (V, E)$ is a page-connectivity graph; B is the number of buffers.

Output: $S =< P_1, P_2, ..., P_r >$ is a page-access sequence with $r \geq |V|$.(P_is need not be distinct)

{Step 1} $< G_{BD}, P_{order} >$=Band-diagonalization(G, B); */* Get B-diagonal graph and ordered set of partitions*/*

{Step 2} $< G_{OBD} >$=Get-off-B-diagonal-entries(G_{BD}); */* Find all the off-B-diagonal edges and nodes from G_{BD} */*

{Step 3} $< VC >$=Find-vertex-cover(G_{OBD}); */* Find the vertex cover VC for the Off-B-diagonal cut-edge E_{OBE} */*

{Step 4} $< S >$=Access-sequence-generator(P_{order}, G_{BD}, VC); */* Generate the page access sequence */*

First, SC derives the Band-diagonalized matrix G_{BD} by permuting the rows and columns of the adjacency matrix representation of PCG to bring in as many edges as possible within the B-diagonal. Secondly, from G_{BD}, SC gets the graph G_{OBD} for off-B-diagonal edges and their corresponding nodes. Thirdly, SC determines the vertex cover for G_{OBD}. Finally, SC generates the page-loading sequence based on the band-diagonalized matrix, the vertex cover of the off-B-diagonal edges, and the partition ordering.

Example Revisited

Consider the join-computation problem discussed in Figures 11 and 12. The input to the SC algorithm is the page-connectivity graph shown in Figure 11(b).

In **step 1**, the nodes in the PCG of Figure 11(b) are rearranged to get as many edges within the B-diagonal as possible. The lightly shaded area in Figure 11(b) shows the B-diagonal. In this example, input graph G and output graph G_{BD} of step 1 are identical for simplicity. The nodes are partitioned in groups of seven nodes, which is one less than the number of memory buffers available for R and S. This reserved memory buffer is used for processing the off B-diagonal edges and the edges between the nodes in adjacent partitions in the loading sequence. Partitions P1={R0-R4,S0-S1}, P2={R5-R6,S2-S6},P3={R7-R11,S7-S8}, P4={R12-R13,S9-S13} are used in this example, as shown in the shaded rectangles in Figure 11(b). The partitions are loaded in the order P1,P2,P3, and then P4. The edges between the nodes within a common partition can be processed with no redundant I/Os. The

[§] A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then either $u \in V'$ or $v \in V'$ (or both). The vertex-cover problem is to find a vertex cover of minimum size in a given graph [9].

edges between the nodes that are in adjacent partitions in the loading sequence and which fall inside the B-diagonal can also be processed without any redundant I/Os, due to the availability of one extra buffer.

The redundant I/Os for the remaining edges can be reduced by computing the vertex cover via **steps 2** and **3**. There are fifteen edges off the B-diagonal with the vertex cover of {S8, R5, R12}. There are five edges between partitions P1 and P3, and they are all incident on node S8. They can be processed with one extra I/O by bringing S8 into the last buffer when the nodes of P1 are in the buffer. Similarly, the ten edges between the nodes in P2 and P4 can be processed in two I/Os. Since five of these are incident on R12 and R5, bringing R12 with P2 and R5 with P4 will take care of all of these edges. Thus, SC produces only three redundant I/Os, which result from the vertex cover {S8,R12,R5} for the fifteen off B-diagonal edges. **Step 4** generates a page access sequence using the execution trace from the previous steps.

In summary, compared with FP, OM, and CO, SC uses a global clustering approach. SC uses symmetric clustering to permute the rows and columns of the adjacency matrix representation of PCG to bring as many edges as possible within the B-diagonal. All of the edges within the B-diagonal can be processed without redundant I/Os. The redundant I/Os for edges outside the B-diagonal are minimized by computing the vertex cover. The nodes in the vertex cover are scheduled with appropriate partitions, and some of these vertices may yield multiple redundant I/Os. Table 10(Appendix B.5) provides a detailed execution trace for interested readers.

4.1 I/O cost of the SC method

The symmetric clustering approach to minimize redundant I/Os can be described in terms of the following problem statement:

Lemma 2 *Given a loading ordered partition* $\{P_1, P_2, \ldots, P_i, P_j, \ldots, P_k\}$ *of* $PCG = (V, E)$*, there is a page access sequence of length* $k = |V|+$ *redundant-I/O to compute the spatial join where the redundant-I/O is given by:*

$$\sum_{V_i \in (vertx\ cover\ of\ outside\ B-diagonal\ edges)} Partition - degree(V_i) \tag{1}$$

where $Partition - degree(V_i)$ is the number of distinct partitions which contain nodes V_j outside of the B-diagonals.

Proof: All of the edges within the main B-diagonals can be processed without redundant I/Os using a contour diagonalization strategy, as illustrated in Figure 10(c). The redundant I/Os for each node in the vertex cover is limited by the number of partitions sharing an off-B-diagonal edge.

4.2 A heuristic for band-diagonalization

The first step of band-diagonalization can be based on either specialized envelope-reduction algorithms [1, 13, 28] or min-cut graph partition algorithms [24, 25]. We use the latter in this paper and plan to explore the former in future work. We describe the heuristic approach that we currently use.

Band-diagonalization

Input: G is a page connectivity graph; B is the number of buffers.
Output: G_{BD} is the B-diagonal connectivity graph; P_{order} is the partition order

$PSet_m$ = Graph-Partition($G, B-1$); /* *Using graph partition software* */
P_{order} = Order-Partition($PSet_m$); /* *Order the partitions using the greedy heuristic* */

Graph-Partition: A min-cut partition algorithm, e.g., metis [24], divides the nodes of the PCG into disjoint subsets while minimizing the number of edges whose nodes are incident in two different partitions. One memory buffer is reserved for bringing in pages from the vertex cover of the off-B-diagonal entries. For example, the min-cut partitioning of PCG for the overlay(R,S) in Figure 11 may yield four partitions for B=8. The partitions shown in Figure 11(b) are P1={R0-R4,S0-S1}, P2={R5-R6,S2-S6},P3={R7-R11,S7-S8}, and P4={R12-R13,S9-S13}, resulting in 69 edges whose incident nodes are in two different partitions. The breakup of these edges by pairs of partitions of incident nodes is shown in Table 4. Formally, the min-cut graph-partitioning algorithm addresses the following problem:

Given: A connectivity graph $G = (V, E)$ with $|V| = n$, and the number of buffers, $B \geq 2$.
Find: A partition of V into p subsets, $V_1, V_2, \ldots, V_p$ such that $V_i \bigcap V_j = \emptyset$ for $i \neq j$ and $\bigcup_i V_i = V$.
Objective: Minimize the size of the set of edges $E_C \subseteq E$ whose incident vertices belong to different subsets.
Constraint: $|V_i| \leq (B-1)$, and the number of partitions, $p = \lceil |V|/(B-1) \rceil$.

