Optimizing Join Index Based Spatial-Join Processing: A Graph Partitioning Approach

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

June 1, 1999

Abstract

A Join Index is a data structure that optimizes the join query processing in spatial databases. Join indices use pre-computation techniques to speed up online query processing and are useful for applications which require low update rates. The cost of spatial 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 the join-index, we introduce a suite of methods based on spatial-clustering to compute the spatial-join. The spatial clustering we employ is based on graph partitioning techniques. For all the methods we derive upper-bounds on the lengths of the page-access sequence. Experimental results with Sequoia 2000 data sets, on a sequential system, show that spatial clustering method outperforms the existing methods based on sorting and online clustering heuristics.

Acronym	Full form	Definition section/page
AGP	Asymmetric Graph Partitioning based heuristic	Section 3
SGP	Symmetric Graph Partitioning Based heuristic	Section 5
FP	Fotouhi and Pramanik's heuristic	Section 2
OM	Omiecinski's heuristic	Section 2
Chan	Chan's heuristic	Section 2
Sorting	Sorting heuristic	Section 2
OPAS-FB	Optimal Page Access Sequence with Fixed Buffer	Section 1
PCG	Page Connectivity Graph	Section 1
B-diagonal matrix M	$M[i,j] = 1 \Rightarrow \lvert i-j \rvert \leq \lfloor \frac{B}{2} \rfloor$	Section 5

Table 1: Table of Acronym

Keywords: Data Clustering, Join Index, Partitioning, Hyper Graph, Query Processing, Spatial Join.

1 Introduction

The join operation is a fundamental operation in databases, and it has been a subject of intense scrutiny by leading database researchers. Much work has been done in optimizing join operation [13, 26]. A join index [2, 8, 20, 24, 30, 33] is a data structure that facilitates rapid join-query processing. For data sets which are updated infrequently and use pre-computation and materialization techniques to speed up online query processing, the join index can be particularly useful. A fully materialized relationship keeps the complete result of a likely query as a separate relation. Then when a query requests the stored relationship, the result is immediately sent out, resulting in a very fast response time. But the fully materialized relationship has high storage overhead, while a partially materialized relationship (e.g. a join index) stores part of the result to reduce storage overhead, and then requires more processing during the query processing step.

The join-index is typically represented as a bi-partite graph between the pages of encumbent relations or their surrogates to compute the join. When the number of buffer pages is fixed, the join-computation problem is transformed into determining an optimal 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 for the case when the buffer size is equal to two pages [25, 28], and consequently, it is unlikely that a polynomial time solution exists for this problem. Solutions in literature use a global clustering method to group pages in one or both tables involved in join to reduce total page access. Available heuristics either group pages of a single table via global sorting [33] or use incremental clustering methods [6, 9, 27]. We introduce two new heuristics for this problem. One heuristic uses global clustering method to group pages in both tables. The other one uses global clustering for pages of a single table using join-index information. Both methods use min-cut graph partitioning[†] as clustering algorithm. The former outperforms the incremental clustering methods while the latter outperforms global sorting heuristics for spatial join. .

1.1 Basic concept of a Join Index

Consider a database with two relations Facility and Forest Stand. Facility has a point attribute representing its location, and Forest Stand has a rectangle attributes representing bounding box of its extend. The polygon representing its extend may be stored separately. A point consists of the x and y coordinates on the map. A rectangle consists of two points, which are the bottom left and top right corners of the rectangle region on a map.

In Figure 1(a), points $a1, a2, a3, b1, b2$ represent facility locations and polygons $A1, A2, B1, B2, C1, C2$ represent the bounding boxes of the boundary of forest boundaries of forest stands. The circle around each location shows the area within distance D from a facility. The rectangle around each forest boundary represents 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 attributes of unique id, $R.ID$, the location (x,y coordinates), and other non-spatial attributes. Relation S represents the forest stands via unique identifier, $S.ID$, the MOBR and non-spatial attributes. MOBR$(X_{LL}, Y_{LL}, X_{UR}, Y_{UR})$,

[†]Recent advances have provided scalable graph-partitioning software such as Metis [19], which can handle large graphs relevant to database in relative reasonable response time, e.g. few seconds. We have had good experiment using it for database problems [22, 31]

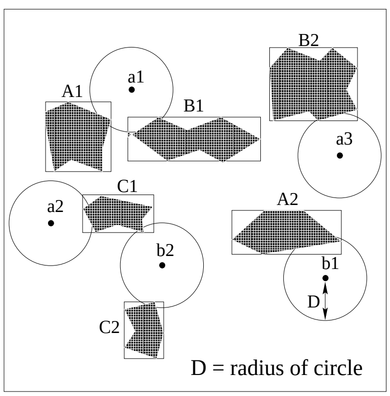
is represented via the coordinates of lower-left corner point(X_{LL}, Y_{LL}) and the upper right corner point(X_{UR}, Y_{UR}).

Now, consider the following query:

Q: "Find all forest stands which are within a distance D from a facility".

This query will require a join on the Facility and Forest Stand relations based on their spatial attributes,

Spatial join algorithm [1, 4, 5, 14, 23] may be used to find the pairs (Facility, Forest-stand) which satisfy the query Q. Alternatively, a join-index may be used to materialize a subset of result to speed up processing for future occurrence of Q if there are few updates to spatial data and the query Q is frequently requested. 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 the 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 to simplify the diagrams in this paper.



(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: Construct Join Index from two relations

1.2 Join Index, Page Connectivity Graph, Join Processing

A join index describes a relationship between the objects of two relations. Assume that each tuple of a relation has a surrogate which uniquely identifies that tuple. A join index is a sequence of pairs of surrogates, where 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. Then the join index is 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 their spatial attribute. 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. The average degree of all the nodes depends on the join selectivity: the higher the degree, the higher the join selectivity.

When the join relationship between two relations is described at the page level, we get a page-connectivity graph. A Page-Connectivity Graph (PCG) [25] $B_G = (V_1, V_2, E)$ is a bipartite graph where vertices V_1 represent the pages from the first relation, and vertices V_2 represent the pages from the second relation. The set of edges is constructed as follows: an edge is added between page (node) v_1^i and page (node) v_2^j, iff 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 2 shows a page-connectivity graph for a join index where nodes a, b represent the pages of relation R, and nodes A, B, C represent the pages of relation S. An edge represents a page-join between a pair of pages. A page-join represents the corresponding join between the tuples in the two pages.

A *min-cut* node partition [15, 21] of graph $G = (V, E)$ 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 [19, 17] for this problem have become available in recent years. They can be used to spatially cluster pages in PCG.

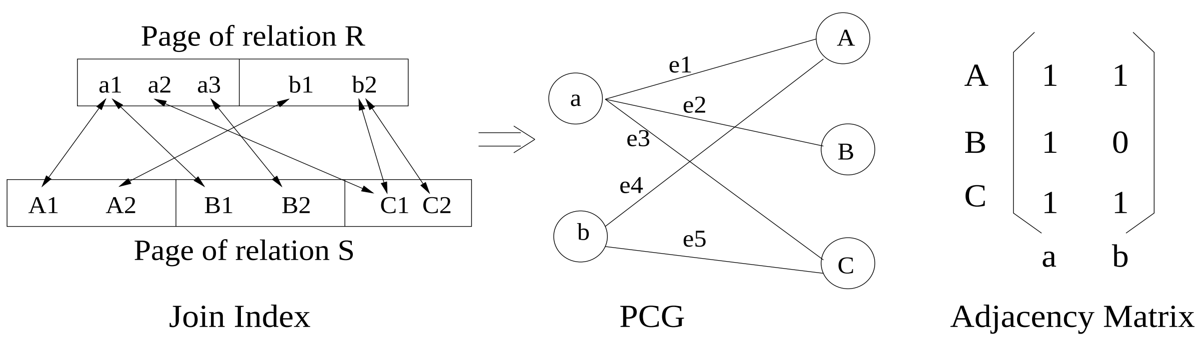


Figure 2: Construction of a Page Connectivity Graph(PCG) from a Join Index.

A join index helps speedup the join processing, as it keeps track of all of the pairs of tuples which satisfy the join predicate. Given a join index JI, one can use the derived PCG to schedule an efficient page access sequence to fetch the data pages. The CPU cost is fixed, as 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 illustrates 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 2. 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, as 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. In Figure 2, with three buffer spaces, (a, B, A, C, b) is a page-access sequence which has 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.

1.3 Problem Definition, Scope, Outline

Given that the I/O cost depends on the page sequence, the following optimization problem is defined for join processing, using join-index in sequential systems. The objective is to determine an ordered list of page accesses which minimize the total page accesses, given a buffer of size B. Here it is important to guarantee that there will never be more than B pages in main memory. We call this the Optimal Page Access Sequence with a Fixed Buffer (OPAS-FB) problem [25]. This problem 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 2 for $B = 2$ is (a, A, b, C, a, B), which results in six page accesses.

