1-20-2006

An R*-Tree Based Semi-Dynamic Clustering Method for the Efficient Processing of Spatial Join in a Shared-Nothing Parallel Database System

Gayatri Ganpaa
University of New Orleans

Recommended Citation

Ganpaa, Gayatri, "An R*-Tree Based Semi-Dynamic Clustering Method for the Efficient Processing of Spatial Join in a Shared-Nothing Parallel Database System" (2006). *University of New Orleans Theses and Dissertations*. 298.
https://scholarworks.uno.edu/td/298

AN R*-TREE BASED SEMI-DYNAMIC CLUSTERING METHOD FOR THE EFFICIENT PROCESSING OF SPATIAL JOIN IN A SHARED-NOTHING PARALLEL DATABASE SYSTEM

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Masters of Science
in
The Department of Computer Science

by

Gayatri Ganpaa
B.Tech, Jawaharlal Nehru Technological University, India, 2003

December 2005

Acknowledgement

This thesis is the result of one and half years of work whereby I have been accompanied and assisted by many people. I have now the opportunity to express my gratitude to all of them. Instead of listing names after names, however, I will just say one big thanks to all the people who deserve to be thanked but not mentioned here by name.

I am highly indebted to my advisor, Professor Mahdi Abdelguerfi. He deserves my heartfelt appreciation for the warm and constant support he has given me to accomplish this thesis. In addition, I would like to thank Dr. Shengru Tu and Dr. Nauman Chaudhry for being on my thesis committee.

A special thanks goes to Mr. Venkata Mahadevan whose continuous support in getting this thesis prepared is invaluable. If it were not for his well-organized instruction, I would not imagine completing this successfully.

Most of all, I would like to thank my family and friends, for their unending support and help they provided to me. To them, this thesis is dedicated.

Table of Contents

List of Figures

List of Tables

Abstract

The growing importance of geospatial databases has made it essential to perform complex spatial queries efficiently. To achieve acceptable performance levels, database systems have been increasingly required to make use of *parallelism*. The *spatial join* is a computationally expensive operator. Efficient implementation of the join operator is, thus, desirable. The work presented in this document attempts to improve the performance of spatial join queries by *distributing* the data set across several nodes of a cluster and executing queries across these nodes in parallel.

This document discusses a new parallel algorithm that implements the spatial join in an efficient manner. This algorithm is compared to an existing parallel spatial-join algorithm, *the clone join*. Both algorithms have been implemented on a *Beowulf* cluster and compared using real datasets. An extensive experimental analysis reveals that the proposed algorithm exhibits superior performance both in declustering time as well as in the execution time of the join query.

Chapter 1 Introduction

The use of geospatial data arises in many applications including Cartography, Computer Aided Design (CAD), Computer Vision, and Robotics to name but a few. Geospatial data sets are often large and are being constantly gathered by numerous satellites and other data collection devices. In order for the data collected to be useful, it needs to be processed and analyzed. This data is typically stored in a spatial database to facilitate processing and analysis. However, due to the massive amount of data being stored, several problems can arise. The ability to store and query this enormous amount of data is critical but may lead to performance degradation. Therefore, faster data retrieval and computation mechanisms are now required.

Performance problems with large databases have been widely documented by researchers and several techniques [3, 4, 13, 15, 16, 17, 18, 20, 23, 25] have been devised to cope with this. Indeed, traditional database management architectures have difficulty meeting the I/O and compute performance levels needed to handle large volumes of geospatial data. To achieve acceptable performance levels, database systems have been increasingly required to make use of Parallelism [8]. One form of parallelism involves the use of compute clusters.

A cluster is simply a collection of compute nodes interconnected via some sort of network. Clusters can improve the performance of geospatial queries by exploiting parallelism. The most popular type of compute clusters in use today is based on the Beowulf paradigm. Beowulf compute clusters are shared nothing machines in which all of the compute nodes, which are called slave nodes, are isolated on a high-speed private network that is not directly visible to the outside world. A single computer connected to the outside world (called the master node) lets a user login to the cluster and submit jobs for processing i.e. by spawning processes that will execute on the slave nodes. Beowulf clusters help speedup program execution time, which is made possible by splitting a task into several sub-tasks that can run in parallel on the slave nodes. As it pertains to database architectures, many schemes have been developed to distribute data across several databases (nodes). Also, there are many algorithms that have been researched to perform the Spatial Join operation. These schemes should theoretically improve the execution time of geospatial queries by performing program tasks in parallel. Several schemes and their implementation details are discussed in this document.

The work presented in this document attempts to improve the performance of spatial queries by distributing the data set across several nodes of a cluster and executing queries across these nodes in parallel (in particular, the work is aimed at reducing the time required for Spatial Join Queries). The rest of this thesis is organized as follows. Chapter 2 provides an overview of spatial databases and distributed spatial databases. Chapter 3 discusses the hardware used (including the Beowulf cluster), software used, and the test data used. Chapter 4 describes the entire process of distributing, querying and visualizing the spatial data; it also describes the user interface developed for performing these operations. Chapter 5 discusses the various declustering techniques, Join algorithms, and rendering operations. Chapter 6 describes the experiments performed and also discusses the results. Chapter 7 concludes with suggestions for future work.

Chapter 2 Overview of Parallel Spatial Databases & Survey of Previous Work

This chapter reviews spatial databases and discusses the need for parallel spatial databases. Also surveyed are various declustering algorithms and join algorithms. The first section gives a brief overview of spatial databases, spatial data types, spatial queries, and spatial indexing. The second section discusses the need for parallel spatial databases and describes the methodology used to deploy parallel spatial databases. The third section surveys declustering and join algorithms.

2.1 Overview of Spatial Databases

2.1.1 Spatial Database

A spatial database [11, 12] describes the location and shape of geographic features. A spatial database system is a database system with additional capabilities for handling spatial data. In particular, a spatial database system provides spatial data types in its data model and query language. Spatial indexing capabilities are also provided.

2.1.2 Spatial SQL

SQL is a standard query language for Relational DBMSs. In this case, it has been extended with spatial types to create spatial SQL. Databases have also been extended to support these new spatial data types. Current ORDBMSs support user defined data types, including spatial types.

The Open GIS Consortium [21] was founded by various software vendors to formulate industry-wide standards related to GIS interoperability. The OGIS standard recommends a set of spatial data types and functions which are crucial for spatial data querying. In this work, PostgreSQL/PostGIS is the spatial database system used. PostgreSQL [2] is an ORDBMS that is extended to support spatial data types. PostGIS is an extension to PostgreSQL which allows GIS (Geographic Information Systems) objects to be stored in the database. PostGIS also includes support for GiST-based R-Tree spatial indexes, and functions for analysis and processing of GIS objects. The GIS objects supported by PostGIS are a superset of the "Simple Features" defined by the OpenGIS Consortium (OGC). The latest version of PostGIS supports all the objects and functions specified in the OGC "Simple Features for SQL" specification [21].

2.1.3 Spatial Data Types

The GIS objects supported by PostGIS are a superset of the Simple Features defined by the OGC. The following are the spatial types supported by the PostGIS [1] Database:

Point: Discrete location represented as a coordinate pair. For example, a city may be represented as a point in a large geographic area. E.g.: (0 0)

LineString: Set of ordered coordinates represented by a string of coordinates. Examples could be rail road tracks and streams. E.g.: (0 0, 1 1, 1 2)

Polygon: Closed feature whose boundary encloses a homogeneous area represented by a closed string of coordinates which encompass an area. Examples could be land use areas and lakes. E.g.: (0 0, 4 0, 4 4, 0 4, 0 0)

MultiLineString: Collection of Lines. E.g.: ((0 0, 1 1, 1 2), (2 3, 3 2, 5 4))

MultiPoint: Collection of Point Objects. E.g.: (0 0, 2 3)

MultiPolygon: Collection of a set of polygons. E.g.: (((0 0, 4 0, 4 4, 0 4, 0 0), (1 1, 2 1, 2 2, 1 2, 1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))

GeometryCollection: Collection of different objects. E.g.: (POINT (2 3), LINESTRING ((2 3, 3 4)))

The objects in space have to be represented by a spatial database system. These could be represented as any of the above data types. For example, the objects in space could be thought of as cities, streams, buildings, etc.

2.1.4 Types of Spatial Queries

In general, spatial queries can be classified as *single scan* and *multi scan queries*:

1. Single Scan: Single scan queries usually necessitate a single scan through the relation they operate on. Therefore, the execution time is usually linear with respect to the number of objects stored in the corresponding relation. Window & distance queries are examples of single scan queries.

"Find all the features within a Bounding Box" is an example of a window query. For a distance query, it is possible to express queries such as "return all features within 100 units of a particular point". The above queries in PostGIS are as follows:

Window Query: SELECT gid, the_geom FROM <table_name> WHERE the_geom && GeometryFromText (`BOX3D (X1 Y1, X2 Y2) `:: box3d, -1)

Distance Query: SELECT gid, the_geom FROM <table_name> WHERE distance (the_geom, GeometryFromText (`POINT(X Y) `, -1)) < 100

The first query is a bounding box query – it specifies that all features that lie within the specified bounding box should be returned. The second query specifies that the feature ids and features that lie within 100 units from a given point should be returned.

2. Multi Scan: Multi scan queries involve instances where objects have to be accessed several times. Therefore, execution time is generally not linear but super linear with respect to the number of objects. For example, a spatial join operation in a relational database system is a multi scan query. A spatial join is a join which compares any two objects through a predicate on their spatial attribute values.

Consider two relations *hydrolin* and *rail.* Each of these relations has the geometry column: the_geom of type: MULTILINESTRING. "Find all the railways which are going across a river" is an example of a spatial Join. In this example, we are using a spatial distance function as the join condition.

Spatial Join: SELECT h.gid, r.gid FROM hydrolin as h, rail as r WHERE distance (h.the_geom, r.the_geom) = 0

Again, consider two relations *bc_roads* and *bc_municipality*. The query "What is the length of the roads contained within each municipality?" is an example of a spatial join. In this example, a spatial interaction condition "contains" is used as the join condition.

Spatial Join: SELECT m.name, sum (length (r.the_geom))/1000 as roads_km FROM bc_roads AS r, bc_municipality AS m WHERE overlaps (r.the_geom, the_geom) AND contains (m.the_geom, r.the_geom) GROUP BY m.name ORDER BY roads_km

The spatial join is a computationally intensive operator to implement. The efficient implementation of the spatial join operator is, thus, desirable. Spatial Joins are usually performed in two steps: Filter step & *refinemen*t step. In the filter step, an approximation of the spatial object, for example the minimum bounding rectangle, is used to remove the features which are not part of the result, and produce candidates that are a superset of the actual result. In the refinement step, each candidate is examined to check if it is part of the result – this is a CPU-intensive operation. This check requires running a CPU-intensive computational geometry algorithm, and consumes most of the time involved in a join operation.

2.1.5 Spatial Indexing

Spatial indexing enables efficient access to the data. There are two major trends for indexing spatial data. The first trend uses space-driven structures; these are indices based on partitioning of the embedding 2D space into rectangular cells, independent of the distribution of objects. The most popular space-driven structure is the Quadtree and its variants. The second trend uses data-driven structures; these indices are based on partitioning the set of objects independent of the space. Examples of data-driven structures are the R-tree and its variants. R-trees are more popular than Quadtree's. This is because unlike the Quadtree, an R-tree adapts to data distribution while keeping the tree balanced. Therefore, the body of work focuses only on the R-tree and its derivatives.

R-trees [26], introduced by Guttman, are an extension of the B-tree designed for efficient indexing of multidimensional objects with spatial extent. An R-tree is a height balanced tree. A leaf node contains an array of entries, [MBR, OID] where MBR is the Minimum Bounding Rectangle of the object in the database and OID is the Object identifier. A non leaf node contains an array of entries of the form [MBR, child pointer] where child pointer is the address of a lower node in the R-tree and MBR covers all the MBRs in the lower node's entries. An example of a R-tree is shown in Figure 2.1.

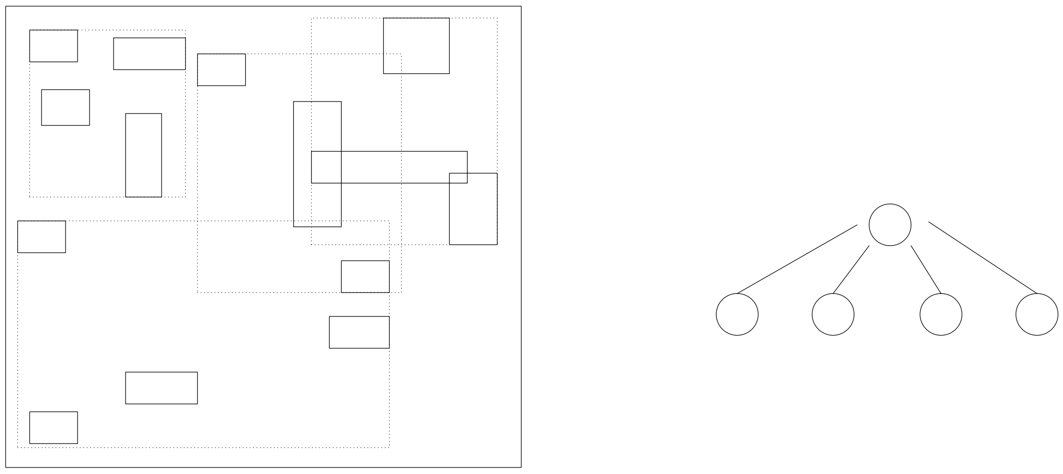


Figure 2.1 Example of a R-tree

The insertion algorithm for an R-tree will place a new entry E in the leaf node whose MBR needs the least enlargement to include E. During a node overflow, the split algorithm will split the node that overflows into two nodes. Then, the algorithm selects two entries which are the most distant ones. These entries are the first entries of two nodes. The remaining entries are assigned using as the criterion the minimum area required to cover the new entry.

The R*-tree [5] was introduced by Beckmann. The insertion algorithm follows the nodes in which the MBR has the minimum increase of overlap. The R*-tree's split algorithm chooses a split that results in a minimum overlap between the MBRs, whereas the R-trees's split algorithm chooses the split that results in the least enlargement of the MBRs. The reinsertion algorithm increases storage utilization and improves the quality of the partition making it almost independent of the sequence of insertions. The R*-tree method minimizes both the coverage (for intermediate nodes) and the overlap (for leaf nodes).

2.2 Parallel Spatial Databases

2.2.1 Why Parallel Spatial Databases

With the rapid increase in the availability of spatial data from a wide variety of sources like satellite images, mapping agencies, etc., there is an increasing demand for systems that can store and effectively manipulate such large spatial data sets. Spatial database systems are the solution of choice. The ability to store and query this enormous amount of data is critical but may lead to performance degradation. The performance problems associated with large databases have been widely documented by researchers for many years and several techniques have been devised to cope with this. One of the techniques that have gained popularity in recent years is parallel processing of spatial database operations. Spatial database operations are often time-consuming and can involve a large amount of data, so they can generally benefit from parallel processing. Parallelism improves the response time of spatial queries. The design of parallel database systems [13] often provides an impressive speedup when processing spatial queries. Data partitioning allows parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel.

2.2.2 How it works

With parallel database systems, the assumption is that there are multiple CPUs and multiple disks available. In such a scenario, each CPU and disk combination is called a compute node of the system. This shared nothing architecture is well suited for parallel database systems. Beowulf compute clusters are probably the best-known example of shared nothing machines in existence today. Each node of the cluster has a processor and disk and each node also holds a database instance stored on its local disk. The spatial data is declustered into fragments, which are then distributed to the compute nodes. The process of data distribution across multiple disks is called declustering. This multi-processor architecture is used for executing the query in parallel; spatial queries are ideally suited for parallel execution. After declustering the data, any query issued through the master node will be invoked on the slave nodes (in parallel) and these nodes will return the results of the query back to the master node. This parallel technique of querying a spatial database will make the spatial queries execute faster.

2.3 Survey of Declustering Algorithms and Join Algorithms

2.3.1 Survey of Declustering Algorithms

Parallel database systems employ partitioning strategies to distribute database relations across multiple processing nodes. It has been shown in [13] that the data can be distributed using round-robin, hash, and range partitioning schemes. There are many other methods proposed for declustering data. J.M. Patel and D.J.DeWitt in [3] propose a *tiling* scheme to partition the data. This scheme is the spatial analog of virtual processor round-robin partitioning for handling skews in parallel joins which was proposed in [14]. The tiling scheme proposed splits the universe (which is the MBR of all the spatial features in a relation) into tiles. Features in a tile are stored on the disk corresponding to the file. A mapping function like round-robin is used to map the tiles to disks. The problem with this scheme is that it risks data skew. This problem could be reduced by increasing the number of tiles. A similar method for declustering has been proposed for redundancy based declustering of spatial objects in a parallel spatial database in [15]. In [15], the number of tiles is always equal to the number of partitions. The data is synthetically generated and uniformly distributed, so data skew is not considered at all. A declustering algorithm based on tiles is proposed in [4], in which it creates partial spatial surrogates (approximation of the spatial feature) when the spatial features overlaps tiles that are mapped to multiple nodes. This scheme reduces the disk overhead. It has been shown in [16] that a space filling Hilbert curve could be used for declustering Cartesian product files with multiple attributes; this approach could be applied to distribute data across processors for queries which involve only one attribute.

In [18], it has been seen that a good declustering could be achieved by using a variation of the Hilbert based declustering method [16], as applied in the Hilbert packed R-trees [19]. In this method, the data is sorted on the Hilbert values of the centers of their rectangles and then packed into R-tree leaf nodes. These leaves are assigned to the nodes in a round-robin fashion. In [17], declustering algorithms for parallel spatial Joins are proposed. Different declustering schemes are proposed using R*-trees [5]. R*-trees are used to perform a spatial join in a shared-disk environment. Two R*-trees are built on the two relations on which the Join is performed, the leaves of the R*-tree are distributed, and different schemes decide when and on which nodes the leaves are distributed.

In this work, a new Join Algorithm which uses R*-tree declustering is proposed. The R*-tree structure of one relation is used to decluster both the join inputs, where as in the scheme proposed in [17], two R*-tree structures are used to decluster the join inputs. In [17], a spatial Join is performed in a shared disk architecture; this is in contrast to our new Join algorithm which is performed in a shared nothing architecture.

2.3.2 Survey of Join Algorithms

Various spatial join algorithms have been proposed for evaluating spatial joins. Most of the algorithms proposed decluster the relations into a number of fragments. The join is then performed by pair-wise joining of these small fragments. The declustering algorithms proposed for parallel spatial join algorithms generally fall into two categories: *dynamic partitioning* function & *static partitioning* function. A Dynamic partitioning function inserts spatial features into a spatial index, like an R-tree and distributes the leaves of the spatial index to nodes. A static partitioning function divides the space into regions and maps the regions to nodes. The spatial join algorithm proposed in [3], uses a static partitioning function. The work in [20] examined the data partitioning mechanism for parallel spatial joins, in which it uses a static partitioning function as well. In [4], two declustering functions are employed, namely declustering using replication and partial spatial surrogates (approximation of spatial features). The authors designed two Join algorithms: *shadow* join and *clone* join. Shadow join uses only the approximation of spatial features when declustering and clone join uses the exact spatial features. The shadow join is similar to the parallel spatial Join in [20], except that in [20] the MBR of the entire feature is used while declustering. Spatial join processing in [25] is based on grid representation of spatial objects. In [25], spatial data is decomposed by superimposing a grid on the universe. Each element of the grid is transformed into z-values by applying z-order transforms. The z-values are then used to perform the spatial join. In [23, 24], Seeded trees are used to create an index and then perform a tree join algorithm. In [17], R*-trees are used to perform the spatial join in parallel and the algorithm uses two R*-trees to distribute data.

Chapter 3 Hardware and Software Tools Used

This chapter describes the hardware and software tools used for the experiments presented in this thesis. A detailed overview of the Beowulf cluster, programming environment, and the spatial databases used is provided. In addition, the datasets used for testing are also described.

3.1 Beowulf Cluster:

The vehicle utilized for parallel processing of spatial queries is a Beowulf cluster. The Beowulf cluster housed at the Department of Computer Science at the University of New Orleans consists of 72 compute nodes, 1 login/submit node, and 1 file server. 63 of the slave nodes are 2.2 GHz Intel Pentium IV systems with 1 GB of memory, 20 GB of local disk storage, and Fast Ethernet networking; the remaining 9 nodes are 2.4 GHz Intel Pentium IV systems with 1GB of memory, 20 GB of local disk storage, and Gigabit Ethernet networking. The file server is a dual 1.4 GHz SMP Intel Xeon system with 2 GB of memory, 500 GB of disk storage, and Gigabit Ethernet networking. Last, but not least, is the master node, which is a dual 2.2 GHz SMP Intel Xeon system with 2 GB of memory, 300GB of disk storage, and 2 Ethernet interfaces. The interface that links the cluster with the private Beowulf network is Gigabit Ethernet; the external interface is 100BaseTX. All of the systems in the cluster are networked together by a Cisco Catalyst 4000 series switch with 10/100/1000 auto-sensing ports and a 12-Gbps backplane. Debian GNU/Linux (2.4.26 kernel version) is the operating system for all nodes in the cluster.