Recent advances have provided scalable graph-partitioning software such as Metis [24], which can handle the large graphs relevant to databases in a few seconds, a relatively reasonable response time. We have had good results using Metis for database problems [29, 38].

The extra memory buffer is reserved for processing pages of the vertex cover of the off-B-diagonal entries. We bring in these pages of vertex cover one page at a time, since this page will join with the pages of the current partition in the memory. If we reserve a greater buffer size, say x, then multiple pages within vertex cover can be loaded at the same time for computation of their join with each other. However, with fixed buffer size B, the decrease of the partition size from $(B-1)$ to $(B-x)$ may cause more cut-edges between partitions, thus generating more outside B-diagonal entries and enlarging the size of the vertex cover. This is a trade-off which we will explore in future work.

Order-Partition chooses a partition ordering, i.e. a loading sequence, using a partition-interaction matrix M. An entry $M[P_i, P_j]$ lists the number of cut-edges between the nodes in partitions P_i and P_j. An example partition-interaction matrix for the join-index of Figure 11 is shown in Table 4. The procedure uses a simple heuristic to construct the loading sequence. It sorts the entries in $M[P_i, P_j]$ in descending order and arbitrarily breaks the tie. It initially chooses the entry with the largest value, arriving at a loading sequence of length 2. Then, it extends the loading on both sides in a greedy manner. For example, suppose $M[P_3, P_2]$ is selected first. Then, the loading sequence P_2-P_3 can be extended to the right by choosing P_4 and extended to the left by choosing P_1 from among the remaining partitions. The choice is based on the highest value of $M[P_2, P_i]$ and $M[P_3, P_j]$ in the partition-interaction matrix. A better heuristic can be designed to select loading sequences that have a higher number of cut-edges between consecutive partitions. To improve the performance of the proposed SC method, we will consider

these heuristics in future work.

	P_1	P_2	P_3	P_4
P_1	*	18	5	0
P_2	18	*	18	10
P_3	5	18	*	18
P_4	0	10	18	*

Table 4: # of cut-edges between partitions P_j and P_i

We use Figure 13 to illustrate the steps of band-diagonalization using graph-partition and order-partition techniques. Figure 13(a) shows an example PCG. Figure 13(b) converts the original PCG into an adjacency matrix. Note that the three edges (R4-S1), (R1-S4), and (R5-S0) are outside the B-diagonal in the adjacency matrix, assuming the buffer size is five. Metis [24] is used to partition this PCG, and each partition has size $(B-1)$, where B is the number of buffers available. Figure 13(c) shows the result after graph-partition. The order-partition procedure uses the partition-interaction matrix, i.e., Figure 13(d), and determines the greedy loading sequence $P1-P2-P3$. The final result, with only one edge (R5-S0) outside the B-diagonal, is displayed in Figure 13(e).

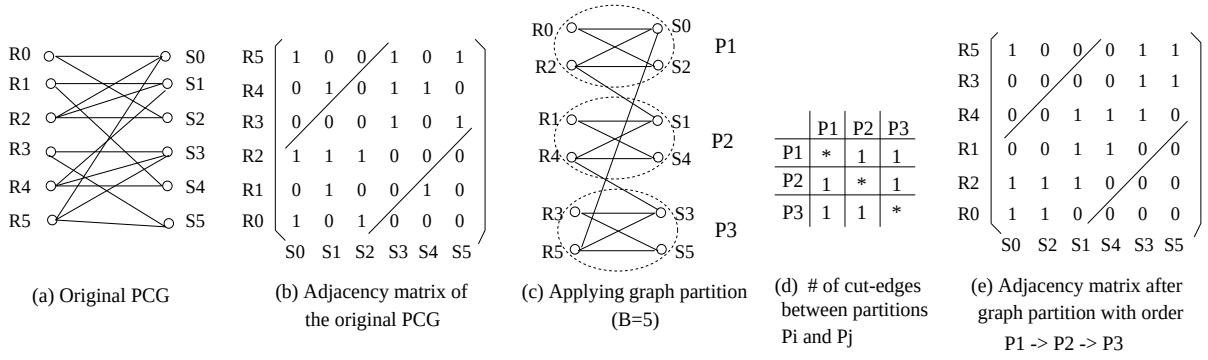
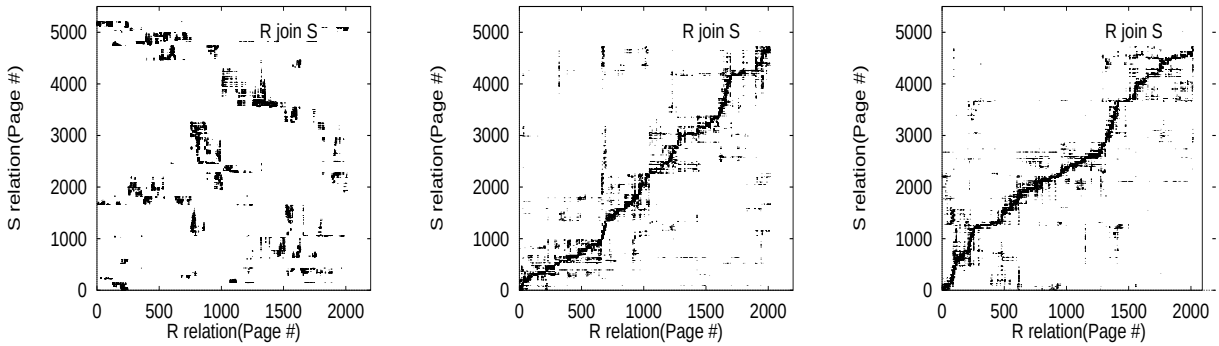


Figure 13: An example for Band-diagonalization

Figure 14 illustrates the effect of band-diagonalization on a real data set. Figure 14(a) is the original PCG relation, where R and S are the two relations to be joined, and where each point in the picture denotes an edge connection between the pages of the two relations. The result after the graph-partition is shown in Figure 14(b): the pages from the R and S relations are relabeled by their partitions. Finally, we order these partitions to bring as many points as possible inside the B-diagonal, as shown in Figure 14(c). In Figure 14(b), 28% of the edges are outside the B-diagonal. After the order-partition, the outside B-diagonal edges are reduced to 22% of the total edges. Edges outside of the B-diagonal can be processed by using a vertex cover, as discussed in the next section.



(a) The original PCG adjacency matrix from a spatial join-index for the Sequoia 2000 dataset

(b) After graph-partition, 28% of the edges are outside the B-diagonal (B=100)

(c) After order-partition, 22% of the edges are outside the B-diagonal (B=100)

Figure 14: Using graph partitioning to derive the B-diagonal

4.3 Vertex Cover Computation

The redundant I/Os in the SC approach are due to the edges, that is, the non-zero matrix elements, outside the B-diagonal of the clustered adjacency-matrix representation of the join-index. These outside-B-diagonal edges are grouped via a vertex-cover algorithm to determine a small set of pages needing redundant I/Os. Determining the minimal vertex cover for a general graph is NP-hard [12]. However, polynomial time algorithms [18] are available for determining the minimal vertex cover for bipartite graphs, such as the PCG for join-indices.