Scope: In this paper, we focus on the OPAS-FB problem. We do not address the update problems associated with managing join indices. For example, a join index may have to change if the underlying base relations change. Also, the the clustering of base relations and tuple-level join-index optimization are out of the scope of this paper.

Outline: The rest of the paper is organized as follows. In Section 2, we describe the related work and our contributions. In Section 3, we propose our first approach, Asymmetric Graph Partitioning based heuristic(**AGP**). In Section 4, AGP is evaluated and compared with Sorting heuristic. In Section 5, the second approach, Symmetric Graph Partitioning based heuristic(**SGP**), is proposed, along with some refinement techniques. In Section 6, we experimentally evaluate the second approach and its refinement techniques *vis-a-vis* each other. In Section 7, we compare our algorithms, AGP and SGP, with other known algorithms for the OPAS-FB problem. In Section 8, we conclude with a summary and future directions.

2 Related Work and Our Contributions

The OPAS-FB problem is known to be NP-hard [25, 28], and heuristic solutions have been proposed in the literature for solving this problem. The heuristics in literature can be broadly divided into two groups, namely asymmetric single table clustering and symmetric two-table on-line clustering. The main approach within **asymmetric single table clustering** is based on sorting one of the tables on the join key. In the following discussion, let R and S be the two relations, with JI being the join index. A ***sorting-based*** asymmetric heuristic presented in [33] reads in as much of the join index (JI) and one relevant relation (R semi-join JI) into memory as possible. Here JI is assumed to be clustered on relation R. 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 to reduce redundant accesses to S. This heuristic is economical, and it ensures that no redundant accesses are performed on relation R, but it may incur redundant accesses to the second relation. Sorting-based heuristic assumes totally ordered join-keys. We extend sorting-based method to spatial domain where the keys (e.g. coordinates in multi-dimensional space) are not totally ordered, by proposing AGP. It uses min-cut graph partitioning of the asymmetric hypergraph representing the join-index. Nodes in this graph represent pages of one table(R). A hyper edge represents a page connection of the other table(S). A hyperedge connects are pages of R with edge to a single page of S. AGP clusters pages of R based on their interaction with pages of S to reduce redundant I/O of pages in S.

The main approach in **symmetric two-table clustering** are based on either Traveling Salesman Problem heuristics or selecting next page or next set of pages to be fetched into memory given the pages in buffer and remaining edges to be processed in the bi-partite page-connectivity graph. The selection is often based on the number of neighbors in memory buffers and number of neighbors on the disk. Details of actual heuristics follow. A traveling salesperson-(TSP) based heuristic [12] 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 memory, as the complete graph grows quadratically with the number of nodes in the smaller of the relations.

Symmetric Heuristic: FP, proposed by Fotouhi and Pramanik [9], is designed for general join graphs. The buffer is initialized with a node which has the smallest degree in the page-connectivity graph. The memory buffer is added with the largest resident degree node. The resident degree of a node A is the number of nodes which are connected to A and are in memory. If there is more than one node with 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, a node with the smallest number of edges with the nodes on the disk can be swapped out.

Symmetric Heuristic: OM, developed by Omiecinski [27], is designed specifically for bipartite join graphs. Initially, choose an R and S node from the page-connectivity graph, e.g., r_i and s_j, to load in the buffer such that (a) (r_i, s_j) is connected. (b)The sum of the degree of r_i and s_j is minimal. The buffer is added with the new node, call it p, using the following strategy: (a)find a node q in the buffer such that the node is connected to the fewest number of nodes outside the buffer, and (b) find the node p, such that (q,p) is connected, and such that the number of edges connecting p to a node not in the buffer is minimal. If a node in the buffer has to be replaced to make room for the new node, then choose the node that (a) is connected to the fewest number of nodes outside the buffer, and (b) is not connected to the new node.

Symmetric Heuristic: Chan [6] first, a SelectSegment heuristic selects the minimal segment that has the shortest non-resident length. From this minimal segment, a SelectPage heuristic chooses the page that has the largest resident degree from this segment. For the selection of victim pages for replacement when buffer is full, the SelectVictim heuristic selects the page with the smallest nonresident degree.

Other heuristics are also proposed for the case with two buffers [25]. A graph-based heuristic is presented in [25] for the case of two buffers. The optimum solution for the case of two buffers is shown to be NP-hard when it is reduced the problem to that of finding a Hamiltonian path problem. Also, a heuristic method is presented by transforming the Hamiltonian-path problem to a Euler path problem.

We propose an off-line clustering approach based on min-cut graph partitioning of bi-partite page connectivity graph(PCG) for the join-index. The idea is to find clusters of pages in PCG in the hope of minimizing redundant I/O as shown in Figure 3. If the node clusters are edge disjoint as in Figure 3(A), i.e. there are no edge between node-clusters, this method will minimize redundant I/O assuming that each node-cluster can fit main memory. SGP is likely to be a stronger clustering methods than on-line heuristics proposed in literature since it uses a global partitioning algorithm based on min-cut graph-partitioning. Our experiments show this trend.

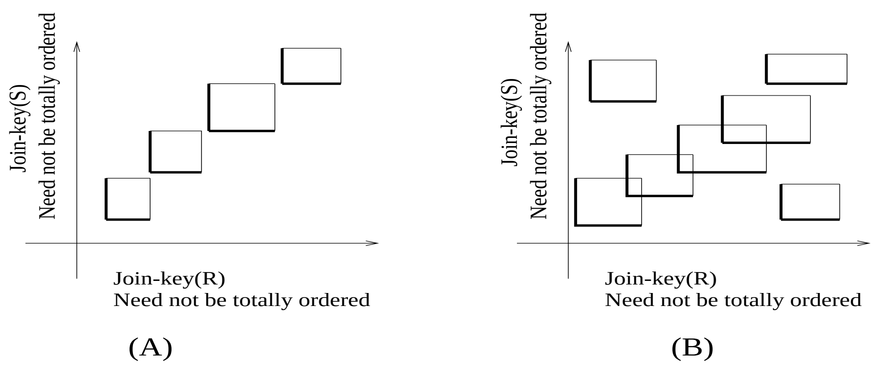


Figure 3: Example of Key-distribution for Join Keys

Our Contributions

In this paper we propose a suite of spatial clustering methods for join processing using the join-index. These methods: AGP, SGP with refinements, are all based on the min-cut graph partitioning techniques. We propose these methods as a new heuristic for solving the OPAS-FB problem in sequential systems. We also derive upper bounds on the number of page accesses needed to compute the spatial-join. We show that the length of the page-access sequence is bounded by the sum of the sizes of the base relations and the size of the cut-set of the page-connectivity graph. Since min-cut graph-partitioning aims to minimize the size of the cut-set, the proposed heuristic is a direct method. We performed our experiments on the Sequoia 2000[13] data set, a popular benchmark data set for spatial databases. Our experiments reveal that in situations of small buffer and high join-selectivity the AGP, which is exclusively base on hypergraph partitioning, often outperforms all other methods. For small buffer size and low join-selectivity, the SGP too outperforms the known competitors.

We also provide three refinements to SGP. First, we experiment with using the hypergraph partitioning algorithm, instead of the simple graph partitioning algorithm to partition the page connectivity graph. Second, we reduce the redundant I/O by properly processing the cut edges between partitions. Finally, we characterize an optimal sequence for loading partitions into the buffer.

3 Proposed Approach 1: Asymmetric Spatial Clustering

3.1 Basic Idea Behind AGP

Sorting-based heuristic ensures that no redundant accesses on the primary relation, but it may incur redundant accesses to the second relation, particularly when the join-key is not totally ordered, e.g. in spatial databases. In such domains, the notion of sorting can be generalized to spatial clustering. AGP clusters page of one table R based on their interaction with pages of S table. Redundant I/O of a page p of S is reduced if many of pages of R with edge connecting to p can be in memory when p is brought to memory.

3.2 Proposed Spatial Clustering Method

Spatial clustering of the tuples in a join index can be viewed as grouping of edges and nodes of corresponding page connectivity graph(PCG). In this section, we focus on asymmetric methods and postpone discussion of symmetric methods to Section 5. The goal of asymmetric clustering methods is to cluster 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 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 PCG. Partitioning of nodes in this hypergraph will form group of pages of the first relation to be loaded together. Goal of minimizing cut hyperedges during partitioning is to reduce the number of page of the second relation that needs to come to memory multiple times.

Consider the example spatial-join problem depicted in Figure 4(a) with two point data-sets, (a,b,c,d) and (A,B,C,D). Assume blocking factor of 1 to simplify the example. The PCG of the join-index for $Distance(i, j) < \frac{L}{\sqrt{2}}$ is shown in Figure 4(c) using the overlay and distance buffer information. The nodes of hypergraph shown in Figure 4(d) consist of 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 computing join using it will have no redundant I/O if 3 buffers are available to hold pages of two relations. In contrast, the partition ((a,b),(c,d)) cuts all four hyperedges and computing join will yield four redundant I/Os if only 3 buffers are available to hold pages of two relations.