This cluster was recently benchmarked using HPL 1.0, a portable, freely available implementation of the standard High Performance Computing Linpack Benchmark. HPL solves a random, dense linear system in double precision (64 bits) arithmetic on distributed memory computers [9]. Benchmarking yielded a "theoretical peak" performance of approximately 63 Gigaflops. The amount of raw computing capacity provided is therefore quite substantial. Figure 1 provides a broad overview of the cluster architecture.

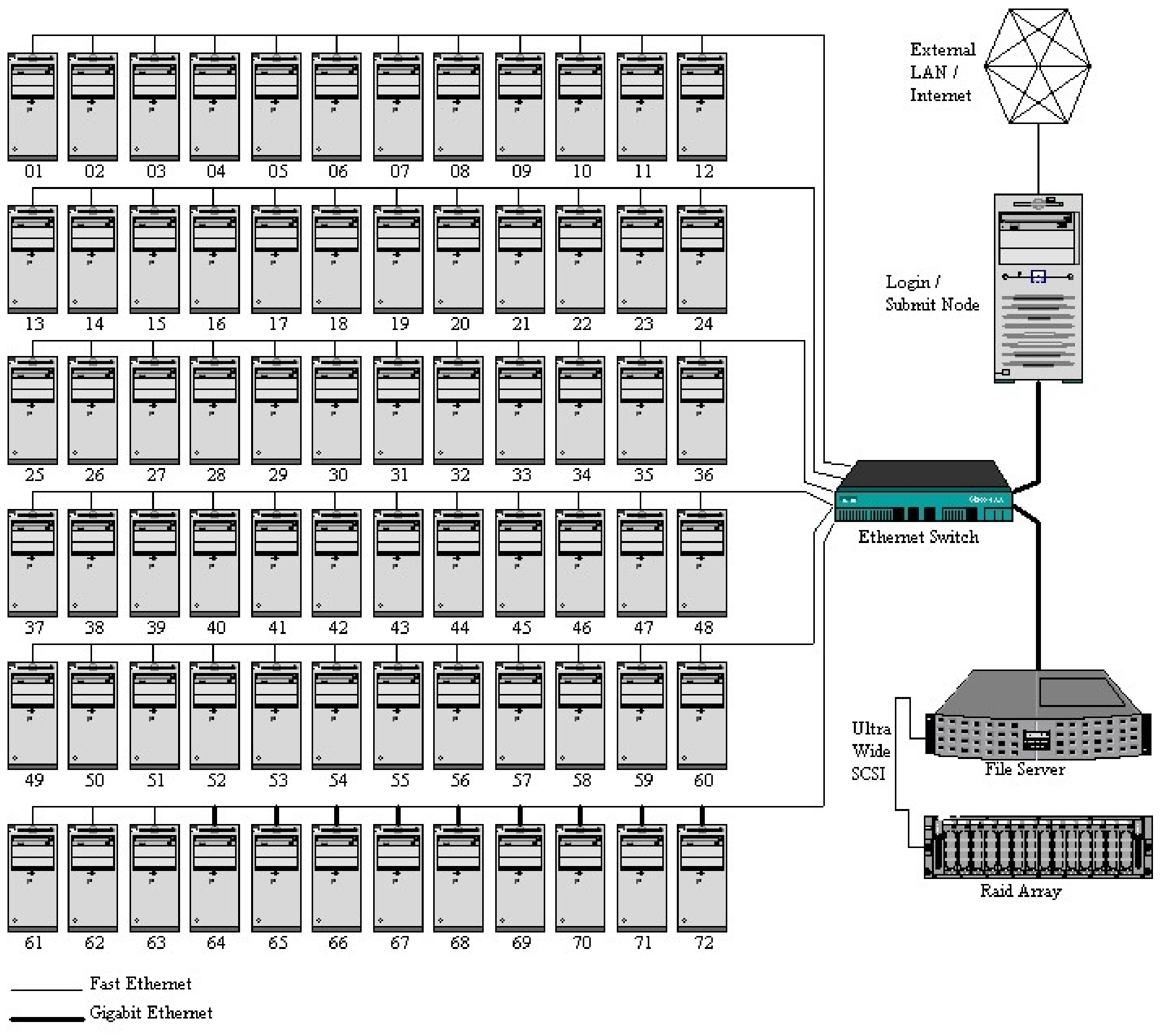


Figure 3.1 Overview of 72-node Beowulf Cluster

There are 3 broad classes of parallel machines: shared memory systems, shared disk systems, and shared nothing systems. In a shared nothing architecture, each processor has its own local memory and disk – nothing is shared and all communication between the processors is accomplished via the communication network. The shared nothing architecture is the most widely used design for building systems to support high performance databases, primarily due to its relatively low cost and flexible design.

Beowulf compute clusters are probably the best-known example of shared nothing machines in existence today. A number of factors can be attributed to this including the emergence of relatively inexpensive but powerful off-the-shelf desktop computers, fast interconnect networks such as Fast Ethernet and Gigabit Ethernet, and the rise of the GNU/Linux operating system. A compute cluster, for all intents and purposes, can be defined as a Pile of Processors [7] interconnected via some sort of network. Each node in the cluster usually has its own processor, memory, and optionally an I/O device such as a disk. However, it is important to note that a cluster is not simply a network of workstations (NOWs) – in a cluster, the compute nodes are delegated only for cluster usage and nodes typically have a dedicated, "cluster-only" interconnect linking them. In the Beowulf paradigm, all of the compute nodes (also called slave nodes) are isolated on a high-speed private network that is not directly visible to the outside world. A single computer called the "master" or "head-end" provides a single entry point to the cluster from the external network. This machine is sometimes referred to as the login or submit node. Essentially, the master node is a system with 2 network interfaces – one connected to the private Beowulf network and the other connected to the regular LAN. Users of the cluster will typically log in only to the master node. From here, they can spawn processes that will execute on the slave nodes. Beowulf clusters are typically applicable to any area of research where a speedup in program execution time is possible by splitting a large job into several sub-tasks that can run concurrently on the compute nodes. This capability provides an intriguing setting for evaluating the use of such a cluster for hosting large GIS databases, where there is an ever-increasing need for greater computing power to process and query the massive amounts of information being stored.

3.2 Spatial Databases: PostgreSQL/ PostGIS

PostgreSQL [2] is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. PostgreSQL has a spatial extension called PostGIS that follows the OpenGIS Consortium's "Simple Features Specification for SQL" [21], a proposed specification to define a standard SQL schema that supports storage, retrieval, query, and update of simple geospatial feature collections via the ODBC API.

PostGIS [1] allows GIS (Geographic Information Systems) objects to be stored in a PostgreSQL database and includes functions for analysis and processing of GIS objects. Point, Line, Polygon, Multipoint, Multiline, MultiPolygon, and GeometryCollection object types can be stored in PostGIS. These are specified in the Open GIS well known Text Format.

PostgreSQL is a good choice for use on a GNU/Linux system (such as our Beowulf cluster) because of the robust, native support that it provides on this platform. Each of the nodes in the Beowulf cluster executes the PostgreSQL server engine and has its own database instance stored on its local disk; the file server is not used because of the potential performance penalty associated with a large number of nodes accessing a shared file system simultaneously. Data will instead be distributed evenly across the slave nodes and the user interface will be executed on the master node. Any query issued through the master node will be invoked on the slave nodes and the slave nodes will return the results of the query back to the master node.

3.3 Programming Environment:

The data visualization component has been programmed in Java using the Geotools [6] Package, which is a leading open source Java library for developing OpenGIS [21] solutions and has been in existence since 1996. As a result, Geotools has been used to develop the interface for viewing geospatial data since it provides a rich set of graphics and rendering methods.

To decluster data across several nodes and to query the databases on all the nodes, Java threads are used to make the queries run concurrently (in parallel) on the nodes. A Java program is run on the master node, which creates threads to connect to the slave nodes, execute the request, and get back the output to the master node. The PostgreSQL JDBC driver is used to connect to the PostGIS database. JFreeChart [10] (a free Java library) is used to generate graphs depicting query performance.

3.4 Datasets used for Testing:

Two collections of datasets were used for testing. Each collection has two datasets. The first collection's two geospatial data sets were obtained from the Bureau of Transportation Statistics (BTS) [22]: the 2002 National Transportation Data Hydrographic and Railway network Features of a collection of adjacent States (Louisiana, Kansas, Mississippi, Arkansas, Texas, Oklahoma, and Missouri). Two datasets from the second collection were obtained from BTS as well: the 2002 National Transportation Data Hydrographic Features of Louisiana State and the 2002 National Transportation Data Railway network of Louisiana State.

The Rail Network is a comprehensive database of the nation's railway system at the 1:100,000 scale. The hydrographic features are a state-by-state database of both important and navigable water features. The Hydrography features include rivers, canals, etc. and the Rail features represent railroads.

The first collection of datasets was for Louisiana State. The Hydrography dataset contains data of Louisiana State. The spatial domain of this data set is west: -94.043189, East: -88.758388, North: 33.019359 and South: 28.855127. The Rail dataset contains data of Louisiana State as well. The spatial domain of this data set is west: -94.042735, East: -89.534031, North: 33.019180 and South: 29.376390. The second collection of datasets was for a collection of adjacent states: Louisiana, Kansas, Mississippi, Arkansas, Texas, Oklahoma, and Missouri. The spatial domain of the Hydrography data set is west: -105.676612, East: -88.098426, North:

40.613582and South: 25.837376. The spatial domain of the Rail dataset is west: -106.606615, East: -88.111893, North: 40.590972and South: 25.891600.

These datasets are obtained as compressed ESRI Shapefile format. When imported into a PostGIS database, the sizes of these datasets are given in Table 3.1 and Table 3.2.

Join queries are performed on each collection of datasets. The Join query joined the Rail dataset with the Hydrography dataset. The query "find all the railways which are going across a river" is a spatial join on Rail and Hydrography. Here, the spatial predicate is whether a railway intersects a river.

When the Shapefiles are imported into PostGIS, each relation contains a feature-id, a few columns describing the data, and the feature-geometry itself. The feature type of the feature-geometry may be one of 7 different types specified by the "Simple Features" specification of the OpenGIS Consortium: *point*, *linestring*, *polygon*, *multipoint*, *multilinestring*, *multipolygon*, and *geometrycollection*. The geometry of the features in these data sets is represented as type MULTILINESTRING. This feature is represented in the database as a set of coordinates, which may be either 2-dimensional or 3-dimensional. The features in these datasets are 2-dimensional.

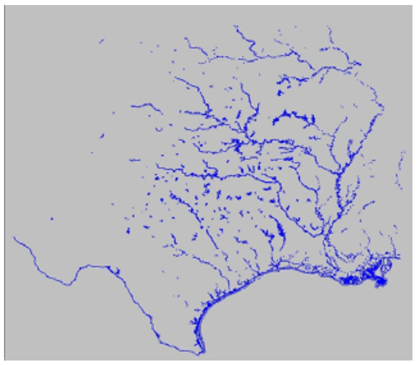
Datasets	# of features	Total Size	Type of Features
Hydrography	31400	20.1MB	MULTILINESTRING
Rail	3543	1.8MB	MULTILINESTRING

Table 3.1 Louisiana TIGER Data Information

Datasets	# of features	Total Size	Type of Features
Hydrography	99737	65MB	MULTILINESTRING
Rail	35492	19MB	MULTILINESTRING

Table 3.2 TIGER data of a collection of adjacent States (Louisiana, Kansas, Mississippi, Arkansas, Texas, Oklahoma, and Missouri) in the US

The datasets are rendered and displayed in the following Figures: Figure 3.2.a and Figure 3.2.b display the hydrographic and railway data of a few states. Figure 3.3.a and Figure 3.3.b display the data of Louisiana State.



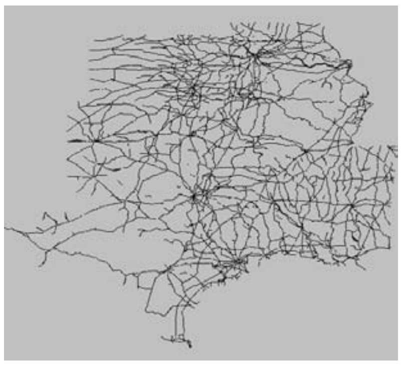
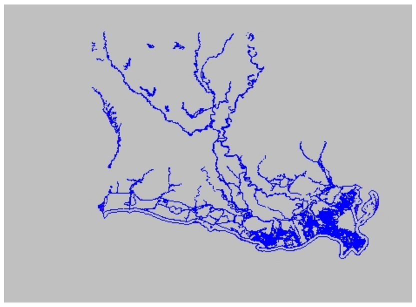


Figure 3.2a Hydro Dataset of a Figure 3.2b Rail Dataset of a collection collection of Sates.of States



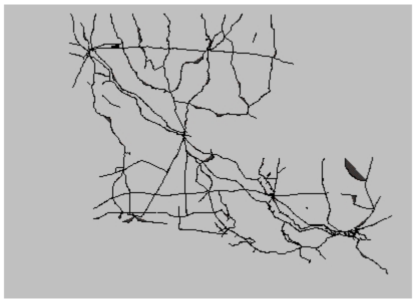


Figure 3.3a Hydro Dataset of Louisiana **Figure 3.3b Rail Dataset of Louisiana**

Chapter 4 Methodology

This chapter describes the processes involved in the efficient parallel processing of geospatial data. The different processes are declustering, querying, and visualization of the spatial data. The user interface developed for visualization and the user interface that was developed for executing commands across several nodes in parallel are also detailed in this chapter.

The first section gives a pictorial representation of the process of declustering, querying, and visualization of spatial data. This is followed by a description of each of the processes in different sections. The last section deals with the user interface developed for declustering the spatial data, and performing the parallel join operation.

4.1 Diagrammatic Representation of the Parallel Processing of Spatial Data

The following are the main processes that are involved in performing a Join in parallel. They are:

1. Declustering
2. Query Execution in Parallel
3. Merge the Results from the Nodes
4. Visualization of the data

Declustering is the process of distributing data across several nodes. Query Execution is done in parallel on the nodes. The results of the query are returned to the master node, which merges the results to get the overall result for the query. The result obtained from merging is visualized using a custom Geospatial Data Viewer.

A program runs on the master node which distributes data from the master node to several slave nodes. After this preparation step, a query can be issued to the master node. This query is then sent to all slave nodes i.e. the query execution is done in parallel across the nodes. This design makes good use of the independent disk and memory subsystems of each slave node. The results from the nodes are then sent back to the master node which merges the result. The result of the query is visualized in the Viewer, a user interface to view spatial data. The steps described previously are shown in Figure 4.1.

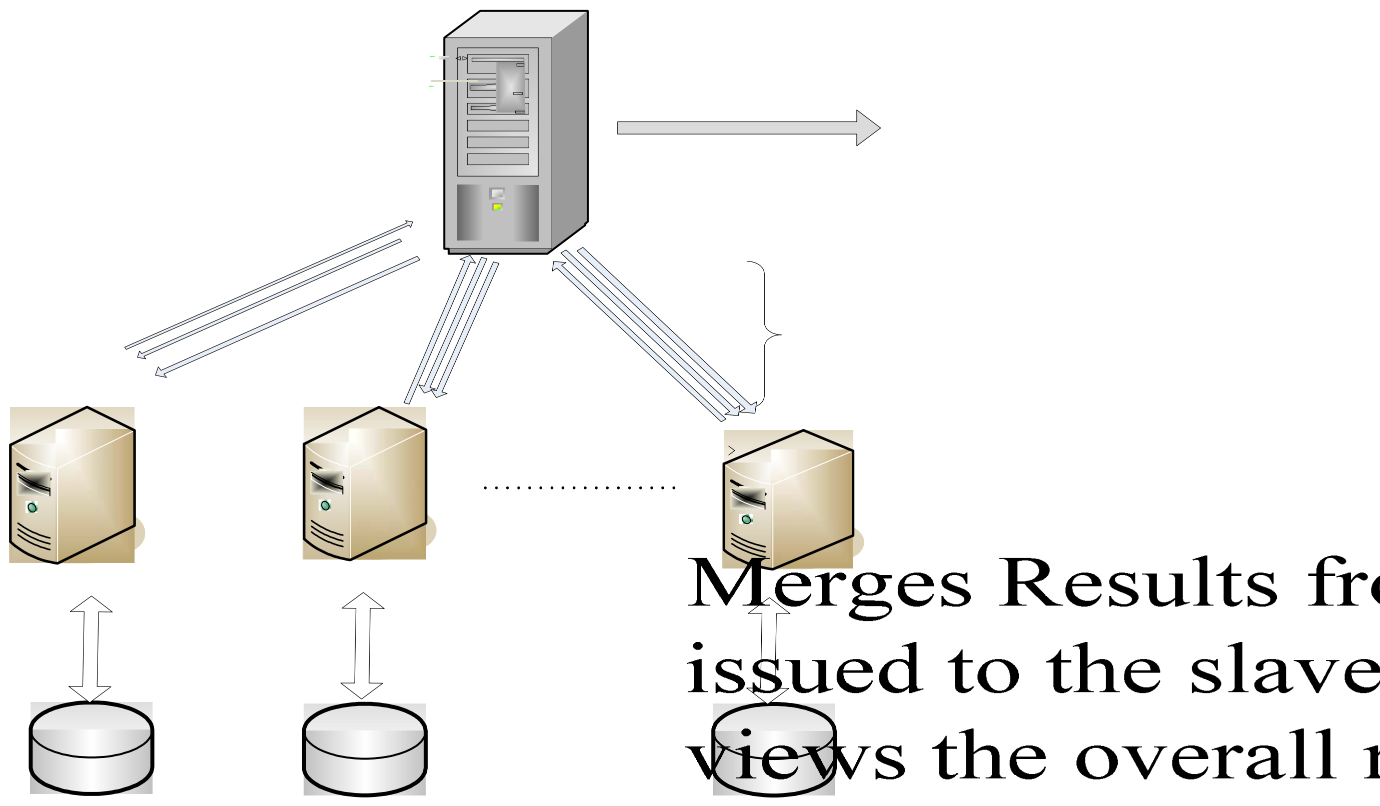


Figure 4.1 Parallel Processing of Spatial Data.

4.2 Declustering

Declustering is the process of distributing data across several nodes. A good declustering technique distributes data such that,

1. There is nearly the same amount of data on each node (reduced data skew).
2. There is minimum replication of data on the nodes.
3. Parallel processing of a spatial join is done efficiently without the need for inter-slave data exchange.

The master node distributes data to several slave nodes. Each slave node has a fragment of the data stored in its database. There are many distribution (partition) techniques available. Some of them are basic round-robin, Tiling [3], Hilbert Curve [16], Hilbert packed R-trees [18], Seeded Trees [23, 24] and Z-order [25] declustering techniques.

The data is partitioned using a partitioning technique. A spatial partitioning function divides both the join inputs into smaller partitions. The join is then performed by pair wise joining of the smaller partitions. Spatial partitioning functions for spatial join algorithms are usually categorized into two types: *static* and *dynamic* declustering techniques.

In the *static* declustering technique, the space is initially decomposed into regions. Each region is mapped to a disk and the features inside a region are stored on the disk the region corresponds to. The tiling technique of [3] is an example of a static declustering technique. In the *dynamic* declustering technique, the features are inserted into a spatial index. The leaves of the spatial index are mapped to disks. In this method, the space is decomposed into regions recursively. There are a minimum and maximum number of features a region can have. Once a region exceeds the number of features than specified, the region is split and the features are re-assigned to the two new regions. This process is done recursively until all the features are inserted into a spatial index. In [17], a dynamic declustering technique based on the R*-tree is proposed in a shared disk environment. It is noted that an index is built for each of the two join relations. The algorithm starts from the roots of both trees and traverses both trees in a depth first order. For each intersecting pair of directory rectangles (minimum bounding rectangle of the data rectangles in the corresponding subtrees), the algorithm follows the corresponding references to the nodes in the lower level of the trees. Results are found when the leaf level is reached. The leaves are then assigned to the disks by one of these mapping functions: a plane sweep order, a round robin assignment or a dynamic assignment.

Figure 4.2 represents the declustering technique. The master node distributes data using a declustering algorithm. After partitioning the data into fragments, a hashing function is used to map the fragments to the nodes. In the case of the tiling scheme [3], the data is divided into a set of tiles. Each tile is then mapped to a node by using a hash function.

To distribute the data, the master node starts server socket programs on each of the slave nodes. The role of this server socket is to receive data from the master node and insert the data received into the database. Once the entire data set is distributed, the server sockets are closed.