The Find-vertex-cover procedure determines the vertex cover for all of the off-B-diagonal edges by a fast but greedy heuristic described below. The heuristic sorts the nodes related to the off-B-diagonal edges by their degree, that is, the number of incident edges. The node with the highest degree is added to the vertex cover and all of the edges incident on this node are dropped. These steps are repeated to cover all of the off-B-diagonal edges. In the future, we plan to use better algorithms which are likely to improve the performance of the proposed SC method.

Find-vertex-cover

Input: $G_{OBD} = (V_{OBD}, E_{OBD})$ is the graph for off-B-diagonal edges and corresponding nodes
Output: VC is the vertex cover for G_{OBD}

while(E_{OBD} is not empty) {
    $V_{highest}$=Find_highest_degree_node(V_{OBD}); /* *Node* $V_{highest}$ *has the highest degree* */
    $VC = VC \bigcup V_{highest}$; /* *Add this node to the set of vertex cover* */
    Update(G_{OBD}); /* *Update* G_{OBD} *by removing node* $V_{highest}$ *and its corresponding edges* */
}

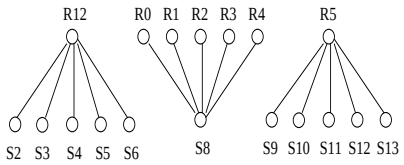


Figure 15: A vertex-cover example

For example, Figure 15 shows the off-B-diagonal edges and their incident nodes derived from Figure 11(b) with the loading sequence $P_1 - P_2 - P_3 - P_4$. By applying the Find-vertex-cover procedure, the vertex cover for these 15 off-B-diagonal edges are nodes R5, R12, and S8. Node R5 covers edges (R5,S9), (R5,S10), (R5,S11), (R5,S12), and (R5, S13). Node R12 covers edges (R12,S2), (R12,S3), (R12,S4), (R12,S5), and (R12,S6). Node S8 covers edges (R0,S8), (R1,S8), (R2,S8), (R3,S8), and (R4,S8).

4.4 Access Sequence Generation

The Access-sequence-generator procedure derives the page-access sequence. It loads each partition in a pre-determined order. When transferring from one partition P_i to the next scheduled partition P_{i+1}, the procedure orders the loading sequence of the nodes using the contour-diagonalization order shown in Figure 10(c). After loading a whole partition, we find all of the off-B-diagonal vertex cover nodes which connect to this partition, and load these nodes, one by one, to compute the join.

Access-sequence-generator

Input: P_{order} is the loading sequence of the partitions;
G_{BD} is the B-diagonal connectivity graph;
VC is the vertex cover for all the Off-B-diagonal edges.
Output: $S =< P_1, P_2, ..., P_r >$ is a page access sequence.

```
for(i = 1; i ≤ |P_order|; i++){
    P_i= GetPartition(P_order, i) /* Get the ith partition */
    if(i==1) { AddPageSequence(S, P_i);/* Add all the nodes within P_1 into the loading sequence */ }
    else { OrderAndAddPageSequence(S, P_{i-1}, P_i);
        /* Order and add the nodes within P_i into the loading sequence by the following rules: */
        /* 1. Add the node within P_i which has the highest connectivity with P_{i-1} in B - diagonal */
        /* 2. Replace the node within P_{i-1} which has finished its join with the nodes in P_i */
    }
    PVC_Set = FindConnected_node_from_VC(P_i,VC);
    /* Find if any Off-B-diagonal vertex cover which connects to this partition */
    AddPageSequence(S, PVC_Set); /* Add these nodes into the loading sequence */
}
```

The computational complexity for SC is $O(|E|) + O(P * log(P)) + O(|V_{OBD}|^2) + O(|V| * log(B-1))$, given a page-connectivity graph $G = (V, E)$ and a buffer size B. The first step, Band-diagonalization, in-

cludes two procedures, namely, Graph-Partition and Order-Partition. The Graph-Partition software we used has computational complexity $O(|E|)$ [26]; the Order-Partition procedure has computational complexity $O(P * log(P))$, where P is the number of partitions, and $P = \lceil \frac{|V|}{B-1} \rceil$. The Find-vertex-cover heuristic for the off-B-diagonal graph $G_{OBD} = (V_{OBD}, E_{OBD})$ has the worst case computational complexity $O(|V_{OBD}|^2)$. The for loop of the Access-sequence-generator executes P times, and the OrderAndAddPageSequence step, with cost $O((B-1) * log(B-1))$, is the dominant cost inside the for loop. The computational complexity for SC is $O(|E|) + O(P * log(P)) + O(|V_{OBD}|^2) + O(|V| * log(B-1))$.

5 Comparative Evaluation of SC, AC and Competitors

The experimental setup is shown in Figure 7. we constructed join-indices for N-nearest-neighbor(Q.A), as well as for distance-based range queries(Q.B) from the Sequoia 2000 [40] data set. Variable parameters included buffer size, page size, and edge ratio. Relation R refers to the GNIS *Point* table, and relation S refers to the Landuse *Polygon* table from the Sequoia 2000 [40] data set. For the sake of brevity, we refer readers to section 2.3.1 for details of the experiment design.

Potential candidate methods for the OPAS-FB problem included AC, SC, FP, OM, CO, and Sorting. However, we did not include the Sorting method since it performed worse than AC on I/O cost measures in our previous experiments.

5.1 Experiment Results

Figures 16 and 17 compare all of the OPAS-FB heuristics for N-nearest-neighbor join-indices and range-query join-indices, respectively. For each experiment, we varied page size, buffer size, and edge ratio.

