DECLUSTERING SPATIAL DATA FOR RANGE AND JOIN QUERIES

by

W. Brett Kennedy

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 1999 by W. Brett Kennedy
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-45949-7
Abstract
Declustering Spatial Data for Range and Join Queries

W. Brett Kennedy
Master of Science
Graduate Department of Computer Science
University of Toronto
1999

The efficiency of performing database queries using multiple computers is dependent on the declustering of the data. Techniques are presented to decluster spatial data to optimize the response times of range and join queries, where the join queries use an intersection predicate. Experiments were performed using rectangular data, from both real and synthetic datasets, using between 1 and 12 nodes in a shared-nothing environment. Range queries were performed with the Filter Tree and joins with $S^3J$. Experiments on single nodes revealed that the execution times of range and join queries can be estimated accurately by considering the input sizes and predicting the output sizes. This can be done accurately with multi-dimensional histograms. Formulas are also presented to predict the times required for communication and other steps necessary for multi-node queries. Further experiments, then, showed that, using this knowledge, and the declustering techniques developed here, it is possible to decluster data such that nodes are assigned the min-max workload. Doing this allows us to realize good response times for multi-node queries using the Filter Tree and $S^3J$. Finally, we demonstrate that our formulas are accurate enough that we are able to determine the best declustering method for large sets of combined range and join queries.
Acknowledgements

I would like to thank my supervisor, Kenneth Sevcik, for his valuable guidance and feedback during my work, and for making my thesis work to be so rewarding. I also thank my second reader, Michiel van de Panne, for his valuable comments. I would also like to thank Nikos Koudas for giving me much support, and many ideas when I was beginning work on this thesis. As well, I would like to thank Paul Lu and Benjamin Gamsa for giving me much support in using MPI and the POW cluster.
# Contents

## 1 Introduction
- 1.1 Overview of the Filter Tree and $S^3J$ .................................................. 3
  - 1.1.1 Range Queries ........................................................................ 6
  - 1.1.2 Join Queries ......................................................................... 7
- 1.2 Data Skew ...................................................................................... 8
- 1.3 Thesis Organization ....................................................................... 9

## 2 Related Work
- 2.1 Indices Used for Spatial Data .......................................................... 10
- 2.2 Range Queries on Spatial Data Using Multiple Nodes ...................... 11
- 2.3 Join Queries on Spatial Data ............................................................ 12
- 2.4 Declustering Data to Multiple Nodes ............................................... 13
  - 2.4.1 Dividing the Data Space Into Fragments .................................. 13
  - 2.4.2 Distributing Fragments to the Remote Nodes ......................... 14
- 2.5 Estimating Selectivity .................................................................... 19
  - 2.5.1 Fractals .................................................................................. 19
  - 2.5.2 Histograms ............................................................................ 20
  - 2.5.3 Analytic Models ...................................................................... 21

## 3 Problem Statement
- 3.1 Replication of Data During Declustering ....................................... 22
  - 3.1.1 Effects of Replication ............................................................... 24
  - 3.1.2 Algorithm to Avoid Replication .............................................. 24
- 3.2 Declustering for Range Queries ..................................................... 25
- 3.3 Declustering for Joins .................................................................... 26
- 3.4 Declustering for Range Queries Followed by Joins ......................... 27
- 3.5 Dividing the Data Space Into Fragments ....................................... 28
  - 3.5.1 Slicing .................................................................................... 28
  - 3.5.2 Brute Force Division ............................................................... 29
  - 3.5.3 Division Based on Skew ........................................................... 30
  - 3.5.4 Variable Fragment-Size Division ............................................. 31
  - 3.5.5 Comparison of Methods to Divide Data into Fragments .......... 32
  - 3.5.6 Comparison of Methods to Distribute Fragments .................... 33
4 Experimental Setup

4.1 Synthetic Data Used

4.2 Real Data Used

4.3 Query Windows Used

4.4 Summary of Experiments

4.5 Experiment 1. Time Requirements of I/O Operations

4.6 Experiment 2. Send and Receive Times Using MPI

4.7 Experiment 3. MPI Times for Different Numbers of Nodes

4.8 Implementation Details

4.8.1 Message Passing

4.8.2 Estimating Selectivities

5 Experimental Results For Range Queries

5.1 Experiment 4. Varying the Size of the Dataset

5.2 Experiment 5. Varying the Query Window Size

5.3 Experiment 6. Large Datasets

5.4 Experiment 7. Estimating Output Size on a Single Node

5.5 Experiment 8. Performing Range Queries Using Multiple Nodes

5.6 Experiment 9. Varying the Number of Nodes and Declustering Level

5.7 Experiment 10. Comparing Declustering With and Without Replication

5.8 Experiment 11. Accuracy of Estimating the Total Time

6 Experimental Results For Joins

6.1 Experiment 12. Time Required By Individual Steps of the Join Operation

6.2 Experiment 13. The Effect of Dataset Size

6.3 Experiment 14. Time Required Where Datasets Have Different Sizes

6.4 Experiment 15. Estimating the Output Size For Joins

6.5 Experiment 16. Estimating Join Times

6.6 Experiment 17. Performing Join Queries Using Multiple Nodes

6.7 Experiment 18. Time Required to Decluster Data

6.8 Experiment 19. Time Required to Return Result Sets

6.9 Experiment 20. Determining the Best Declustering Level

6.10 Experiment 21. The Relationship Between Data Skew and the Best Declustering Level

6.11 Experiment 22. Efficiency of Division Based on Skew

7 Experimental Results For Combined Range and Join Queries

7.1 Experiment 23. Comparing Declustering Complete and Partial Datasets for Join Queries

7.2 Experiment 24. Predicting Response Times for Multi-node Range Queries When Using the Join-based Strategy

7.3 Experiment 25. Accuracy of the Brute Force Division Method For Sets of Range and Join Queries
Chapter 1

Introduction

There are now many systems maintaining large quantities of spatial data, including those working with scientific, remote sensing, GIS, cartography and census data. Due to the widespread use of spatial data, and the large volumes of data commonly stored by these systems, efficient access to this information is becoming increasingly important.

Spatial data is data that represents the location of objects in some physical space. For example, a geographic database may store the location of buildings, rivers, roads or other features in some specific area. The data may be stored as points, lines, rectangles, polygons or more complicated forms. Often series of line segments are appropriate to represent the location of roads, rivers, railway lines, hydro lines and so on. Rectangles may be appropriate to represent the location of buildings, parking lots or lakes. A spatial database will, then, consist of a set of rectangles, points, lines, and possibly other shapes that represent the location of a set of objects. As with non-spatial datasets, each dataset is composed of a set of rows. But with spatial data sets, each row represents one object. Each object is stored as a set of coordinates in the n-dimensional space in which it exists. For example, the rectangular objects used in this thesis are specified by the coordinates of two opposite corners. Another technique to store rectangular data is to store the coordinates of the centre, and the extent of the rectangle in each dimension.

Spatial queries are queries that have to do with the location of objects in one or more datasets. One commonly performed query is the range query, also known as the intersection query. A shape, known as a query window, which occupies some area of the data space, is given, and the database is checked for every object that intersects the query window. Another common query is the join query. This involves two datasets and performs some comparison between every pair of objects in the two sets. Typically an intersection predicate is used, so the result is the set of all pairs of objects, such that one object is from each set, and the two objects intersect. An example of a join may be where there are two datasets, representing the locations of buildings and forested areas in the Greater Toronto Area. A join on the two sets would find all buildings that are within forested areas.

Strategies to improve the efficiency of spatial queries include developing index structures appropriate for spatial data and spatial queries, and developing efficient means to parallelize access [MS98]. With respect to developing different index structures, using the storage and index techniques that are used for non-spatial data is adequate to save the
information, but is not generally adequate to efficiently perform the types of queries we may wish to perform on spatial data. To facilitate range queries, joins, and other spatial queries, numerous index structures have been developed including the R-Tree [Gut84], \( R^+ \) Tree [SRF87], \( R^* \) Tree [BKSS90], k-d-b Tree [Rob81], Hilbert R-Tree [KF94], PMR Quad Tree [NS86], Bang File [Fre87], hB-Tree [LS90], buddy-tree [SK90] and the Filter Tree [SK96]. This thesis will work with Filter Trees, and \( S^3J \) [KS97], an algorithm that is similar to the algorithm used for the Filter Tree, but which works on datasets that are not indexed.

Often sequential query processing does not adequately meet the real-time requirements of applications such as interactive visualization systems or other systems using spatial data [SRK+98]. This motivates the parallelization of database query processing. As databases have become larger and hardware has become less expensive, interest in parallel algorithms for processing data has increased [MS98].

The Filter Tree and \( S^3J \) have been shown to perform well on single nodes, but work has not yet been done to test their performance on multiple nodes. Some authors have, though, examined methods to perform multi-node queries with other index structures, for example Koudas et al. with the R-Tree [KFK96].

Much of the work with multi-node queries has concentrated on developing effective ways to decluster the data, that is, developing ways to give each node a portion of the dataset, such that each object is given to at least one node, and such that the nodes can collectively calculate the correct result set for any given query without exchanging data. For example, a dataset may be declustered by randomly assigning each object to one of, say, ten nodes. To perform a range query, then, the query window may be sent to each of these ten nodes. If each of the ten nodes reports the objects that it was sent that are within the query window, then the combination of these ten result sets will be the correct result set for the query. However, randomly assigning the objects to the ten nodes would likely result in the query being performed in an inefficient manner.

This thesis will deal with the problem of using multiple nodes effectively to reduce the execution times of queries on spatial datasets. For this thesis, we assume that a spatial dataset is initially located on a single node that is connected to several other nodes through some communications network. Each node has a copy of the Filter Tree and \( S^3J \) executables and can execute these on some subset of the original dataset. Performing range queries, join queries or other queries on a dataset involves, then, declustering the dataset across the available nodes, performing the operations at each node, and reconciling the results into a single result set. No modifications were made for this thesis to the range query algorithm for Filter Trees or the \( S^3J \) algorithm other than the extensions necessary to decluster the data and reconcile the results. The efficiency of performing the queries using multiple nodes as opposed to a single node is therefore dependent on the ability to decluster the data in an effective way.

How data is declustered depends on the types of queries that will be performed on the data. If data is being declustered for a single operation, then it is important that the act of declustering is quick and not a significant part of the time to perform the query. It also may be that the data is being declustered for a large number of operations. In the case of join queries, this is not realistic; it is unlikely that an application would perform the same join operation on the same datasets repeatedly. It is, however, quite
likely that data could be declustered to perform multiple range queries, since for each query, a different query window could be used. Here, where the data is to be declustered for an extended period of time, it is not important how much time is spent in the act of declustering; it is more important that the data is declustered well, to optimize the anticipated queries.

The main purpose of this thesis is to examine queries in data-intensive applications, that is, where the datasets are large enough that they should not be moved, except for one initial declustering; after this, only intermediate results should be transferred among nodes [CABK88]. Therefore, for this thesis, we will be concerned with the scenario where two datasets are initially located on a single node, where the data will be declustered only once, and where all queries will be performed using this declustering. Once the datasets are distributed among multiple nodes, a series of range queries are performed on both datasets (with possibly different query windows) and join operations are routinely performed on the results of two range queries. The results of the range queries may be re-declustered across the nodes for single join operations. This scenario allows us to work with only two datasets at a time. It also makes the performing of multiple joins more realistic, since each will be performed on different subsets of the two datasets.

This thesis will present experiments relating to range and join queries, performed on real and synthetic rectangular data in two-dimensional space, using either a single node, or multiple (between 4 and 12) nodes in a single-user environment. We will focus on shared-nothing architectures, since these have been shown to be the most scalable architecture. This is due to the fact that no single resource, other than the network, is shared by the nodes, and therefore large numbers of nodes can be added to the system [WDJ91].

Some of our experiments consider only a part of the query described above. For example, some experiments examine declustering datasets to perform only a series of range queries, or only a single join operation. This is done to gain an understanding of the properties of the Filter Tree and the $S^3 J$ algorithm, and, in the case of experiments with $S^3 J$, the datasets used may be considered to represent data sub-sets, or intermediate results, created by an earlier operation.

Four new techniques are presented to decluster spatial data for the Filter Tree and $S^3 J$, and experiments are performed using two of these techniques, Brute Force Division, and Division Based on Skew. The experiments show that these techniques allow multi-node queries to be performed efficiently. Brute Force Division requires a set of formulas to predict the times for the various steps of the query algorithms for given declusterings. Therefore, several experiments are first presented to develop and demonstrate the accuracy of these formulas.

### 1.1 Overview of the Filter Tree and $S^3 J$

Filter Trees are based on a hierarchal representation of the data space, achieved through a recursive division of the data space in each dimension. The Filter Tree, then, represents the data space using many levels, where each level divides the data space at a finer granularity than the previous level. That is, each level divides the data space into a
greater number of cells than the previous level. The implementation of the Filter Tree studied in this thesis uses twenty levels, numbered 0 through 19. In the case where there are two dimensions, Level 0 has a single cell, Level 1 divides each dimension of this cell in half, and so has 4 cells, Level 2 divides each cell from the previous level in half in each dimension, and so has 16 cells, and so on. Figure 1.1 shows this for the first four levels. In general, with 2-dimensional space, each level \( l \) contains \( 4^l \) cells.