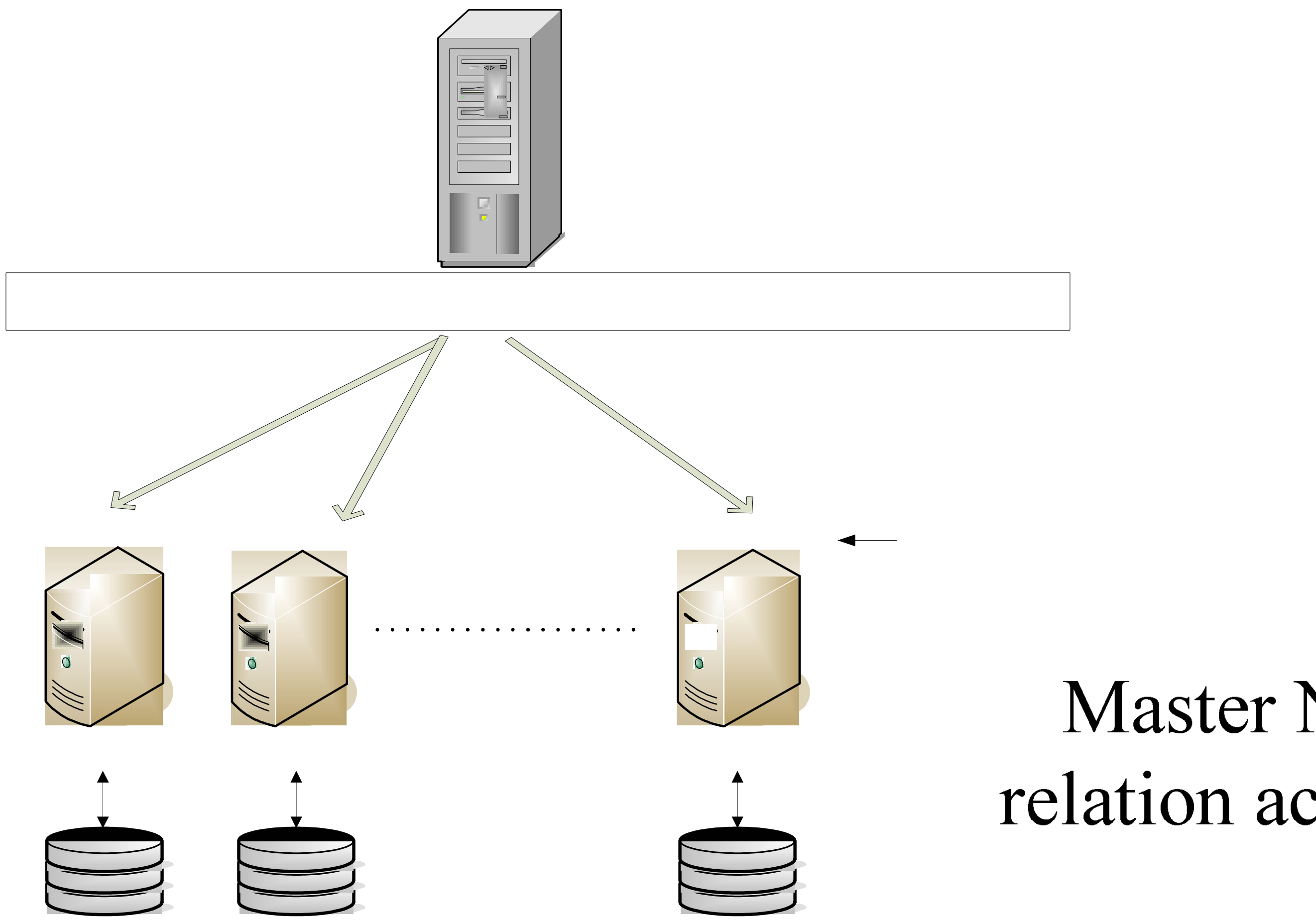


Figure 4.2 Declustering

4.3 Parallel Execution of the Query

Each slave node has its own database instance. Once the data from the master node is distributed, the query is executed. The query execution is done in parallel on each slave node. Figure 4.3 illustrates the execution of the query.

The declustering algo
slave nodes. Whi

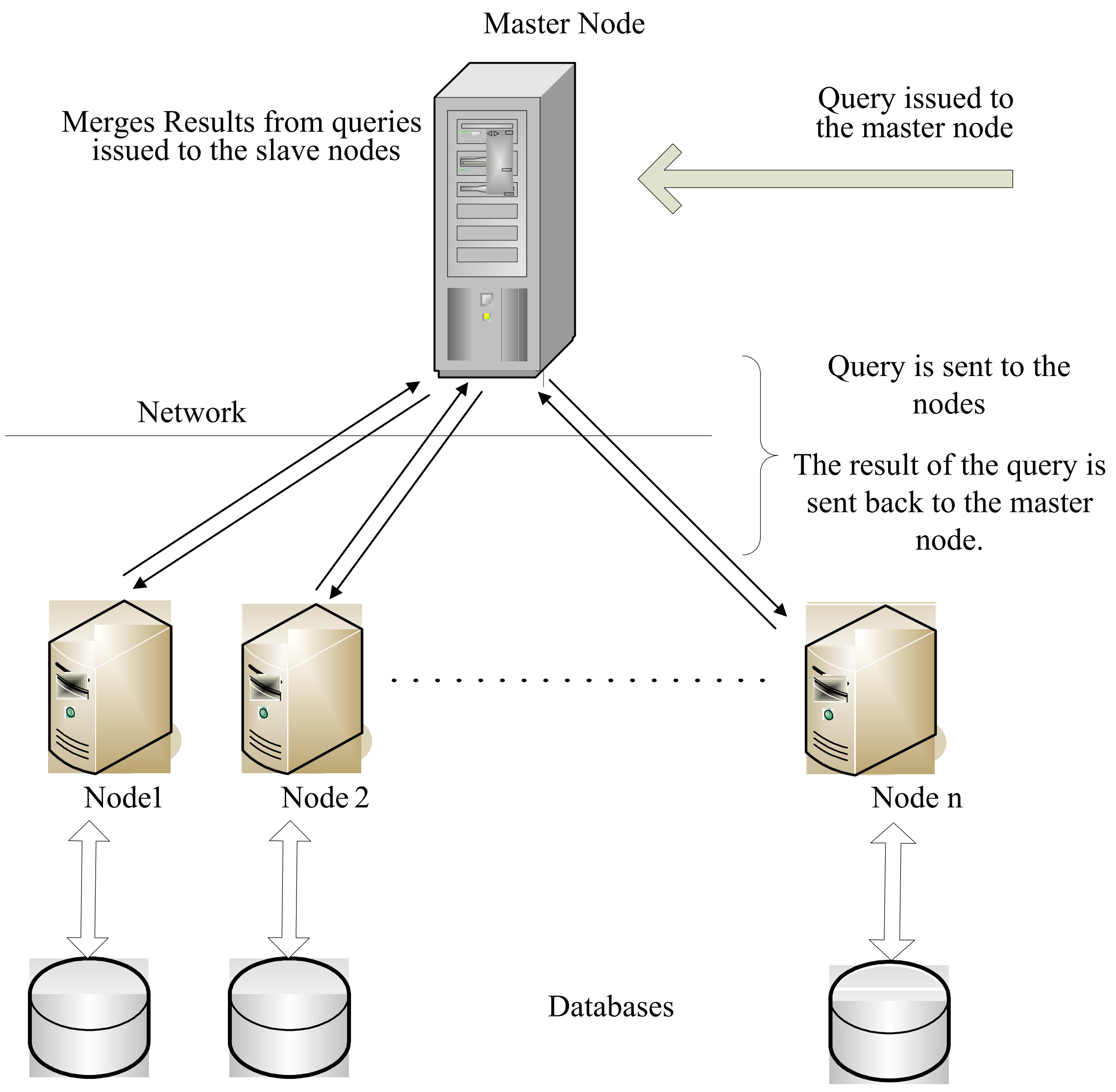

Master Node
Merges Results from queries issued to the slave nodes
Query issued to the master node
Network
Query is sent to the nodes
The result of the query is sent back to the master node.
Node1
Node 2
Node n
Databases

Figure 4.3 Parallel Execution of the Query

The sequence of steps performed is as follows:

1. Query is issued at the master node.
2. The master node spawns several threads which connect to the slave nodes. Each thread connects to a slave node and executes the query on the node. Each query is executed in parallel across the nodes in the cluster that contains fragments of the spatial data.

3. After each query execution, each thread returns the result to the master node.
4. The master node merges the results from all the slave nodes and outputs the overall result.

The result obtained is in textual format. However, the result obtained could also be visualized in the Viewer, which is a user interface to view the spatial data. This is explained in Section 4.4 below.

4.4 Viewing

The output from the execution of the query is in textual format. To view the geographical output, the data is rendered in the viewer. The master node merges the query results from all the slave nodes and renders the overall result in the developed Viewer, pictured in Figure 4.4.

The viewer is a user interface developed for viewing spatial data. The Viewer provides a number of tools to operate on the map. These tools are zooming, panning, print, adding layers from Shapefiles and adding layers from PostGIS [1]. The Geotools [6] library is used to build this interface. Figure 4.4 shows the viewer developed for viewing the geospatial data. Figure 4.5 shows the *adding layers from Shapefiles* option in the viewer. Figure 4.6 shows the *adding layers from PostGIS* option in the viewer.

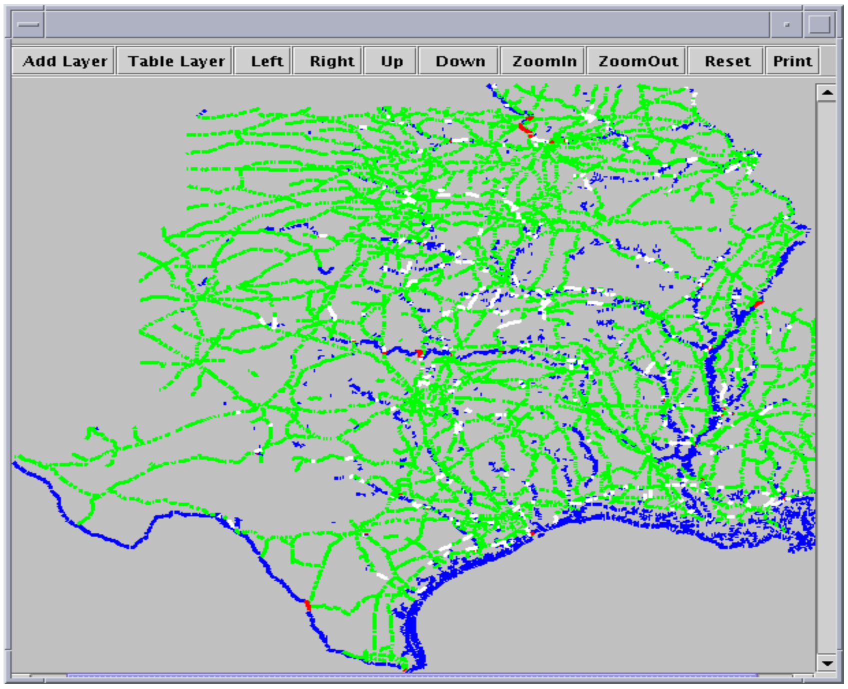
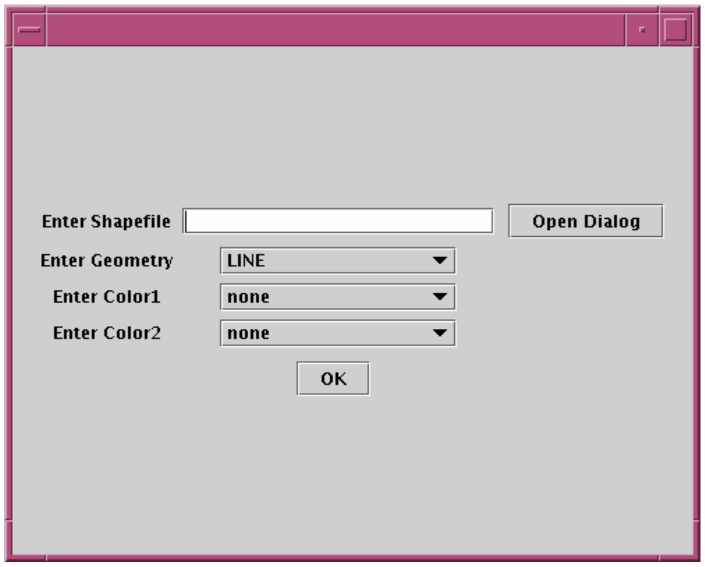


Figure 4.4 Viewer



Enter Shapefile

Enter Geometry LINE

Enter Color1 none

Enter Color2 none

Figure 4.5 Adding Layers from Shapefile

The algorithm to render a Shapefile is given in Section 5.4.1. The algorithm to render data from PostGIS is given in Section 5.4.2. The form and the color of the geometry are specified by the user (see Figure 4.5 & Figure 4.6).

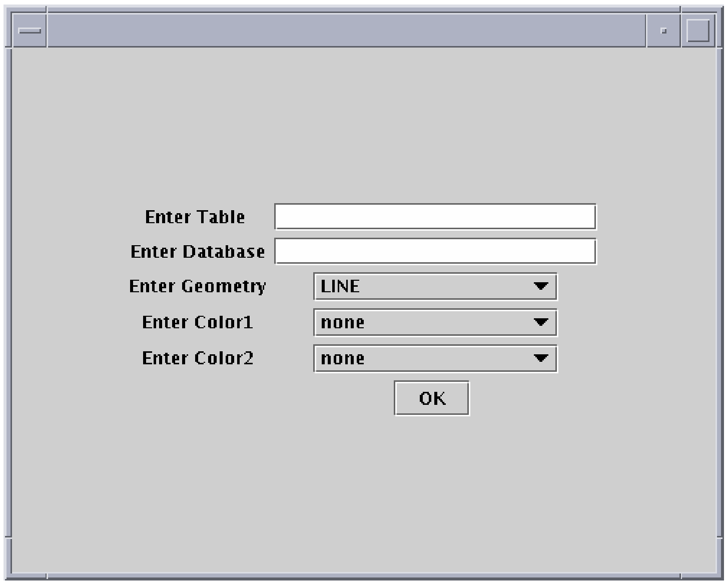


Figure 4.6 Adding Layers from PostGIS

4.5 User Interface

The user interface provides for the loading of spatial data into the master node. It also provides the capability for distributing, querying and updating of the spatial data to multiple compute nodes in a cluster. The user interface enables the creation of tables and databases before distributing the data to the nodes. This interface also lets the user send commands to the nodes for execution. For example, to start a database server on all the nodes, a command is sent through this interface. Parallel spatial join algorithms work on fragments of spatial data stored on the nodes. These algorithms are executed using this interface.

The User Interface developed for the above mentioned processes is seen in Figure 4.7. There are many options in the Menu bar: to create tables and databases, to start the database server (PostgreSQL [2] server), to perform the parallel join algorithms, and to decluster geospatial data. SQL queries, when given in the SQL Query Text Area in the interface, can be executed on the nodes as needed. The number of nodes on which the desired process is executed on is selected using this interface as well.

Figure 4.7 shows the interface developed. Figure 4.8 shows the Table option in the interface which enables the creation of tables in a database.

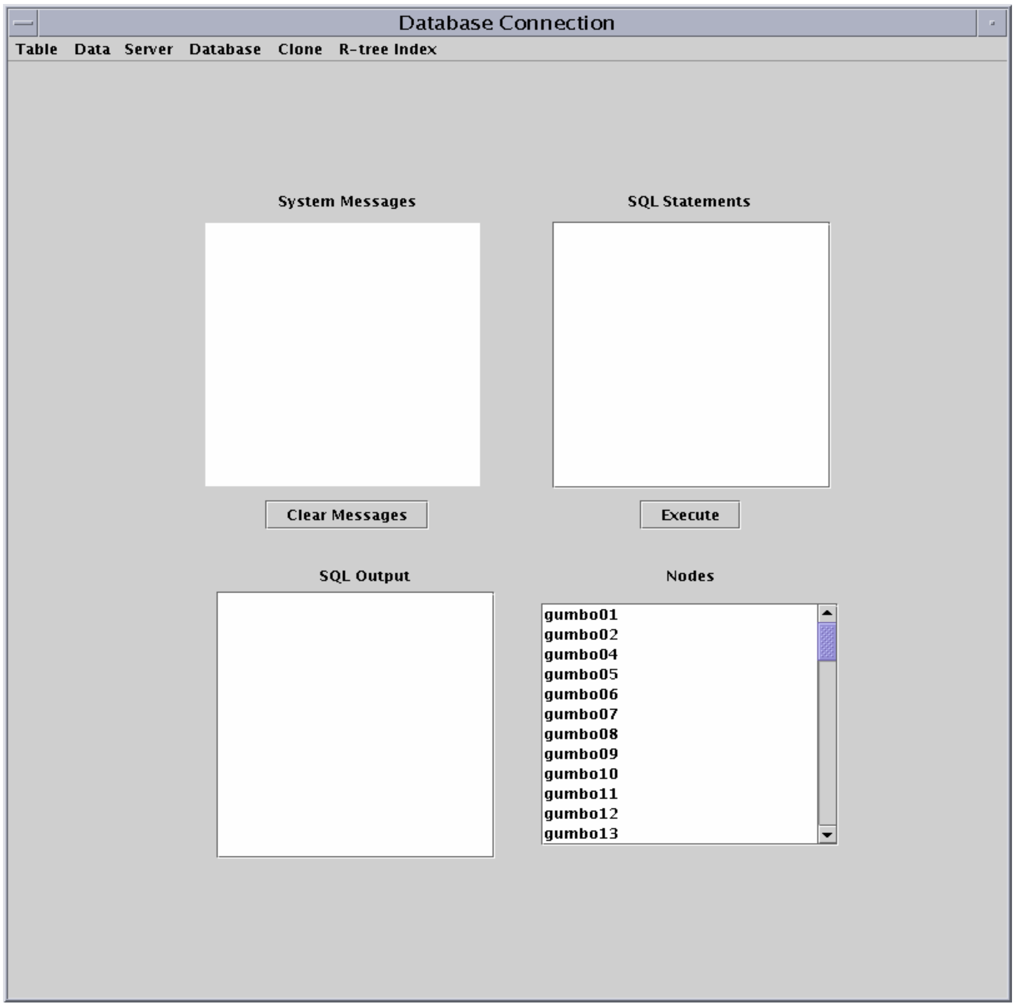


Figure 4.7 User Interface

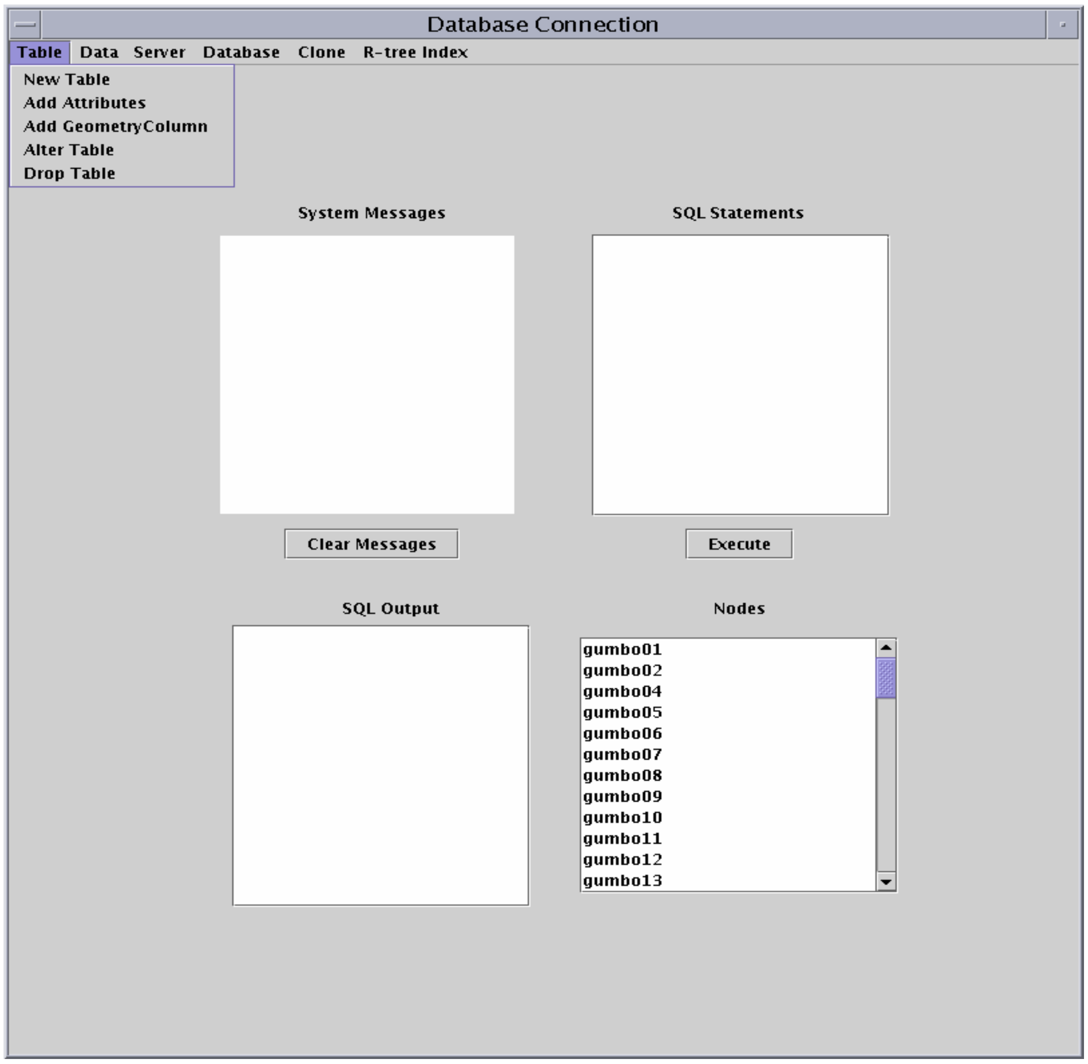


Figure 4.8 Table Option in the Interface

Chapter 5 Discussion of Algorithms

In [4], the two inputs to be joined are declustered using a *static* declustering technique. The space is divided into a set of *tiles* and each tile is mapped to a disk. Each of the two join relations is declustered using this technique. In [17], the join inputs are declustered using a *dynami*c declustering technique. Two R*-tree structures are used to decluster the data. The algorithm starts from the roots of both trees and traverses both of the trees in a depth first order. For each intersecting pair of directory rectangles (minimum bounding rectangle of the data rectangles in the corresponding subtrees), the algorithm follows the corresponding references to the nodes in the lower level of the trees. Results are found when the leaf level is reached. The leaves are then assigned to the disks by either of these: a plane sweep order, a round robin assignment or a dynamic assignment.