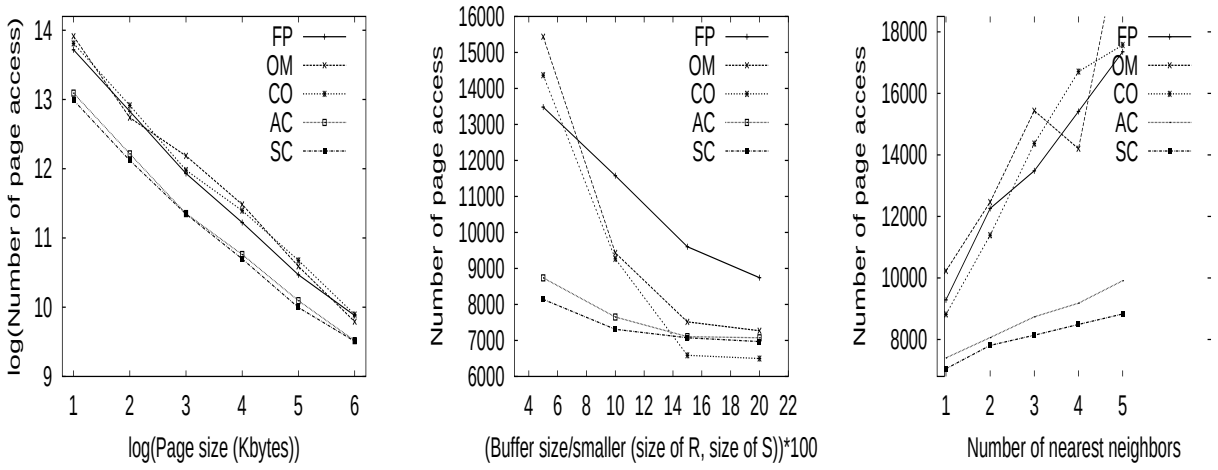
5.1.1 Page size

Page size affects both the clustering of the base relations and the degree of the nodes in the PCG. We studied the effect of page size on the performance of the OPAS-FB methods. Figures 16(a) and 17(a) show the effect of page size, which was varied from 2 Kbytes to 64 Kbytes. The page size and the number of page accesses are shown in the logarithmic form of base two for easy comparison. In the N-nearest-neighbor join-indices, SC and AC outperformed all the other methods using different page sizes. In the Range-query join-indices, SC and AC required fewer page accesses than all the other methods. AC outperformed SC when page size was greater than 64 Kbytes. The OM method performed well with the 4 and 8 Kbytes page size.

5.1.2 Buffer size

Buffer size determines the number of off-B-diagonal edges in the SC method, and the size of the partition in the PCG, i.e., the number of disk-pages that can be bulk-loaded. Larger number of buffers often decrease the number of page accesses.

Figures 16(b) and 17(b) show the effect of buffer size. We varied the number of buffers as a percentage of the number of pages of the smaller relation, and changed the percentages from 5 to 20. As can be seen, AC and SC performed better than all the other methods when the buffer size was relatively low, e.g., 5 to 10 percent. The CO and OM methods did well with large buffer sizes.



(a) Page size: 2K - 64K, Buffer size: 5% of the smaller relation, No of nearest neighbors=3

(b) Buffer size: 5% - 20% of the smaller relation, Page size: 2K, No of nearest neighbors=3(Θ=0.003)

(c) No of nearest neighbors=1-5(Θ=0.002-0.005), Buffer size: 5% of the smaller relation, Page size: 2K

Figure 16: Nearest Neighbor Query: The effect of page size, buffer size, and number of nearest neighbors for different OPAS-FB heuristics

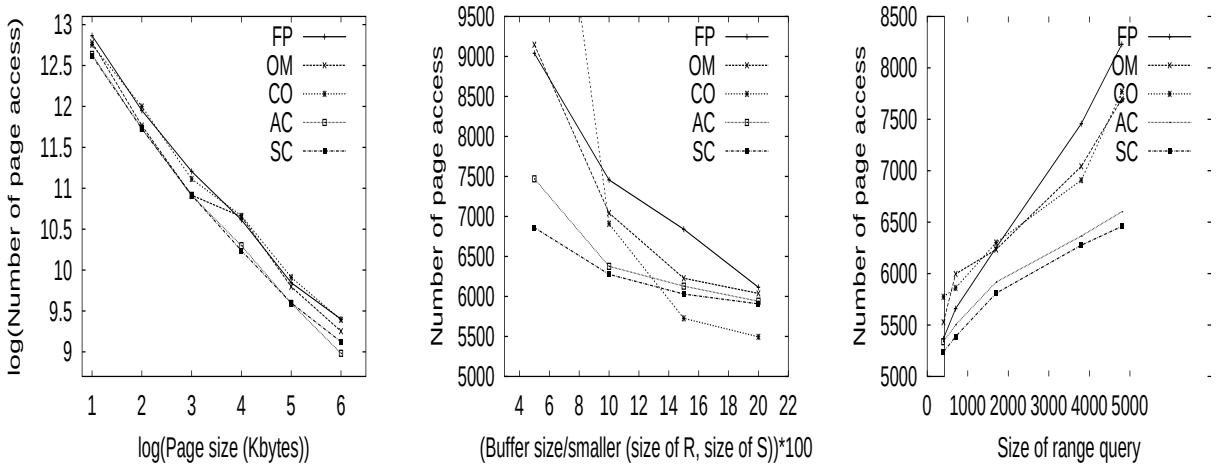
Throughout our experiments, we repeatedly observed that buffer size was the dominant factor determining the relative performance of the AC and SC algorithms. In other words, when the number of buffer size was relatively low, our proposed algorithms tended to outperform other competitors. One possible explanation is that when buffer size is relatively small, methods such as FP, OM, and CO generate many extra page accesses for swapping in and out, while our algorithms apply a global clustering approach to group and load relevant pages together, thus reducing these redundant I/O page accesses.

5.1.3 Edge ratio

The edge ratio is a metric for page-level join selectivity and the degree of connectivity of the bipartite PCG. A high edge ratio graph has pages sharing edges with many other pages, increasing the probability that some of the neighbors are not in memory buffers as well as the likelihood of redundant I/Os.

In this experiment, we changed the edge ratio by both increasing and decreasing the number of nearest neighbors and the size of the range query. The results of the experiment shown in Figures 16(c) and 17(c) indicate that AC and SC uniformly outperformed all other methods, and that SC required fewer page accesses than AC.

Notice that in this experiment, the buffer size was set at a relatively small number for both queries, i.e., 5% in the nearest neighbor query and 10% in the range query. Under this given constraint, as previously stated, SC and AC were expected to perform better than the other methods, even though for all the heuristics, the number of page accesses rises as the edge ratio increases.



(a) Page size: 2K - 64K, Buffer size: 10% of the smaller relation, Range query size = 3800

(b) Buffer size: 5% - 20% of the smaller relation, Page size: 2K, Range query size = 3800(Θ=0.003)

(c) Range query size = 400 - 4800(Θ=0.002-0.003), Buffer size: 10% of the smaller relation, Page size: 2K

Figure 17: Range Query: Effect of page size, buffer size, and number of nearest neighbors for different OPAS-FB heuristics

6 Conclusion and Future Work

In this paper, we introduced two new algorithms for spatial join computation, given a join-index and a fixed buffer size. The key idea is to use spatial clustering. The proposed AC and SC algorithms outperformed the traditional methods in our experiments with the Sequoia 2000 [40] data set, particularly when the size of the memory buffer was small(<10%), relative to the size of the spatial relations. We also provided a formal characterization of an upper bound on the number of redundant I/Os needed by AC and SC.