![Hierarchal Representation of Data Space: Levels 0 to 3](image)

Figure 1.1: Hierarchal Representation of Data Space: Levels 0 to 3

Each object is associated with exactly one level, the lowest level at which the object is contained entirely in one cell. The object is then associated with both this level, and the cell of this level in which the object is contained. For example, if the data space contains two objects, as shown in Figure 1.2, then the object at the centre of the data space does not fit entirely within any cell at Level 1, but it does fit within the single cell at Level 0. Similarly, the object in the upper-right portion of the data space does not fit entirely within any cell at Level 2, but it does fit within a cell at Level 1. The two objects are then associated with the levels and cells as shown in Figure 1.3. In this manner large objects are kept at high levels where there are few cells, and small objects are generally kept at lower levels where there are many cells.

![Data Space containing 2 objects](image)

Figure 1.2: Data Space containing 2 objects
Within each level, the objects are ordered by their Hilbert value. The Hilbert curve is a space-filling curve that visits every cell of the level exactly once, and which never crosses itself. Figure 1.4 shows the Hilbert curves for Levels 1 and 2. Level 0 has only a single cell, which is given Hilbert value 0, and so does not require a curve. At all other levels, the Hilbert curve begins in the lower-left corner and ends in the lower-right corner, and each cell is ordered by its position on the curve. Objects are given the Hilbert value of the cell in which they are contained. For example, the two objects in Figure 1.3 are respectively at Level 0, with Hilbert value 0, and at level 1, with Hilbert value 2. Hilbert values may also be specified as binary fractions of at least $kl$ bits, where $k$ is the dimensionality, and $l$ is the level. Using integers and binary fractions are equivalent, and using either is sufficient. In the sequel we will use integers to represent Hilbert values.

The Hilbert curve orders the cells in such a way that cells that are physically close in the data space tend to be located near each other on the Hilbert curve. Therefore, using the Hilbert curve to order the objects has the property that objects that are physically close will tend to be stored in the same, or in a nearby, data page.

Before queries are performed using the Filter Tree, the datasets must be pre-processed. This creates, for each dataset, one file containing the index, and a set of files containing the data. There is one data file for each level, and within each data file the objects are ordered by their Hilbert values. The index file contains a series of B-trees, one for each level. Each B-tree contains a set of pointers into the relevant data file. Both the index file and the data files are composed of a series of pages, each 4096 bytes in size.
1.1.1 Range Queries

To perform a range query on a single node, a query window is first specified. At any given level, this window will cover certain ranges of Hilbert values. This set of ranges of Hilbert values is calculated at one level of the Filter Tree, known as the containment level. Figure 1.5 gives an example of a query window where Level 3 is used as the containment level. In this case, the window covers the Hilbert ranges: 16 to 20, 23 to 24, 27 to 32 and 35 to 36.

![Query Window At Containment Level 3](image)

Once the ranges of Hilbert values that the window covers at the containment level are found, it is simple to calculate the Hilbert values at all other levels. For example, each cell at the containment level covers exactly four cells at the level below. These four cells can be found by multiplying the Hilbert value at the containment level by four, and then adding zero, one, two and three. For example, Cell 1 at Level 1 corresponds to Cells 4, 5, 6 and 7 at Level 2. So, in the case above, where one of the ranges of Hilbert values covered by the window is 16 to 20 at Level 3, we can calculate that the window covers Cells 64 to 83 at Level 4. A similar method can be used to find the cells two levels down. This involves multiplying by 16, then adding 0 to 15, and so forth.

To find the corresponding cells at higher levels, we instead divide by four and then apply the floor function. For example, in the case above, the range 27 to 32 at Level 3 corresponds to Cells 6 to 8 at Level 2, Cells 1 to 2 at Level 1 and Cell 0 at Level 0.

Once the ranges of Hilbert values are calculated for each level, the data files of the Filter Tree are searched one level at a time. For each level, the index file is examined to determine which data pages must be read. The necessary data pages are read and each object is compared with the query window. A buffer is used to store both index and data pages. A more complete description of the range query, as performed on the Filter Tree, is given in Appendix B.
1.1.2 Join Queries

The Filter Tree also allows spatial joins to be performed on two datasets. However, since the act of creating the index and data files from a dataset is quite expensive, it is not efficient to use the Filter Tree to perform only a single join operation. Since we will be considering in this thesis the case where each join is performed on different range query results, it is more efficient to use the $S^3J$ algorithm.

The $S^3J$ organizes the data in a manner that is simpler than the Filter Tree, in that B-trees are not used. It is, therefore, able to prepare the data for joins much quicker, but actually performs the joins more slowly than the algorithm using the Filter Tree. The $S^3J$ begins by creating a set of temporary files, referred to as level files. One level file is created for each level, for each dataset. It then populates these with the objects from the two datasets. For each dataset, $S^3J$ examines each object and calculates its level and Hilbert value, and writes the object to the appropriate level file. This step is known as the Partition Step. The next step is the Sort Step, where each level file is sorted by the Hilbert values.

The third step is the Join Step. To perform this, $S^3J$ first reads one page from each level file of each dataset into the buffer. This requires $S^3J$'s buffer to be at least $2L$ pages, where $L$ is the number of levels. In performing the join, each level file has its pages brought into memory in order, such that each page is brought in when the join has finished with the previous page of that level file, and each page remains in memory until the join has finished with it. In this way, each page is read into memory exactly once.

The algorithm works by first taking one cell from the first set and comparing it to the corresponding cell in the opposite set, and the cells at higher levels that enclose this cell. For example, if we consider Cell 14 of Level 2 of the first dataset, we would compare this to the following cells in the second dataset: Cell 14 of Level 2, Cell 3 of Level 1 and Cell 0 of Level 0. There are no other cells, at these levels, with which an object in Level 2, Cell 14 of the first dataset can intersect. There are, however, cells at lower levels in which objects may intersect objects in this cell. These join-pairs will be found when these lower cells are compared with cells in the first set. The algorithm alternates between comparing a cell in the first set with a set of cells in the second set, and comparing a cell in the second set with a set of cells in the first set. In either case, which specific cell is used is determined by finding the cell that has the smallest Hilbert value, for the relevant dataset, that is currently in memory and that has not been compared to the other dataset.

As pairs of intersecting objects are found, they are written to an output page, which is kept in memory and written to disk whenever it becomes full. A more complete description of the three steps of $S^3J$ is given in Appendix A.

Using the Filter Tree involves the creation of data files, and an index file that allows quick access to the data. We will consider an environment where the data and index files exist indefinitely. Although rows may be inserted, deleted or modified, updates to the Filter Tree can be done efficiently. Therefore, the time required to create and update the index and data files from the original dataset is not significant. With the $S^3J$, however, the level files created from the original input sets are useful only for one join operation. Therefore, the time required to create these files and sort them is important.
and is considered here.

Since Filter Trees have a more sophisticated structure, including a B-tree index, than that used by $S^3J$, and they therefore allow the queries to be performed more efficiently, they are more appropriate where multiple queries will be performed on a dataset. The $S^3J$ is capable of performing only joins, and is suitable where only one join operation will be performed. In cases such as joining two range query results, it is not practical to build a Filter Tree. Therefore, we will use Filter Trees for the range queries and $S^3J$ for the joins.

With both range queries and joins, if the data has more complicated shapes than rectangles, then both Filter and Refinement steps are necessary. The objects are, in this case, bound by MBRs, and the Filter Step works only with these, in the manner described above. With range queries, the Filter Step finds all MBRs that intersect the query window, and with join queries, the Filter Step finds all pairs of MBRs that intersect. The Refinement Step is then performed on each object to determine which actually intersect. With significantly complex shapes, the Refinement Step can be quite costly. With rectangular data, only a single step, equivalent to the Filter Step is necessary.

### 1.2 Data Skew

Walton et al. have identified two types of skew: intrinsic (attribute) skew and partition skew [WDJ91]. Attribute skew has to do with the distribution of the data within the data space, and cannot be controlled. When data is spread evenly over the data space, it is considered to have a uniform distribution. If, instead, the data is concentrated more densely in some areas than others, it is considered non-uniform and is referred to as being skewed (i.e., having an intrinsic skew). With non-spatial data, there are several distributions that are commonly used to model intrinsic skew, such as the Zipf [Zip49] and Normal distributions. With spatial data, it is not as clear how to model data skew. Some authors have used a Zipfian distribution by creating synthetic data where each coordinate of the objects has a Zipfian distribution, so that the data is concentrated in the lower-left corner [TS96]. Koudas used two synthetic distributions to simulate real data skew, referred to as “equal area coverage” and “triangular distribution” [Kou97].

Partition skew has to do with how data is distributed between nodes. Walton et al. found that intrinsic skew is the cause of partition skew, and partition skew is what affects the query times, but the relationship between intrinsic and partition skew is difficult to identify. Four classes of partition skew have been identified: Tuple Placement Skew, Selectivity Skew, Redistribution Skew and Join Product Skew [WDJ91]. In the case of the main focus of this thesis, performing range queries and joining their results, these correspond to the initial distribution of the two complete datasets (before the range queries), the distribution of the range query results, the distribution of these results after they are re-declustered for the $S^3J$ operation, and the distribution of the join results.

Since applications cannot alter the intrinsic skew of datasets, they must distribute the data in an appropriate way given the skew, to mitigate the effects of intrinsic skew by minimizing the partition skew. If a single query is performed on multiple nodes, and the data is uniform, the performance improvement should be roughly proportional to the
number of nodes [WDJ91]. However it is possible for there to be very little speedup if the data is skewed [DNSS92]. This can occur where most of the data is sent to only a small fraction of the nodes, leaving the other nodes to be idle for much of the time the query is being processed. One of the disadvantages of the shared-nothing architecture is its susceptibility to data skew [LC93]. In other architectures, where copies of the same data are available to all processors, data skew is not an issue. Much of the work of this thesis is aimed at addressing the effects of skew on query response time.

1.3 Thesis Organization

The remainder of the thesis is outlined as follows:

Chapter 2: Related Work. This chapter examines other structures to maintain spatial data, work performing range and join queries on spatial data using single and multiple nodes, work declustering data to multiple nodes, and work estimating the selectivities of queries.

Chapter 3: Problem Statement. This chapter (i) explains the issues involved with declustering data for the Filter Tree and $S^3J$, including replication; (ii) presents four methods to divide the data space into fragments; (iii) compares these four methods; and (iv) compares the methods described in Chapter 2 to distribute fragments to nodes.

Chapter 4: Experimental Setup. This chapter describes the data used for all experiments, the query windows used for the range queries, and the algorithms used to perform range, join and combined range and join queries. It describes how histograms are used to predict the selectivities of queries. It also describes the communication and I/O costs of the environment in which the experiments are performed.

Chapter 5: Experimental Results for Range Queries. This chapter provides the results of timing experiments for range queries on single and multiple nodes. These test formulas that estimate the time required for range queries, and test how well our techniques work to decluster data for multi-node range queries.

Chapter 6: Experimental Results for Joins. This chapter provides the results of timing experiments for joins on single and multiple nodes. These test formulas that estimate the time required for joins, and test how well our techniques work to decluster data for multi-node join queries.

Chapter 7: Experimental Results for Combined Range and Join Queries. This chapter applies the results found in Chapters 5 and 6 to the problem of declustering for a series of range and join queries.

Chapter 8: Conclusions and Future Work. This chapter provides a summary of the work done for the thesis, experimental results and the significance of the results. It then lists several directions in which the work presented in this thesis may be expanded.
Chapter 2

Related Work

2.1 Indices Used for Spatial Data

Among the more common index structures used for spatial data are the k-d-b Tree, the R-Tree, and its variants, including the $R^+$-Tree, $R^*$-Tree and Hilbert R-Tree. Filter Trees were developed largely as an alternative to these indices. R-Trees are an extension of B-trees, suitable for spatial data. As with B-trees, the R-Tree is composed of a set of nodes. At the top there is a single node, which points to the set of nodes at the next level, which in turn point to the set of nodes at the next level and so on. Each non-leaf node of the index tree is composed of a set of pointers of the form $\langle \text{ptr}, R \rangle$, where $\text{ptr}$ is a pointer to a child node, and $R$ is the minimum bounding rectangle (MBR) that contains all objects in the child node (and its children). Leaf nodes also contain $\langle \text{ptr}, R \rangle$ pairs, where $\text{ptr}$ is a pointer to another file that contains the object, and $R$ is the MBR for the single object. R-Trees have order $(m, M)$, where each node, with the exception of the root, contains between $m$ and $M$ $\langle \text{ptr}, R \rangle$ pairs. For a given set of data, there is no one unique R-Tree; the tree depends on the values of $m$ and $M$ and on the order in which the objects were inserted into the tree.