In this chapter, a parallel join algorithm based on a *semi-dynamic* declustering technique is proposed. The proposed algorithm uses an R*-tree to decluster the first relation. The second relation is declustered statically using a tiling like approach. The leaves of the R*-tree, built on the first relation, is used to decluster the second relation. The features of the second relation that intersect with a leaf are stored on the node that the leaf corresponds to.

The proposed algorithm is compared with two different versions of the *clone join* algorithm proposed in [3]. It is noted that the clone join is a parallel join algorithm that makes use of a *static* declustering technique based on tiling. The comparison of our algorithm with the tiling based clone join is motivated by the well-established fact that *static* declustering techniques perform better [20] than their *dynamic* counterparts.

The first section describes the spatial declustering techniques. Tiling scheme [3] and the R*-tree based semi-dynamic approach are described. Implementation details of these techniques are also provided. The second section reviews parallel spatial joins in general. The third section discusses the join algorithms used. The fourth section describes how the Geotools classes were used to render Shapefiles and data from the PostGIS databases. This section also highlights the other features of the viewer.

5.1 Spatial Declustering Techniques

Two different techniques are implemented. The first one is the tiling technique described in [4]. The second one is the proposed semi-dynamic approach which uses a single R*-tree to decluster the data from both the join inputs.

5.1.1 Tiling Technique (Declustering using Replication)

The tiling technique implemented in this chapter was originally proposed by Patel & DeWitt in [4]. This technique is implemented to compare it with the new algorithm presented in this work.

5.1.1.1 Declustering Algorithm using the Tiling technique

The universe of a relation is defined as the Minimum Bounding Rectangle that covers all the spatial attributes of the relation. The universe of the relation to be distributed is divided into a number of tiles of the same size. Each tile is mapped to a node according to some hash function; to test this algorithm, a round robin function is used. Spatial objects that are within a tile are stored on the node it is mapped to. Spatial objects which overlap multiple tiles are stored on the nodes that correspond to these tiles. So, the spatial objects that are within the first tile would be stored on the node it corresponds to, and the objects that are within second tile will be stored on the node it is mapped to and so on. If a spatial object overlaps more than one tile, it is stored in both the nodes that map to these tiles. The number of tiles chosen should be no less than the number of nodes.

This scheme presents two disadvantages, namely:

1. Data distribution skew
2. Replication

Data fragments stored on the various nodes may vary greatly in size, resulting in *data distribution skew*. Also, because a spatial object could be stored on more than one node, this results in *replication* of the same object.

The universe is a rectangle that covers all the features in a relation. It may contain some regions where there are very few features. So, if the universe is divided into a smaller number of tiles, then more data may be inserted on a node compared to the other nodes as shown in Figure 5.1. One solution is to decrease the inequalities between the nodes by increasing the number of tiles. As the number of tiles increases, the data distribution skew reduces. However, because the universe is divided into more tiles, many features may overlap more than one tile resulting in increased replication. Spatial objects, which overlap more than one tile, are replicated in many nodes. So, when the number of tiles increases, the percentage of replication grows.

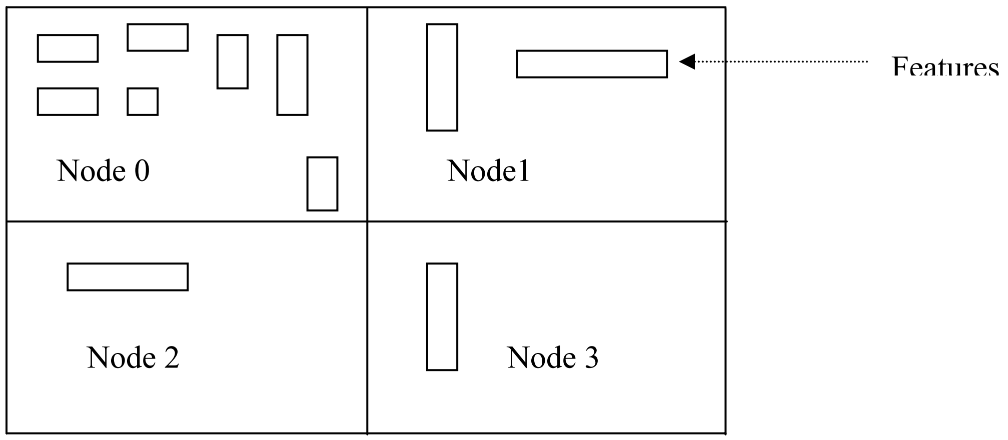
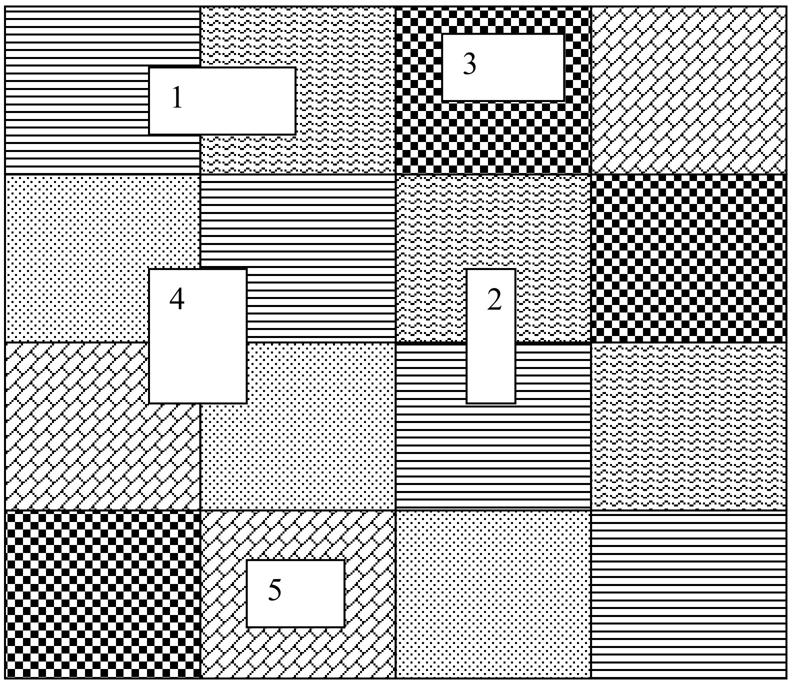


Figure 5.1 Data Distribution Skew

The universe is divided into tiles i.e., it's divided into a number of rows and columns. A hash function is usually utilized to map tiles onto nodes. An example of a hash function is round robin. Given *c* columns and *n* compute nodes, a round robin function will map the tile with the column number *i* and row number *j* onto node *(i+j*c) mod n.*

The example of Figure 5.2 assumes 5 nodes and the universe composed of 16 tiles. A relation is being declustered across 5 nodes using 16 tiles. In the Figure, 1, 2, 3, 4 & 5 are the first 5 features of the relation. Feature 1 is stored on both Node 0 and Node1. Feature 2 is also stored on both Node 0 and Node1. Feature 3 is stored on Node 2. Feature 4 is stored on Node 0, Node 1, Node 3, and Node 4. Feature 5 is stored on Node 3.



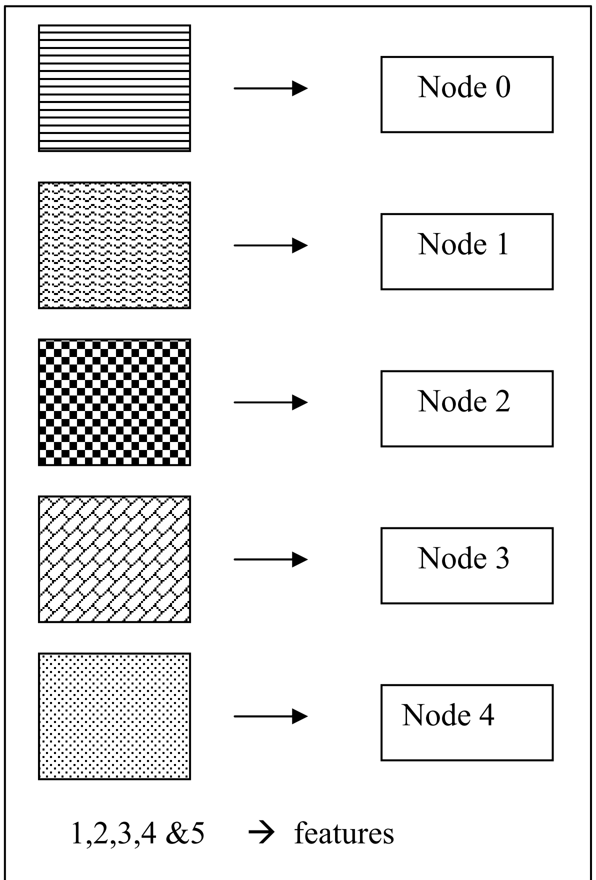


Figure 5.2 Declustering Tiling Technique.

As it can be seen from Figure 5.2, there is both data replication and data distribution skew. Feature 4 is stored on Nodes 0, 3 & 4. Also Feature 1 is stored on Node 0 and Node 1, which shows a feature is replicated on more than one node. Also, Node 0 has three features whereas Node 2 has just one feature which shows Data Distribution Skew. This is listed in Table 5.1.

Nodes	Features
0	1,2,4
1	1,2
2	3
3	4,5
4	4

Table 5.1 Tiling Scheme Data Distribution

5.1.1.2 Implementation Details

Using PostGIS [1] *&&* (Overlaps operator) and the PostGIS [1] *extent* function, the declustering algorithm for the tiling technique is implemented in Java. Suppose A, B are two features. If A's bounding box overlaps B's bounding box, then the *&&* operator returns true. The extent function takes a geometry column as an argument and will return a BOX3D giving the maximum extend of all features in the table. The above mentioned algorithm is implemented as follows:

- Get the universe of the table to be distributed using the *extent* function in PostGIS [1]. Then use the *xmin* (Box3D), *ymin* (Box3D), *xmax* (Box3D) and *ymax* (Box3D) functions in PostGIS [1] which gives x_{min}, y_{min} x_{max}, y_{max} points.
- Given these points, the universe (Minimum Bounding Rectangle of all features) is divided into a number (# of rows * # of columns) of tiles.
- Make a Bounding Box, Box3D representation of each tile.
- For each bounding box use the *&&* Overlaps operator in PostGIS [1] and get all the spatial objects overlapping this bounding box and store the obtained geometries in the node corresponding to that tile using the hash function described above.

The above steps are performed for declustering one relation. For a spatial join, two tables have to be declustered. This algorithm is used for spatial join because the same algorithm could be applied to two different tables without any difficulties. In that case, the universe should now be the Minimum Bounding Rectangle that covers all the spatial features of both the relations.

5.1.2 R*-tree based Semi-Dynamic Approach

This is a new algorithm proposed in this paper. The spatial data is declustered using an R*-tree. The mapping of leaves to compute nodes is done via one of the two hash functions we propose. Each of the two hash functions has its distinct advantages as explained below.

5.1.2.1 Declustering algorithm using the R*-tree based Semi-Dynamic approach

Using this algorithm, the problem of data replication and distribution skew are reduced. An R*-tree [5] is built on the relation to be declustered. The R*-tree is an indexing scheme for spatial data. The leaves of this tree are treated as tiles and are distributed across various nodes. Each leaf of the R*-tree is mapped to a node according to some hash function. The leaves of the tree are numbered from the left to the right. There are two hash functions that have been used. The features within the leaves are stored on the node the leaf is mapped to.

For the first hash function, k (where $k = \lceil \text{Total leaves/ \# of Nodes} \rceil$) successive leaves are taken and are stored on each node. Consider an R*-tree built on a relation which has total leaves as 21. The number of nodes the data has to be distributed onto is 3. Then k =7 successive leaves are stored on each node.

For the second hash function, each leaf is stored on a node in a round robin fashion. For example, leaf number *p* is stored on the node *p mod n*, where n is the number of nodes.

5.1.2.2 Implementation Details

The semi-dynamic R*-tree based declustering algorithm is implemented in Java. *xmin* (feature), *ymin* (feature), *xmax* (feature) and *ymax* (feature) functions in PostGIS [1] are used. These functions return the coordinates of the bounding box of the feature.

1. An external R*-tree is built on the relation to be distributed. The entries in this R*-tree are the Minimum Bounding Rectangle (MBR) of the features in the relation. To find the MBRs, the above mentioned functions of PostGIS are used.
2. Each leaf of the R*-tree is mapped to a node; the features that are inside a leaf are located by querying the R*-tree for the entries in the leaf. These features are inserted into the node the leaf corresponds to, which is determined by the hashing function used.

The above mentioned approach is to decluster one relation. For a spatial join, two relations have to be declustered using the R*-tree structure of usually the smallest relation and is described below.

For the join algorithm, an R*-tree is built on one of the two relations (usually the smallest) on which the join is being performed. The leaves of this R*-tree are distributed as described above for the first relation. The features of the second relation are distributed statically using the tiling like approach. The leaves of the R*-tree built on the first relation are assumed as tiles, and thus, the features of the second relation which overlap the leaves are stored on the nodes corresponding to the leaves. Since the leaves of the R*-tree do not overlap, there would not be any replication for the first relation, though there would be some replication for the second relation. The second relation is not completely declustered in this approach; only the features overlapping the leaves of the R*-tree structure of the first relation are distributed since such features are only the candidates for the Join.

The implementation details of the process of declustering the two relations for spatial join are given below. To decluster two relations in this approach, the first relation is declustered as explained in section 5.1.2.1, i.e. an R*-tree is built on it. So for every leaf of the R*-tree of the first relation, the features that are within the leaf are stored on the node the leaf correspond to. The features in the second relation which overlap this leaf are stored on the same node.

The examples of Figure 5.3 & Figure5.4 assume Relation 1 has 8 features (*1, 2, 3 ... 8*), and Relation 2 has 7 features (*1, 2, 3 …7*). Figure 5.3 shows the R*-tree built on the first relation: it has 4 leaves: *a, b, c* and *d*. It also shows the features numbered from 1 through 8. Each leaf is mapped to a node according to the hash functions as described above.

So for the first hash function, the features which are contained in leaves *a* and *b* are stored on the first node. The features contained in the leaves *c* and *d* are stored on the second node. For the second hash function, the features in leaves *a* and *c* are stored on the first node and the features which are stored on leaves *b* and *d* are stored on the second node.

The R*-tree structure of the first relation is used to decluster the second relation. According to the first hash function, the features which overlap the leaves *a* and *b* are stored on the first node and the features which overlap leaves *c* and *d* are stored on the second node. For the second hash function, features which overlap *a* and *c* are stored on first node and features which overlap *b* and *d* are stored on second node. Figure 5.4 illustrates the declustering for the second relation based on the R*-tree built on the first relation.

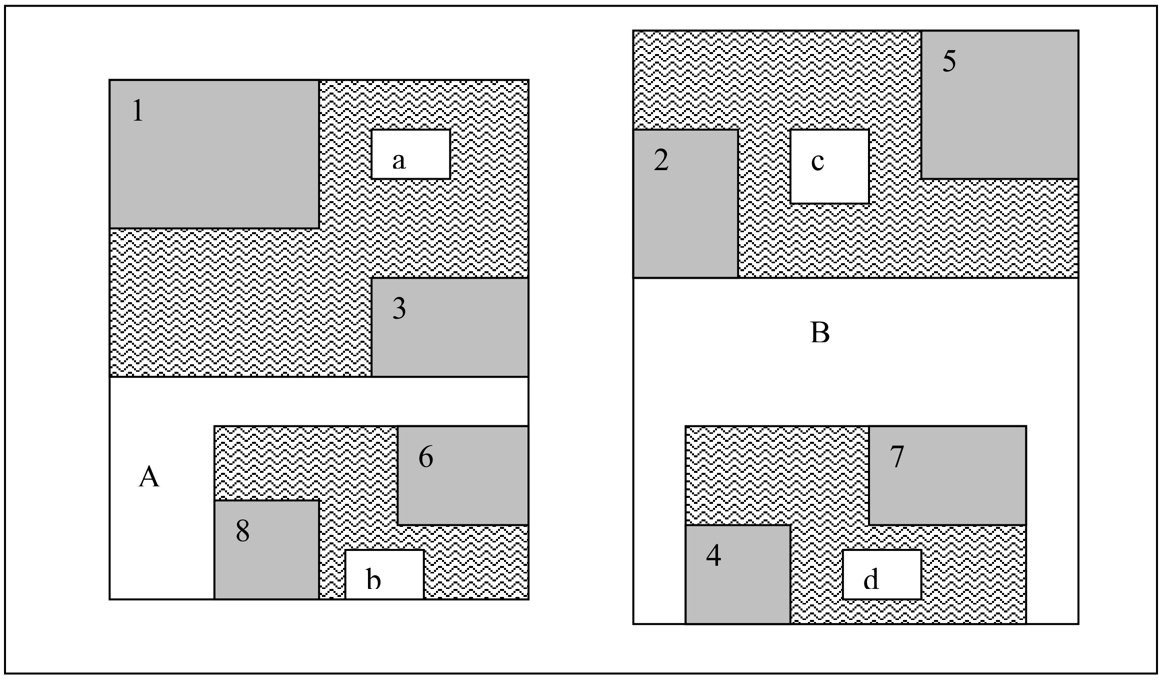


Figure 5.3 R*-tree built on the first relation

▨ ⇨	Feature
▩ ⇨	Leaves

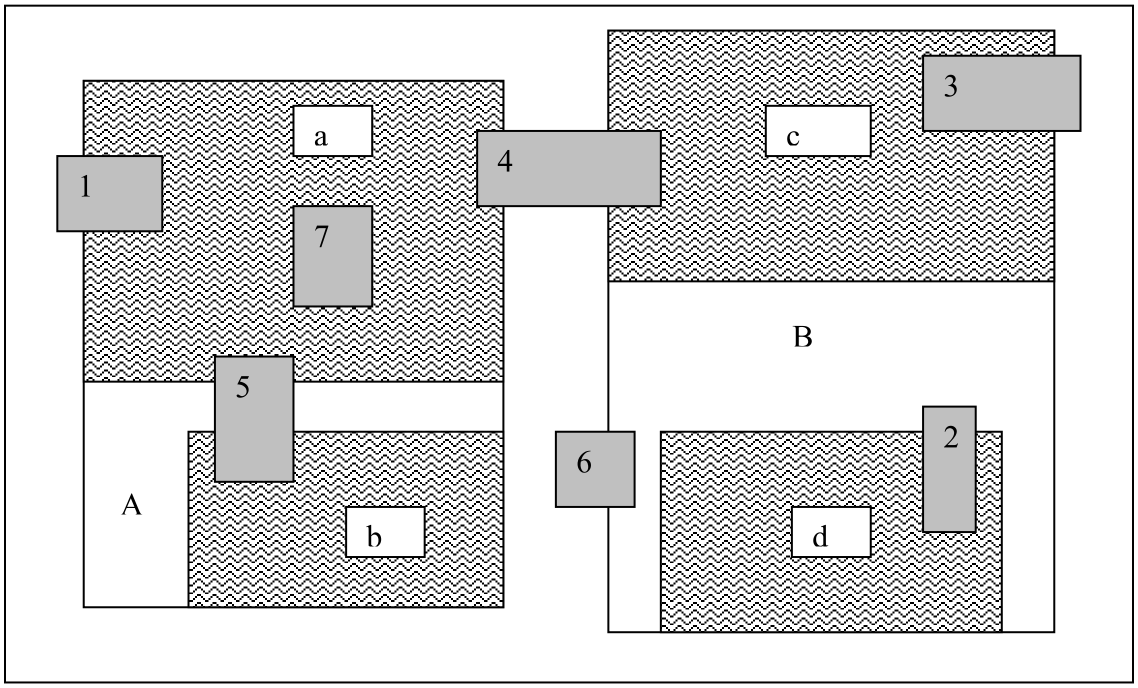


Figure 5.4 Declustering Second Relation based on the R*-tree Structure of the First Relation

As it is seen from Figures 5.4 & 5.5 above, the features of the two relations are distributed among two nodes as shown in Tables 5.2 and 5.3.