In the future we would like to improve some of the heuristics chosen in the implementation of AC and SC, as discussed throughout this paper. We would also like to look at related issues regarding maintenance of join-indices in the face of updates and also the interaction of join-indices with the join-computation algorithm. Finally, we are interested in exploring the usefulness of AC and SC in data warehouses [19, 20] (e.g., processing star-joins using the STARindex [19]) and spatial data mining(e.g., neighborhood index [10]).

Acknowledgments

This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work was also supported in part by NSF grant #9631539. We would like to

thank Kim Koffolt for improving the readability of this paper. We also thank Xuan Liu, Xinhong Tan and Weili Wu for their technical comments.

References

[1] S. T. Barnard, A. Pothen, and H. D. Simon. A Spectral Algorithm for Envelope Reduction of Sparse Matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334, 1995.

[2] L. Becker, K. Hinrichs, and U. Finke. A New Algorithm for Computing Joins With Grid Files. In *Proceedings of International Conference on Data Engineering*, 1993.

[3] L. Belady, R. Nelson, and G. Shedler. An anomaly in the space-time characteristics of certain programs running in paging machines. *Communications of the ACM*, 12(6):349–353, June 1969.

[4] C. Berge. *Graphs and Hypergraphs*. American Elsevier, New York, 1976.

[5] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Multi-Step Processing of Spatial Joins. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1994.

[6] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-trees. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1993.

[7] C. Y. Chan and B. C. Ooi. Efficient Scheduling of Page Access in Index-Based Join Processing. *IEEE Transactions on Knowledge and Data Engineering*, 9(6):1005–1011, November/December 1997.

[8] H. T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, Sweden*, pages 127–141, August 1985.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. The MIT Press, 1991.

[10] M. Ester and J. Sander S. Gundlach, H. Kriegel. Database Primitives for Spatial Data Mining. *Proc. Int. Conf. on Databases in Office, Engineering and Science, Freiburg, Germany*, 1999.

[11] F. Fotouhi and S. Pramanik. Optimal Secondary Storage Access Sequence for Performing Relational Join. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):318–328, September 1989.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1993.

[13] A. George and A. Pothen. An Analysis of Spectral Envelope-Reduction via Quadratic Assignment Problems. *SIAM Journal of Matrix Analysis and its Applications*, 18(3):706–732, 1997.

[14] P. Goyal, H.F. Li, E. Regener, and F. Sadri. Scheduling of Page Fetches in Join Operation Using Bc-Trees. In *Proceedings of International Conference on Data Engineering*, 1988.

[15] G. Graefe. Query Evaluation Techniques for Large Databases. *Computing Surveys*, 25(2):73–170, 1993.

[16] O. Gunther. Efficient Computation of Spatial Joins. In *Proceedings of International Conference on Data Engineering*, 1993.

[17] L. Hagen and A. Kahng. Fast Spectral Methods for Ratio Cut Partitioning and Clustering. In *Proceedings of IEEE International Conference on Computer Aided Design*, 1991.

[18] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matching of graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[19] Informix. *White Papers, http://www.informix.com/informix/solutions/dw/redbrick/wpapers/star.html*.

[20] W. H. Inmon. *Building the Data Warehouse*. John Wiley & Sons Inc, 1992.

[21] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. hMetis Home Page. *http://www-users.cs.umn.edu/~karypis/metis/hmetis/main.html*.

[22] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. *Proceedings ACM/IEEE Design Automation Conference*, 1997.

[23] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 7(1):69–79, March 1999.

[24] G. Karypis and V. Kumar. Metis Home Page. *http://www-users.cs.umn.edu/~karypis/metis/metis/main.html*.

[25] G. Karypis and V. Kumar. Parallel Multilevel Graph Partitioning. In *Proceedings of Supercomputing*, November 1996.

[26] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[27] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 1970.

[28] G. Kumfert and A. Pothen. Two Improved Algorithms for Envelope and Wavefront Reduction. *BIT*, 37(3):001–032, 1997.

[29] D. R. Liu and S. Shekhar. A Similarity Graph-Based Approach to Declustering Problem and its Applications Toward Parallelizing Grid Files. In *Proceedings of the Eleventh International Conference on Data Engineering, IEEE*, pages 373–381, March 1995.

[30] M. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 209–220, 1994.

[31] T. Merrett, Y. Kimbayasi, and H. Yasuura. Scheduling of Page-Fetches in Join Operations. In *Proceedings of the 7th International Conference on Very Large Databases*, 1981.

[32] P. Mishra and M.H. Eich. Join Processing in Relational Databases. *Computing Surveys*, 24(1):63–113, 1992.

[33] E. R. Omiecinski. Heuristics for Join Processing Using Nonclustered Indexes. *IEEE Transactions on Software Engineering*, 15(1):18–25, January 1989.

[34] S. Pramanik and D. Ittner. Use of Graph Theoretic Models for Optimal Relational Database Accesses to Perform Join. *ACM Transactions on Database Systems*, 10(1):57–74, March 1985.

[35] D. Rotem. Spatial Join Indices. In *Proceedings of International Conference on Data Engineering*, 1991.

[36] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the 8th International Conference on Very Large Data Bases, Mexico City, Mexico*, pages 257–262, September 1982.

[37] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C.T. Lu. Spatial Databases: Accomplishments and Research Needs. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):45–55, 1999.

[38] S. Shekhar and D. R. Liu. CCAM: A Connectivity-Clustered Access Method for Networks and Networks Computions. *IEEE Trans. on Knowledge and Data Engineering*, 9(1), January 1997.

[39] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[40] M. Stonebraker, J. Frew, and J. Dozier. The Sequoia 2000 Project. In *Proceedings of the Third International Symposium on Large Spatial Databases*, 1993.

[41] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

A Summary trace for different algorithms in Figure 12

Table 5 provides a summary trace of various heuristics. Each row represents a new page being fetched into main memory. Thus, the number of rows represents the total number of pages fetched. For simplicity, the clustering of I/O for multiple pages is not modeled. An entry $(+R0)$ for Sorting in iteration 1 means that page $R0$ was fetched. An entry $(+S1, = S1)$ for iteration 9 of Sorting implies that $S1$ was fetched into memory$(+S1)$ and that all edges incident on $S1$ were processed with the pages available in the buffer. The set of pages available in the buffer in this iteration are $\{R0, R1, R2, \ldots, R6\}$, since we have fetched those in previous iterations. The buffer containing $S1$ can be reused in the next iteration to bring in $S2$, as shown by entry $(+S2)$ for Sorting in iteration 10. The next interesting entry is $(-S2, +S3)$ in iteration 11 for Sorting, where the buffer containing $S2$ is overwritten by incoming page $S3$ even though some edges (e.g. $R7 - S2$) incident cannot be processed right away. This is due to a buffer size fixed at eight. Note that $S2$ returns to memory in iteration 29 to process the edge $(R7 - S2)$. This leads to a redundant I/O. Note that because the graph is highly connected in spatial order, AC does not re-cluster the R relation, and generates the same loading sequence as the Sorting heuristic.