We formally describe AGP now via following pseudo-code.

AGP 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,* $|HE_r| = |V_s|$ */

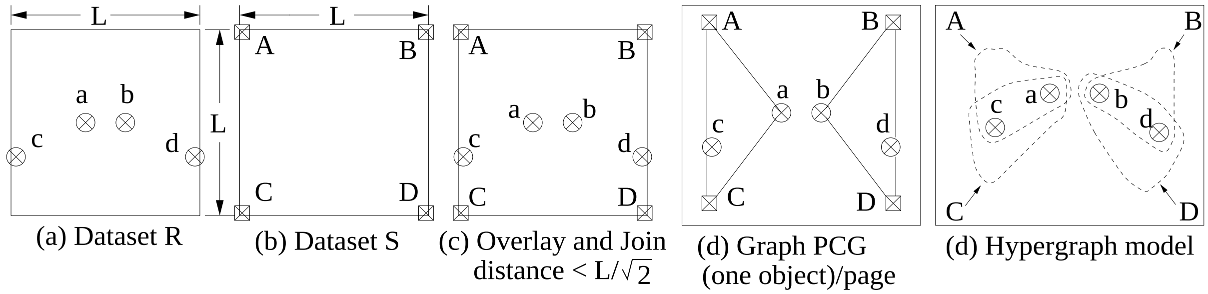


Figure 4: Construction of a one-side hyper graph from the data set

```
/* 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) /* Minimize the number of hyperedge-cut set */
i=0;
while ((P_{i_r}=SelectUnprossedPartition(PSet_r))!=NULL) /* Select the un-processed partition */
{
    AddPageSequence(S, P_{i_r});/* Add all the pages in P_{i_r} into the loading sequence */
    P_{i_s} = Sort-Eliminate-Dup(G, P_{i_r});
    /* Sort and eliminate the duplicated pages in V_s of G which connect to P_{i_r} */
    AddPageSequence(S, P_{i_s});/* Add all the pages in P_{i_s} into the loading sequence */
    P_{i_r}.flag = "processed"; /* Mark this partition as "processed" */
    i++;
}
```

The procedure DeriveHypergraph(G) works as follows. Nodes of the first relation R form the nodes of the hypergraph. For each node v of the second relation, it builds a hyperedge to encompass a set of nodes on the first relation(R) connected to v in G. We partition this hyper-graph using the min-cut hyper-graph partitioning algorithm, hMetis [17, 18], which is a multi-level hypergraph partitioning algorithm that has been shown to produce high quality bi-sections on a wide range of problems arising 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, we load 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 AGP 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 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 join, where $f(v)$ denotes number of partitions of V_r that have an edge to node v in V_s.*

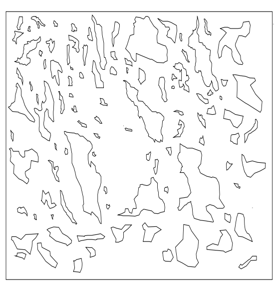
Proof: Each node v in V_s is connected to $f(v)$ partitions of V_r. Therefore, each node v in V_s has to be loaded $f(v)$ times into the buffer to do the join. Total number of redundant I/O is $\sum_{v \in V_s}(f(v) - 1)$. Total I/O cost $= |V_r| + |V_s| + \sum_{v \in V_s}(f(v) - 1) = |V_r| + |V_s| +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

We note that min-cut hypergraph partitioning algorithm, e.g. hMetis, minimizes the number of hyperedge connecting nodes across clusters. It does not distinguish between a hyperedge spanning four clusters or two clusters. While AGP outperforms sorting based heuristic already, the performance of AGP will improve when better algorithm for hypergraph partitioning are available which minimizes total number of cuts on cut-hyperedges. We plan to explore this in future work.

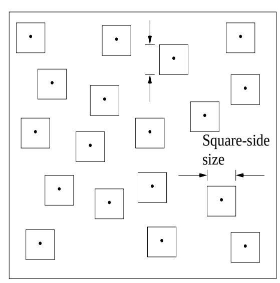
4 Comparing of Asymmetric Approaches: Sorting and AGP

4.1 Experiment Design

We now compare the performance of Sorting heuristic and AGP. For the evaluation we use a join index derived from spatial data derived from the Sequoia 2000 [32] dataset. We selected two data sets as our base relations: the *Points*, containing 62,584 California place names with their associated locations(Longitude and Latitude), extracted from the US Geological Survey's Geographic Names Information System(GNIS); and the *Polygons* with 4388 records, representing the Cropland and Pasture landuse in California. Throughout Section 4 and 8, the *Point* and *Polygon* relations will be referenced as R and S, respectively. The point are buffered to be a rectangle for adjusting the edge ratio [6]. 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. We plot a small but representative portion of the data set as in Figure 5. The join of these two relations with the "overlap" predicate produces a join index.



(a) Polygons for Landuse



(b) Buffer of GNIS Points

Figure 5: Example of the Sequoia 2000 data set

The variable parameters are the Buffer size, Page size and Edge ratio. The metric of evaluation is the number of page accesses required by each algorithm to implement the join.

Figure 6 summarizes the experiment steps. Derived data-sets consist of squares with different side-lengths, e.g. 3km, 6km, 8km, 18km. Joining these square data-sets with polygon yields join-index which are converted to equivalent page connectivity graphs. These page connectivity graphs are input to "Page Access Sequence Generator", which simulate behavior of sort based and AGP algorithm for given buffer size. The page access sequence and total page I/O are tracked for each combination of join algorithm, page size, buffer size, and edge ratio to derive the experiment results.

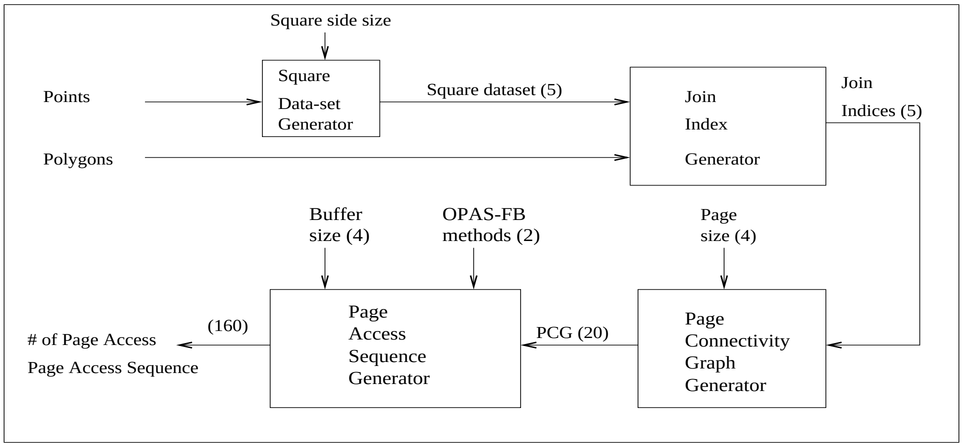


Figure 6: Experimental setup and design.

4.2 Experiment results

Figure 7 shows the comparison between the AGP and Sorting heuristic. The AGP method is uniformly better than the Sorting heuristic. Figure 7(a) shows the impact of the page size, varying from 2 kbytes to 8 kbytes, the difference between AGP and Sorting decreases as the page size increases. The reason is that when the page size increases, the number of pages decreases, and clustering efficiency improves for all methods, reducing the gap between performance.

Figure 7(b) shows the effect of buffer size (as a fraction of the size of the smaller relation) on the the I/O performance of AGP and sorting based method. The performance gap between the two methods goes up and down. As long as the buffer is smaller than the smaller of the two relations involved in join, both AGP and sort-based approach uses most of the buffers to load pages of only one relation. This leads to poor buffer utilization for both. The difference in performance comes from the difference in clustering ability.

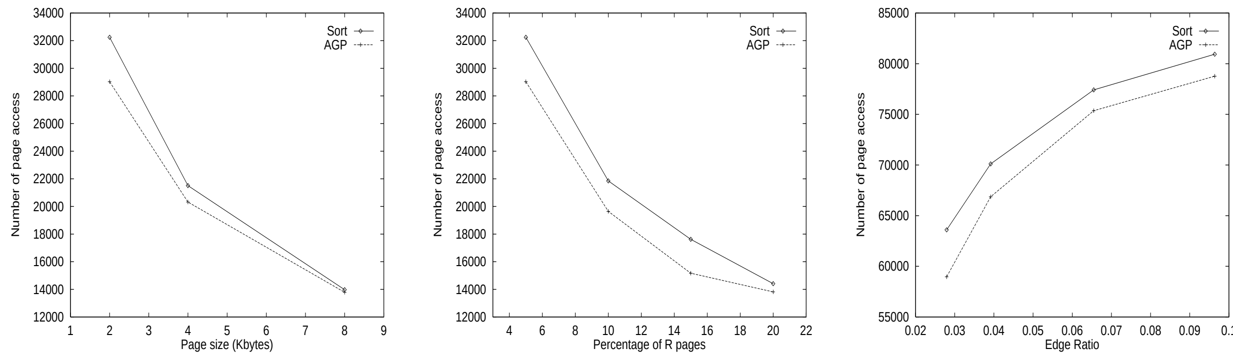
Figure 7(c) shows the effect of edge ratio. AGP outperforms Sorting-based approach uniformly in our experiments. The gap between the performance of two methods does not show any trend.