Relation/ Node	Node1	Node 2
Relation 1	1, 3, 6, 8	2, 4, 5, 7
Relation 2	1, 4, 5, 7	2, 3, 4

Table 5.2 First Hash Function

Relation/Node	Node 1	Node 2
Relation 1	1, 2, 3, 5	4, 6, 7, 8
Relation 2	1, 3, 4, 5, 7	2, 5

Table 5.3 Second Hash Function

It can be observed from Tables 5.2 & 5.3 that there is no replication for the first relation, but there is replication for the second relation. Indeed for the second relation, in Table 5.3, feature 5 is stored on both node 1 and node 2. Also for the second relation in Table 5.2, feature 4 is stored on both the nodes.

The above two hashing methods each have advantages and disadvantages. With the first one, if the answers of the spatial join request are concentrated geographically on one part of the universe (say part A in Figure 5.3), then these answers will be computed by only one node (node 1 in our example). This leads to data distribution skew. This disadvantage is reduced with the second hashing function. The replication rate, using a real dataset (see Chapter 6), of the second relation which uses the R*-tree of the first relation is seen in table 5.4. With the first hash function, the replication of the second relation is lower. But if the number of features a leaf can have is increased than the replication rate with both the hash functions is almost the same.

Number of features (min/max) per leaf	10/20	35/70	100/200
Hashing func. 1 (range hash)	101.3	82.7	28.7
Hashing func 2 (round-robin)	200.4	99.8	25.6

Table 5.4 Replication Rate with the Two Hashing Functions.

5.2 Spatial Joins

5.2.1 Spatial Join Steps

Spatial joins typically operate in two steps, shown in Figure 5.5:

Filter Step: In this step, an approximation of each spatial object, the minimum bounding rectangle is used to eliminate those tuples that cannot be part of the result; this produces a set of candidate pairs for the spatial join.

Refinement Step: In this step, each candidate is examined to check if it part of the result; this is a CPU-intensive algorithm since it has to be checked with the exact geometry.

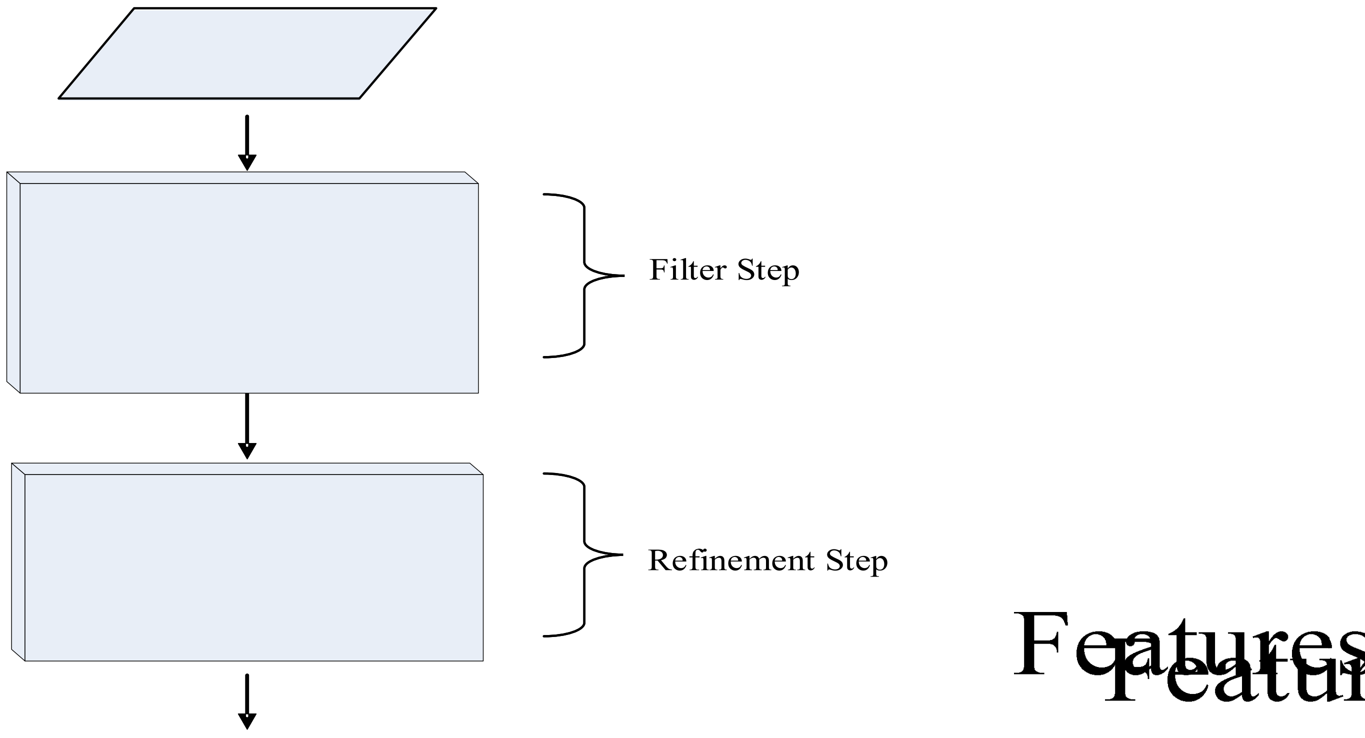


Figure 5.5: Spatial Join Operation

5.2.2 Spatial Join in Parallel

For Spatial Join in parallel, the following steps are usually performed:

1. Declustering step
2. Filter & Refinement steps

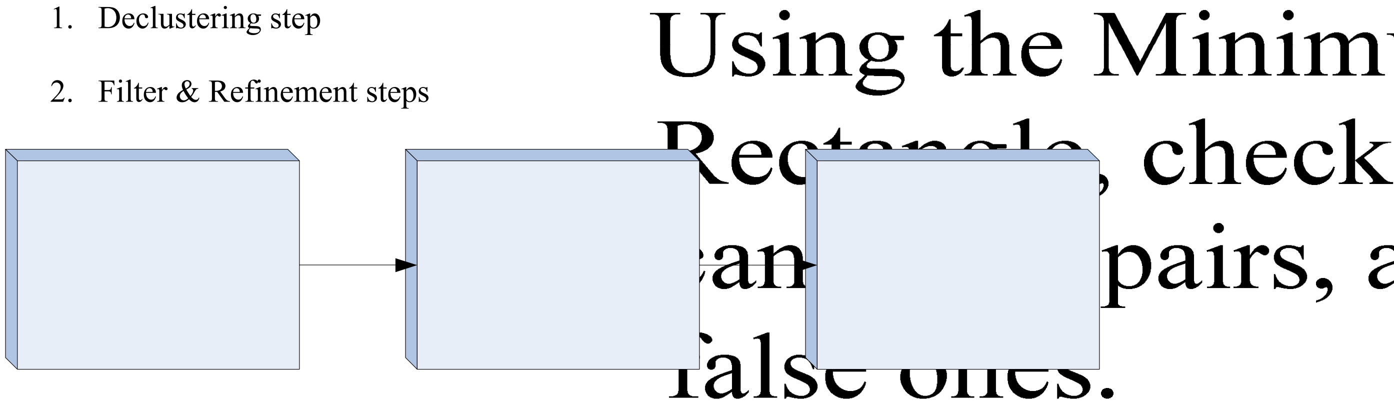


Figure 5.6 Parallel Spatial Join Operation

The first step depends on the declustering algorithm used; both tiling as well as R*- tree declustering are used. The declustering algorithms are explained in section 5.1. Next, the filter and refinement steps are performed. Figure 5.6 shows the sequence of steps.

Filter Step:

This step is performed on each node. A plane sweep algorithm [12] runs on each node to perform this filter step: Let the two relations on which the join has to be preformed be R and S. This step is performed on the MBR's of the features. This step eliminates the false hits, and gives the candidate pairs (the object identifier pairs (OID pairs)). All the features of R and S are sorted in ascending order according to the x values of their lower-left corners, *xl* of their MBR's. The first feature in this set is picked; let it be from the relation R, r. We now have to search in the sorted list S for all the rectangles, looking for the ones whose lower x values, *xl* are smaller than the x values of upper-right corner, *xu* of r, until an MBR in S is such that it has its *xl* value greater than the *xu* value of r. The resulting pairs thus obtained are the ones which overlap with r along the x-axis. These pairs are checked to see if they actually intersect, by checking to see if they overlap along the y axis. If they do overlap then the pairs are added to the result of the filter step. 'r' is marked as done and removed from the sorted set, and the processing continues with the next element in the sorted set. A sweep line is assumed to move though the sorted set. This process continues until one of the relations has been fully processed.

Figure 5.7 gives a plane sweeping example. This example assumes there are 5 features in R, and 4 features in S. Let the features in R be {R1, R2, R3, R4, R5} and in S be {S1, S2, S3, S4};

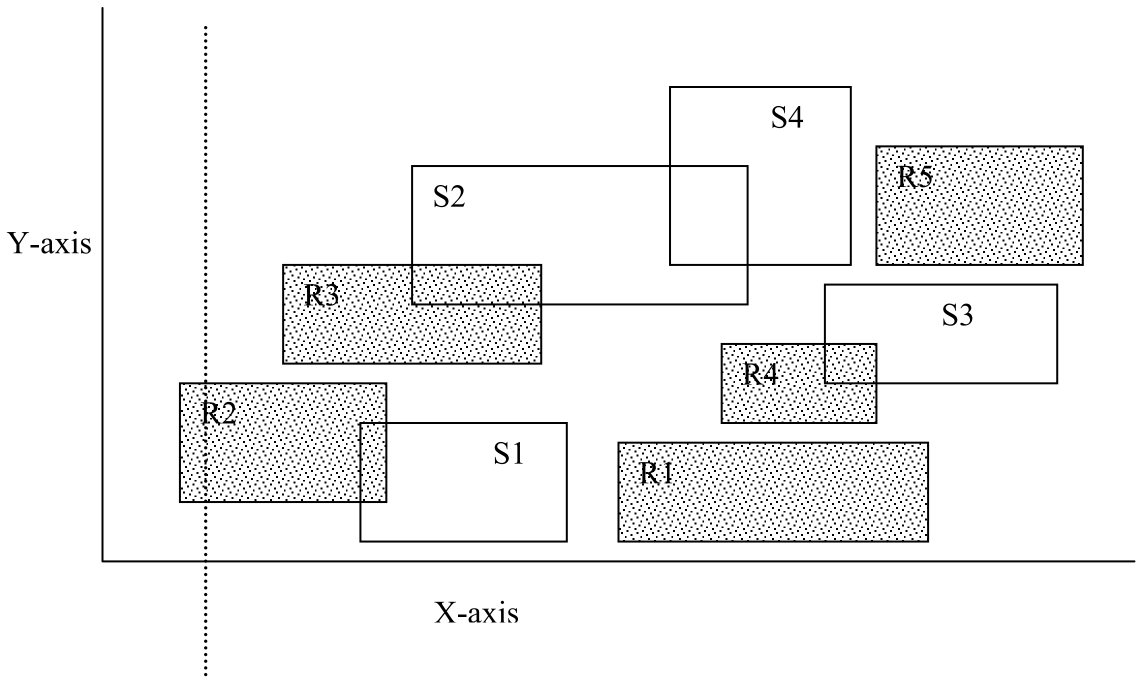


Figure 5.7 Plane Sweep Algorithm for the Filter Step

The entries are first sorted according to the *xl* values:

Sorted Set: {R2, R3, S1, S2, R1, S4, R4, S3, R5}

The process starts off with the first feature from the Sorted Set. In this case, R2 is selected initially to begin the process:

1. R2 is checked with each element in the set {S1, S2, S4, S3} until it reaches an S whose *xl* value is greater than *xu* value of R2. In this case, it stops at S2.
2. The pairs generated so far are checked along the y axis for intersection. In this case the pair (R2, S1) is checked along the y-axis. It does overlap so it is added to the result of the filter step.
3. The algorithm continues with the next entry in the Sorted Set.

So for this example: the resulting pairs would be: {(R2, S1), (R3, S2), (R4, S3)}. This algorithm is performed on the data contained in each node.

Refinement Step:

In this step, each candidate pair is examined to check if the features actually intersect. This check requires running a CPU-intensive computational geometry algorithm, and is the part of the overall algorithm which takes the most time. To know if two features are really overlapping, the *distance* function of PostGIS [1] could be used. The distance function takes two geometry features as arguments and returns the Euclidean distance between two geometries in projected units. If the distance is zero, we can say that the two features are intersecting. Otherwise, the two features are disjoint. The result of filter step is a set of OID pairs in the form (OID-R1, OID-R2), such that the MBR of the feature corresponding to the OID-R1 overlaps with the MBR of the feature corresponding to OID-R2. The refinement steps checks the features corresponding to these OIDs if they really overlap by using the distance function.

The refinement step gives the result as Object Identifier (OID) pairs in text which has the form (OID-R1, OID-R2). Each OID pair contains the Object Identifier (Feature-id) of the feature in the first relation and the OID of the feature in the second relation. To view the data, features are fetched from the relations stored on the main node. The data is first fetched from the first relation. To make this process faster, OID pairs are sorted according to the OID of the first relation. These OID pairs are sorted and the features are fetched from the first relation. When the features are being fetched from the second relation, the OID pairs are sorted according to the OID of the second relation. The data fetched is stored in temporary tables and viewed using the Viewer described in section 4.4.

5.3 Join Algorithms

5.3.1 Join Algorithm for Tiling Technique

The clone join algorithm [4] uses the tiling technique (described previously) to perform spatial join. The two join inputs are declustered using the tiling technique, and then the filter and refinement steps are applied as explained in the previous sections. Two variants of the clone join algorithm [4] are investigated.

5.3.1.1 Clone Join Algorithm, Variant 1 (CJAV 1)

This method is a version of the clone join algorithm [4]. The master node distributes the data to the nodes according to the tiling technique described previously. The filter step is performed on each node. The output of the filter step is a set of OID pairs. Each OID pair contains the Object Identifier (Feature-id) of the feature in the first relation and the OID of the feature in the second relation. This output is sent to the master node which removes the duplicate OID pairs. The duplicate OID pairs are removed from all the nodes which have the OID pair, except from one node. On which node the OID pair should be retained is picked randomly. The resulting OID pairs which are retained on each node after removing the duplicates are sent back to the respective node to perform the refinement step.

The refinement step is performed on each node. The refinement step removes the false hits. The output which is again a set of OID pairs is sent to the master node. The master node sorts OID pairs according to the OID of the first relation, so that when the features are fetched from the first relation, it would be faster. This sequence of steps is illustrated in Figure 5.8 below.

In this method, before the refinement step, the master node has to remove the duplicate OID pairs in the nodes and send back the remaining OID pairs to the respective nodes. Therefore, the master node should maintain information about which OID pair belongs to which node.

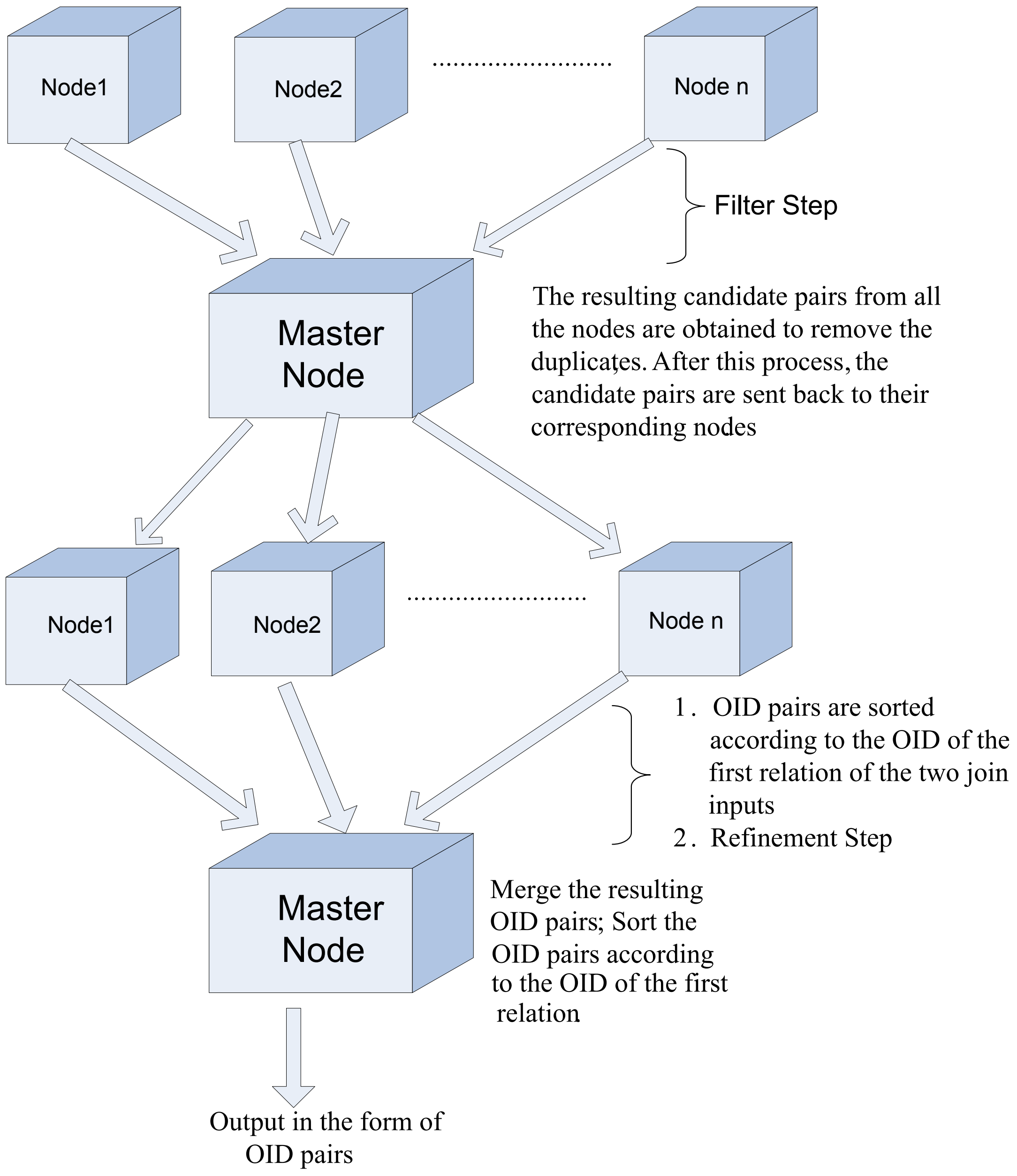

Node1
Node2
Node n
Filter Step
Master Node
The resulting candidate pairs from all the nodes are obtained to remove the duplicates. After this process, the candidate pairs are sent back to their corresponding nodes
Node1
Node2
Node n
1. OID pairs are sorted according to the OID of the first relation of the two join inputs
2. Refinement Step
Master Node
Merge the resulting OID pairs; Sort the OID pairs according to the OID of the first relation
Output in the form of OID pairs

Figure 5.8 Clone Join Algorithm, Variant 1 (CJAV1).

5.3.1.2 Clone Join Algorithm, Variant 2 (CJAV2)

In this method, the master node distributes the data to the nodes using the tiling technique described previously. The filter step is run on each node. The output of the filter step is a set of OID pairs. Each OID pair contains the Object Identifier (Feature-id) of the first join input and the OID of the second join input. The refinement step is performed on each node, and then the resulting candidate pairs are sent to the master node which removes the duplicates and sorts the OID pairs according OID's of the first relation. The output would be a set of OID pairs which are the result of the join. These steps are illustrated in Figure 5.9 below.

In this method, the master node doesn't have to remember which OID pair belongs to which node, but the problem with this method is that the Refinement step is performed on all the OID pairs prior to removing duplicates.