Traces of other algorithms are shown in other columns using 8 buffers. Note that the number of last rows with a $'+'$ entry(page fetch) designates the total number of page I/Os for an algorithm. In other words, AC/Sorting-based algorithm has 40 I/Os, CO's heuristic has 35, OM has 33, FF has 45 and SC has 31, as shown in the last row labeled I/O count.

Table 6, 7, 8, 9, and 10 in Appendix B provide the detailed execution traces for AC/Sorting-based, CO, OM, FP, and SC, respectively, for interested readers.

	Algorithms for computing Join with a Join Index				
Iteration	**AC/Sorting-based**	**CO**	**OM**	**FP**	**SC**
1	+R0	+R0	+R0	+R0	+R0
2	+R1	+S0	+S0	+S0	+R1
3	+R2	+S1	+S1	+R1	+R2
4	+R3	+S2	+S2	+S1	+R3
5	+R4	+S8,=R0	+S8,=R0	+R2	+R4
6	+R5	+R1	+R1	+S2	+S0
7	+R6	+S3,=R1	+S3,=R1	+R3	+S1
8	+S0,=S0	+R2	+R2	+S8,=R0	+S8
9	+S1,=S1	+S4,=R2	+S4,=R2	+R4	-S8,+S2,=R0
10	+S2	+R3	+R3	-R1,+R6	+S3,=R1
11	-S2,+S3	+S5,=R3	+S5,=R3	-S0,+S3	+S4,=R2
12	-S3,+S4	+R4	+R4	-R2,+R7	+R5,=S0
13	-S4,+S5	-S0,+S6,=R4	-R4,+R5,=S0	-S1,+S4	+S5,=R3
14	-S5,+S6	+R6	+R6,=S1	-R3,+R12	+R6,=S1
15	-S6,+S7	-S1,+S7,=R6	+R7	-S2,+S5	+S6,=R4
16	-S7,+S8	+R7	-R6,+R12,=S2	-R4,+S8	+R12
17	-S8,+S9	-S2,+S9,=R7	+R8,=S3	-R6,+S7	-R12,+R7,=S2
18	-S9,+S10	+R8	+R9,=S4	-R7,+R9	+R8,=S3
19	-S10,+S11	-S3,+S10,=R8	+R10,=S5	-S3,+S10	+R9,=S4
20	-S11,+S12	+R9	+R13	-R8,+R10	+R10,=S5
21	-S12,+S13,=R0	-S4,+S11,=R9	-R7,+R11,=S8	-S4,+S11	+S7
22	-S13,+R7,=R1	+R10	+S10,=S10	-R9,+R11	-R5,+R11,=S6
23	+R8,=R2	-S5,+S12,=R10	+S9	-S8,+R5	+S8,=R6
24	+R9,=R3	+R11	-R8,+R7,=S9	-S5,+S12	+S9,=R7
25	+R10,=R4	-S6,+S13,=R11	+S7	-S7,+S9	+S10,=R8
26	+R11,=R5	+R13,=R13	-S7,+S6,=R7	-R10,+R13,=S10	+S11,=R9
27	+R12,=R6	+R12,=S8	+S11,=R9,=S11	+S13,-S13,=S11	+R12,=S7
28	+R13	+R5,=S7	+S12,=R10,=S12	+S6,=R11,=R12	+S12,=R10
29	+S2,=S2	+S0,=S0,=S9	+S13,=R5,=R11	+R7,=R7,=S12	+R13,=S8
30	+S3,=S3	+S1,=S1,=S10	+R4,=R4,=R12	+R8,=R8	+S13,=R11
31	+S4,=S4	+S2,=S2,=S11	+R6,=R13,=S13	+R9,=R9,=S9	+R5
32	+S5,=S5	+S3,=S3,=S12	+R8,=S6	+S0,=S0	
33	+S6,=S6	+S4,=S4,=S13	+S7	+S1,=S1	
34	+S7,=S7	+S5,=S5		+S2,=S2	
35	+S8,=S8	+S6		+S8,=R13,=S8	
36	+S9,=S9			+R4,=R4	
37	+S10,=S10			+R10,=R10	
38	+S11,=S11			+S3	
39	+S12,=S12			+R1,=R1,=S3	
40	+S13			+S4,=R5	
41				+R2,=R2,=S4	
42				+R6,=S6	
43				+S7,=R6,=S7	
44				+R3	
45				+S5	
I/O count	**40**	**35**	**33**	**45**	**31**

Table 5: Summary of results for all algorithms(Buffer size=8, -:Swap Out, +:Add, =: Done)

B Trace for different algorithms in Figure 12

B.1 Execution Trace for AC/Sorting method

Table 6 shows the behavior of the Sorting method for computing a join, given the join-index of Figure 11. Table 6 has five columns. The first column shows the iteration number. The second column shows the node swapped out in the current iteration. The third column shows the node selected and brought into the memory buffers. The fourth and fifth columns show the pages of R and S in the main memory buffer. The last column shows the nodes which have been processed completely and need not come into the memory buffer again.

Iteration	Swap out	Add	Page of R in Buffer	Page of S in Buffer	Done
1		R0	R0		
2		R1	R0,R1		
3		R2	R0,R1,R2		
4		R3	R0,R1,R2,R3		
5		R4	R0,R1,R2,R3,R4		
6		R5	R0,R1,R2,R3,R4,R5		
7		R6	R0,R1,R2,R3,R4,R5,R6		
8		S0	R0,R1,R2,R3,R4,R5,R6	S0	S0
9		S1	R0,R1,R2,R3,R4,R5,R6	S1	S1
10		S2	R0,R1,R2,R3,R4,R5,R6	S2	
11	S2	S3	R0,R1,R2,R3,R4,R5,R6	S3	
12	S3	S4	R0,R1,R2,R3,R4,R5,R6	S4	
13	S4	S5	R0,R1,R2,R3,R4,R5,R6	S5	
14	S5	S6	R0,R1,R2,R3,R4,R5,R6	S6	
15	S6	S7	R0,R1,R2,R3,R4,R5,R6	S7	
16	S7	S8	R0,R1,R2,R3,R4,R5,R6	S8	
17	S8	S9	R0,R1,R2,R3,R4,R5,R6	S9	
18	S9	S10	R0,R1,R2,R3,R4,R5,R6	S10	
19	S10	S11	R0,R1,R2,R3,R4,R5,R6	S11	
20	S11	S12	R0,R1,R2,R3,R4,R5,R6	S12	
21	S12	S13	R0,R1,R2,R3,R4,R5,R6	S13	R0,R1,R2,R3,R4,R5,R6
22	S13	R7	R7		
23		R8	R7,R8		
24		R9	R7,R8,R9		
25		R10	R7,R8,R9,R10		
26		R11	R7,R8,R9,R10,R11		
27		R12	R7,R8,R9,R10,R11,R12		
28		R13	R7,R8,R9,R10,R11,R12,R13		
29		S2	R7,R8,R9,R10,R11,R12,R13	S2	S2
30		S3	R7,R8,R9,R10,R11,R12,R13	S3	S3
31		S4	R7,R8,R9,R10,R11,R12,R13	S4	S4
32		S5	R7,R8,R9,R10,R11,R12,R13	S5	S5
33		S6	R7,R8,R9,R10,R11,R12,R13	S6	S6
34		S7	R7,R8,R9,R10,R11,R12,R13	S7	S7
35		S8	R7,R8,R9,R10,R11,R12,R13	S8	S8
36		S9	R7,R8,R9,R10,R11,R12,R13	S9	S9
37		S10	R7,R8,R9,R10,R11,R12,R13	S10	S10
38		S11	R7,R8,R9,R10,R11,R12,R13	S11	S11
39		S12	R7,R8,R9,R10,R11,R12,R13	S12	S12
40		S13	R7,R8,R9,R10,R11,R12,R13	S13	R7,R8,R9,R10,R11,R12,R13,S13