Sorting heuristic can be considered to be a special case of spatial clustering. In addition, preprocessing step of Sorting is cheaper than spatial clustering, thus there is a trade off between join performance and pre-processing cost. The proposed AGP is useful when there are few updates and preprocessing can be done once, and multiple join requests follow.

5 Proposed Approach 2: Symmetric Spatial Clustering

5.1 Motivation

While AGP is an improvement over Sorting based method, it has a few drawbacks. Its buffer utilization can be poor particularly for relatively large buffer size since it gives almost the entire buffer space to one relation. Secondly, the choice of the favored relation is not trivial in many situations. We illustrate

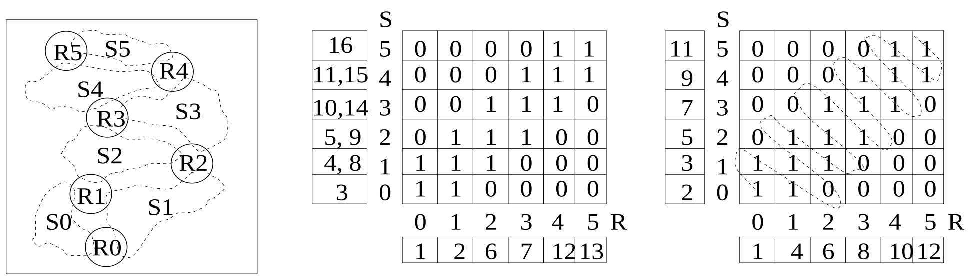


(a) Page size: 2K - 8K, Buffer size: 5% of the R relation, Edge ratio: 0.00669

(b) Buffer size: 5% - 20% of the R relation, Page size: 2K, Edge ratio: 0.00669

(c) Edge ratio: 0.02794 - 0.09638, Buffer size: 5% of R relation, Page size: 2K

Figure 7: Effect of Page size, Buffer size, Edge ratio on AGP and Sorting heuristic



(a) Spatial Join (b)Sorting/AGP (c)Symmetric method

Figure 8: Comparison of symmetric and asymmetric methods

these with the help of a spatial join problem shown in Figure 8. Figure 8(a) shows a polygon set with 6 polygons R0..R5 and a point data set with 6 points. The adjacency matrix M_{PCG} representation of join-index is shown in Figure 8(b) along with the page access sequence for sorting based algorithm with three memory buffer. Sorting requires 15 I/O including 3 redundant I/Os on S1, S2, and S3. Figure 8(c) shows a different page access sequence exploiting the symmetry of this problem. The symmetric method alternates between pages of the two relations to compute join with 12 I/O (i.e. no redundant I/O). This property can be generalized to other B-diagonal adjacency matrix where $\{M_{PCG}[i,j]=1\} \Rightarrow \{|i-j| \leq \lfloor B/2 \rfloor\}$. B is the number of buffers available for pages of R and S. Symmetric method can process B-diagonal adjacency matrix with no redundant I/O given B buffers for R and S. symmetric methods, e.g. FP [9],OM [27],Chan [6], can address these deficiencies. However, most symmetric methods proposed in literature are incremental, considering local information in PCG. We now propose a symmetric spatial clustering method, SGP, which exploit global information across entire PCG.

5.2 Framework

Symmetric spatial clustering approach to minimize redundant I/O can be described in terms of the following problem statement:

Given: A page connectivity graph in adjacency matrix M_{PCG} form and buffer size B to hold pages of R and S.

Find: A permutation of rows and a permutation of columns of M_{PCG}.

Objective:

$$\textbf{Minimize} \sum_{V_i \in (vertx\ cover\ of\ outside\ B\ diagonal\ edges)} \left\lceil \frac{number\ of\ outside\ B - diagonal\ edges\ incident\ on\ V_i}{B} \right\rceil \quad (1)$$

Where outside B-diagonal edge $(M[i,j]) = 1$ iff $|i-j| > \lfloor \frac{B}{2} \rfloor$

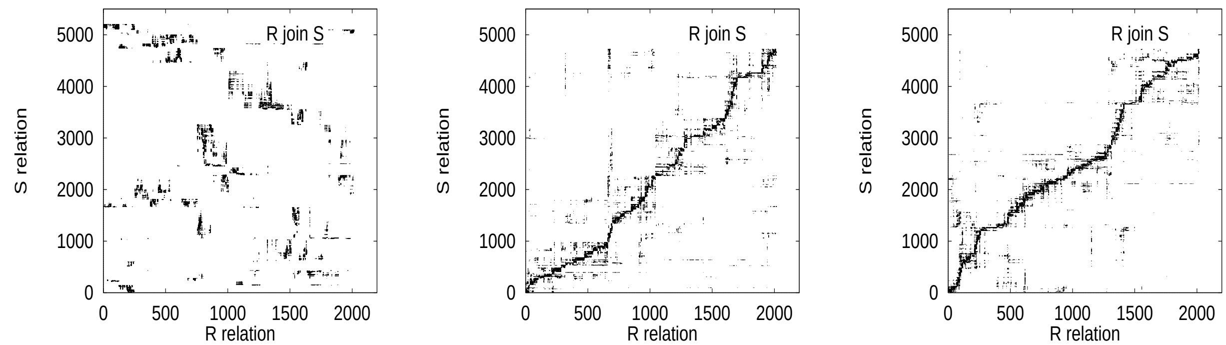
Constraint: Memory buffer $\leq B$

The redundant I/Os in this approach are due to the edge(i.e. non-zero matrix elements) outside B diagonal of a spatially clustered adjacency-matrix representation of join-index. These outside B-diagonal edge are grouped via a vertex cover to determine the set of page requiring redundant I/O. A minimum vertex cover of outside B-diagonal edge determines the redundant I/O if degree of these pages are smaller than the number of buffers available to cache page of R and S. Otherwise, some of these pages lead to redundant I/O as captured by the objective function.

This problem formalization provides a conceptual framework around B-diagonalization and vertex cover of off B-diagonal edges. This problem is computationally difficult due to its reliance on NP-hard sub-problem(e.g. minimal vertex cover). This is not a surprise in view of NP-hardness of OPAS-FB problem. Clearly heuristic solutions are needed to control computational overhead. One may devise a direct heuristic for this problem, e.g. as a search problem in the space of permutations of rows and column of M_{PCG}. This will require careful engineering to ensure scalability since PCGs can be large. We choose to take advantage of a well-engineered heuristic family, Metis [19], for a related problem, namely min-cut graph partitioning. Metis uses a hierarchical approach using graph coarsening to scale up. It can partition sparse graphs with millions of nodes within minutes providing fairly competent solutions and thus is being used for clustering problem in databases [31], data mining, etc.

We augment Metis by other steps to capture other important properties of our problem formalization as follows. A B-diagonal form is created via proper ordering of the partitions of PCG derived from Metis. The ordering tries to bring as many cut edges inside B-diagonal as possible, as described in 5.6. A vertex cover heuristic is used to group off B-diagonal edges into a small set of nodes, for reducing redundant I/O. These nodes are scheduled with specific partitions to determined the page access sequence as described in 5.5.

We use Figure 9 as an exmple to show the steps for deriving B-diagonal using graph partition technique. Figure 9(a) is the original PCG relation, where R and S are two relations to be joined, and each point in the graph denotes a edge connection between these two relations. We use Metis [19] to partition this PCG, each partition has size $(B-1)$, where B is the number of buffer available. We show the result after partition in Figure 9(b), the R and S relations are relabeled from the first partition to the last partition. Finally, we re-order these partitions to bring as many points inside B-diagonal, as shown in Figure 9(c). In Figure 9(b), there are 28 percent of points outside B-diagonal, after partition reordering, we reduce these points to be 22 percent of the total points.



(a) The original PCG connection

(b) After graph partition

(c) After graph partition and reordering

Figure 9: Using graph partition to derive the B-diagonal

5.3 Basic Idea: Simple SGP

The SGP method clusters the nodes of the Page-Connectivity Graph(PCG) via *min-cut* graph partitioning software Metis. The adjective "symmetric" refers to the fact that page clusters include pages from both tables R and S with no preference to either table. The min-cut partition algorithm partitions the nodes of the PCG into disjoint subsets, while minimizing the number of edges whose nodes are incident in two different partitions. Since only the nodes that are incident on the edges belonging to the edge-cut set can contribute to redundant I/O, minimizing the size of the edge-cut set provides a tight bound on the number of redundant I/O. We can formalize the properties of the SGP method as follows:

Symmetric Min-cut Graph Partitioning of the Page Connectivity Graph (SGP)

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, $p = \lceil |V|/(B - 1) \rceil$.