Insertions are similar to those in B-trees, in terms of the tree remaining balanced, and changes being propagated up the tree. The R-Tree begins with an empty tree which is populated by inserting objects one at a time. As objects are inserted, there are different strategies available to choose the leaf node to which they will be added, such as finding the leaf with the MBR that needs to be expanded the least. As well, when a leaf node overflows, it must be split, and this split must be propagated up the tree. When a leaf node is split, two new leaf nodes are created to replace it, and the $M + 1$ objects are distributed between these two leaves. There are different strategies used to split a node. It may be split such that the total areas of the two new MBR's is minimized, or it may be split such that the area common to the MBR's of the two new nodes is minimized [Gut84].

The $R^*$-Tree is similar to the R-Tree except it utilizes more sophisticated algorithms for inserting and splitting nodes. The insertion algorithm seeks to find the node where the resulting MBR has the least overlap with its siblings. The split algorithm splits a region such that the minimum overlap between the MBR's of the two new nodes results,
which requires considering every possible split in each dimension. $R^*$-Trees also execute a forced re-insert when nodes overflow; certain members are deleted and then re-inserted, which usually gives the tree a better structure but is more expensive in terms of the time to perform inserts [BKSS90].

The structures for the $R^+$-Tree and k-d-b Tree are very similar. They are distinct from R-Trees and $R^*$-Trees in that the data space is decomposed into non-overlapping regions, so that no MBRs have any overlap, but objects may be contained partially in more than one region. This necessitates storing sub-objects in the leaf-nodes, with the effect that each object may be stored multiple times. The difference between a k-d-b Tree and a $R^+$-Tree is that once the space has been divided, the $R^+$-Tree attempts to shrink regions where possible, so that each region is just large enough to contain the objects within it; the k-d-b Tree does not perform this step. The k-d-b Tree can be built faster, but the $R^+$-Tree allows faster query times, since the number of search paths is generally lower [SRF87, Rob81].

The design of the Hilbert R-Tree is based on the assumption that the most efficient form of the R-Tree is where the size of the MBRs in the non-leaf nodes can be minimized. This allows range queries to prune the search space more effectively. To achieve this, the Hilbert R-Tree seeks to group objects that are physically close together as much as is possible. This is achieved by sorting objects by the Hilbert values of their centres and forming groups of objects that are adjacent in sort order.

The k-d-b Tree and each of the R-Tree variants execute range queries in the same manner: a search begins at the root node and proceeds down paths in which the MBRs intersect the query window.

### 2.2 Range Queries on Spatial Data Using Multiple Nodes

Koudas, Faloutsos and Kamel studied declustering spatial data for range queries using R-Trees [KFK96]. Their major focus was to determine the best unit of declustering, which would activate an optimum number of processors for each query, and would result in the minimum query response times. They found that, given the environment in which they performed their experiments, it is not necessarily optimal to decluster data such that nearby objects are sent to different nodes. For example, a range query that returns a small number of objects, say ten, should not retrieve these from ten different nodes, due to the communication costs necessary to pass data among ten nodes. In this case, the optimum number of nodes is likely much less than ten. It is desirable to activate only this smaller number of nodes by ensuring that the ten objects are declustered to this number of nodes.

They also found that for larger queries, it is desirable to have the data divided evenly among the nodes. This may be achieved, where the data space is divided into chunks, by placing chunks that are physically close on different nodes. They therefore sought to find the optimal chunk size such that all objects within a given chunk are assigned to the same node, and chunks that are physically close are assigned to different nodes. They
found that in most cases one page was the optimal chunk size. They also found that assigning a Hilbert ordering to the chunks was effective in assigning nearby chunks to different nodes. This allowed them to decluster such that small queries were performed by a small number of nodes and large queries were performed by a large number of nodes, and so, average response times were minimized.

### 2.3 Join Queries on Spatial Data

Several methods, other than the Filter Tree and $S^3J$, to perform join queries on spatial data have been proposed. These include use of the R-Tree, as well as Partition Based Spatial Merge (PBSM) [PD96], and Scalable Sweeping-Based Spatial Join [APR+98]. The last two are designed for cases where a join is to be performed on a pair of non-indexed datasets, and so are analogous to $S^3J$. The R-Tree is itself an index, and so is analogous to the Filter Tree.

Brinkhoff et al. proposed an efficient algorithm to perform join queries using the R-Tree [BKS93]. This begins at the root nodes of the two datasets and calculates the overlap of their MBRs. The algorithm then proceeds through the tree in a depth-first manner until it locates all pairs of MBRs in the leaf nodes that intersect. As it searches the trees, it locates the intersecting area of each MBR with the MBRs of the relevant nodes in the opposite tree, and in this way is able to prune much of the search space. Huang et al. later proposed an improvement on this using a breadth-first search [HJR97].

Patel and DeWitt proposed PBSM, which divides the data space into a number of partitions, such that the partitions cover the entire data space and do not overlap, and such that the objects within each partition may fit entirely in memory. It then joins the two sets using a plane-sweeping algorithm on each partition, one at a time. A plane-sweeping algorithm locates matching pairs by moving a sweepline along one axis of the data space. In this way, all intersecting objects are found, since each must be crossed simultaneously by the sweepline when it is at some position. Since objects may be partially within two or more partitions, they will be replicated in each of these partitions, and duplicate join-pairs may be reported. The PBSM addresses this by sorting all join-pairs and removing duplicates as the final step [PD96].

The Scalable Sweeping-Based Spatial Join, proposed by Arge et al., begins by ordering all objects by their lower boundary in the $y$ dimension. The algorithm then uses a plane sweeping algorithm to find all join-pairs. Arge et al. found that in general, all objects that are crossed by the sweep line, when it is in any given position, will fit into main memory. In some cases, however, this will not be possible and the data space must first be divided into vertical strips. The Scalable Sweeping-Based Spatial Join uses random data sampling to determine if there exists a high concentration of objects anywhere in the data space that would necessitate the division of the data space into strips. It also uses the sample to determine where the strips can be defined, so that all join-pairs may be found in main memory [APR+98].
2.4 Declustering Data to Multiple Nodes

The key decision when performing queries on multiple computers is how to decluster the data. This involves the two smaller questions: how to divide the data space into fragments, and how to distribute these fragments to the nodes. Methods to divide the data space include: Multi-Attribute Grid Declustering (MAGIC) [GDQ92], Range Partitioning, and Hybrid Range Partitioning Strategy (HRPS) [GD90b]. As well, four new techniques were developed for this thesis: Division Based on Skew, Brute Force Division, Slicing, and Variable Fragment-Size Division.

Most of these techniques use virtual processors, which implies that the data is divided into more fragments than there are nodes. For example, if there are four processors, the data may be divided into more than four parts, perhaps sixteen or sixty four parts. If there are four processors, and sixteen virtual processors, then each of the four processors will get some subset of the sixteen parts. Using virtual processors often means the act of declustering requires more time, but it allows the loads to be better balanced where there is data skew [DNSS92]. Using virtual processors also necessitates a technique to distribute the fragments to the nodes.

Methods to distribute the fragments include: round-robin, hash functions, List Scheduling [Gra66], Coordinate Modulo Distribution (CMD) [LSR92], Field-wise Exclusive- or (FX) [KP88], Error Correcting Codes (ECC) [FM91], the Hilbert Curve Method (HCAM) [FB93] and the Vector Based Method (VECTOR) [CR93]. Some techniques to improve the distribution of fragments include Incremental Techniques and Dynamic Load-Balancing Techniques.

2.4.1 Dividing the Data Space Into Fragments

MAGIC

Ghandeharizadeh et al. proposed MAGIC to decluster non-spatial data for range queries, but since it's performed on multiple attributes, it may be considered equivalent to declustering for spatial data. For example, if MAGIC uses two partitioning attributes, then this is equivalent to declustering a two-dimensional data space. MAGIC divides the data space into a grid, by dividing each attribute (dimension) into a series of ranges. This creates a set of cells, each of which may be assigned to a processor. To divide the data space into cells, MAGIC considers a sample set of queries. From these, it characterizes the average query, \( Q_{ave} \) by calculating the average number of tuples returned, and the average CPU and I/O times for the set of queries. It then calculates \( M \), the ideal number of processors to use in processing \( Q_{ave} \), given the communication overhead of the network. By setting the size of the cells such that each cell contains \( \frac{1}{M} \) times the number of tuples returned by \( Q_{ave} \), MAGIC ensures that \( Q_{ave} \) is performed by \( M \) nodes. The cells are then assigned to the processors in such a way as to balance the two goals of placing physically close cells on different nodes, and assigning each node an approximately equal workload [GDQ92].
Range Partitioning

Range Partitioning is a linear technique, which are techniques commonly used to decluster non-spatial data based on a single partitioning attribute. With Range Partitioning, each node receives the rows representing a certain range of values for the partitioning attribute [GDQ92]. It is not necessary that the boundaries of each range be evenly distributed in the attribute domain; they may be chosen such that each range contains an approximately equal number of rows [DNSS92]. Since Range Partitioning creates a number of fragments equal to the number of nodes, there is no issue of how to distribute the fragments.

With spatial data, Range Partitioning implies that each node is sent a single fragment of data representing one connected area of the data space. Range partitioning could mean giving each node a certain range of values along one axis, or could refer to giving a certain range of Hilbert values to each node. We have developed a form of Range Partitioning based on Hilbert ordering called Slicing, which is discussed in Section 3.5.1.

Hybrid Range Partitioning

One problem with Range Partitioning is that with range queries, it often does not allow the desired degree of parallelism. This is because any given query window will likely cover the ranges assigned to only a small number of nodes. The Hybrid Range Partitioning Scheme (HRPS) also declusters a dataset based on the values of a single partitioning attribute, which, as with Range Partitioning, may be the Hilbert values. The HRPS is a hybrid of Range Partitioning and round-robin declustering, where the partitioning attribute is divided into small fragments, which are distributed in a round-robin fashion to the nodes. This gives each node a set of small ranges of records, which allows queries with low selectivities to be performed on a small number of nodes, and queries with high selectivities to be performed on a large number of nodes.

To determine the size of the fragments, HRPS first determines the optimum number of nodes to execute the average query. The average query is found by considering the frequency, and CPU, I/O and network costs for a set of queries, in a similar way as MAGIC. The HRPS then develops a formula for the response time of the average query, which is specified in terms of the number of participating nodes. Setting the first derivative to zero gives an estimate of the optimum number of participating nodes. The fragment size is then calculated from the optimum number of participating nodes and the size of the average query [GD90b].

2.4.2 Distributing Fragments to the Remote Nodes

This section describes several methods to distribute fragments to nodes, including linear techniques and techniques for Cartesian Product files. Cartesian Product files are files where the data space has been divided into a grid, and each record belongs to exactly one cell, such that its attributes are within the specified range of attribute values for that cell [FB93]. In the case of the Filter Tree and $S^3J$, each level file may be considered to be a Cartesian Product file. The goal with these techniques is to place cells that are physically close on different nodes.
The techniques for Cartesian Product files considered here are CMD, ECC, FX, HCAM and VECTOR. The linear techniques considered here are round-robin, hash functions and List Scheduling.

**Round-robin**

Round-robin is often applied to non-spatial data to decluster based on a single attribute. Rows that have the first value for the partitioning attribute are sent to the first node, rows with the second value to the second node, and so on. With spatial data, this can be done based on Hilbert value, in which case round-robin is equivalent to HCAM, described below.

Round-robin can result in a load imbalance if there is a data skew. However, it is fast to execute, and is generally appropriate where there is little data skew. How susceptible to data skew round-robin is depends on how many distinct values there are for the partitioning attribute and how many nodes there are. For example, if there are 8 nodes and 16 distinct values, each node would get 2 sets of rows, and if some values for the partitioning attribute were much more frequent than other values, there could be significant load imbalance. However, if there were 8 nodes but 800 distinct values, each node would receive 100 sets of rows, and the odds of significant load imbalance would be much smaller.

**List Scheduling**

List Scheduling is a technique where a set of tasks are given in some arbitrary order, and are assigned to a set of processors using a greedy algorithm. That is, the tasks are considered one at a time, in the order in which they are given, and each is assigned to the processor which is currently estimated to have the smallest workload.

Graham studied List Scheduling and determined that if we are given a set of tasks in some order, and assign the tasks to nodes using a greedy algorithm, then the resulting assignment of tasks to nodes will have a competitive ratio of $2 - \frac{1}{n}$, as compared to the optimal assignment, where $n$ is the number of nodes [Gra66, Aza97]. When List Scheduling is applied to declustering spatial data, we assign each cell of a given level to a set of nodes, considering the cells in any order, such that we assign each cell to the node that currently has the smallest estimated workload. Doing this, the node that receives the most rows will receive at most $2 - \frac{1}{n}$ times the maximum number of rows given to any node using the optimal declustering.