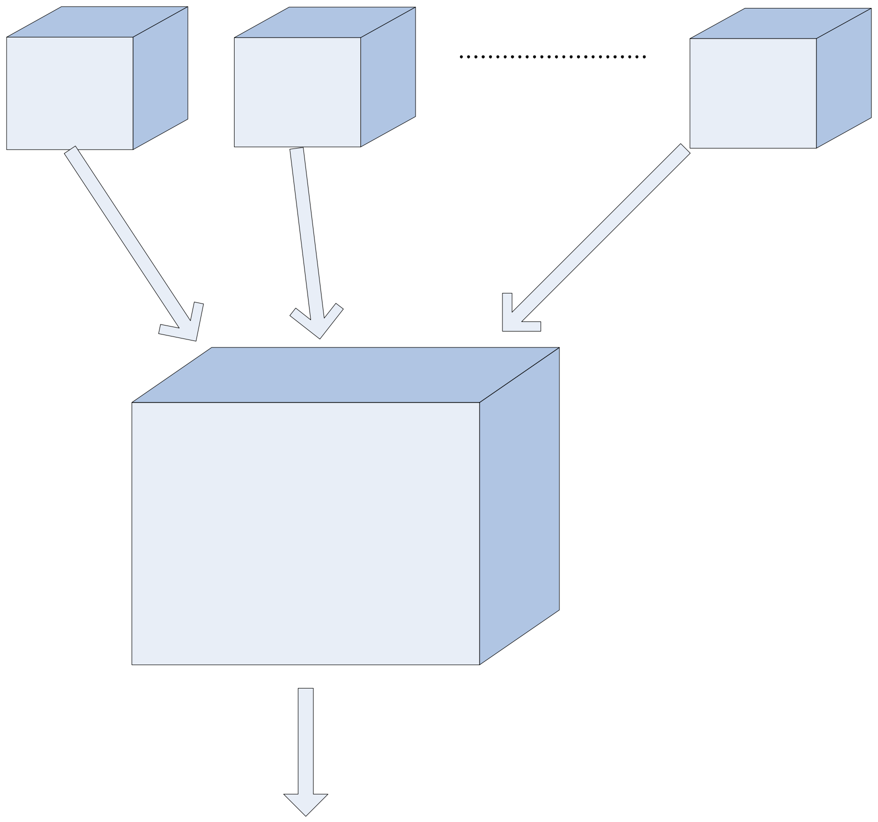


Figure 5.9 Clone Join Algorithm, Variant 2(CJAV2)

The above two variants are now compared. CJAV1 is more useful if the replication rate is high. CJAV1 removes duplicate OID pairs before performing the refinement step. So, CJAV1 reduces the OID pairs for the refinement, which could make the execution faster. On the other hand, if the replication rate is not that high, then sending the OID pairs to and from the master node would add communication overhead. In this case, CJAV2 would seem to be more appropriate to use.

5.3.2 Join Algorithm Based on Semi-Dynamic Approach

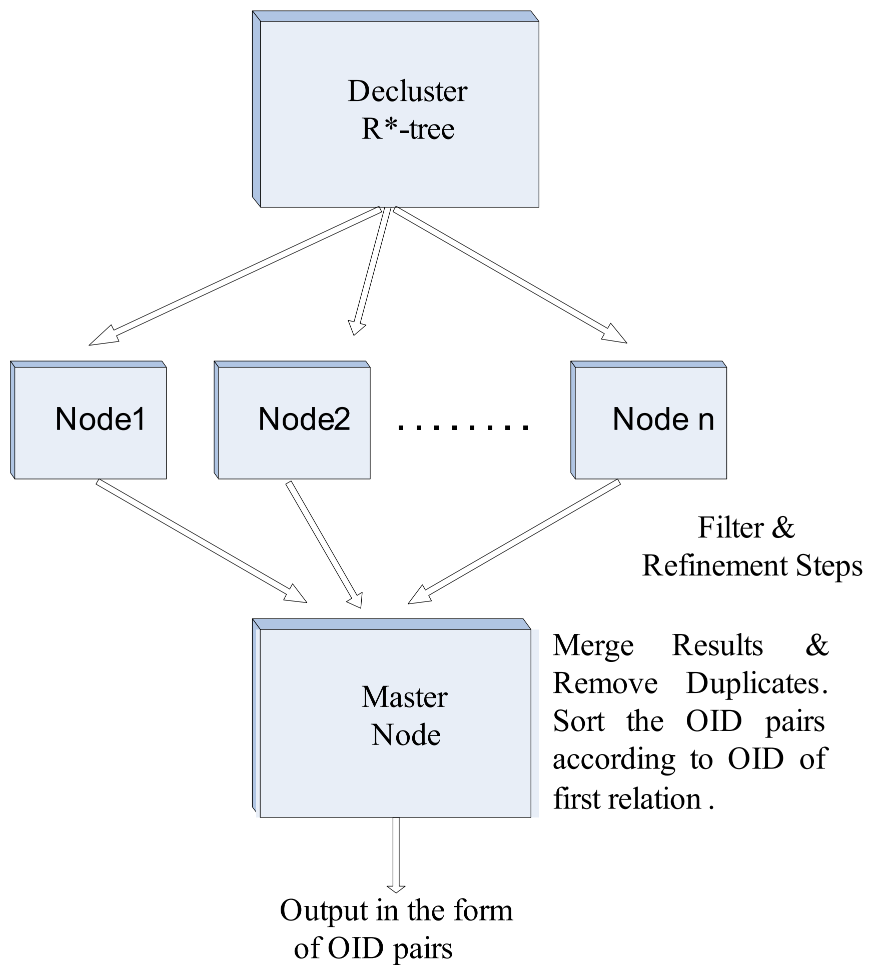

Decluster R*-tree
Node1
Node2
. . . .
Node n
Filter & Refinement Steps
Master Node
Merge Results & Remove Duplicates. Sort the OID pairs according to OID of first relation .
Output in the form of OID pairs

Figure 5.10 Join Algorithm based on Semi-Dynamic Approach

The data is first distributed using the R*-tree based Semi-Dynamic declustering scheme described previously. Then, the join algorithm is applied on each node. The join algorithm performs the filter and Refinement steps on each node. On each node, the result is a set of OID pairs. The resulting pairs from all the nodes are merged by the master node. The duplicates are removed and the OID pairs are sorted to give the overall result. Figure 5.10 shows the sequence of steps involved. The difference between the proposed algorithm and the CJAV2 is that the declustering scheme used is different. This is observed from Figure 5.11.

An extensive experimental comparative analysis of the proposed algorithm with the two variants (CJAV1 & CJAV2) of the clone join algorithm is performed in Chapter 6.

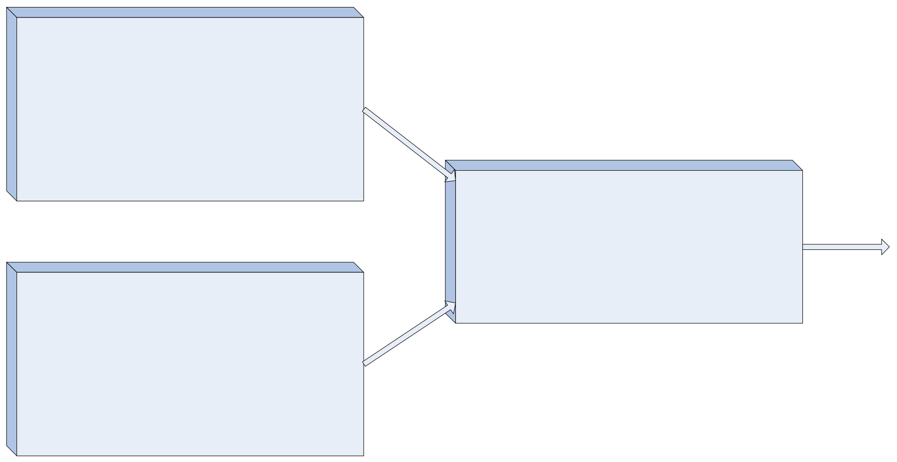


Figure 5.11 Difference between Join Algorithm Based on Semi-Dynamic Approach and CJAV2

5.4 Rendering

Geospatial data exists is many different formats. A Shapefile is a geospatial data format that is widely used. Geospatial data could be contained in a database as well. This section discusses rendering of the data from the above mentioned sources, Shapefiles & PostGIS databases.

The data obtained from the Join algorithms discussed above is in textual format; the output is a set of OID pairs. This data has to be rendered in order to view the result graphically. One way to view the data would be to store the features corresponding to the OID's in PostGIS, and render it using the Geotools [6] library as explained in section 5.4.2 below.

The steps needed to render data using the Geotools library can be summarized as follows:

1. Collect the features to be rendered. The first step would depend on whether the data is stored in a Shapefile or PostGIS.
2. Prepare Styles for the features. This step specifies the form and color data has to be rendered in.
3. Create a layer consisting of the features and the styles that have to be applied to these features.
4. Add the layer to the map and display the map in a graphical user interface.

The section below elaborates on how to render the data using the classes of the Geotools library.

5.4.1 Rendering a Shapefile

The Geotools library is used to render the data from a Shapefile. The features are obtained from the Shapefile itself.

To get the features from the Shapefile, the following has to be done:

- Create a *ShapefileDataStore* instance by passing it the path where the Shapefile exists.
- Create a FeatureSource object from this datastore.

To render the features in some form and color, the styles for the features need to be prepared.

- Prepare Styles for the features to be drawn. Depending on whether a Line or Point has to be drawn, the *LineSymbolizer* or *PointSymbolize*r classes could be used.

Next, a layer consisting of the features and the styles that have to be applied to the features need to be made:

- Build the map, i.e. create a *MapContext* and add a layer consisting of the feature source and style instances, using its addLayer method.

Finally the map is displayed:

- Show the map by creating the *StyledMapPane* and JFrame instances and setting the pane with the correct *MapContext*. Then create a scroll pane for this mapPane in the Frame to enable scrolling of large maps.

5.4.2 Rendering from PostGIS

The data in the database can also be viewed using the Geotools library. The features are obtained from a relation in the PostGIS database. To get the features from the PostGIS database the following has to be done:

- Connect to the PostGIS database by using the *PostgisConnectionFactory* class, and then use the getConnectionPool method to get the ConnectionPool instance.
- Create the *PostgisDataStore* by passing the connection pool instance. Create a FeatureSource object from this datastore instance.

After obtaining the features, styles can be applied to these features and rendered as explained in the Section 5.4.1.

5.4.3 Viewer

There are many tools that can be used on the map: zooming, panning, printing, and adding layers from Shapefiles and PostGIS data sources. Using the Geotools library, an interface was developed in Java which has the capability to perform all the operations mentioned above on a map. This interface is the Viewer which allows viewing of the data and the ability to apply various operations on the displayed map.

The following operations could be performed on the map:

1. Zooming: lets the user to zoom in and zoom out of the map.

   For this task, the user has to click on the map.

   a. Zoom In to Point: zooms in on the view and centers on clicked point.
   b. Zoom Out From Point: zooms out and centers on the clicked point.
2. Panning: moves the view to the left, right, up, and down. These options are added to the viewer.
3. Reset: sets the visible area to the full extent of the layers.
4. Print: Open a print dialog to print the map.
5. Add Layers from Shapefile: The user specifies the Shapefile to be rendered and also the style; this creates a layer and adds it to the GUI.
6. Add layers from PostGIS: The user specifies the table and the database from which the data has to be rendered as well as the style; this creates a layer and adds it to the GUI.

The data obtained from the Join algorithms discussed above is in textual format; the output is a set of OID pairs. The output of the Join algorithm could be viewed using the developed Viewer.

Chapter 6 Results and Discussion of Results

This chapter discusses the experiments conducted and analyses the results obtained. The first section contains the results obtained using the tiling scheme. It discusses the declustering results as well as the results of the two versions of clone join algorithms. The second section contains the results of a R*-tree based semi-dynamic join algorithm. Finally, the two declustering techniques are compared.

6.1 Tiling Scheme

The two collections of datasets mentioned in Chapter 3 were used for testing. Both datasets (dataset1 & dataset2) contain Hydrography and Rail information. Dataset1 is limited to the state of Louisiana, while Dataset2 is much larger and includes the following adjacent states: Louisiana, Kansas, Mississippi, and Arkansas, Texas, Oklahoma, and Missouri states. When these datasets are imported into the PostGIS database, the relations are named as *hydrolin* and *rail.*

6.1.1 Results on Declustering

This scheme as discussed in section 5.1 presents two disadvantages, namely replication and data distribution skew. A spatial object could be stored on more than one node, which results in replication of the same object i.e. data replication. Also, data fragments stored on the various nodes may vary greatly in size, resulting in data distribution skew. A good spatial declustering scheme would be one that assigns nearly an equal number of features to each node and, would consequently, have a *coefficient of variation* close to zero.

The ratio of the standard deviation and the mean is called the *Coefficient of Variation.* This reflects the variability of the data on several nodes when the data is distributed. A good declustering method is expected to have a low *coefficient of variation* (CV) of the features per node as well as a low percentage of replication of the features.

The two data sets were distributed onto various nodes with different test parameters. In the tiling scheme, the universe is divided into tiles and these tiles are distributed across various nodes. For dataset1 the number of tiles is varied from 50 to 5,000. For dataset2 the number of tiles is varied from 50 to 7,200.

Both Figures 6.1 and 6.2 show the coefficient of variation of the number of features per node versus the number of tiles. The x-axis represents the number of tiles, and the y-axis represents the coefficient of variation. In both figures, each curve represents the graph generated for a specified set of compute nodes. Each curve represents the number of nodes the data was distributed across. Nodes-4 means that the result was taken when the number of nodes selected was 4.

When the number of tiles is low, the CV is high reflecting the fact that data on different nodes varies greatly in size. As the number of tiles increases, the distribution becomes nearly uniform. Figure 6.1 and Figure 6.2 shows the CV of features per node for dataset1 and dataset2 respectively. Both Figures use the coefficient of variation of the distribution of the features in each node as a metric. These Figures illustrate the fact that the coefficient of variation of features per node decreases as the number of tiles increases. These observations are in line with results obtained in [3,4] using different datasets.

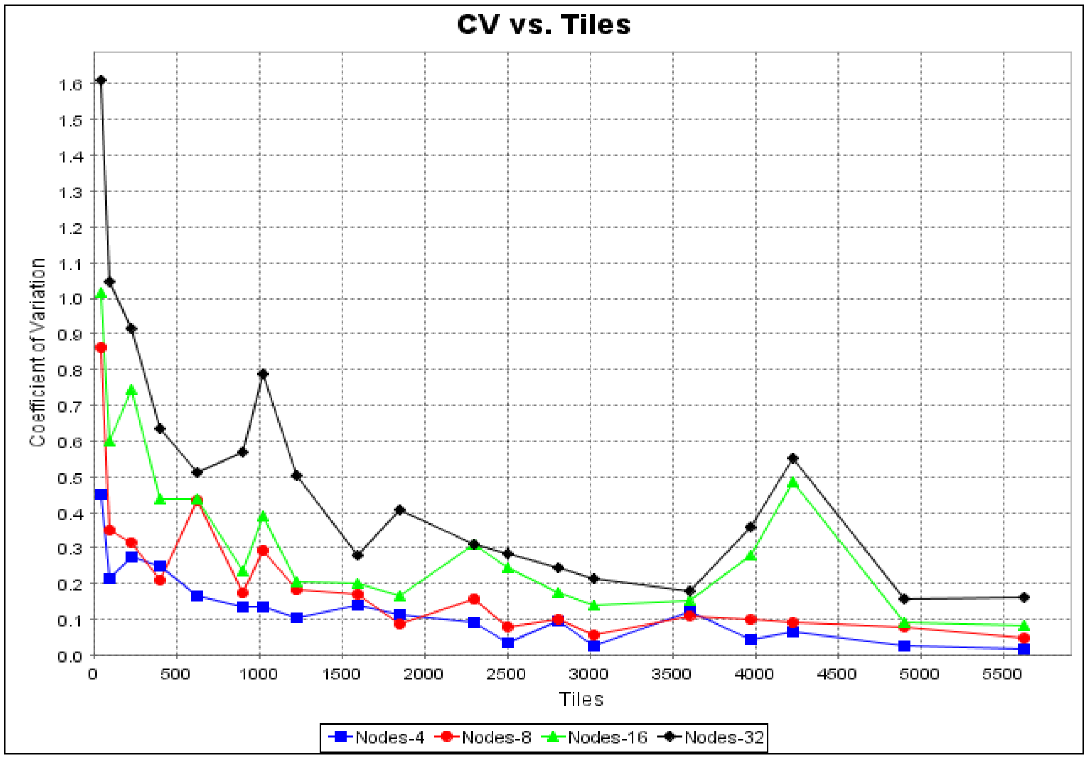


Figure 6.1 Co-efficient of Variation vs. Tiles for dataset1

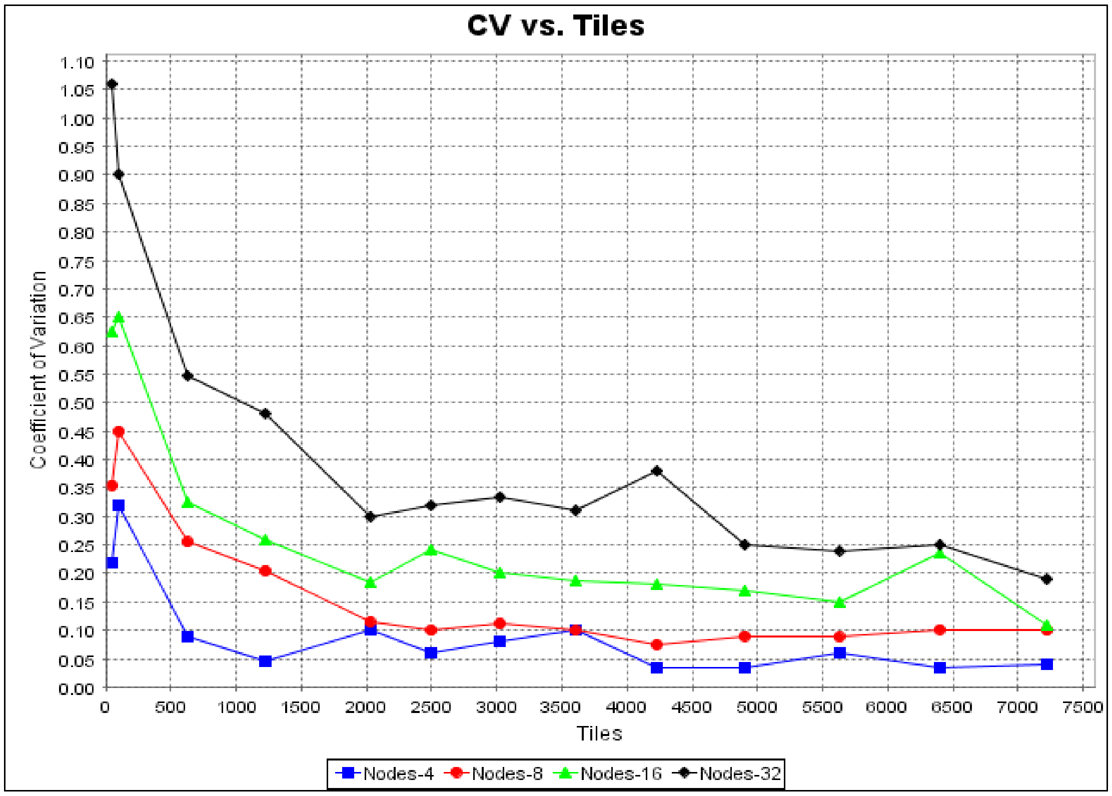


Figure 6.2 Co-efficient of Variation vs. Tiles for dataset2

For a fixed number of tiles, it is observed from Figure 6.1 and 6.2 that the coefficient of variation is less when fewer compute nodes are used. This is because the distribution of tiles that cover dense regions is better with a smaller number of nodes. Also, it is seen that the curves generated are not smooth. The irregularities on the curve were due to the limitations of the round-robin hashing function used. Indeed, when the number of columns was a multiple of the number of nodes, all tiles which have the same column number were stored on the same node; this is equivalent to having less tiles. These irregularities are also observed in [3,4].

Figure 6.3 and Figure 6.4 below show the percentage of replication of the features when the number of tiles increases for dataset1 and datset2 respectively. The x-axis represents the number of tiles, and the y-axis represents the percentage of replication. When the number of tiles increases, spatial features, which overlap many tiles, are replicated on many nodes. So, when the number of tiles increases, the percentage of replication grows. The irregularities in the curves of Figure 6.3 and Figure 6.4 are due to the same reason explained previously for Figures 6.1 and 6.2.