We now describe an algorithm, Simple SGP, for determining a page-access sequence, given a partition of the page-connectivity graph. The pseudo-code is shown in the following SGP algorithm. In this algorithm, each partition is loaded into memory to process all the joins completely within that partition. Whenever a partition has an associated cut edge, one node of the edge is already in memory. Then we only need to bring the other node that corresponds to the cut edge into memory. Due to the construction of the partitions (the number of pages in each partition is less than the number of buffers in memory), there is one buffer space available for bringing in one page to process a cut-edge. The cut edges associated with the partition are processed one at a time by using the empty buffer space to store a page needed to process the selected cut-edge.

For example, Figure 10 is an example of a partition of the PCG graph shown in Figure 2. Assuming that the buffer size is 3, each partition V_1, V_2, and V_3 can contain up to two nodes of the PCG. The edge-cut set consists of the three edges $\{e_2, e_3, e_4\}$. The Simple SGP algorithm on this partition proceeds as follows: We randomly load a partition into the buffer, say V_1, and first perform the join internal to the partition, $\{a, A\}$. We then follow the edge-cut e_2, load page B in the spare buffer and materialize the join $\{a, B\}$. We then follow the edge-cut e_3, swap page B for C in the buffer and materialize the join $\{a, C\}$. Finally, all joins emanating from V_1 are materialized by loading page b at the end of edge-cut e_4. We then move on to the next randomly selected remaining partition until all are exhausted.

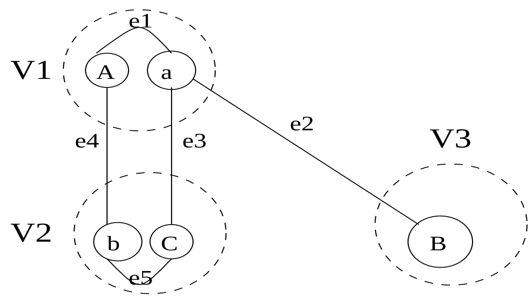


Figure 10: A min-cut partition of the graph

Lemma 2 *Given a partition $\{V_1, V_2, \ldots, V_p\}$ of the page-connectivity graph $PCG = (V, E)$, there is a page-access sequence of length at most $K \leq |E_C| + |V|$ to process the join, assuming $|V_i|$ is less than the number of buffers. If $E_C = 0$, the algorithm Simple SGP is guaranteed to result in an optimal page-access sequence.(Note that $|V| = |V_r| + |V_s|$ where V_r and V_s are the nodes of relation R and S begin joined.)*

Proof: The Simple SGP algorithm results in at most $|V| + |E_C|$ page accesses. Since each cut-edge can result in at most one extra page access, the total number of page accesses is bound by the total number of pages in all of the partitions and the number of cut edges. Therefore, there exists a page-access sequence of size at most K, where $K \leq |V| + |E_C|$. ■

Simple SGP Algorithm

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

assert($B \geq 2$); /* *Number of buffer* */

P_Set = Metis-Partition $(G, B-1)$ /* *Minimize the number of edge-cut set* */

i=1;

while ((P_i=SelectUnprocessedPartition(P_Set))!=NULL)

/* *Select the un-processed partition* */

{

AddPageSequence(S, P_i);/* *Add all the nodes in P_i into the loading sequence* */

```
        CP_Set = FindConnectPartition(P_i)); /* Find all the partitions which connect with P_i */
        for(j = 1; j ≤ |CP_Set|; j++)
        {
            CP_j=CP_Set[j]; /* Get the j-th connected partition */
            if( CP_j.flag != "processed")/* This partition has not been processed */
            {
                Nodes=GetIncidentNodes(P_i,CP_j);/* Find all the nodes in CP_j which connect to P_i */
                AddPageSequence(S, Nodes); /* Add all these nodes into the loading sequence*/
            }
        }
        P_i.flag = "processed"; /* Mark this partition as "processed" */
        i++;
    }
```

5.4 Refinement 1 for Simple SGP: Reducing Cut Edges

Instead of using a simple graph for determining PCG-node partition, we build a hyper-graph for the PCG and partition this hyper graph.

We use the page-connectivity graph model of the join index to construct a hyper-graph from the join index. The hyper-graph is derived from the original bipartite graph. To construct a pure hyperedge graph, for each node in the bipartite graph, we build a hyperedge to encompass this node and its connecting nodes in the other relation. To construct a hyper-edge graph, in addition to these pure hyperedges, we add hyperedges for each edge in the original bipartite graph. Figure 11 illustrates this construction with the help of an example.

A *min-cut* node partition [15, 21] of a hyper graph G partitions the nodes in V_1 and V_2 into disjoint subsets while minimizing the number of hyper edges whose incident nodes are in different partitions. The h-cut-set of a min-cut partition is the set of hyper-edges whose incident nodes are in two different partitions. Fast and efficient heuristic algorithms for this problem have become available in recent years. In our experiment, we use hMETIS [17], which is a set of programs for partitioning hypergraphs.

Lemma 3 *The savings in redundant I/O for utilizing the hyper-graph partitioning algorithm instead of simple graph partitioning is:*

$$Savings = \max(E_c - E_h, 0).$$

Proof: The hyper-graph partitioning algorithm results in a different edge-cut set, E_h for the PCG. The potential savings is $E_c - E_h$ for the simple SGP algorithm. ■

5.5 Refinement 2: Vertex Cover Strategy, Given loading sequence

The goal of this refinement is to reduce the redundant I/O by properly processing the cut edges between partitions. The redundant I/O caused by the cut edges can be saved by using three ideas. **First**, knowing the next partition P_j to be loaded while processing a partition P_i can eliminate redundant I/O for edges

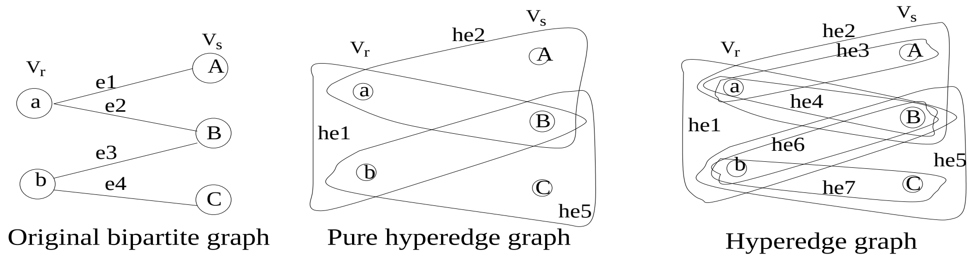


Figure 11: Construction of a hyper graph from the bipartite graph

between P_i and P_j. For example, in figure 12(a), if we load the partitions in this order $\{P1, P2, P3\}$, we can always find a processed node to be replaced by a new node in the next partition, thus eliminate the redundant I/O for processing cut-edge between consecutive partitions. While transferring from partition P_i to P_j, we delete the node in P_i which is either not incident on an edge-cut to P_j, or whose join across the edge-cut to the P_j has been materialized, and add the node in P_j which has the highest connectivity with the nodes in P_i.

Lemma 4 *Given a partition of the page-connectivity graph $PCG = (V, E)$, and the loading sequence of these partition is $\{V_1, V_2, \ldots, V_i, V_j, \ldots, V_p\}$. If the adjacent matrix of any two consecutive partitions V_i, V_j can be permutated in the triangular form, the redundant I/O between the two partitions can be saved.*

Proof: All edges within the main B-diagonals can be processed without redundant I/O.
Corollary of Lemma Number of redundant I/O $\leq$ Number of cut-edge outside B-diagonal.

Second, edges incident on a common page can be processed together. In figure 12(b), there are four edge cut between $P1$ and $P2$. However, there are only two nodes in each partition involved in these edge cut. The redundant I/O is bounded by the distinct nodes involved in the edge-cut set.

Third, the cut-edges between partitions P_i and P_k are cheaper to process with partition P_i in memory, if the number of distinct pages of P_i incident on the cut-edge is more than those of P_k. For example, in figure 12(c), it will be better if we process these edge-cut when $P2$ is in memory, with only one redundant I/O. However, if we process these edge-cute when $P1$ is in memory, the redundant I/O cost will be four.