**Hash Functions**

Hash functions are commonly used to decluster non-spatial data by applying a hash function to each distinct value of the partitioning attribute, and, using this, mapping attribute values to buckets. The number of buckets may be equal to or greater than the number of nodes. With spatial data, a hash function may be applied to the Hilbert values of the objects. If the number of buckets is greater than the number of nodes, the buckets must be distributed to the nodes. To do this a scheme such as round-robin, or List Scheduling may be used.
CMD

Coordinate Modulo Distribution (CMD) begins by giving cells a series of index ids, one for each dimension of the data space. This is done by numbering the cells in each dimension such that the index ids go from 0 to the number of nodes minus one, and then repeat, as shown in Figure 2.1 (a). The CMD assigns each fragment to a node by summing the index ids in each dimension and calculating this sum mod \( N \), where \( N \) is the number of nodes. In two-dimensional space CMD creates a pattern of diagonal stripes, where each stripe represents one destination node \([DS82, LSR92, FB93]\). The CMD is quite similar to another technique to distribute fragments called Disk Modulo (DM), which was originally proposed by Du and Sobolewski \([DS82, LSR92]\). Figure 2.1 (a) shows the distribution of fragments to four nodes if the data space is two-dimensional and divided into 64 fragments.

\[
\begin{array}{ccccccc}
3 & 3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(a) CMD

\[
\begin{array}{cccccc}
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(b) ECC

\[
\begin{array}{cccccc}
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(c) FX

\[
\begin{array}{cccccc}
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(d) HCAM

\[
\begin{array}{cccccc}
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3 & 3 & 3 & 3 & 3 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(e) VECTOR

Figure 2.1: Distribution of Fragments to 4 Nodes Using CMD, ECC, FX, HCAM and VECTOR

ECC

Error Correcting Codes (ECC) requires the number of divisions of the data space in each dimension to be a power of two. Each fragment is represented by the concatenation of the binary representations of the index ids. With ECC, the index ids range from 0 to the number of fragments in each dimension minus one. For example, in two-dimensional space, fragment (2,3) is represented by the concatenation of 010 and 011, 010011. Each fragment is then represented by \( km \) bits, where \( k \) is the number of dimensions, and \( 2^m \) is the number of fragments in each dimension. The \( km \) bits are treated as Error Correcting Codes, and the last \( \log_2 N \), where \( N \) is the number of nodes, represent parity check bits. The remaining bits are information bits. Mappings of fragments to nodes can be found by using the \( N \) different values expressible in the \( \log_2 N \) bits \([FM91]\). Figure 2.1 (b) shows the distribution of fragments to four nodes if the data space is two-dimensional and divided into 64 fragments.

FX

Field-wise Exclusive-or (FX) assumes the number of nodes is a power of two. Though it will work where the number is not a power of two, the performance will suffer \([MS98]\). The FX assigns index ids in the same manner as ECC. It finds the destination node
for each fragment by taking the exclusive-or of the binary representation of each index id, and then taking the last $\log_2 N$ bits to represent the id of the destination node. For example, if the fragment has index id (3,7), and there are 4 nodes, the destination node is found by:

$$3 \text{ XOR } 7 = 011_2 \text{ XOR } 111_2 = 100_2.$$  

We take the last $\log_2 4 = 2$ bits to be the destination node. In this case we have node 00$_2 = 0$ [FB93].

Figure 2.1 (c) shows the distribution of fragments to four nodes if the data space is two-dimensional and divided into 64 fragments.

**HCAM**

The Hilbert Curve Method (HCAM) orders each fragment based on its Hilbert value and distributes the fragments to the nodes round-robin based on this ordering. The Hilbert ordering is known to have excellent clustering properties, and therefore is more effective than other methods in ensuring that fragments that are physically close in the data space are sent to different nodes. The HCAM is also more flexible than some declustering methods, because it allows for any number of fragments and any number of nodes [FB93]. Figure 2.1 (d) shows the distribution of fragments to four nodes if the data space is two-dimensional and divided into 64 fragments.

**VECTOR**

The goal with the Vector Based Method (VECTOR) is to maximize the area that can be covered by a range query such that each node must perform only one disk read. VECTOR is designed primarily to meet this goal, and not to necessarily perform well on average for range queries. It was designed for visualization systems that require that all accesses be within a certain time period, and so is more concerned with worst case performance than average case performance, or with cases where the range queries require more disk pages to be read than there are nodes. VECTOR is best suited to two-dimensional image and cartographic databases, and is limited since no algorithm has been developed to extend it beyond two dimensions [MS98].

VECTOR uses two vectors, $u = (a, b)$ and $v = (c, d)$. Each fragment is assigned to a node such that once a fragment is assigned to a node, all fragments whose distance away, in the two-dimensional data space, is $mu + nv$, for all $m$ and $n$, are assigned to the same node. The two vectors, $u$ and $v$, may be viewed as forming a parallelogram, of area $N$, where $N$ is the number of nodes. The algorithm requires, then, that $|ad - bc| = N$. Since the goal is to place fragments that are close on different nodes, we wish also for fragments that are sent to the same node to be far apart. This is achieved by maximizing $u + v$ such that the above constraint is met [CR93]. Figure 2.1 (e) shows the distribution of fragments to four nodes if the data space is two-dimensional and divided into 64 fragments, using $u = (2,0)$ and $v = (1,-2)$.

To see how the cells are distributed in Figure 2.1 (e), consider first the cells given to Node 0. If the cell in position (0,0) is given to Node 0, each cell that is any multiple of (2,0) and (1,-2) away is also given to Node 0. This causes all cells with coordinates of
the form \((x + ma + nc, y + mc + nd)\) to be given to Node 0. For example, setting \(m = 1\) and \(n = 0\) gives \((2,0)\); setting \(m = 2\) and \(n = 0\) gives \((4,0)\); setting \(m = 1\) and \(n = -1\) gives \((1,2)\) and so on. Figure 2.1 (e) shows that each of these cells are given to Node 0. Once every cell that must be assigned to Node 0 is assigned in this manner, we choose an unassigned cell for Node 1, and, using every value of \(m\) and \(n\), assign all the appropriate cells to Node 1 in the same way. This then continues for each of the other nodes.

**Improvements to Methods to Distribute Fragments**

Several improvements can be made to the above techniques to reduce the load imbalance due to data skew, including Incremental Techniques and Dynamic Load Balancing. Incremental Techniques work by, at regular checkpoints, examining the amount of work performed by each node, to determine how well balanced the nodes are. If the imbalance is too great, then data can be moved from the busiest to the least-busy nodes. Generally, when this is done, data will only be moved after the first few checkpoints. This incurs overhead from the act of checking the loads, and moving data between nodes, but will improve the load balance over time. If the data is to be declustered for a long period of time, the work to redistribute some portion of the data a few times is not significant [HTY95].

A similar technique is known as Dynamic Load-Balancing. While incremental techniques set check points at intervals during a sequence of queries, to examine the total amount of work done by each node since the previous checkpoint, Dynamic Load-Balancing attempts to adjust the load balance in the middle of one query, and applies only to that query. Each node may, after performing its portion of the query, check other nodes to determine if they are still executing the same query. Nodes that still have a sufficient amount to do may send some data to the nodes that have finished. In this way, the workload may be made reasonably even among the nodes [SRK+98].

**Comparison of Techniques to Distribute Fragments**

DeWitt et al. compared Range Partitioning, hash functions, round-robin, and List Scheduling for join queries on non-spatial data [DNSS92]. Their experiments varied the skew of the partitioning attribute. All values of this attribute were distributed uniformly except the most frequent value, whose frequency they varied. Round-robin and List Scheduling use virtual processors while hash did not. Range Partitioning, by definition, does not use virtual processors. Their implementation of Range Partitioning used sampling to determine the divisions between the partitions. They found hash partitioning to be the best technique where the data was not skewed, since this did not require the overhead of virtual processors or sampling. DeWitt et al. also found that for skewed distributions, round-robin was superior to List Scheduling, since the improved declustering achieved by List Scheduling was not offset by the extra overhead [DNSS92].

Several authors have presented comparisons for range queries using CMD, ECC, FX, HCAM and VECTOR [HS94, MS98, FB93]. These experiments did not consider data skew or correlations between attributes, and did not consider how the data was divided into fragments; the papers simply assumed all fragments were 4K bytes.
Himatsingka and Srivastava compared CMD, ECC, FX and HCAM on uniform data, and found certain strategies to be better in certain circumstances, although none was consistently superior to the others. Also, all four performed very similarly, and close to optimal, for large queries on large databases [HS94].

Faloutsos et al. also compared these four techniques for range queries. The tests covered a wide variety of data sizes and query window sizes, and used either 8, 16 or 32 nodes. They did not specify the distribution of their test data. It can be assumed, then, that it was likely uniform data. These tests showed HCAM to perform as well or better than DM, FX and ECC [FB93].

Moon et al. compared DM, FX, HCAM and VECTOR and found that DM is nearly optimal for range queries for small numbers of nodes, but that DM and FX scale poorly as nodes are added. They found, however, that HCAM and VECTOR scale well. Their experiments in two dimensions showed that as more nodes were added to the system, HCAM and VECTOR’s response times continued to improve, but DM and FX’s did not. For two dimensions, VECTOR was shown to be marginally better than HCAM, though HCAM also performed close to optimally for all experiments performed. For three and four dimensions, HCAM consistently performed the best. They did not specify the distribution of their test data [MS98].

2.5 Estimating Selectivity

Estimating the workloads for remote nodes may be necessary to achieve a good load balance when declustering. Estimating the workload is done by estimating the size of the subsets sent to each node and the size of their output sets. Several methods exist to calculate these estimates. Here we will consider multi-dimensional histograms, analytic models and fractals.

In the last decade there has been a hundred-fold increase in CPU speeds, but only a ten-fold increase in hard-drive speeds [BSG+99]. As well, improvements in processor speed are expected to continue to outpace improvements in disk speeds [HS94]. Therefore, it makes sense to base the workload estimates largely on I/O. While some operations are now CPU bound, and others are I/O bound, in the future the same operations will likely be more and more I/O bound. Therefore, estimating workload based on the size of the input file(s) and the estimated result sizes appears quite reasonable. Experiments below confirm that even where operations are CPU bound, the total time may still be accurately predicted based on the size of the input and output sets.

2.5.1 Fractals

Bellusi and Faloutsos observed that datasets generally behave as fractals, and the average number of neighbors, within distance $\epsilon$, $(n_b)$ each point (where the data is point data) has may be shown to follow a power law. Using this observation, and determining a specific power, the correlation dimension ($D_2$), it is possible to accurately predict query selectivities. The $D_2$ is a specific instance of the general fractal dimension. It is used

\[1\text{though they used DM instead of CMD}\]
where sets of points exist in some data space that has been divided evenly into cells, and represents the ratio of the squared sum of the cell occupancies to the size of the cells. Once $D_2$ is calculated, it may be used to determine the average number of neighbors each point has. That is, $\bar{n}_b$ can be derived directly from $D_2$. This, then, may be used to estimate the selectivity of biased range queries or self-joins.

In the case of random range queries, the selectivity is well-known: the area of the query window divided by the area of the data space, multiplied by the size of the dataset. But for biased range queries, where the centre of a query window must correspond to a point in the dataset, the selectivity is $\frac{\bar{n}_b + 1}{N}$, and for self-joins the selectivity is $\frac{\bar{n}_b}{N - 1}$, where $N$ is the size of the dataset. Since these formulas require only $\bar{n}_b$, determining this quantity is sufficient for estimating result sizes.

To determine $D_2$, Bellusi and Faloutsos present a box-counting algorithm, which may be executed over the range of cell sizes where the dataset exhibits self-similarity. They also demonstrate that $\bar{n}_b$ may be derived from $D_2$ by showing that $\bar{n}_b$ is proportional to $\epsilon^{D_2}$, which implies $\bar{n}_b$ is equal to $\epsilon^{D_2}$ multiplied by a constant. When the query window is square, this constant is $(N - 1) \times 2^{D_2}$, where $N$ is the size of the dataset. With other shapes, the constant may be found by identifying the square window with equal area as the query window. Using this, Bellusi and Faloutsos derive a formula for estimating $\bar{n}_b$, given $\epsilon$: 

$$\bar{n}_b = (N - 1) \times (2\epsilon)^{D_2}.$$ 

Experiments showed that the formula of Bellusi and Faloutsos consistently estimated $\bar{n}_b$ within 12%. No range queries or self-joins were actually performed, but since they rely on virtually the same formulas that were presented in this paper, it may be assumed their selectivities would be predicted equally accurately.

What is not clear, however, is how many datasets actually behave as fractals. Bellusi and Faloutsos state that it is quite common, but when datasets are not fractal, none of the procedures described here can be applied. As well, whether the data is fractal or not cannot be determined until $D_2$ is calculated, which is an $O(N\log N)$ operation \cite{BF95}.