Table 6: Example: the AC/Sorting-based method(Buffer size=8)

B.2 Execution Trace for the CO's method

Table 7 shows the behavior of CO's method for computing a join, given the join-index of Figure 11. Table 7 has five columns. The first column shows the iteration number. The second column shows the node swapped out in the current iteration. The third column shows the node selected and brought into the memory buffers. The fourth and fifth columns show the pages of R and S in the main memory buffer. The last column shows the nodes which have been processed completely and need not come into the memory buffer again.

Iteration	Swap out	Add	R Buffer	S Buffer	Done
1		R0	R0		
2		S0	R0	S0	
3		S1	R0	S0,S1	
4		S2	R0	S0,S1,S2	
5		S8	R0	S1,S2,**S8**	R0
6		R1	R1	S0,S1,S2,**S8**	
7		S3	R1	S0,S1,S2,**S8**,S3	R1
8		R2	R2	S0,S1,S2,**S8**,S3	
9		S4	R2	S0,S1,S2,**S8**,S3,S4	R2
10		R3	R3	S0,S1,S2,**S8**,S3,S4	
11		S5	R3	S0,S1,S2,**S8**,S3,S4,S5	R3
12		R4	R4	S0,S1,S2,**S8**,S3,S4,S5	
13	S0	S6	R4	S1,S2,**S8**,S3,S4,S5,S6	R4
14		R6	R6	S1,S2,**S8**,S3,S4,S5,S6	
15	S1	S7	R6	S2,S8,S3,S4,S5,S6,S7	R6
16		R7	R7	S2,S8,S3,S4,S5,S6,S7	
17	S2	S9	R7	S8,S3,S4,S5,S6,S7,S9	R7
18		R8	R8	S8,S3,S4,S5,S6,S7,S9	
19	S3	S10	R8	S8,S4,S5,S6,S7,S9,S10	R8
20		R9	R9	S8,S4,S5,S6,S7,S9,S10	
21	S4	S11	R9	S8,S5,S6,S7,S9,S10,S11	R9
22		R10	R10	S8,S5,S6,S7,S9,S10,S11	
23	S5	S12	R10	S8,S6,S7,S9,S10,S11,S12	R10
24		R11	R11	S8,S6,S7,S9,S10,S11,S12	
25	S6	S13	R11	S8,S7,S9,S10,S11,S12,S13	R11
26		R13	R13	S8,S7,S9,S10,S11,S12,S13	R13
27		R12	R12	S8,S7,S9,S10,S11,S12,S13	S8
28		R5	R5,**R12**	S7,S9,S10,S11,S12,S13	S7,S9,S10,S11,S12,S13
29		S0	R5,**R12**	S0	S0
30		S1	R5,**R12**	S1	S1
31		S2	R5,**R12**	S2	S2
32		S3	R5,**R12**	S3	S3
33		S4	R5,**R12**	S4	S4
34		S5	R5,**R12**	S5	S5
35		S6	R5,**R12**	S6	R12,R5,S6

Table 7: Example: the CO's method(Buffer size=8)

B.3 Execution Trace for OM's method

Table 8 shows the behavior of OM's method for computing a join, given the join-index of Figure 11. Table 8 has five columns. The first column shows the iteration number. The second column shows the node swapped out in the current iteration. The third column shows the node selected and brought into the memory buffers. The fourth and fifth columns show the pages of R and S in the main memory buffer. The last column shows the nodes which have been processed completely and need not come into the memory buffer again.

Iteration	Swap out	Add	R Buffer	S Buffer	Done
1		R0	R0		
2		S0	R0	S0	
3		S1	R0	S0,S1	
4		S2	R0	S0,S1,S2	
5		S8	R0	S0,S1,S2,**S8**	R0
6		R1	R1	S0,S1,S2,**S8**	
7		S3	R1	S0,S1,S2,**S8**,S3	R1
8		R2	R2	S0,S1,S2,**S8**,S3	
9		S4	R2	S0,S1,S2,**S8**,S3,S4	R2
10		R3	R3	S0,S1,S2,**S8**,S3,S4	
11		S5	R3	S0,S1,S2,**S8**,S3,S4,S5	R3
12		R4	R4	S0,S1,S2,**S8**,S3,S4,S5	
13	R4	R5	R5	S0,S1,S2,**S8**,S3,S4,S5	S0
14		R6	R5,R6	S1,S2,**S8**,S3,S4,S5	S1
15		R7	R5,R6,R7	S2,**S8**,S3,S4,S5	
16	R6	R12	R5,R7,**R12**	S2,**S8**,S3,S4,S5	S2
17		R8	R5,R7,**R12**,R8	**S8**,S3,S4,S5	S3
18		R9	R5,R7,**R12**,R8,R9	**S8**,S4,S5	S4
19		R10	R5,R7,**R12**,R8,R9,R10	**S8**,S5	S5
20		R13	R5,R7,**R12**,R8,R9,R10,R13	S8	
21	R7	R11	R5,R12,R8,R9,R10,R13,R11	S8	S8
22		S10	R5,R12,R8,R9,R10,R13,R11	S10	S10
23		S9	R5,R12,R8,R9,R10,R13,R11	S9	
24	R8	R7	R5,R12,R9,R10,R13,R11,R7	S9	S9
25		S7	R5,R12,R9,R10,R13,R11,R7	S7	
26	S7	S6	R5,R12,R9,R10,R13,R11,R7	S6	R7
27		S11	R5,R12,R9,R10,R13,R11	S6,**S11**	R9,S11
28		S12	R5,R12,R10,R13,R11	S6,**S12**	R10,S12
29		S13	R5,R12,R13,R11	S6,**S13**	R5,R11,R12,R13,S13
30		R4	R4	S6	R4
31		R6	R6	S6	
32		R8	R6,**R8**	S6	S6
33		S7	R6,**R8**	S7	R6,R8,S7

Table 8: Example: the OM's method(Buffer size =8)

B.4 Execution Trace for FP's method

Table 9 shows the behavior of FP's method for computing a join, given the join-index of Figure 11. Table 9 has five columns. The first column shows the iteration number. The second column shows the node swapped out in the current iteration. The third column shows the node selected and brought into the memory buffers. The fourth and fifth columns show the pages of R and S in the main memory buffer. The last column shows the nodes which have been processed completely and need not come into the memory buffer again.