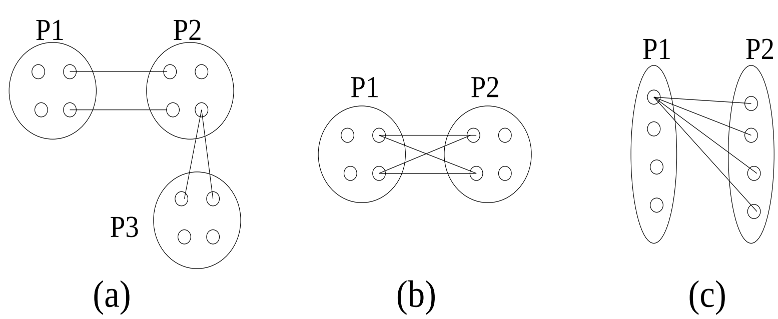


Figure 12: Examples of refinement 3

5.6 Refinement 3: Choose Proper Loading Sequence

If some pages of the current partition in memory are connected to pages in the next partition to be loaded, then these pages need not be fetched again, as they are already in memory. Hence the order in which different partitions are loaded into memory influences the number of page fetches in a page-access sequence. In practice, a sequence of partitions which maximizes the number of connected pages between consecutive partitions is desired because it reduces the length of the page access sequence. For this, the following Longest Path heuristic can be used. Construct a complete graph G_p(the Weighted Partition Graph(WPG)) with one node for each of the partitions. The weight on each edge between the nodes is the **minimum** of the number of distinct pages, connected between the partitions, that correspond to the nodes. For example, Figure 13 is the WPG derived from Figure 10. The weight on the edge connecting two nodes is equivalent to the number of pages that need not be fetched from disk when one node(partition) is in memory and the other one is next on the loading schedule. Thus a partition schedule corresponding to the Longest Path on the WPG leads to a minimum number of page accesses to load the partitions that correspond to all the nodes of the WPG G_p.

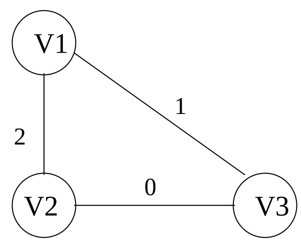


Figure 13: The Weighted Partition Graph(WPG). $\{V_2, V_1, V_3\}$ is an example of the Longest Path.

Besides, for cut edges between non-adjacent partitions in the sequence, the redundant I/O is not bound by the number of cut edges. Rather, we choose the smaller distinct pages incident on the cut edges between the pairs of partitions. This means we do the actual join by loading each page in the smaller distinct pages of one partition to the one buffer space. We formalize the above observation in the following lemma.

Lemma 5 *Given a weighted partition graph $WPG = (G_p, E_p)$ derived from the partition $\{V_1, V_2, \ldots, V_p\}$ of the page-connectivity graph $PCG = (V, E)$ and a Longest Path on the WPG, there is a page-access sequence of length $K \leq |V| + |TW| - |LP|$, where $|TW|$ is the total weight of the edges of WPG, and $|LP|$ is the weight of the Longest Path.*

Proof: $|TW|$ is the actual redundant I/O. As long as we can find a node of the current partition which is either not incident on an edge-cut to the next scheduled partition, or whose join across the edge-cut to the next scheduled partition in the buffer has been materialized, then we can replace this node with a node from the next partition on the loading schedule to the buffer. This provides us with a cumulative savings of $|LP|$, which can be subtracted from the redundant I/O. ■

The Longest Circuit problem is known to be NP-Complete [11]. We describe two trivial heuristics in Appendix A whose experimental evaluation *vis-a-vis* each other will be discussed in Section 6. There are other, more sophisticated heuristics available, and those may improve the performance of our methods. We plan to explore those in our future work.

5.7 Final Algorithm

The final algorithm, SGP, incorporates all three refinements. The algorithm first partitions the PCG using hypergraph partitioning and simple graph partitioning. It chooses the one that results in a smaller edge-cut set. Then, it orders the resulting partitions using the longest circuit heuristic. Finally, it loads the partitions as dictated by the longest circuit. After loading each partition to the buffer, first, it processes all the join within this partition, then, it processes the joins caused by the cut edges with the connected partitions. For each partition in the connected partition set, it compares the number of distinct pages incident on the cut edges between these two partitions, and chooses the cheaper one to process. When transferring from the one partition P_i to the next scheduled partition P_{i+1}, it orders the loading sequence of nodes using the following strategies: (a)Add the node within P_{i+1} which has the highest connectivity with P_i. (b)Replace the node within P_i which is either not incident on an edge-cut to P_{i+1}, or whose join across the edge-cut to P_{i+1} has been materialized. The pseudo-code and detailed descriptions for SGP algorithm are shown in Appendix B.

6 Experimental Evaluation of Refinement

6.1 Experimental Design

We now evaluate the performance of refinement strategies 1, 2 and 3 and compared it with Simple SGP.

The fixed parameter for this experimental evaluation is the graph partitioning algorithm Metis [19] and hypergraph partitioning algorithm hMetis [17]. The variable parameters are the Buffer size, Page size and Edge Ratio. The metric of evaluation is the number of page accesses required by each algorithm to implement the join. Figure 14 shows the various process steps of the experiment design.

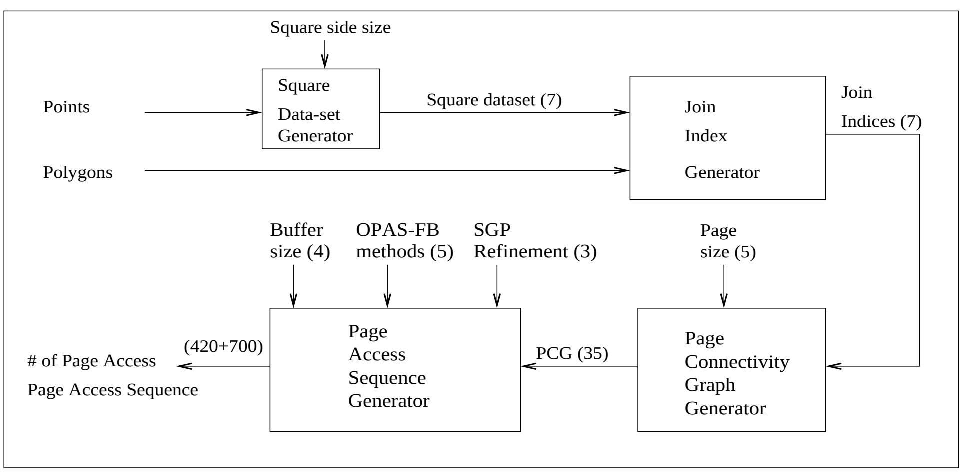


Figure 14: Experimental setup and design.

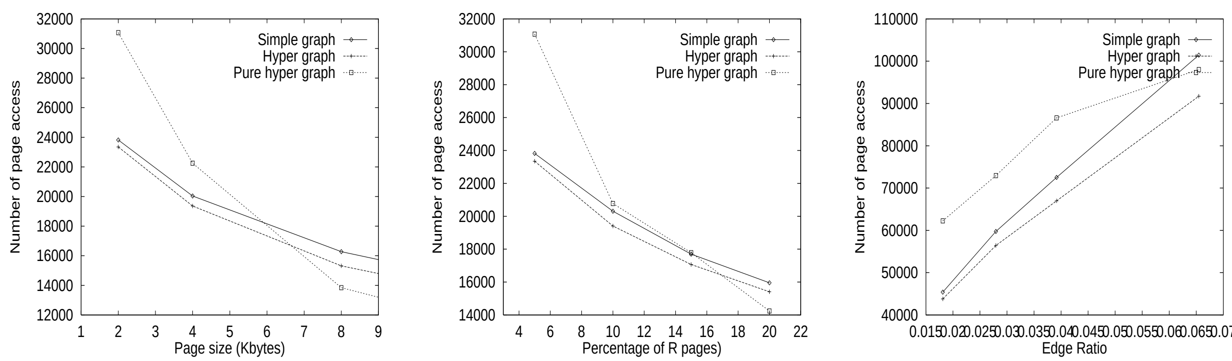
6.2 Experiment Results on Effect of Refinements on SGP

6.2.1 Effect of Refinement 1: Does graph model matter?

We test the relative improvement of using different hyper-graph structure partitions(hMetis) [17], compared with using the simple graph partition(Metis) [19]. All of the comparisons are done by using the Simple SGP algorithm. For different experiments, we vary the buffer size, page size and square side size.

Effect of page size

With square-side size at 3000 meter, e.g. edge ratio at 0.00669, and buffer size fixed at five percent of the number of pages of the R relation we vary the page size from 2k to 16k. The Hyper graph partition results in fewer page accesses compared to the simple graph partitioning. The Pure hypergraph has the lowest number of page accesses when the page size is greater than 8k. (Figure 15(a))



(a) Page size: 2K - 16K, Buffer size: 5% of the R relation), Edge ratio: 0.00669

(b) Buffer size: 5% - 20% of the R relation, Page size: 2K, Edge ratio: 0.00669

(c) Edge ratio: 0.01812 - 0.06548, Buffer size: 5% of R relation, Page size: 2K

Figure 15: Effect of page size, buffer size, and edge ratio with Refinement 1 for Simple SGP

Effect of buffer size

With edge ratio at 0.00669, and page size at 2k bytes, we increase the buffer size from 5 to 20 percent of the number of pages of the R relation. The Hyper graph partition results in better performance, compared to the the simple graph partition. The Pure hypergraph partition has the worst performance when the buffer size is low. However, when the buffer size is larger than a threshold(15 percent), the Pure hypergraph has the best performance, as shown in Figure 15(b).