### 2.5.2 Histograms

Muralikrishna and DeWitt have shown Equi-Depth histograms to be useful for estimating the selectivities of range queries for two and three-dimensional datasets \cite{MD88}. Their technique works by first deciding how many buckets are desired, then how many divisions in each dimension are desired. For example, in the case of spatial data (where each dimension is equally important), if there are to be sixteen buckets and there are two dimensions, there would then be four divisions in each dimension. The positions of the divisions in each dimension can be determined either by examining the entire dataset or sampling it, in order to have a roughly equal number of objects within each division. This requires first sorting the dataset by the first dimension, then by the second and so on. Experiments conducted by Muralikrishna and DeWitt showed that the Equi-Depth histogram always predicted the selectivity within 9% and was generally accurate within 2%. It appears that the Equi-Depth histogram is, then, a useful tool for estimating the selectivity of multi-attribute queries. However, Muralikrishna and DeWitt include no analysis of the algorithm's execution time, and no technique to determine how many
Poosala and Ioannidis studied multi-dimensional histograms to predict query selectivity for multi-dimensional non-spatial data [PI97]. One approach that they considered was Hilbert numbering, in which each point in the data space is ordered by its Hilbert value, and each bucket corresponds to a certain range of Hilbert values. The bucket boundaries are selected using the MaxDiff partition constraint, which causes the boundaries to be placed where there is the greatest difference between the frequencies of the adjacent points. They also considered PHASED, which is a variation on the Equi-Depth histograms proposed by Muralikrishna and DeWitt, and an approach they developed called MHIST. MHIST decomposes the data space into a set of non-overlapping buckets by first considering the entire data space as a single region, then splitting one region into two until there are the desired number of buckets. Splits are made by examining the range of values along each edge of each partition, and determining which edge contains the greatest variance. The split is made perpendicular to this edge to minimize the variance of the two resulting edges. Poosala and Ioannidis's experiments showed MHIST to provide the most accurate selectivity estimates of the three approaches considered.

2.5.3 Analytic Models

Theodoridis and Sellis developed an analytic model to predict both the number of disk accesses necessary to perform range queries using R-Trees, and the selectivity of the queries [TS96]. Their model requires only the size of the dataset, the density, and the values of \( m \) and \( M \), which define the size of the nodes used by the R-Tree. The density is found by calculating the total size of the objects, divided by the size of the data space. By using the density, they are able to predict the average size of the MBRs within the R-Tree nodes at each level, and can use this to calculate the number of disk accesses for a given query, assuming each node is one page, and therefore requires one disk access.

The expected number of disk accesses, for any given query window, is equal to one (to access the root node) plus the expected number of nodes at each level that intersect the query window. The number of levels may be determined from the size of the data set, \( m \) and \( M \). As well, the expected number of objects covered by a query window can be calculated from the size of the window, the number of objects, and the average size of the objects, assuming a uniform distribution. From this it is possible to calculate the average size of the MBRs covered by each non-leaf node.

To improve the accuracy of the estimates, Theodoridis and Sellis calculated, instead of a single value for the density, a density grid, which was equivalent to a Equi-Depth histogram. Without the use of the density grid, the model would be able to estimate only the average results for queries, and not results for specific windows.

For the work in this thesis, we use histograms to predict the selectivities of queries and an analytic model to predict query response times. The analytic model used here does not predict numbers of I/O operations, but does, in a similar way, estimate the total response time based on the numbers of steps that must be performed.
Chapter 3

Problem Statement

The main problem addressed in this thesis is to perform an initial declustering on two datasets, perform a series of range queries on these sets, and to occasionally join the results of two range queries. This leads to a set of sub-problems:

1. What is the best technique to perform the initial declustering? That is, how is the data space divided into fragments, and how are these distributed to the nodes?

2. When is it preferable to decluster the two sets to optimize for range queries, and then re-decluster the range query results for the joins, and when is it preferable to perform the initial declustering to optimize for join queries, and allow the range queries to be performed non-optimally, but avoid re-declustering to perform the joins?

3. When is it useful to base the declustering on the data's skew, and when is it useful to base the declustering on the expected workload?

3.1 Replication of Data During Declustering

If two datasets are declustered to multiple nodes to perform a parallel join using the $S^3J$ algorithm, then inevitably, some cells must be sent to multiple nodes. To illustrate this, consider where there are four nodes. Each node must be given a subset of the cells, and each node must receive the same set of cells for both datasets.

If the two datasets each occupy only Levels 0 and 1, then there is only one possible way to decluster the data: at Level 1, Cell 0 must go to one node, Cell 1 to another and so on. We can assume Node 0 receives Cell 0 and so on. Node 0 can then find pairs of objects in Level 1, Cell 0 of the first and second data sets. But an object in Level 1, Cell 0 of the first dataset may intersect an object at Level 0 of the second dataset. Also, an object in Level 1, Cell 0 of the second dataset may intersect an object at Level 0 of the first dataset. Therefore Node 0 must also receive Level 0 of both datasets. In the same way, all 4 nodes will need the single cell of Level 0, so this cell is replicated at all four nodes.
If the two datasets instead had data at Levels 0, 1 and 2, then there are many ways to decluster the data to four nodes. The data could be declustered at Level 1 as above, or it could be declustered at Level 2, where there are 16 cells. We assume that each node will get at least one cell, and that other than that, any declustering is possible. If the first four cells go to one node, the next four go to another node and so on, then the declustering is the same as above, and only the cell at Level 0 is replicated. Otherwise, some cells at Level 1 are replicated as well. For example, if, at Level 2, Cell 0 goes to Node 0, Cells 1 and 2 go to Node 1 and Cell 3 goes to Node 2, then, at Level 1, Cell 0 (which covers these four cells at Level 2) must go to Nodes 0, 1 and 2.

If the data is declustered at Level 1, then the cells at Level 2 simply go to the same nodes as the cells at Level 1 of which they are a part. There is never replication at the level that is declustered, or at lower levels. Given this, in general, the lower the level at which the data is declustered, the more replication will occur. However, this may be offset by the fact that the data is divided more evenly.

Note that declustering at any given level is equivalent to declustering at the next lower level using fragments of four cells, declustering at two levels down using fragments of sixteen cells, and so on. For convenience, we will usually speak of declustering at a certain level, and treating each cell at this level as a fragment.

The more nodes there are, the more total replication there will be. For example, when there are four nodes, then the data may be declustered at Level 1. Increasing the number of nodes to sixteen will mean the data must be declustered at Level 2 or lower, which causes the cells at Level 1 as well as Level 0 to now be replicated. However, if the data is declustered at Level 5, then no more cells will be replicated if four or sixteen nodes are used, but the cells that are replicated must be sent to more nodes. For example, the cell at Level 0 will go to sixteen nodes instead of four.

In general, $R$, the degree of replication, by level is:

$$R = \begin{cases} 4^n : & t \leq l - n \\ 4^{l-t} : & l > t > l - n \\ 1 : & t \geq l \end{cases}$$

where:

$R$ is the number of nodes each cell is sent to.
$l$ is the declustering level.
$t$ is the current level.
$4^n$ is the number of nodes.

This indicates that from Level 0 to Level $l - n$, every cell is sent to every node, between Level $l - n$ and Level $n$, every cell is sent to either 4, 16, 64, or so on, nodes, and at the declustering level and lower, the cells are not replicated, and are sent to only one node each.

When declustering data, it is possible to decluster at a level where the number of cells equals the number of nodes, or to decluster at a level where there are more cells than there are nodes, which allows us to to use virtual processors. The use of virtual processors usually allows the data to be distributed more evenly among the nodes, but requires more replication.
3.1.1 Effects of Replication

Replication has three effects. First, the remote nodes must read more data, which increases the I/O they must perform, increasing their total execution times. Second, the communication costs are increased. Communication costs are not relevant where the declustering is to be done for a large number of operations, but can be important where the declustering is done for a single operation. The experiments described below specify the extra communication and I/O costs of replicating data.

The third effect is that results may contain replicated join-pairs. If several nodes receive some of the same cells, then each of these nodes may calculate the same join-pairs. To avoid this, the remote nodes must either be aware of the replication of cells, or duplicates must be removed when the result sets of the remote nodes are reconciled.

3.1.2 Algorithm to Avoid Replication

Each node can be informed as to which of the cells that it receives are replicated, and can therefore avoid reporting duplicate pairs in the query results. A simple way to do this, when the data has been declustered at one specific level, is to send each remote node two pieces of information: the number of nodes and the level at which the data was declustered. If each node also knows what number it is, it can calculate which pairs of cells it is responsible for reporting the join-pairs of, using a round-robin ordering.

An example of this is where there are eight nodes, and the data is declustered at Level 3 using HCAM, as shown in Figure 3.1. The numbers in the cells represent which nodes receive each cell.

<table>
<thead>
<tr>
<th>Level 3</th>
<th>Level 2</th>
<th>Level 1</th>
<th>Level 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 6 4 7 3 2 5 6</td>
<td>4 5 1 2</td>
<td>0 - 7</td>
<td>0 - 7</td>
</tr>
<tr>
<td>7 0 6 7 2 3 4 5</td>
<td>6 7 2 3</td>
<td>0 - 7</td>
<td>0 - 7</td>
</tr>
<tr>
<td>3 4 5 6 1 2 3 4</td>
<td>4 5 1 2</td>
<td>3 6 7 2</td>
<td>0 - 7</td>
</tr>
<tr>
<td>0 1 6 8 9 10 11 12</td>
<td>5 6 7 1</td>
<td>2 3 6 7</td>
<td>0 - 7</td>
</tr>
</tbody>
</table>

Figure 3.1: Distribution of Fragments to 8 Nodes Using HCAM

Cells at Levels 3 and lower, in this case, are not replicated. Therefore, cells at these levels can be compared with other cells at these (non-replicated) levels without generating duplicate join-pairs. As well, cells at these levels can be compared with cells at the higher (replicated) levels without generating duplicates, since of the two cells considered, only one is replicated. For the replicated levels, however, only one node may generate join-pairs for any pair of cells. Therefore, each node is responsible for a certain set of cells at each level. These cells are compared to the corresponding cells in the other dataset at the same and higher levels.

Figure 3.2 shows which nodes are responsible for which replicated cells in this case. The responsible nodes are found by considering, at each level, which set of cells are given
to which set of nodes, and assigning responsibility round-robin. For example, Figure 3.1 shows that at Level 2, there are eight cells that are given to Nodes 0 - 3. The responsibility for these are assigned round-robin, so that each of these four nodes is responsible for two cells at Level 2. At Level 1, each of the four cells is given to all eight nodes, so the responsibility is assigned round-robin to the eight nodes, which covers only the first four. Similarly, all eight nodes receive the single cell at Level 0, but only one node is responsible for this cell. Given this, Node 1, for example, will receive eight cells at Level 2, all four at Level 1 and the single cell at Level 0. Of these, it is responsible for two cells at Level 2, and one cell at Level 1.

Figure 3.3 shows the pairs of replicated cells that Node 1 will examine for join-pairs. The cells with thick edges are the cells for which Node 1 is responsible. Since each node receives the same set of cells for both datasets, the same cells are replicated for both datasets, and the same nodes are responsible for the replicated cells for both datasets. At Level 2, Node 1 is responsible for two cells. These it compares to all relevant cells in the other set, whether it is responsible for the other cells or not. These two cells are each compared to exactly one cell at each of the equal and higher levels of the other dataset, the same as would be done normally. The same is true of Level 1; the single cell for which it is responsible at this level is compared between the two data sets and both instances of this cell are compared to Level 0 in the opposite data set.

### 3.2 Declustering for Range Queries

With range queries, there is no issue of replication. As long as each object in a dataset is sent to exactly one node, then a query window can be given to each node, and the union of their result sets will be the correct answer to the range query. If the data is being declustered simply to perform range queries, then it is not necessary to ensure that, for two datasets, the corresponding cells go to the same nodes. It is also not necessary to ensure that, if one cell is given to a certain node, the corresponding cells at the other levels of the dataset are sent to the same node. Therefore, it is much simpler to balance the load when data is to be declustered for only range queries than when the data is to be declustered for joins.

With range queries, the goal is not to achieve good load balancing across the nodes, but is to achieve good load balancing within each query window across the nodes. Ex-
The experiments below demonstrate that this is done by ensuring that an approximately equal number of objects within each query window are found at each node. To achieve this, the data should be divided into fragments that are as small as possible, and these should be distributed such that fragments that are physically close to each other are, as much as possible, sent to different nodes. This ensures that, on average, the objects found within any given query window are spread evenly over the nodes. This is true whether the data has a uniform or skewed distribution.

### 3.3 Declustering for Joins

The differences in declustering for joins, as opposed to range queries, are: (i) both datasets must be declustered in precisely the same way; (ii) each level of a dataset must be distributed to match the declustering at the declustering level; (iii) data must be replicated; and (iv) there is not the issue of wanting to place neighboring fragments on different nodes.