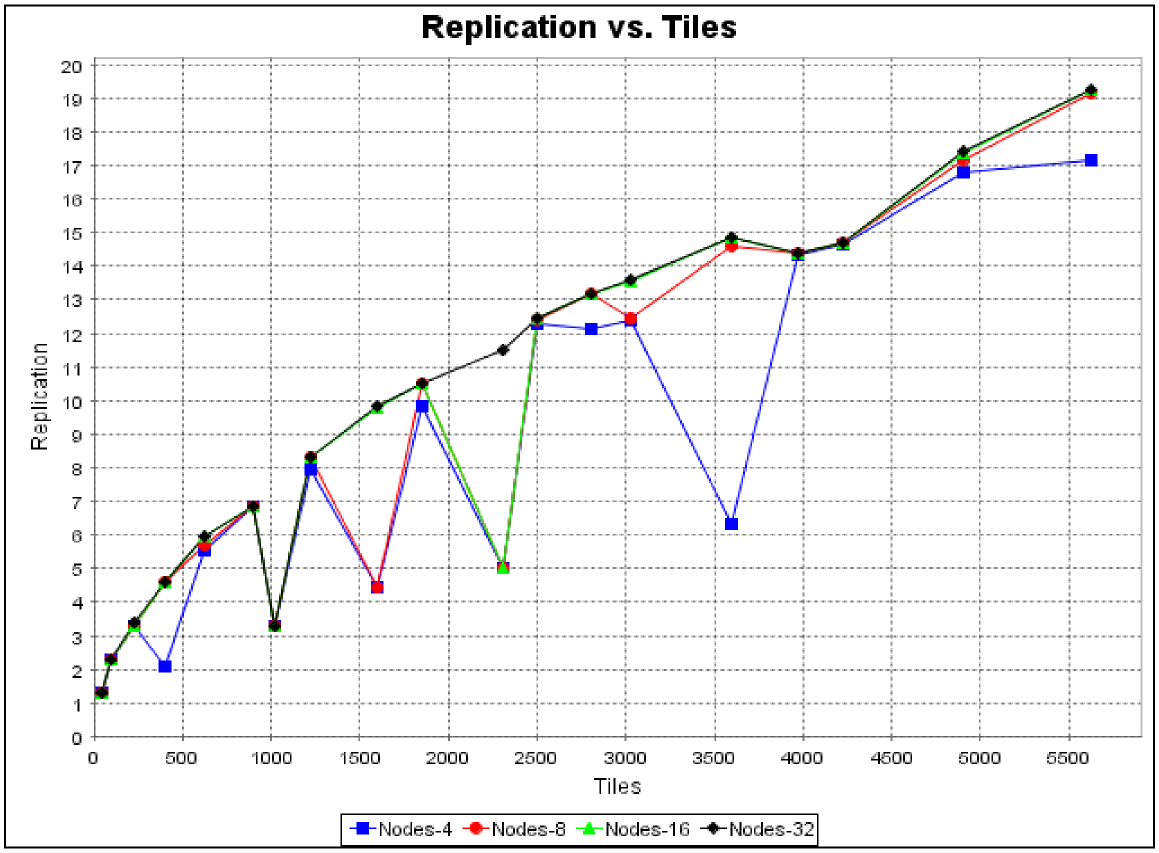


Figure 6.3 Replication vs. Tiles for dataset1

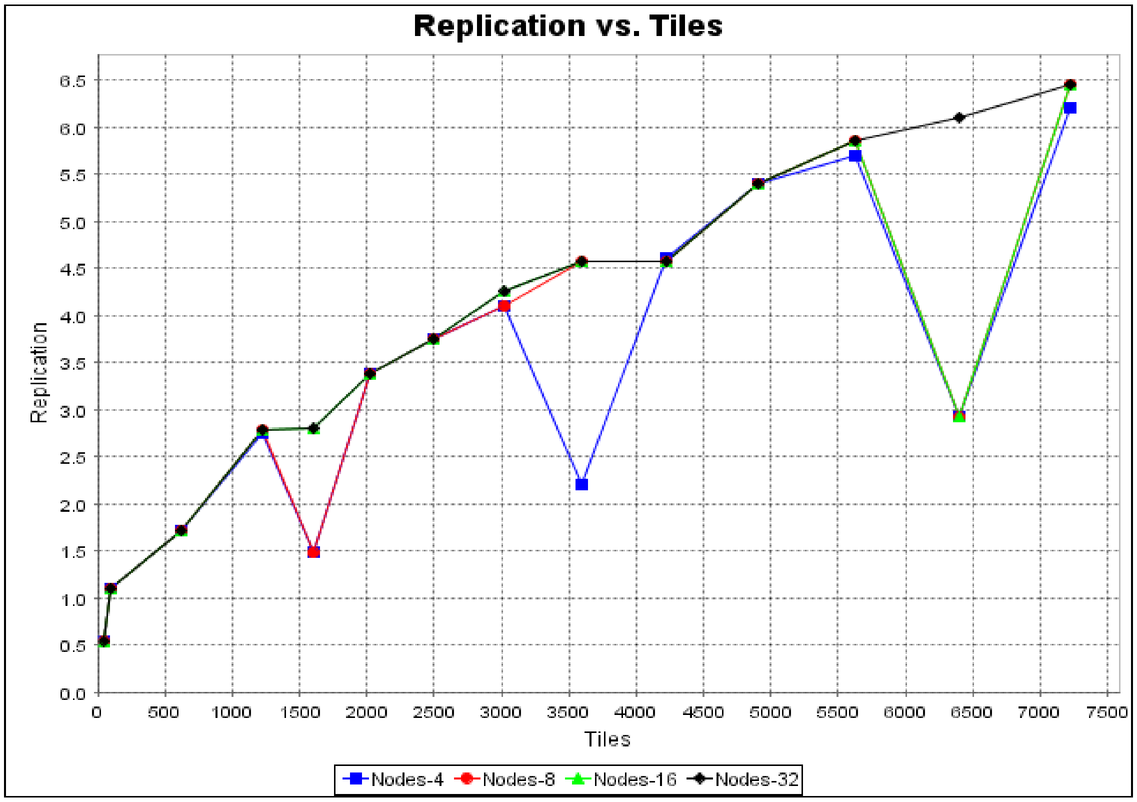


Figure 6.4 Replication vs. Tiles for dataset2

6.1.2 Results of Join Algorithm for Tiling Scheme

The two versions of Clone Join algorithms (CJAV1, CJAV2) introduced previously in section 5.3 are tested. The performance of these algorithms is tested using dataset1 and dataset2. The spatial join is performed on the *Hydrolin* and the *Rail* relations.

6.1.2.1 Variation of the Number of Tiles

The two relations were declustered using the Tiling Technique on 16 nodes with a varying number of tiles. Figure 6.5 shows the result obtained. The execution time of the Join Operation was taken using the CJAV2 algorithm.

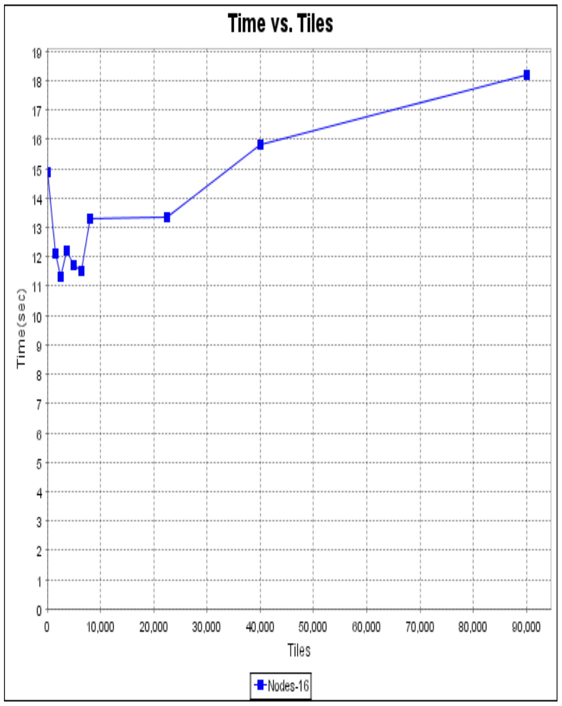


Figure 6.5a Time vs. # of Tiles for dataset1. CJAV2 using 16 nodes

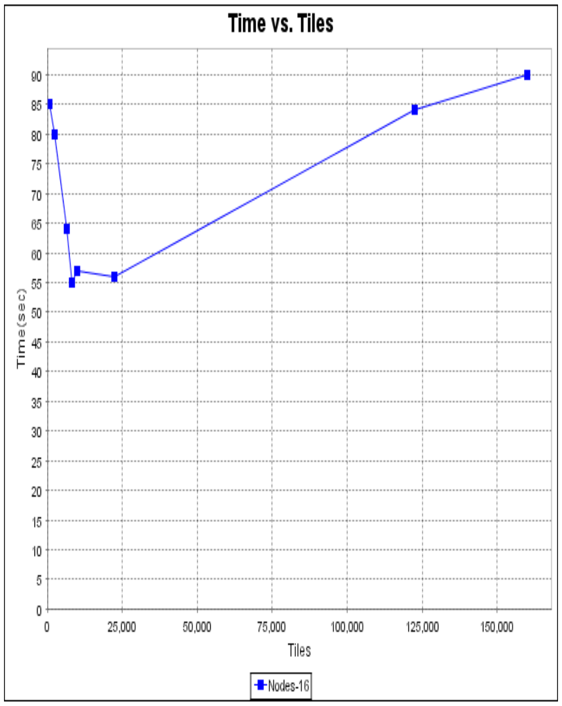


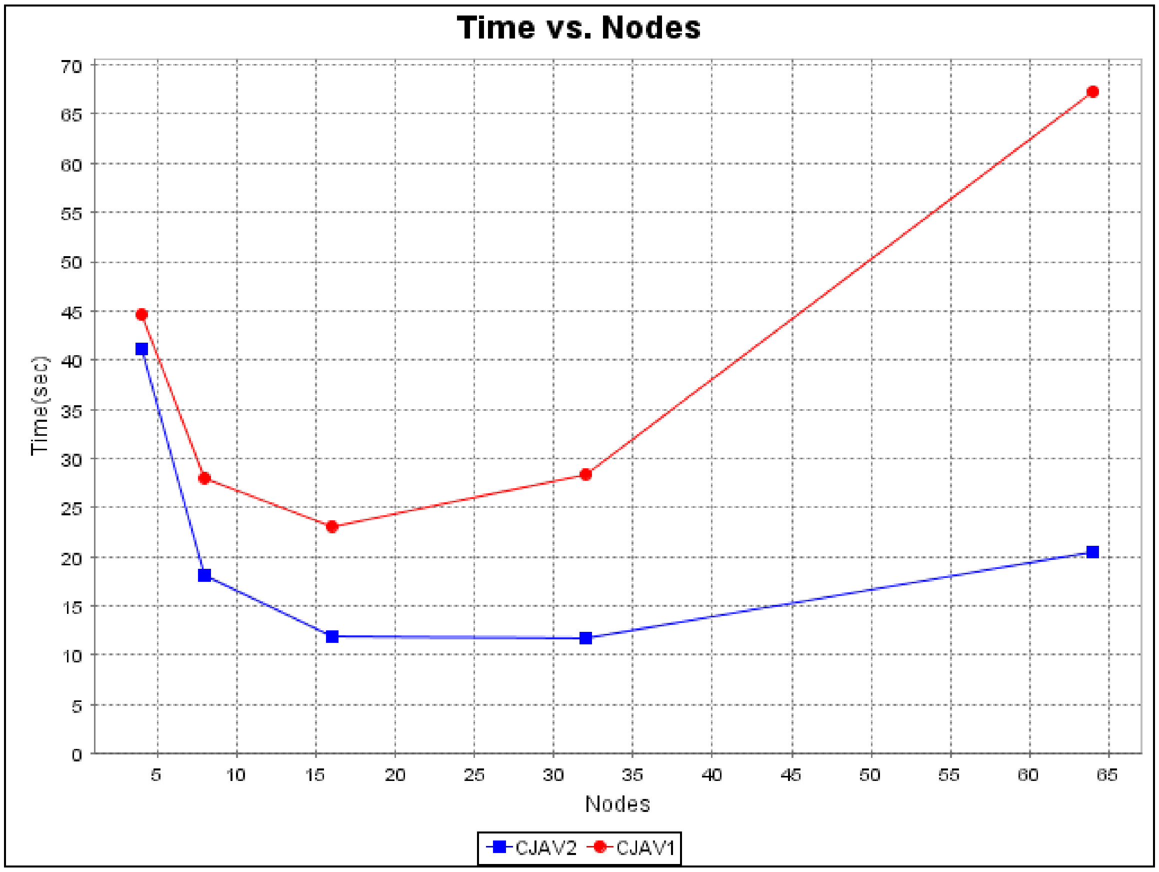
Figure 6.5b Time vs. # of Tiles for dataset2. CJAV2 using 16 nodes.

From Figures 6.5a & 6.5b, it is seen that the time taken for the Join of *Hydrolin* and *Rail* starts decreasing as the number of tiles increase. This is due to a decrease in the coefficient of variation. The time then starts increasing as the number of tiles gets much higher, which is due to an increase in replication. The CJAV1 algorithm exhibits the same behavior.

6.1.2.2 Variation of the Number of Nodes

Both dataset1 and dataset2 were declustered using tiling. The number of tiles was kept constant (3,600 & 8,100 for Dataset1 & Dataset2) respectively. The Join query was tested under five different cluster configurations: 4, 8, 16, 32 and 64 slave nodes for both dataset1 & dataset2. The datasets were also loaded onto one slave node (reference system) and tested.

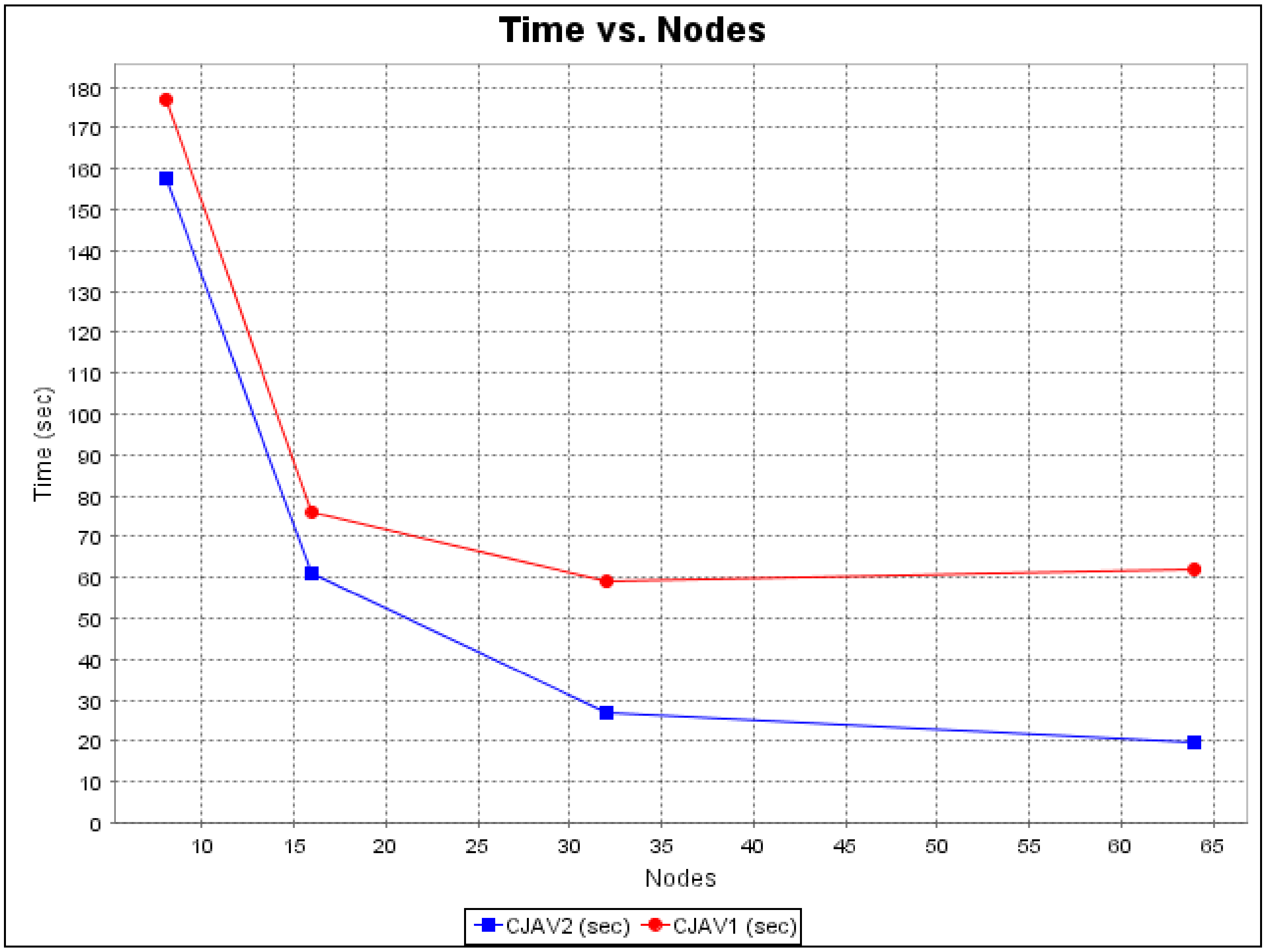
Figures 6.6 & 6.7 summarize the result for dataset1 and dataset2 respectively.



Nodes/Time	CJAV2(sec)	CJAV1(sec)
1	390.3	399.7
4	41.1	44.6
8	18.1	28.0
16	11.8	23.1
32	11.6	28.4
64	20.4	67.3

Figure 6.6 Time(ms) vs. Nodes for dataset1 (3,600 tiles)

From Figure 6.6, it is observed that with Dataset1 CJAV2 outperforms CJAV1. The performance gain becomes more pronounced as the number of compute nodes is increased. The performance of CJAV2/ (CJAV1) shows a gain up to 32/ (16) nodes; beyond this number, a degradation of performance is observed. This is probably due to an increase in communication overhead when more nodes are involved in the querying. That is, network communication overhead caused by a large number of nodes returning query results to the master node mitigates any processing gains achieved by having more processors and disks. However, the use of a greater number of nodes is expected to be beneficial for much larger datasets.



Nodes/Time (sec)	CJAV2	CJAV1
1	6,797.7	6,804.6
4	554.3	567.1
8	158	177
16	56	76
32	26	59
64	20	62

Figure: 6.7 Time (ms) vs. Nodes for dataset2 (8,100 tiles)

Figure 6.7, shows the same experiment with the larger dataset, dataset2. A similar trend is observed except that the performance gain of CJAV2 over CJAV1 is higher with this larger dataset. It is also observed that the scalability of CJAV2 is improved with this larger dataset.

6.1.2.3 Comparison of the Two Clone Join Algorithms

The two join algorithms are compared, as seen in Figure 6.7. The Join query was tested with five different cluster configurations: 1, 8, 16, 32 and 64 slave nodes.

It is noted that for the CJAV2 algorithm, the filter and refinement steps of the spatial join are performed on the nodes without sending the filter result to the master node. In contrast, for the CJAV1 algorithm, the Filter step output is sent back to the master node which removes duplicates, and sends back the resulting output to the nodes for the refinement step. As a result, in the case of algorithm CJAV1, the master node receives OID pairs from each node and then performs a duplicate removal operation. Afterwards, the master node distributes the duplicate free set of OID pairs back to the nodes they originated from. These steps introduce additional communication and processing overhead. This overhead is more pronounced when the replication percentage is low.

However, CJAV1 may be useful with very large datasets that generate a larger number of duplicates. In this case, it may be beneficial to remove the duplicate OID pairs before proceeding with the refinement step.

6.2 R*-tree based Semi-Dynamic Approach

6.2.1 Declustering Results of a single relation

With R*-tree based declustering, there is no duplication of the features as the leaves of the R*-tree do not overlap. Using this approach the CV of features per node is reduced compared to the tiling technique. An R*-tree with varying *fan out* (M) is built on the *hydrolin* relation (for both dataset1 & dataset2).