Change the edge ratio

We fix the buffer size to 5 percent of the number of pages of the R relation, and page size at 2k bytes, then we vary the edge ratio by extending the size of the square side in the point dataset. The Hyper graph partition performs better than the Pure hypergraph and the simple graph as shown in Figure 15(c).

When the edge ratio is larger than a threshold (0.06), The Pure-hyper graph outperforms the simple graph.

6.2.2 Effect of Refinement 2 and 3 on SGP: Does cut-set processing strategy matter? Does loading Sequence matter?

In this experiment, we fix the edge ratio and page size, vary the buffer size and use different linearalization algorithms . The result of algorithms is shown in Fig 16. As we can see from Fig 16, refinement 2, the three ideas for proper cut set processing between partitions, does improves the performance under different buffer sizes. The one-way greedy heuristic and the sorting heuristic in refinement 3 generate nearly the same result, and all perform better than the simple SGP algorithm. The reason is that the simple SGP randomly linearalizes the loading sequence of the partitions without doing any optimization.

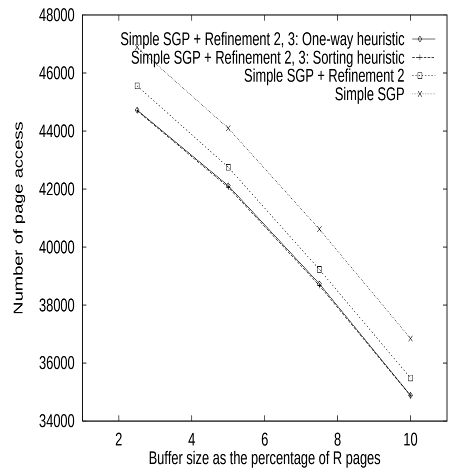


Figure 16: Page access number using different linearalization algorithm for Simple SGP

7 Comparative Evaluation of SGP, AGP and Competitors

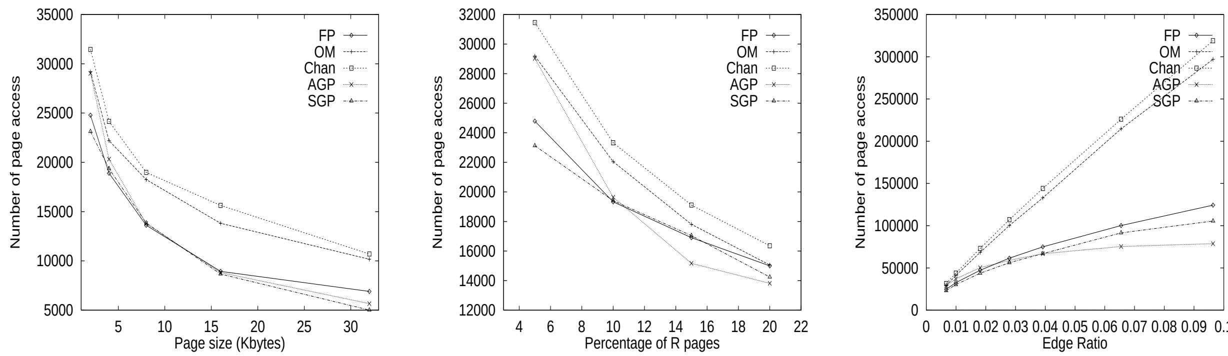
7.1 Experimental Design

The experimental setup is shown in Figure 14. The candidates for the OPAS-FB method are FP, OM, Chan, Sort, SGP, and AGP algorithm. There are some basic constraints on the experiment which are worth mentioning at the outset. These constraints are due to the properties of the data set at hand. If the relationship type between the two relations is $1:1$ or $1:N$, then the sorting and AGP algorithm leads to the optimal page access sequence. Even in the situation where the relationship type is $M:N$, the sorting and AGP algorithm can give a good performance, provided that sorting on one relation can lead to a good clustering of pages in the other relation. We used a subset of the Sequoia data set that consisted of two relations: Point and Polygon. We converted the Point dataset into an axis-parallel rectangular dataset. The orientation of each rectangle was chosen at random. We used the **intersection** binary relationship as the spatial join predicate. Converting the point data into a rectangular data set transformed the $N:1$ point-polygon relationship into an $M:N$ rectangle-polygon relationship.

7.2 Experiment results

7.2.1 Page size

Page size affects the clustering of the base relations and also the degree of the nodes in the PCGs. We study the effect of page size on the performance of the OPAS-FB methods. We fixed the buffer space at five percent of the number of pages of relation R, and edge ratio at 0.00669. We varied the page size from 2k to 32k. Figure 17(a) shows the result of this experiment. The SGP method outperforms the other methods for most values of page size. It comes second to FF for a few page size.



(a) Page size: 2K - 32K, Buffer size: 5% of the R relation), Edge ratio: 0.00669

(b) Buffer size: 5% - 20% of the R relation, Page size: 2K, Edge ratio: 0.00669

(c) Edge ratio: 0.00669 - 0.09638, Buffer size: 5% of R relation, Page size: 2K

Figure 17: Effect of Page Size, Buffer Size, and Edge Ratio for different OPAS-FB heuristics

7.2.2 Buffer size

In this experiment, we fixed the edge ratio at 0.00669, and the page size at 2k. We varied the number of buffers as a percentage of the number of pages of relation R. The percentage is changed from 5 to 20. Fig 17(b) shows the number of page accesses recorded by each of the six methods. A large buffer size improves the performance of each method, while the AGP heuristic overall has the best performance, even though SGP and OM do well at some places.

7.2.3 Edge Ratio

In this experiment, we buffer size set at 5 percent and page size set at 2K. We changed the edge ratio by increasing/decreasing the size of the square side of relation R. The result of the experiment shows in Figure 17(c). The SGP method results in a lower number of page accesses than the other methods for lower edge ratio, and AGP does best for higher values of edge ratio.

The example for the effect of Edge Ratio

We use an example to illustrate the effect of the characteristic of the PCG for suitability of asymmetric and symmetric methods for join processing given a join index. The buffer size is fixed at five for this exam-

R									
7	0	0	0	0	0	0	0	1	
6	0	0	0	0	0	0	1	0	
5	0	0	0	0	0	1	0	0	
4	0	0	0	0	1	0	0	0	
3	0	0	0	1	0	0	0	0	
2	0	0	1	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	
0	1	0	0	0	0	0	0	0	
	0	1	2	3	4	5	6	7	S

(a)Thickness=1

R									
7	0	0	0	0	0	1	1	1	
6	0	0	0	0	1	1	1	1	
5	0	0	0	1	1	1	1	1	
4	0	0	1	1	1	1	1	0	
3	0	1	1	1	1	1	0	0	
2	1	1	1	1	1	0	0	0	
1	1	1	1	1	0	0	0	0	
0	1	1	1	0	0	0	0	0	
	0	1	2	3	4	5	6	7	S

(b)Thickness=5

R									
7	0	0	0	0	0	1	1	1	
6	0	0	0	0	1	1	1	1	
5	0	0	0	1	1	1	1	1	
4	0	0	1	1	1	1	1	1	
3	0	1	1	1	1	1	1	0	
2	1	1	1	1	1	1	0	0	
1	1	1	1	1	1	0	0	0	
0	1	1	1	1	0	0	0	0	
	0	1	2	3	4	5	6	7	S

(c)Thickness=6

R									
7	0	0	0	0	1	1	1	1	
6	0	0	0	1	1	1	1	1	
5	0	0	1	1	1	1	1	1	
4	0	1	1	1	1	1	1	1	
3	1	1	1	1	1	1	1	0	
2	1	1	1	1	1	1	0	0	
1	1	1	1	1	1	0	0	0	
0	1	1	1	1	0	0	0	0	
	0	1	2	3	4	5	6	7	S

(c)Thickness=7

R									
7	1	1	1	1	1	1	1	1	
6	1	1	1	1	1	1	1	1	
5	1	1	1	1	1	1	1	1	
4	1	1	1	1	1	1	1	1	
3	1	1	1	1	1	1	1	1	
2	1	1	1	1	1	1	1	1	
1	1	1	1	1	1	1	1	1	
0	1	1	1	1	1	1	1	1	
	0	1	2	3	4	5	6	7	S

(c)Thickness=15

Figure 18: Table for different edge ratio

Class	1:1	M:N	M:N	M:N	M:N
Edge Ratio	8/64	34/64	39/64	44/64	1
AGP/Sorting	16	20	21	22	24
OM	16	16	18	21	35
FP	16	16	23	26	32
Chan	16	16	18	23	35
SGP	16	16	21	26	34

Table 2: I/O count for different methods

ple. The thickness of the diagonal increases from one to fifteen, e.g. from sparse matrix to full connected graph, as shown in Figure 18. Table 2 shows the number of I/O needed by symmetric and asymmetric method to compute join. When the thickness of diagonal is small, e.g. 1, all methods have the same I/O count, and the sorting heuristic is the cheapest to find the page access sequence. As the diagonal-thickness increase, the performance of sorting deteriorate, and symmetric methods(OM,FP,Chan,SGP) outperform asymmetric heuristics, e.g. AGP, Sorting. Finally, as the graph gets closer to being fully connected, the thickness of the diagonal is greater than B, asymmetric methods outperform symmetric methods. We summarize our experience with various kinds of join problem toward choosing a method to compute join using join-index as in Table 3.