Also, Join Product Skew must be considered. Join Product Skew occurs when both
Datasets are skewed in the same way, which causes the nodes that receive the dense areas of the data space to have larger output sets than the other nodes. When this may occur, the only thing that can be done is to divide the dense areas of the data space as evenly as possible, which implies declustering at a low level. Partition Skew is also an issue with range queries, but since join queries can result in $O(n^2)$ output sizes, the problem can be more severe.

When declustering for joins, the goal is to minimize the response time. This means minimizing the work done by the node given the largest amount of work, which is achieved by balancing the two sub-goals of assigning the nodes equal work and minimizing the replication.

### 3.4 Declustering for Range Queries Followed by Joins

In general, the goals for range queries and joins cannot be met simultaneously, and we must either decluster just for range queries, and re-decluster for joins, or we must decluster while balancing the goals for range and joins queries such that the overall time for combined queries is minimized.

This leads to two possible strategies to perform combined queries. The first is to perform the initial declustering as if only range queries were to be performed. When range query results are to be joined, each node must send its result set back to a single node, which then re-declusters the data in a way that is optimized for this particular join. This will be referred to in the sequel as the "Range-based Strategy".

The second is to perform the initial declustering as if only joins were to be performed, meaning that each dataset is declustered in the same way, cells are replicated and so on. This would, in general, make the performance of the range queries less efficient, and would cause there to be replication after the range query step as well as after the join step, but would allow the joins to be performed without re-declustering. This will be referred to in the sequel as the "Join-based Strategy".

As is stated above, when declustering to optimize the range queries, the same method is used whether the data is uniform or skewed and we decluster at the lowest level possible. However, when performing the initial declustering to support joins, or when re-declustering for a single join operation, the data skew is relevant. With uniform distributions, the method by which to decluster is straightforward: the highest possible level should be used. That is, the highest level such that there are as many cells as there are nodes should be used. This will minimize the replication, and since the data is uniform, each node will have an approximately equal workload. Since uniform data allows us to use a number of fragments equal to the number of nodes, there is no issue as to how to distribute the fragments; each node receives exactly one, and all other levels are distributed accordingly.

However, with skewed data, it is probable that declustering at the highest level possible will result in load imbalance. The more skewed the datasets are, the lower the level that must be used to approach an equal load across the nodes. We wish to minimize the maximum amount of work done by any node. The lower the level at which the data is declustered, the more evenly balanced the nodes will be (unless the data happens to be
to have its densest areas spaced evenly at the same interval at which we decluster), but the greater the replication. These two factors must, therefore, be balanced to find the ideal level at which to decluster, where we can achieve the min-max workload.

3.5 Dividing the Data Space Into Fragments

As is stated above, when declustering to optimize for range queries, the technique used should divide the data into fragments that are as small as possible, and decluster these such that the fragments that are close to each other tend to be sent to different nodes. This would suggest the best way to divide the data space is to, at each level, consider each cell a fragment and distribute these using a technique such as HCAM or CMD. This allows some large fragments at the higher levels, but there is typically little data at the higher levels, and the majorit of the fragments will be quite small. It may be preferable to project every object onto a very low level based on the centre of the object, and to decluster every object at this level. This would allow objects that are at high levels to be distributed to different nodes, despite being assigned to the same cell at the higher level. However, experiments below indicate that distributing each object based on the level to which it belongs can achieve very good balance.

When performing the initial declustering to allow joins, and when re-declustering data for a single join, the best technique to use for declustering is not as straightforward. Four techniques to divide the data space into fragments are presented here: Slicing, Brute Force Division, Division Based on Skew, and Variable Fragment-Size Division. Brute Force Division and Division Based on Skew both choose some level at which to decluster, and therefore use virtual processors and require choosing a method to distribute the fragments to the nodes.

3.5.1 Slicing

One simple declustering method is a variation of Range Partitioning which we have called Slicing. When performing a declustering using Slicing, we wish to find \( n - 1 \) boundaries between fragments, where \( n \) is the number of nodes, such that each fragment has approximately the same number of objects. Finding the \( n - 1 \) boundaries can be done by creating a multi-dimensional histogram, where the data space is represented at some granularity, and by projecting every object in the two sets onto this histogram. One quicker technique to locate the boundaries, assuming the data has already been divided into level files and sorted, is to locate the largest level files of the two datasets and to slice these into equal-size parts. This is likely to be not only quicker than using histograms, but also more accurate.

A simple example of Slicing is a self-join where the data is to be declustered to four nodes and where the largest level file is at Level 10. The slicing algorithm would first find the size of this level file. If it is, for example, 16K bytes, then 4K would be sent to each remote node. The algorithm would have to, however, first check that the divisions made in this manner do not split any cells. That is, any objects in the same cell should be sent to the same node. Therefore, the amount of data sent to each node would not be
necessarily exactly 4K, but should be very close. This method assumes that each object requires the same number of bytes, as is the case with the implementation used in this thesis.

Level 10 has 1,048,576 cells, numbered 0 through 1,048,575. If, splitting in this way, we found, for example, the first 4K covered Cells 0 to 8000, the next 4K covered cells 8001 to 8600, the following 4K 8601 to 1,047,000, and the last 4K 1,047,001 to 1,408,575, then the four nodes would receive these four ranges of cells at Level 10, and all other levels would be declustered accordingly.

Using Slicing would often allow there to be little replication, since it generates only the minimum necessary number of fragments at the declustering level. However, Slicing does allow unnecessary replication to occur, since it can result in fragments with non-regular shapes, as shown in Figure 3.4 (a). Therefore, Slicing may be more suited to situations where replication is not a concern, for example, where it is possible to perform the slicing at a level such that there is very little data in the higher levels.

If replication is a concern, it can be reduced here by making the shapes more square. Figure 3.4 (b) shows how (a) can be modified only slightly to require less replication. Another method to reduce the amount of replication, is to simply slice a level file at a higher level.

![Figure 3.4: Declustering Data to Two Nodes Using Slicing](image)

### 3.5.2 Brute Force Division

Brute Force Division attempts to balance the two sub-goals of declustering for joins: load-balancing and reducing replication. It may also be used to balance these two sub-goals for combined range and join queries, and so may be used to find the best division into fragments when anticipating a certain proportion of range queries and joins. It may, then, be used for the initial declustering, where we do not wish to re-decluster for joins.
Brute Force Division seeks to find the best level at which to decluster, where each cell represents one fragment and the fragments can be distributed to the nodes in some manner (such as CMD or List Scheduling), and each other level is declustered accordingly. This is the level where we have the min-max expected workload at any node.

If considering only a join operation, to determine the best level at which to decluster, the first step is to pick a technique to distribute the fragments to the nodes. We can then estimate the workload of each node if the data is declustered at the highest possible level (for example, if there are sixteen nodes then the highest level at which we can decluster is Level 2), and find the maximum of these workloads. We can then perform the same estimate for each level below this level. We would then know which level gives us the min-max workload, and can choose this to be the best level at which to decluster.

If considering a series of range queries and joins, in the same way, we consider each level, from the highest possible level, to the lowest, and find which of these levels gives the best response time for this set of queries. If we are declustering for an indefinite number of queries, we cannot find the best declustering level with complete accuracy. But if we have a small set of queries that are representative of the queries that will be performed, then we can estimate the best declustering level by finding the best declustering level for these queries. This can be done by, at each level, finding the sum, over all queries, of the maximum times taken by any of the nodes. The best declustering level is the one for which this sum is the smallest.

### 3.5.3 Division Based on Skew

When considering joins, the more skewed the data is, the lower the level at which it is appropriate to decluster. It can be determined experimentally what the most appropriate level at which to decluster is, given the skew of the two datasets. Using Division Based on Skew involves measuring the skew of two datasets and choosing a declustering level based on some heuristic.

#### Measuring Skew

To measure the skew of a dataset, the entire dataset may be examined or it may be sampled. For the Filter Tree and S3J, the quickest way to acquire a sample of the data is to use one of the level files, the most accurate being the largest. This assumes that the data at each level is distributed in roughly the same manner, that is, larger objects are not distributed differently than smaller objects. This may not always be true, but this method is faster than sampling each of the level files, and is likely accurate enough to perform a single join operation.

Given a sample, several methods of measuring skew exist. Two possibilities considered here are standard deviation and percentiles. The standard deviation is calculated by reading in every object in the largest level file and projecting these onto an appropriate level. For this thesis, we use Level 6. If the largest level file is used, this will generally be a level lower than Level 6, usually between Levels 13 and 18 (at least in the implementation studied here, which has twenty levels). If the level that is used is lower than Level 6, then each cell at this level can be mapped directly to one cell at Level 6. If the majority
of the data for a dataset is at levels higher than Level 6, then an alternative technique must be used. This can be easily developed, but since it is considered a rarity, it is not pursued here.

A data structure representing the cell occupancies of the 4096 cells at Level 6 is then created and populated based on the data in the largest level file. The standard deviation of the number of records per cell within this data structure can then be calculated. The larger the standard deviation, the more skewed the data.

If the same data structure is sorted in decreasing order of occupancies, then various percentiles can be calculated. For example the 10th percentile can be calculated by finding how many of the 4096 cells contain the first 10% of the objects. Regardless of the percentile used, the smaller the percentile, the more skewed the data. Standard deviation is faster to compute, since it does not require the sort operation.

### 3.5.4 Variable Fragment-Size Division

This method starts by dividing the data at the highest possible level, given the number of nodes, and then estimating the workload for each node. If the loads are too imbalanced (that is, not within a certain predefined tolerance on their differences), then an area belonging to the node with the largest workload is divided into four parts (assuming the data space is two-dimensional). The fragment that is divided into four is the fragment representing the largest portion of this node's workload, and the work that this area represents is subtracted from the total work for this node. The four fragments are then, one at a time, given to the nodes with the smallest workloads, and the estimated work for each node is recalculated, considering the increased replication, after each reassignment. The process continues until the min-max workload is no longer improved, or all workloads are within the predefined tolerance.

![Figure 3.5: Division of Data Space For Four Nodes Using Variable Fragment-Size Division](image)

Figure 3.5 gives an example of how the data may be declustered to four nodes. Since there are four nodes, the data space would initially be divided into four fragments, which implies declustering at Level 1, as shown in Figure 3.5 (a). If the workloads are not within the given tolerance, the node with the largest workload will have its fragment divided into four. If this is node 1, then the work associated with this fragment would be subtracted from Node 1's estimated workload. We now must assign the four new fragments to nodes. One at a time, we calculate which node has the smallest workload.
and assign one fragment to this node, then update its estimated workload. This can give a declustering as shown in Figure 3.5 (b). If the workloads are still not within the given tolerance we must again find which node has the greatest workload. If this is Node 3, then we must find which of its two fragments represents the greatest amount of its work. This fragment is then divided into four and these new fragments are assigned to nodes, in this case giving three to Node 0, and one to Node 2, as shown in Figure 3.5 (c). If the load balance is still not within the given tolerance, the process will execute this step again, giving the declustering shown in Figure 3.5 (d). If this gives a load balance within the given tolerance, the process will terminate.

This technique is useful where only parts of the data space are densely populated and need to be divided at a fine level. The areas that are not densely populated are divided at a coarse level. Therefore there is only replication in the areas of the data space that require fine divisions.

### 3.5.5 Comparison of Methods to Divide Data into Fragments

Slicing allows the data within even large query windows to be placed on a single node, or on only a small number of nodes, and so is not appropriate for performing the initial declustering. Slicing is, therefore, only useful in the case where data is being re-declustered for a single join. It could likely be performed quite quickly, but since it does not consider replication, it can result in poor performance in some cases. However, it would likely, on average, require no more replication than HCAM, CMD and so on. Where it is appropriate, Slicing is the fastest method to decluster, and the most effective way to equally divide a dataset among multiple nodes. Slicing is not investigated in this thesis, due to time constraints, but is expected to perform well under at least some conditions.

Variable Fragment-Size Division can give better load balancing than the other methods, but since it allows for large fragments, it can also lead to very poor load balancing for range queries. It is therefore not suited for the initial declustering. As well, due to the fact that it is a much more expensive operation than Slicing or Division Based on Skew, it is not suited to re-declustering for single joins. We believe it would be the best technique where two datasets are declustered to perform a large number of join operations, but since this case is not considered in this thesis, no experiments using Variable Fragment-Size Division are performed.

Brute Force Division is expensive to perform, and is therefore not appropriate to re-decluster for a single join, but will work well for the initial declustering, where the time to calculate the best declustering level is less important.

Division Based on Skew, by definition, cannot be as accurate as Brute-Force Division, but can be performed much faster. Therefore, it is not appropriate for the initial declustering, but may be for re-declustering for single joins.

It appears then that the best choices for the initial declustering are to decluster at Level 19 (the lowest level) when using the Range-based Strategy, and Brute Force Division when using the Join-based Strategy. The best choice to re-decluster for joins appears to be Division Based on Skew. With Brute Force Division and Division Based on Skew, and when declustering at Level 19, the question remains of how best to distribute the
fragments to the nodes.