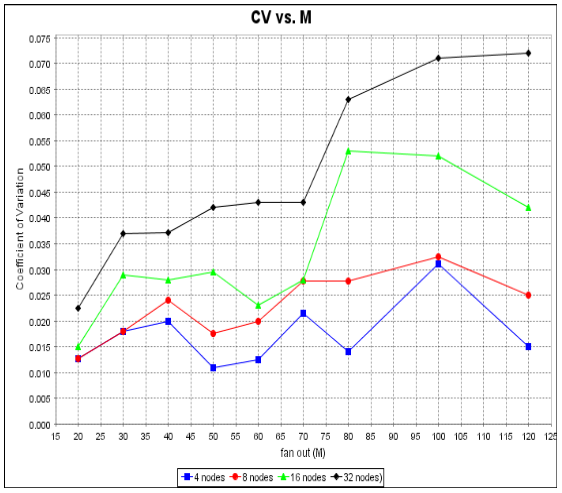


Figure 6.8 CV vs. M (R*-tree on hydrolin) using dataset1

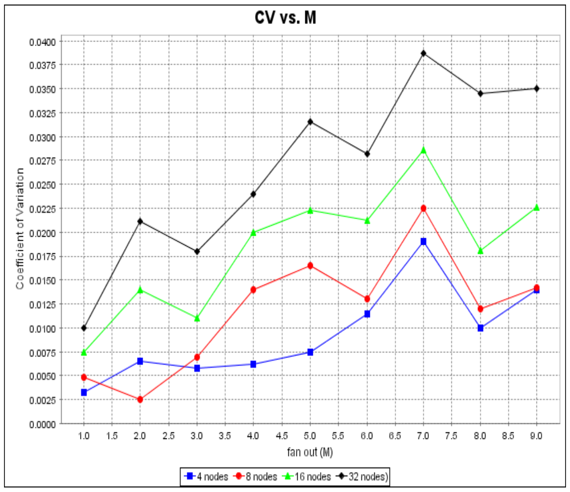


Figure 6.9 CV vs. M (R*-tree on hydrolin) using dataset2.

It is observed from Figures 6.8 & 6.9, that the coefficient of variation is lower for the R*-tree based semi-dynamic declustering. This is verified by comparing the above Figures with Figures 6.1 & 6.2. For dataset1, CV ranges from 0.01 to 0.07 which indicates that the data distribution is nearly even among the nodes. The same is observed when dataset2 is used to build an R*-tree on *hydrolin*.

It is noted that a similar trend (lower CV) is also observed when an R*-tree is built on the *rail* relation.

6.2.2 Declustering Results of two relations for a Join

The declustering of two relations was performed for different parameter values to choose the best fan out to perform the join.

To perform the Join using the R*-tree based semi-dynamic declustering (discussed in chapter 5), the R*-tree structure of one of the relations is used to decluster both the relations, such that there will be no replication on one of the relations. However, the other relation is subject to replication and data distribution skew. This occurs because this relation is declustered statically using a tiling like scheme. The leaves of the first relation's R*-tree are treated as tiles for the second relation to be declustered.

The join is performed on relations *Hydrolin* and *rail*. As discussed before, the main parameter used to build an R*-tree is the fan out, M. Best parameter values are found such that the second relation declustered has minimum replication of data as well as a nearly even distribution of data.

The R*-tree was built on the *hydrolin*, the larger of the two relations. This R*-tree structure was used to distribute the *rail* dataset. This approach was tested with various values for the R*-tree's fanout to find the best distribution. The same process is repeated for the R*-tree on the *rail,* the smaller of the two relations. The two mapping functions used to map the leaves to disks are round robin and range hash, as described previously in section 5. For this purpose, each of the four variants was tested. It is observed from Figure 6.10 that building an R*-tree on the smaller relation, *rail,* reduces replication of data. In addition, the declustering time is reduced by building an R*-tree on the smaller relation.

Based on the above discussion, the semi-dynamic join approach will be based on an R*-tree built on the *rail* relation. Two mapping functions (round robin and range mapping) were used to map features to compute nodes. The results of this algorithm are compared with CJAV2, as this algorithm performs better than CJAV1 for both dataset1 and dataset2.

The comparative analysis for both datasets is discussed below. CV and Replication values are noted for different parameters of the *hydrolin* relation. The best parameter is selected from these results and tested for the join query.

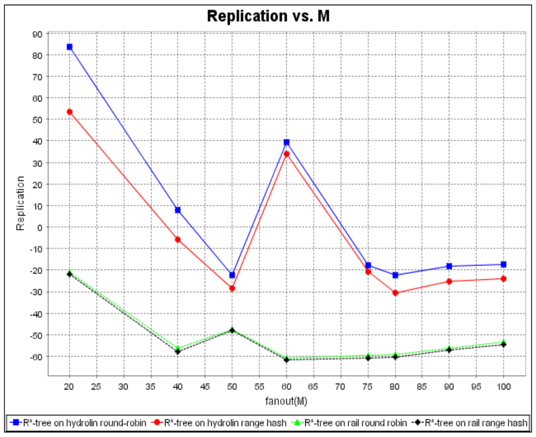
Results with dataset1

From Figure 6.10, it is observed that the Replication is initially high for lower values of M, but exhibits a sharp decrease as M increases. The reason for this is explained below.

In this case, hydrolin is declustered statically using a tiling like scheme. The same declustering pattern of the Tiling scheme is observed for this relation. The different tiles are leaves of the R*-tree. For smaller fanout (M) values, the universe is divided into a larger number of leaves. As M increases, the number of leaves decreases. Hence, replication is high for lower M and decreases as M is increased.

From Figure 6.11, it is observed that the CV for the range hash function is greater than the round-robin. This occurs because some part of the map may have considerably less features, and hence data distribution skew will be more prevalent for the range hash function.

M=40 & M=90 are selected for testing the Join query as the Replication and CV are both low for these two M values.



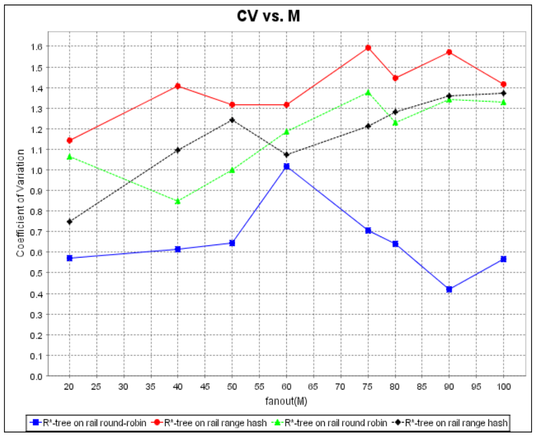


Figure 6.10 Replication vs. M (16 nodes) for dataset1

Figure 6.11 CV vs. M (16 nodes) for dataset1

<u>Results with dataset2:</u>

In Figures 6.12 & 6.13 the fanout, M ranges from [20 to 300]. It is observed from figures 6.12a and 6.12b that the Replication is high for smaller M, and reduces as M increases. The reason for this is the same as explained in the case of dataset1.

Also from Figures 6.12a and 6.12b, the behavior of the mapping functions is noted. Replication for round-robin starts off high. As M increases, the replication is almost the same for both mapping functions. For the range hash, a range of leaves is stored on a single node, which does not store the duplicates. For the round-robin hash function, since each leaf is distributed to a node in a round robin fashion, the features that overlap more than one leaf are stored in multiple nodes. Hence, initially the replication for round-robin is high. However as M increases, replication decreases. This occurs because for larger M, the number of leaves is reduced. Thus, replication is nearly same for both hash functions.

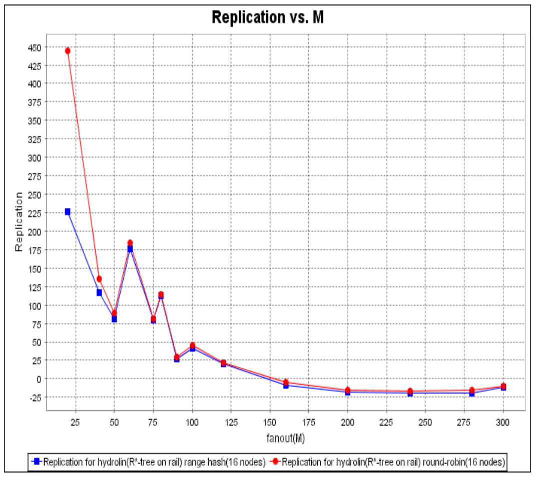


Figure 6.12a Replication vs. M for hydrolin (R*-tree on rail, 16 nodes)

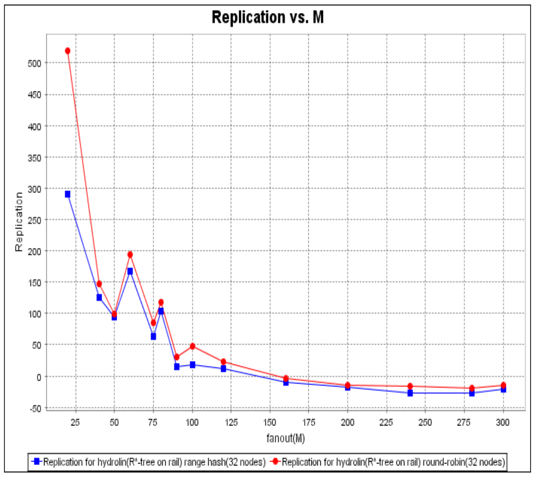


Figure 6.12b Replication vs. M for hydrolin (R*-tree on rail, 32 nodes)

Figure 6.12 Replication vs. M for R*-tree Based Semi-Dynamic Approach

The results from Figures 6.13a, 6.13b demonstrate that CV of range-hash is high, illustrating the fact that this hash function is more subject to data skew. This is due to the same reason as explained previously for dataset1.

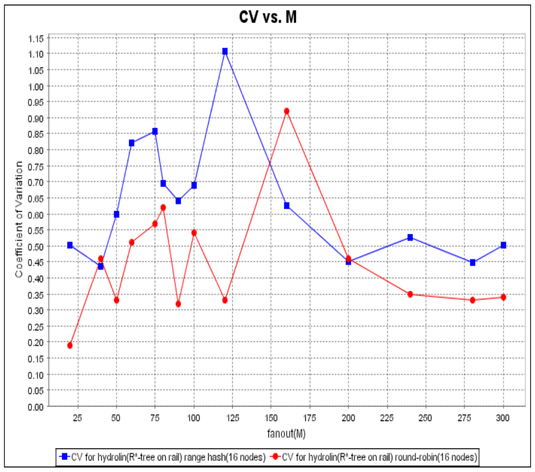


Figure 6.13a CV vs. M for hydrolin (R*-tree on rail, 16 nodes)

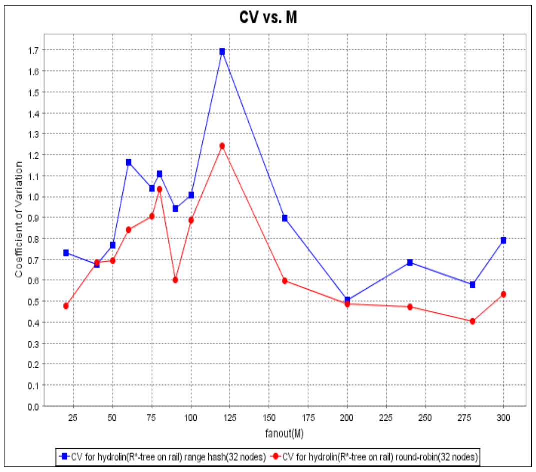


Figure 6.13b CV vs. M for hydrolin (R*-tree on rail, 32 nodes)

Figure 6.13 Coefficient of Variation vs. M for R*-tree Based Semi-Dynamic Approach

It is observed from the above Figures that for fanout, **M=280**, both CV and replication are low for both the variants. The join query is performed using this value.

6.3 Comparison of the Declustering Techniques

For both the declustering techniques used, the declustering time as well as the time for the join query is noted. The declustering time results are shown below.

6.3.1 Comparison of the Declustering Times

Time/Variants	**R*-tree based Semi-Dynamic approach (R*-tree on *rail*, round-robin)**		**R*-tree based Semi-Dynamic approach (R*-tree on *rail*, range-hash)**		**Tiling Technique**
	M=40	**M=90**	**M=40**	**M=90**	**Tiles=3,600**
Time(sec)	116.7	128.7	151.4	152.3	816.8

Table 6.1 Decluster Times of all Variants Using dataset1

Time/Variants	**R*-tree based Semi-Dynamic approach (R*-tree on *rail*, round-robin) M=280**	**R*-tree based Semi-Dynamic approach (R*-tree on *rail*, range-hash) M=280**	**Tiling Technique Tiles=8,100**
Time(sec)	760.6	723.8	5,647.0

Table 6.2 Decluster Times of all Variants Using dataset2

From the above two Tables 6.1 & 6.2, it is observed that the tiling declustering technique takes considerably more time than the proposed R*-tree based semi-dynamic clustering techniques. The performance advantage of the proposed clustering technique becomes more pronounced as the size of the data set is increased.

It should be noted that in the experiment, both algorithms (Tiling and the R*-tree-based semi-dynamic declustering method) have been run under their most favorable parameter values. Indeed, as shown previously, the tiling method performs best when 3600 (8100) tiles are used with dataset1 (dataset2). Similarly, the R*-tree based semi-dynamic approach performs best when *rail* is declustered using an R*-tree with a fan out of 40 (280) for Dataset1 (Dataset2).

6.3.2 Comparison of the Join Results

The comparative analysis below only shows the execution time of each of the two algorithms. It does not include the time to decluster the two relations (*rail* and *hydrolin*) to be spatially joined. The comparison of the declustering time of each of the two methods was performed in the previous section.

<u>Results using dataset1</u>

The spatial join query was tested under the following cluster configurations: 1, 4, 8, 16, 32 and 64 slave nodes for each of the two datasets. As previously done, both algorithms were run under their most favorable parameters values.

Nodes/ Variants	**R*-tree on *rail*, round-robin M=40**	**R*-tree on *rail*, range hash M=40**	**CJAV2 3,600 tiles**
1	99.1	99.1	390.3
4	19.2	17.7	41.1
8	10.9	12.4	18.1
16	11.7	11.8	11.8
32	11.6	13.6	11.6
64	19.7	19.7	20.4

Table 6.3 Comparison of 2 Variants of R*-tree Based Semi-Dynamic Approach with CJAV2 using dataset1

It is noted from Table 6.3 that the R*-tree based semi-dynamic approach outperforms CJAV2 for all cluster configurations. There is a tangible performance gain achieved as the number of compute nodes increase from 1 to 8. For a higher number of nodes, the gain in time is more modest. This is attributed mostly to the small size of the dataset. It is noted that for both algorithms the execution time using 64 nodes is more than that of 32; this is due to the communication overhead as explained previously for the clone join versions.

Results for Dataset2

Nodes/ Variant	**R*-tree on *rail*, round-robin M=280**	**R*-tree on *rail*, range hash M=280**	**CJAV2 8,100 tiles**
1	4,844.4	4,844.4	6,797.7
4	403.8	409.2	554.3
8	134.7	133.1	158.3
16	46.5	50.0	55.7
32	21.7	25.2	25.8
64	18.6	23.1	19.6

Table 6.4 Comparison of 2 Variants of R*-tree Based Semi-Dynamic Approach and CJAV2 using dataset2

Table 6.4 tabulates the result for dataset2. A similar trend is also observed with this larger dataset. That is, the proposed method always outperforms CJAV2, but as the number of nodes increases, the performance gap tends to decrease. It is also observed that round-robin performs better than the range hash function in almost all cases. Higher CV is the most probable reason for this, as replication for a fanout of 280 is almost the same for both hash functions.

The above extensive comparative experimental analysis confirms that the R*-tree based semi-dynamic approach is more efficient than the clone join algorithm (CJAV2). Its performance is superior both in declustering time as well as in the execution time of the spatial join query.

Chapter 7 Conclusion

In this thesis, a new declustering strategy using a semi-dynamic approach was proposed. The proposed scheme builds an R*-tree on one of the two relations to be joined. The leaves of this R*-tree are mapped onto compute nodes. The features of the second relation are distributed statically using a tiling like approach. The leaves of the R*-tree built on the first relation are treated as tiles, and thus, the features of the second relation which overlap the leaves are stored on the nodes corresponding to the leaves. Based on this declustering strategy, a new R*-tree based semi-dynamic parallel join algorithm and two versions of the existing clone join algorithm were investigated. A comparative performance analysis of these algorithms was done using real datasets. These algorithms were implemented and run on a Beowulf cluster. The experimental results show that the proposed algorithm outperforms the clone join algorithm. Its performance is superior both in declustering time as well as in the execution time of the spatial Join query.

Future work includes the design and implementation of *adaptive* parallel algorithms for both single scan and multiple scan spatial queries. These algorithms are expected to take into account the size of the datasets, the distribution of features, the main memory size of each slave node and other parameters (to be determined) to determine the best execution strategy. These algorithms are expected to form the basis of an *adaptive parallel query processor*.

References

1. PostGIS Manual. <http://postgis.refractions.net/documentation/>
2. PostgreSQL Online Documentation <http://www.postgresql.org/docs/>
3. Jignesh M. Patel., and David J. Dewitt. "Partial Based Spatial-Merge Join." Proceedings of the ACM Sigmoid 1996, June 1996. 259-270
4. Jignesh M. Patel., and David J. Dewitt. "Clone Join and Shadow Join: Two Parallel Spatial Join Algorithms", Proceedings of the 8th ACM International Symposium on Advances in GIS, ACM Press, 2000. 54-61.
5. Beckmann, N., Kreigel, and R., Seeger, B. "The R*-tree: An Efficient and Robust Access method for Points and rectangles." Proceedings of the ACM SIGMOD International Conference on Management of Data, 1990 322-331.
6. Geotools Library and Documentation. <http://www.geotools.org/Tutorials>
7. Ridge, Daniel, et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." Proceedings IEEE Aerospace, 1997. Goddard Space Flight Center, CACR Caltech.
8. Abdelguerfi M., and Wong K.F. Parallel Database techniques. Ed. manuscript, IEEE Computer Society Press, 1998. ISBN 0-8186-8398-8.
9. Jack J. Dongara, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: Past, Present, and Future. 2001.
10. Gilbert, David, and Thomas Morgner. JFreeChart Java Library. <http://www.jfree.org/jfreechart/index.php>
11. Guting, Ralf Harmut. Introduction to Spatial Database Systems <http://www.dpi.inpe.br/geopro/referencias/guting_spatialdbms.pdf>
12. Shekar, Sahshi, and Sanjay Chawla. Spatial Databases – A Tour. Prentice Hall, 2003. ISBN 013-017480-7
13. DeWitt, David J., and Jim Gray. "Parallel Database Systems: The future of High performance database processing." Appeared in Communications of the ACM, Jan 1992. Vol. 36, No. 6, 85-98,
14. DeWitt, David J., et al. "Practical Skew Handling in Parallel Joins." Proceedings of the 18th International Conference on Very Large Data Bases, August 1992. 27-40,
15. Tan, K L., and J. X. Yu. "A Performance Study of Declustering Strategies for Parallel Spatial Databases." Proceedings of the 6th International Conference on Database and Expert Systems Applications, London, United Kingdom, September 1995. 157-166.
16. Christos Faloutsos, Pravin Bhagwat. "Declustering using fractals." Proceedings of the second international conference on Parallel and distributed information systems, January 1993, San Diego, California, United States. 18-25.
17. Thomas Brinkhoff, Hans-Peter Kriegal, and Bernhard Seeger. "Parallel Processing of Spatial Joins Using R-trees." Proceeding of the 12th international Conference of Data Engineering. Washington – Brussels – Tokyo, Feb 1996 IEEE Computer Society. 258-265.

18. Nikos Koudas, Christos Faloutsos, and Ibrahim Kamel. “Declustering spatial databases on a multi-computer architecture.” Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology, 1996. London, UK. 592-614.
19. Kamel I., and C. Faloutsos. “On packing rtrees.” Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM), November 1993. 490-499
20. X. Zhou, D.J. Abel and D. Truffet. “Data partitioning for Parallel Spatial Join Processing.” Proceeding of the 5th Intl. Symp. On Large Spatial Databases, Berlin, Germany, July 1997. 178-196,
21. OpenGIS Consortium Inc., “OpenGIS Simple Features Specification for SQL”, OpenGIS Consortium Inc., 1999. <http://www.opengeospatial.org>
22. Bureau of Transportation Statistics (BTS), “2002 National Transportation Atlas Data Shapefile Download Center Selected States” <http://www.bts.gov/programs/geographic_information_services/download_sites/ntad02/statedownloadform.html>
23. Lo M L., and C. V. Ravishankar. “Spatial Joins Using Seeded Trees.” Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, May 1994. 209-220.
24. Lo M. L., and C. V. Ravishankar. “Generating Seeded Trees From Data Sets”. Proceedings of the Fourth International Symposium on Advances in Spatial Databases, August 1995. 328-347
25. J. A. Orenstein. “Spatial Query Processing in an Object-Oriented Database System”. Proceedings of the 1986 ACM SIGMOD Conference on Management of Data, 1986. 326-336.
26. Guttman, Antonin. “R-trees: A Dynamic index structure for Spatial Searching.” Proceedings of the ACM SIGMOID International Conference on Management of Data, 1984. 47-57

Vita

Gayatri Ganpaa was born in the city of Hyderabad in India, on May 15, 1982. She got her Bachelors of Technology in Computer Science & Information Technology from Jawaharlal Nehru Technological University in May, 2003. She joined the Masters of Sciences in Computer Science at the University of New Orleans in 2004. During this time, she worked as a Teaching Assistant of the Computer Science Department.