8 Conclusion and Future Work

In this paper, we introduce spatial clustering methods for minimizing redundant I/O, given a fixed buffer. We also propose three refinements which further improve the performance of the basic algorithm. The proposed AGP and SGP heuristic usually outperform the Sort-based heuristic and Graph-based heuristics. In the future, we would like to explore algorithms for multi-way join-indexes using the base data sets consisting of *point*, *line* and *polygon* data types. We also plan to do experiments with different spatial predicates like direction and distance. Finally, we would also like to investigate the relationship between a multi-way join and the hyper-graph partitioning model, and to establish upper bounds for the page-access schedule on the multi-way join.

The min-cut hypergraph partitioning package(hMetis) we use in our experiment minimizes the num-

<table>
<tr><td>Domain</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>One-dimension</td><td>No method has redundant Cheapest=Sorting</td><td rowspan="2"></td><td rowspan="2">No symmetric method has redundant I/O Cheapest=FP</td><td rowspan="2"></td><td rowspan="2">Small vertex cover for off B diagonal edges ⇒ SGP,

Some cases, OM,FP, or Chan may be slightly better.</td><td></td><td>Asymmetric methods have less redundant I/O. Cheapest = Sorting.</td></tr>
<tr><td>Multi-dimension</td><td>No method but sorting has redundant I/O Cheapest = AGP</td><td></td><td>AGP has least I/O</td></tr>
<tr><td>Relationship Cardinality</td><td>1:1
1:N</td><td colspan="6">M:N</td></tr>
<tr><td>Thickness of diagonal</td><td>1</td><td colspan="3">$\leq B$</td><td colspan="2">$> B$, but few entries outside B diagonal</td><td>Fully connected</td></tr>
</table>

Table 3: Summary

ber of hyperedge connecting nodes across clusters. However, it does not distinguish a hypergraph cutting by four clusters or two clusters. While our experiment shows that AGP outperforms Sorting based heuristic already, the performance of AGP will be improved when better algorithm for hypergraph partitioning are available which minimizes total number of cuts on cut-hyperedges.

We used two trivial heuristics to solve the Longest Circuit problem in Refinement 3 for SGP. There are other more sophisticated heuristic available [29], and they may improve the performance of our methods. We plan to test these heuristic.

Data Warehouses [3, 16] process large volumes of data obtained from operational and legacy systems. Data warehouses clean and transform the data so that changes can trends can be inferred from the data. The volume of data in these data warehouses is very large and the data is updated infrequently, due to the historical nature of the data. Data warehouses often use pre computation and materalization techniques like STARindex [3] to speedup online query processing. We would like to apply our graph partitioning approach as in Join Index to STARindex.

The data sets used in our experiments are *Point*(California place names) and *Polygon*(Cropland and Pasture landuse in CA) data, derived from Sequoia 2000 [32] benchmarks data sets. There are other data set in this benchmark that can be used in our experiments for performance analysis. For example, the *Streams* data contains 201,659 stream segments, extracted from the the US Geological Survey's Digital Line Graph hydrography data for California. For polygon data, there are Agricultural Land, Forest Land, Wetland, Rangeland, Barren Land, etc. The *Streams* layer data can do the map overlay operations(union, intersection, identity) [7, 10], with the *Polygon* layers, and the Join Index can be precomputed for processing later spatial join requests.

Acknowledgements

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. Thanks to Christiane McCarthy for helping to improve the readability of the paper.

References

[1] 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.

[2] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Trans. Knowledge and Data Eng.*, 1(2):196–214, June 1989.

[3] Red Brick. *White Papers, http://www.redbrick.com.*

[4] 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.

[5] 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 1990.

[6] Chee Yong Chan and Beng Chin Ooi. Efficient Scheduling of Page Access in Index-Based Join Processing. *IEEE Transactions on Knowledge and Data Engineering*, November/December 1997.

[7] Yue-Hong Chou. *Explorint Spatial Analysis in Geographic Information Systems.* Onword Press, 1996.

[8] B.C. Desai. Perfromance of a Composite Attribute and Join Index. *IEEE Trans. Software Eng.*, 15(2):142–152, February 1989.

[9] F. Fotouhi and S. Pramanik. Optimal Secondary Storage Access Sequence for Performing Relational Join. *IEEE Transactions on Knowledge and Data Engineering*, September 1989.

[10] A. Frank. Overlay processing in spatial information systems. *Computer-Assisted Cartography*, 1988.

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

[12] 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.

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

[14] O. Gunther, L. Becker, K. Hinrichs, and U. Finke. Efficient Computation of Spatial Joins. In *Proceedings of International Conference on Data Engineering*, 1993.

[15] 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.

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

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

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

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

[20] A. Kemper and G. Moerkotte. Access Support in Object Bases. In *Proc. ACM SIGMOD, Atlantic City, N.J.*, pages 364–374, 1990.

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

[22] 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.

[23] M. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *Proceedings of the Fourth International Symposium on Large Spatial Databases*, 1995.

[24] H. Lu, R. Luo, and B.C. Ooi. Spatial Query Processing by Approximations. *J. Australian Computer Science Comm.*, 17(2):132–142, 1995.

[25] 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.

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

[27] Edward R. Omiecinski. Heuristics for Join Processing Using Nonclustered Indexes. *IEEE Transactions on Software Engineering*, 15, January 1989.

[28] S. Pramanik and D. Ittner. Use of Graph Theoritic Models for Optimal Relational Database Accesses to Perform Join. *ACM Transactions on Database Systems*, March 1985.

[29] Gerhard Reinelt. *The Traveling Salesman: Computatinal Solutins for TSP Applications.* Springer-Verlag, 1994.

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

[31] 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.

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

[33] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, pages 218–246, December 1987.

A Longest Circuit heuristic

One-way greedy heuristic

1. Choose the node Vp_i connecting to the longest length edge as the start node.
2. For all of the nodes connected to Vp_i, find the longest length edge with corresponding node Vp_j; set Vp_i to Vp_j.
3. repeat step 2.

Sorting heuristic

1. Sort the edges in graph Gp in descending order
2. Scan these edges; construct a set of (n-1) edges with linear property.

In this method, we initially choose the edge with longest length and two nodes associated with it. Then, we do a greedy search on both sides and construct the linear order.

B SGP Algorithm

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

```
assert(B ≥ 2); /* Number of buffer greater than two */
HG = ConstructHyperGraph(G); /* Construct Hypergraph HG from G */
PSet_m = Metis-Partition (G, B − 1); /* Simple graph partition, using Metis */
PSet_h = hMetis-Partition (HG, B − 1);/* Hyper graph partition, using hMetis */
E_m = Edge-Cut-No(G, PSet_m); /* Number of edge cut using Metis algorithm */
E_h = Edge-Cut-No(G, PSet_h); /* Number of edge cut using hMetis algorithm */
if(E_m ≤ E_h){
/* Order the partitions using the one way greedy heuristic */
    P_order = One-Way-Longest-Circuit(PSet_m); }
else { P_order = One-Way-Longest-Circuit(PSet_h); }
/* Load the partition into buffer as determined by the longest circuit heuristic */
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} */
        /* 2. Replace the node within P_{i−1} which has finished its join with the nodes in P_i */
    }
    CP_Set = FindConnectPartition(P_i ,GetPartition(P_order, i + 1));
    /* Find all the partitions which have cut-edge set with P_i, except the next loading partition P_{i+1} */
    for(j = 1; j ≤ |CP_Set|; j + +){
        CP_j=CP_Set[j]; /* Get the j-th connected partition */
        Nodes[CP_j, i]=GetIncidentNodes(CP_j,i);
        /* Find all the distinct nodes in partition CP_j which connect to partition i */
        Nodes[i, CP_j]=GetIncidentNodes(i,CP_j);
        /* Find all the distinct nodes in partition i which connect to partition CP_j */
        if((|Nodes[CP_j, i]| ≤ |Nodes[i, CP_j]|)&&(Nodes[CP_j, i].flag! = "processed")){
        /* Refinement 2 */
            AddPageSequence(S, Nodes[CP_j, i]); /* Add these nodes into the loading sequence */
            Nodes[CP_j, i].flag = "processed";
            Nodes[i, CP_j].flag = "processed";
            /* Mark the flag of the nodes between two partitions as "processed" */
        }
    }
}
```