3.5.6 Comparison of Methods to Distribute Fragments

Since there are already a large number of good techniques to distribute fragments to nodes, including those discussed in Chapter 2, no new techniques were developed for this thesis. We will therefore compare the techniques above in terms of their suitability to the initial declustering using Brute Force Division, and to re-declustering using Division Based on Skew.

In general, of the methods we have considered, round-robin and hash functions are relatively fast to compute, List Scheduling gives good load balancing, and CMD, ECC, FX, HCAM and VECTOR are effective in placing fragments that are close to each other in the data space on different nodes.

Hash functions have almost the same advantages and disadvantages as round-robin. The additional step of calculating the bucket for each row typically causes them to take only marginally longer. Like round-robin, they are susceptible to data skew; they will allow some buckets to be far larger than others. We would, therefore, expect similar load balancing resulting from these. Given the nature of the Filter Tree and $S^3J$, performing round-robin declustering would be simpler if based on Hilbert values than if based on a coordinate of the data space. Using Hilbert values, round-robin is equivalent to HCAM. HCAM has been shown to result in good load balancing when used for range queries, and would likely result in good load balancing for join queries as well. Therefore, we can assume HCAM is preferable to round-robin based on other orderings, or hash-based methods.

List Scheduling would likely give the best load-balancing, but would not be suited for the initial declustering since it does nothing to prevent fragments that are physically close from being assigned to the same node, and could therefore result in poor performance for the range queries. It is also not suited to re-decluster for single joins, due to the time it requires. Therefore, the choice for the initial declustering, and to re-decluster is among CMD, ECC, FX, HCAM and VECTOR.

That HCAM, CMD, ECC, FX and VECTOR attempt to place fragments that are close to each other on different nodes is desirable when we only perform range queries, but, where we must replicate the data, this will also maximize the replication. However, if there are between four and sixteen nodes, and the data is two-dimensional, then only the degree of replication at the level immediately above the declustering level will be affected. All cells belonging to all other levels above the declustering level will be distributed to all nodes in any case.

All comparisons of HCAM, CMD, ECC, FX and VECTOR for range queries indicate either that they are roughly equal in quality, or that HCAM or VECTOR is the preferred method. VECTOR, from the tests performed, appears to be only marginally better than HCAM, is limited to two dimensions and is not as easy to apply to Filter Tree and $S^3J$ as HCAM. Therefore, of the techniques to distribute fragments, HCAM seems to be the best choice for range queries.

When using the Join-based Strategy, we must balance the goals of optimizing for range queries and optimizing for joins. To consider the joins, we should choose, again,
one of HCAM, CMD, ECC, FX and VECTOR. Since HCAM does the best of these in ensuring the physically close fragments are assigned to different nodes, it will result in the most replication, and is in this sense, the worst of the five methods. But, since it tends to place close fragments on different nodes, it would be the best in term of load balancing, since it would divide dense clusters of data among the nodes the most evenly. It is expected that this improvement in load balancing would more than offset the small increase in replication (relative to the other methods), and it, therefore, appears that HCAM is the best technique to use with Brute Force Division for the initial declustering.

When performing a re-declustering using Division Based on Skew, the choice is again among HCAM, CMD, ECC, FX and VECTOR. For the same reasons as stated with respect to the initial declustering, HCAM appears to be the best choice. As well, since the Filter Tree and $S^3J$ already store the Hilbert value for every object, HCAM can be performed faster than the other four methods, which is important when declustering for single operations.
Chapter 4

Experimental Setup

The experiments were performed on a cluster of sixteen RS6000 machines, each running AIX Version 4.1, and interconnected by a FORE ATM network. Each computer was an IBM RISC System/6000 Model 43P, 133 MHz with a 512 K L2 cache and 96 MB of RAM.

Communication was performed using the Message Passing Interface (MPI), a standardized and portable message-passing system which allows arrays of data to be sent between pairs of nodes, or to be broadcast to multiple nodes. The experiments were performed using MPICH Version 1.1.1, an implementation of MPI Version 2.0.

4.1 Synthetic Data Used

Two sets of synthetic data were used in all experiments. The first had a diagonal distribution. Five different skews, represented by their skew number, were used. The lower the skew number, the more concentrated the points were around the main diagonal, and therefore the more skewed the data was. To generate the five datasets, points were generated randomly (such that they were distributed uniformly through the data space) and were deleted with higher probability the farther they were from the main diagonal. For each skew number, the probability of being discarded increased with the distance from the main diagonal, but, given any distance from the main diagonal, the lower the skew number, the more likely points were to be deleted. A more complete description of the algorithm to generate the synthetic data is given in Appendix D. Table 4.1 shows, for each of the five skews used, the number of points discarded to generate a set of ten random points. The files of this set are referred to as Diagonal1, Diagonal2 and so on. These files were created with the following sizes: 10,000, 25,000, 50,000 and 100,000. These sizes were used since they are similar to the sizes of the real datasets used.

With skew $\# = 1$, almost all points are quite close to main diagonal. With skew $\# = 3$, most of data space is occupied, but is more dense around the main diagonal. With skew $\# = 5$, most of data space is occupied, and only the upper-left corner and lower-right corner are sparse. This is very close to a uniform distribution. The five Diagonal sets are shown in Figure 4.1.

The second set of synthetic data contains six files, each with 50,000 objects. These
files are not intended to be similar to real data; they were designed simply to estimate the time required for a join as a function of the characteristics of the data. These files are referred to as Data1, Data2, and so on, and are described in Table 4.2.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data1</td>
<td>Small rectangles evenly distributed, no overlap</td>
</tr>
<tr>
<td>Data2</td>
<td>Large rectangles evenly distributed, no overlap</td>
</tr>
<tr>
<td>Data3</td>
<td>Each rectangle a different size, extensive overlap</td>
</tr>
<tr>
<td>Data4</td>
<td>Small rectangles, each partially overlapping two or more others.</td>
</tr>
<tr>
<td></td>
<td>Data is contained in a smaller portion of the data space than Data1</td>
</tr>
<tr>
<td>Data5</td>
<td>Large rectangles, each partially overlapping two or more others.</td>
</tr>
<tr>
<td></td>
<td>Data is contained in a smaller portion of the data space than Data2</td>
</tr>
<tr>
<td>Data6</td>
<td>Similar to Data3, but greater overlap, and the data is contained</td>
</tr>
<tr>
<td></td>
<td>in a smaller portion of the data space</td>
</tr>
</tbody>
</table>

Table 4.2: Second Set of Synthetic Files

Two additional synthetic sets were used, called Uniform and SuperSkew. Uniform has 49,952 rows which are placed almost exactly evenly through the data space. Each row is a square rectangle, with the extent of each side being 0.0000095 of the extent of the data space. This size was used because it is a typical object size found in the real data sets. The squares are arranged in 223 evenly-spaced rows of 224 evenly-spaced columns. The SuperSkew set was created with various sizes, to have every object the identical size and location, located in the exact centre of the data space. The extents of the objects here are also 0.0000095 of the extent of the data space.

4.2 Real Data Used

We used five real datasets, referred to as LB, MG, Fluid, Citadel, and Islands for the experiments.

LB and MG are from the TIGER dataset, compiled by the U.S. Bureau of Census. LB represents roads in Long Beach County, California, and has 53,145 rows. MG represents
roads in Montgomery County, Maryland, and has 39,231 rows. Since these sets represent roads segments (each row corresponds to the MBR bounding one road segment), there is little overlap between the objects in any one set, and so no area of the data space is as densely occupied as occurs in datasets where there is overlap. For this reason, the TIGER sets may be considered not to be typical of spatial data. However, it is possible that they are typical of datasets that have the inherent property of having little overlap, such as datasets representing roads, powerlines, rivers, coastlines and so on.

Fluid is a subset of a set of points obtained from a computational fluid dynamics simulation. Specifically, the points are solutions of a Naiver Stocks equation solving the flow for a Boeing 747 landing. Fluid was originally a 3-dimensional set, and has been projected onto two dimensions. It contains 120,206 rows. The objects are densely grouped in three clusters, all towards the centre of the data space. The data space is quite sparse other than these three clusters.

The Citadel dataset was compiled by the Brazilian telephone company and represents the geographic location of telecommunications facilities. It has 66,837 rows. The data occupies the entire data space, but is significantly denser in the upper-right quarter. There is one particularly dense cluster in the upper-right corner of the data space, occupying approximately 1/64 of the data space.

Islands is from the Sequoia dataset. This file represents the MBRs for polygons representing the location of islands. The data occupies the entire data space, but is significantly denser along the top and right edges. It has 14,375 rows.

For each real dataset, the data was normalized to fill the entire data space. In general, it is possible, in some cases before normalizing, for the data to occupy what is roughly a rectangular area, but for the grid to be positioned at an angle to this rectangle. In this case, when the data is normalized so as to fit just within the data space, there would result some empty space. It is possible, then, before normalizing the data, to determine if the data points occupy a roughly rectangular area, and to align the grid with this rectangle. This was not done for these datasets.

Plots of the five diagonal and five real sets are shown in Figure 4.1. Figure 4.1 shows 100 points for each set. These points were found by sorting the sets by Hilbert value, and selecting the points at regular intervals.

**Modified Real Data**

We also altered the skew of the LB and MG datasets by moving the data towards the main diagonal. These real datasets were skewed to three different extents: 25%, 50%, and 75%, meaning each object was moved 25% of way from its actual position to the main diagonal, 50% of the way or 75% of the way. The data moved 75% of the way was the most concentrated around the main diagonal and therefore was the most skewed. Doing this allowed us to perform experiments where data is skewed to different degrees, but is still realistic, in the sense that we started with actual data.

Table 4.3 shows the standard deviation and three percentiles for each of the real and diagonal sets, and the modified LB and MG sets. The standard deviations and percentiles are calculated in the manner described in Section 3.5.3. Table 4.3 demonstrates that the standard deviation and percentiles each give consistent and reasonable estimates of the
skews of the sets. In the sequel we will use only standard deviation to measure skew.

![Figure 4.1: Diagonal and Real Datasets](image)

<table>
<thead>
<tr>
<th>File</th>
<th>Std. Dev.</th>
<th>P10</th>
<th>P25</th>
<th>P50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>4.12</td>
<td>2.90</td>
<td>8.42</td>
<td>19.39</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>3.13</td>
<td>4.59</td>
<td>13.00</td>
<td>30.32</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>2.72</td>
<td>5.49</td>
<td>15.82</td>
<td>37.39</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>1.89</td>
<td>8.53</td>
<td>24.97</td>
<td>59.80</td>
</tr>
<tr>
<td>Diagonal5</td>
<td>1.15</td>
<td>12.69</td>
<td>38.02</td>
<td>93.29</td>
</tr>
<tr>
<td>LB</td>
<td>3.91</td>
<td>2.63</td>
<td>8.55</td>
<td>20.93</td>
</tr>
<tr>
<td>LB.shift25</td>
<td>4.59</td>
<td>1.99</td>
<td>6.67</td>
<td>16.31</td>
</tr>
<tr>
<td>LB.shift50</td>
<td>5.57</td>
<td>1.40</td>
<td>4.77</td>
<td>11.74</td>
</tr>
<tr>
<td>LB.shift5</td>
<td>7.57</td>
<td>0.82</td>
<td>2.73</td>
<td>6.47</td>
</tr>
<tr>
<td>MG</td>
<td>2.67</td>
<td>5.25</td>
<td>16.00</td>
<td>39.66</td>
</tr>
<tr>
<td>MG.shift25</td>
<td>3.10</td>
<td>4.14</td>
<td>12.32</td>
<td>30.71</td>
</tr>
<tr>
<td>MG.shift50</td>
<td>3.74</td>
<td>3.07</td>
<td>9.06</td>
<td>22.27</td>
</tr>
<tr>
<td>MG.shift75</td>
<td>5.15</td>
<td>1.70</td>
<td>5.06</td>
<td>12.50</td>
</tr>
<tr>
<td>Fluid</td>
<td>70.50</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Citadel</td>
<td>24.17</td>
<td>0.01</td>
<td>0.17</td>
<td>0.61</td>
</tr>
<tr>
<td>Islands</td>
<td>6.27</td>
<td>0.41</td>
<td>3.23</td>
<td>10.12</td>
</tr>
</tbody>
</table>

Table 4.3: Skews of Real, Synthetic and Modified Real Files
4.3 Query Windows Used

Five different query windows were used for range queries, shown in Figure 4.4. Each was square and centred in the centre of the data space. Query 1 (Q1) was the smallest, and Query 5 (Q5) was the largest. Figure 4.4 also shows the fraction of the data space each window covers.

Placing the query windows in the centre does make it simpler to decluster the data than where the query windows are away from the centre, in that the data may in some cases be declustered at a higher level. For example, if using a small window, such as Q1, if the window is centred and we decluster at Level 1, the window will intersect four cells, and depending on the declustering algorithm and the number of nodes, presumably four nodes will perform the query. However, if the window is off-centre, it will only intersect one cell. In general, centred windows allow the data to be declustered more evenly at high levels than off-centre windows. Nevertheless, at low levels, even small windows are declustered equally evenly, whether they are centred or not. For example at Level 9 there are 262,144 cells. Q1 covers 0.0016 of the data space, which is 419.4 cells. Given that this many cells are intersected by the window, it does not matter if the window is centred or not.