Iteration	Swap out	Add	R Buffer	S Buffer	Done
1		R0	R0		
2		S0	R0	S0	
3		R1	R0,R1	S0	
4		S1	R0,R1	S0,S1	
5		R2	R0,R1,R2	S0,S1	
6		S2	R0,R1,R2	S0,S1,S2	
7		R3	R0,R1,R2,R3	S0,S1,S2	
8		S8	R0,R1,R2,R3	S0,S1,S2,**S8**	R0
9		R4	R1,R2,R3,R4	S0,S1,S2,**S8**	
10	R1	R6	R2,R3,R4,**R6**	S0,S1,S2,**S8**	
11	S0	S3	R2,R3,R4,**R6**	S1,S2,**S8**,S3	
12	R2	R7	R3,R4,**R6**,R7	S1,S2,**S8**,S3	
13	S1	S4	R3,R4,**R6**,R7	S2,**S8**,S3,S4	
14	R3	R12	R4,**R6**,R7,R12	S2,**S8**,S3,S4	
15	S2	S5	R4,**R6**,R7,R12	**S8**,S3,S4,S5	
16	R4	R8	R6,R7,**R12**,R8	**S8**,S3,S4,S5	
17	R6	S7	R7,**R12**,R8	S8,S3,S4,S5,**S7**	
18	R7	R9	**R12**,R8,R9	S8,S3,S4,S5,**S7**	
19	S3	S10	**R12**,R8,R9	S8,S4,S5,**S7**,**S10**	
20	R8	R10	**R12**,R9,R10	S8,S4,S5,**S7**,**S10**	
21	S4	S11	**R12**,R9,R10	S8,S5,**S7**,**S10**,S11	
22	R9	R11	R12,R10,R11	S8,S5,**S7**,**S10**,S11	
23	S8	R5	R12,**R10**,R11,R5	S5,**S7**,**S10**,S11	
24	S5	S12	R12,**R10**,R11,R5	S7,**S10**,S11,S12	
25	S7	S9	R12,**R10**,R11,R5	S10,S11,S12,S9	
26	R10	R13	R12,**R11**,R5,R13	S10,S11,S12,S9	S10,S11,S12
27		S13	R12,**R11**,R5,R13	S9,S13	S13
28		S6	R12,**R11**,R5,R13	**S9**,S6	R11,R12
29		R7	R5,**R13**,**R7**	**S9**,S6	R7
30		R8	R5,**R13**,**R8**	**S9**,S6	R8
31		R9	R5,**R13**,**R9**	**S9**,S6	R9,S9
32		S0	R5,**R13**	**S6**,S0	S0
33		S1	R5,**R13**	**S6**,S1	S1
34		S2	R5,**R13**	**S6**,S2	S2
35		S8	R5,**R13**	S6,**S8**	R13,S8
36		R4	R5,R4	S6	R4
37		R10	R5,**R10**	S6	R10
38		S3	R5	**S6**,S3	
39		R1	**R5**,R1	**S6**,S3	R1,S3
40		S4	R5	**S6**,S4	R5
41		R2	R2	**S6**,S4	R2,S4
42		R6	R6	S6	S6
43		S7	R6	S7	R6,S7
44		R3	R3		
45		S5	S5		R3,S5

Table 9: Example: the FP's method(Buffer size =8)

B.5 Execution Trace for SC

Table 10 shows the behavior of SC's method for computing a join, given the join-index of Figure 11. Table 10 has five columns. The first column shows the iteration number. The second column shows the node swapped out in the current iteration. The third column shows the node selected and brought into the memory buffers. The fourth and fifth columns show the pages of R and S in the main memory buffer. The sixth column shows the nodes which have been processed completely and need not come into the memory buffer again. The last column shows the partition number.

Iteration	Swap out	Add	R Buffer	S Buffer	Done	Partition
1		R0	R0			1
2		R1	R0,R1			1
3		R2	R0,R1,R2			1
4		R3	R0,R1,R2,R3			1
5		R4	R0,R1,R2,R3,R4			1
6		S0	R0,R1,R2,R3,R4	S0		1
7		S1	R0,R1,R2,R3,R4	S0,S1		1
8		S8	R0,R1,R2,R3,R4	S0,S1,**S8**		1
9	S8	S2	R0,R1,R2,R3,R4	S0,S1,S2	R0	2
10		S3	R1,R2,R3,R4	S0,S1,S2,S3	R1	2
11		S4	R2,R3,R4	S0,S1,S2,S3,S4	R2	2
12		R5	R3,R4,R5	S0,S1,S2,S3,S4	S0	2
13		S5	R3,R4,R5	S1,S2,S3,S4,S5	R3	2
14		R6	R4,R5,R6	S1,S2,S3,S4,S5	S1	2
15		S6	R4,R5,R6	S2,S3,S4,S5,S6	R4	2
16		R12	R5,R6,**R12**	S2,S3,S4,S5,S6		2
17	R12	R7	R5,R6,R7	S2,S3,S4,S5,S6	S2	3
18		R8	R5,R6,R7,R8	S3,S4,S5,S6	S3	3
19		R9	R5,R6,R7,R8,R9	S4,S5,S6	S4	3
20		R10	R5,R6,R7,R8,R9,R10	S5,S6	S5	3
21		S7	R5,R6,R7,R8,R9,R10	S6,S7		3
22	R5	R11	R6,R7,R8,R9,R10,R11	S6,S7	S6	3
23		S8	R6,R7,R8,R9,R10,R11	S7,S8	R6	3
24		S9	R7,R8,R9,R10,R11	S7,S8,S9	R7	4
25		S10	R8,R9,R10,R11	S7,S8,S9,S10	R8	4
26		S11	R9,R10,R11	S7,S8,S9,S10,S11	R9	4
27		R12	R10,R11,R12	S7,S8,S9,S10,S11	S7	4
28		S12	R10,R11,R12	S8,S9,S10,S11,S12	R10	4
29		R13	R11,R12,R13	S8,S9,S10,S11,S12	S8	4
30		S13	R11,R12,R13	S9,S10,S11,S12,S13	R11,R12,R13	4
31		R5	R5	S9,S10,S11,S12,S13	R5,S9,S10,S11,S12,S13	4

Table 10: Example: the SC method(Buffer size =8)