<table>
<thead>
<tr>
<th>Window</th>
<th>Lower Coord.</th>
<th>Upper Coord.</th>
<th>Fraction of Data Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.48</td>
<td>0.52</td>
<td>0.0016</td>
</tr>
<tr>
<td>Q2</td>
<td>0.40</td>
<td>0.60</td>
<td>0.04</td>
</tr>
<tr>
<td>Q3</td>
<td>0.35</td>
<td>0.65</td>
<td>0.09</td>
</tr>
<tr>
<td>Q4</td>
<td>0.30</td>
<td>0.70</td>
<td>0.16</td>
</tr>
<tr>
<td>Q5</td>
<td>0.25</td>
<td>0.75</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 4.4: Query Windows

4.4 Summary of Experiments

The experiments performed seek primarily to determine if Brute Force Division and Division Based on Skew using HCAM are effective ways to decluster the data, and to find which of the Range-based and Join-based Strategies, described above, is preferable under which circumstances.

If only range queries were to be performed, then one implementation would be the algorithm shown in Figure 4.2. The node that received the query would receive each of the partial range query results from the remote nodes. Since the data is declustered without replication, the combination of the partial results is the correct result set. Similarly, if only a single join query were performed, one implementation would be the algorithm shown in Figure 4.3.

Figure 4.4 shows an algorithm to perform two range queries and join their results using the Range-based Strategy. Comparing Figures 4.2, 4.3 and 4.4 reveals that the
- One node receives a query window
- Query window sent to all nodes
- Each node performs range query on its portion of the data
- Each node sends its result set to the original node
- The original node writes the result set to disk

**Figure 4.2: Algorithm to Perform Range Queries**

- Partition and sort the two datasets
- Determine best declustering level based on data skew
- Decluster data with replication to other nodes using HCAM
- Each node performs join on its portion of data, removing the duplicate pairs
- Each node sends its result set to the original node
- The original node writes the result set to disk

**Figure 4.3: Algorithm to Perform Join Queries**

- One node receives two query windows
- Send first query window to all nodes
- Each node performs range query on their portion of the data
- Each node sends its result set to the original node
- The original node writes the result set to disk
- Send second query window, and receive result set in the same way
- Partition and sort the two result sets
- Determine best declustering level based on data skew
- Decluster data with replication to other nodes using HCAM
- Each node performs join on its portion of data, removing the duplicate pairs
- Each node sends its result set to the original node
- The original node writes the join result set to disk

**Figure 4.4: Algorithm to Perform Two Range Queries and Join the Results: The Range-based Strategy**

The first part of this strategy is identical to the case where there are only range queries, and the second part is identical to where there are only joins.

An algorithm to implement the Join-based Strategy is shown in Figure 4.5. The first part of this strategy is the same as where only range queries are performed, except the range queries take longer due to the replication and, presumably, poorer load balancing; and the writing to disk is done by each node, instead of only a single node. We assume that the range queries are not necessarily performed simply to perform the join, and therefore the results are returned to the primary node, which writes these to disk. This may be done in parallel with the remote nodes beginning the second part, the join. The second part of this strategy is the same as the case where only joins are performed, except the partition and sort steps are performed by each node on a portion of the range query results, instead of by a single node on the entire range query results.
- One node receives two query windows
- Send first query window to all nodes
- Each node performs range query on their portion of the data
- Result sets are written to local disks
- Result sets are also returned to the primary node
- Send second query window, and perform range query in the same way
- Each node performs partition, sort and join steps on its portion of the range query results, removing duplicate pairs
- Each node sends its join results to the original node
- The original node writes the result set to disk

Figure 4.5: Algorithm to Perform Two Range Queries and Join the Results: The Join-based Strategy

In order to ultimately prove the efficiency of the Brute Force Division technique using HCAM, we demonstrate that, given a set of range and join queries to be performed on two datasets, the Brute Force Division algorithm is able to determine the best declustering strategy and level. For the Brute Force Division technique to determine the best strategy and level, it must be able to predict the time required by each of the steps given in Figures 4.4 and 4.5. Experiments are conducted to develop and prove the accuracy of formulas to predict each of these steps.

Experiments were done first to analyse just the range query and just the join query, on single nodes and on multiple nodes. Several experiments were performed with the Filter Tree and $S^3J$ on a single node to determine what portions of the overall execution times were taken by the different steps of the algorithms, and how long queries took when given datasets with different sizes and different skews. This allowed us to determine what affected the execution times for these queries, and so, to determine what would affect the execution times when they are declustered across multiple nodes. These experiments demonstrated that the major factor for determining the time for range queries is output size, and for join queries the major factors are input and output sizes.

Since Brute Force Division requires estimating workloads, and since workloads were shown to depend on output size, experiments are also done on single nodes to demonstrate the usefulness of histograms to estimate output sizes.

Experiments were then done, with range and join queries, to determine how long the various steps of performing multi-node queries take. For range queries, this meant determining how much time was spent at the remote nodes calculating the result set, and how much time was taken by the primary node receiving and processing the results. Experiments also determined how well the loads could be balanced given different skews, window sizes, numbers of nodes and declustering levels. As well, it was determined how much extra work is required when data is replicated.

For join queries, experiments determined the time required to decluster datasets, for the remote nodes to calculate the result sets, and for the primary node to receive and process the results. Experiments also determined how we can determine the best declustering level, to balance workload and minimize replication, given the data skew of
the two sets. Experiments then demonstrated that Division Based on Skew is an effective technique to decluster for single join queries.

Based on these experiments, we developed a model to estimate the times taken by range, join, and combined queries. Further experiments demonstrated the accuracy of these estimates, and also demonstrated that these estimates could be used by the Brute Force Division technique to accurately determine the best declustering level when using the Join-based Strategy, and that the Brute Force Division technique could determine accurately when the Range-based or Join-based Strategy is preferable.

4.5 Experiment 1. Time Requirements of I/O Operations

With the Filter Tree and $S^3J$, all I/O is done with the ReadPage and WritePage functions, which read and write blocks of 4096 bytes. Experiment 1 demonstrates the time requirements of these two operations.

The first test started with an empty file and performed 10,000 writes followed by 10,000 sequential reads on this file. Ten trials of the test were performed, and the times for the 10,000 writes and reads are given in Table 4.5. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Trial</th>
<th>Writes</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,348</td>
<td>3,537</td>
</tr>
<tr>
<td>2</td>
<td>6,950</td>
<td>6,038</td>
</tr>
<tr>
<td>3</td>
<td>2,350</td>
<td>5,741</td>
</tr>
<tr>
<td>4</td>
<td>4,648</td>
<td>3,238</td>
</tr>
<tr>
<td>5</td>
<td>4,957</td>
<td>4,341</td>
</tr>
<tr>
<td>6</td>
<td>3,172</td>
<td>7,038</td>
</tr>
<tr>
<td>7</td>
<td>2,849</td>
<td>3,241</td>
</tr>
<tr>
<td>8</td>
<td>5,551</td>
<td>5,440</td>
</tr>
<tr>
<td>9</td>
<td>6,750</td>
<td>6,339</td>
</tr>
<tr>
<td>10</td>
<td>3,951</td>
<td>4,441</td>
</tr>
<tr>
<td>avg</td>
<td>4,353</td>
<td>4,939</td>
</tr>
</tbody>
</table>

Table 4.5: Times To Read and Write 10,000 4096-Byte Pages (Times in ms)

This gives an average time per individual write of 0.435 ms and per individual read of 0.494 ms.

A second test was similar, except that within the loop for the reads and writes, we also perform a series of multiplications. The test was first run without the reads and writes, to determine the time taken by the multiplications, and this time was then subtracted from the total times reported. The test allowed us to determine if I/O takes less time if a large number of operations can be done consecutively, without any CPU calculations between the I/O operations. This test gave a time per individual write of 0.443 ms and and per individual read of 0.495 ms, and so, the two tests showed only slightly different
results for the read and write operations. The experiment, therefore, shows that the time to read one page is about \(0.43\) ms and the time to write one page is about \(0.49\) ms.

Since the computers used for this experiment use disk caching, the times given here represent the elapsed time to read and write data to and from the kernel buffer. The kernel buffer shares memory space with the virtual memory system dynamically and therefore changes size. In the environment used here, it is typically 16 MB. In the case of writes, the time to actually write the data to disk does not affect this time, since it occurs asynchronously later. Reads also access only the kernel buffer, unless the requested page is not in this buffer. Therefore the times presented here are not for actual disk reads and writes. They are, however, representative of the I/O times for the Filter Tree and \(S^3J\), since these benefit from the same disk caching. In the sequel we will use the terms logical read, logical write and logical-I/O to refer to reads and writes to the kernel buffer.

### 4.6 Experiment 2. Send and Receive Times Using MPI

The data that is declustered to perform range and join queries will consist of sets of integers. Experiment 2 determines the time to send arrays of integers between pairs of nodes. A sleep statement was added to the node that received the data, so that it slept for 100 seconds before it began to receive the data. This ensured that the times recorded for the receives include only the act of receiving, since the data was guaranteed to be already available.

The results are the average of ten runs, and are shown in Table 4.6. Each run gave identical results, within a second for each time. Times for sends and receives, and the times per byte are given in milliseconds. Integers are four bytes, and therefore Total Bytes is found by multiplying the number of arrays sent, by the array size, by four. Table 4.6 shows that the times to send and receive arrays of data are approximately equal. It also demonstrates that the time per byte depends on the array size, and so, it is faster to send data as a few large arrays than as many small arrays. Based on this experiment, we can determine the communication time for messages in the experiments below. Since larger messages may be sent faster, we can assume large messages will be sent when possible, and there is, therefore, generally a transfer rate of \(0.00018\) ms per byte.

### 4.7 Experiment 3. MPI Times for Different Numbers of Nodes

Experiment 3 involved sending arrays of 500 integers to 1, 4, 6, 8 or 12 nodes. Each time a total of 24 arrays were sent. The results of this experiment are shown in Table 4.7. This experiment indicates that, if a node is sending a fixed amount of data, it requires roughly the same amount of time to send to 4, 6, 8 or 12 nodes, as long as an equal number of equal-sized arrays are sent. It is marginally faster to have more receivers, probably due
Table 4.6: Time to Send and Receive Arrays of Data Using MPI (Times in ms)

<table>
<thead>
<tr>
<th>Sent Arrays</th>
<th>Array Size</th>
<th>Total Bytes</th>
<th>Send Time</th>
<th>Recv. Time</th>
<th>Time / Byte</th>
<th>MB / sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>1</td>
<td>40,000</td>
<td>3,013</td>
<td>2,994</td>
<td>0.075</td>
<td>0.013</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>400,000</td>
<td>30,333</td>
<td>30,311</td>
<td>0.075</td>
<td>0.013</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1</td>
<td>4,000,000</td>
<td>285,941</td>
<td>285,977</td>
<td>0.071</td>
<td>0.013</td>
</tr>
<tr>
<td>100,000</td>
<td>10</td>
<td>4,000,000</td>
<td>17,010</td>
<td>17,045</td>
<td>0.0043</td>
<td>0.235</td>
</tr>
<tr>
<td>10,000</td>
<td>100</td>
<td>4,000,000</td>
<td>2,136</td>
<td>2,103</td>
<td>0.00053</td>
<td>1.878</td>
</tr>
<tr>
<td>5,000</td>
<td>1000</td>
<td>20,000,000</td>
<td>3,872</td>
<td>3,875</td>
<td>0.00019</td>
<td>5.168</td>
</tr>
<tr>
<td>2,500</td>
<td>2000</td>
<td>20,000,000</td>
<td>3,907</td>
<td>3,902</td>
<td>0.00019</td>
<td>5.128</td>
</tr>
<tr>
<td>5,000</td>
<td>2000</td>
<td>40,000,000</td>
<td>7,748</td>
<td>7,738</td>
<td>0.00019</td>
<td>5.168</td>
</tr>
<tr>
<td>2,500</td>
<td>4000</td>
<td>40,000,000</td>
<td>7,169</td>
<td>7,167</td>
<td>0.00018</td>
<td>5.586</td>
</tr>
</tbody>
</table>

Table 4.6: Time to Send and Receive Arrays of Data Using MPI (Times in ms)

to greater parallelism, but the difference is small.

Table 4.7: Sending to Various Numbers of Nodes Using MPI

<table>
<thead>
<tr>
<th>Number Nodes Receiving</th>
<th>Arrays Per Receiver</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>18.5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10.5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10.3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>10.1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>10.1</td>
</tr>
</tbody>
</table>

Table 4.7: Sending to Various Numbers of Nodes Using MPI

4.8 Implementation Details

Performing multi-node range and join queries required expanding the code for the Filter Tree and $S^3J$. For the Filter Tree, approximately 1,200 lines were added to perform the initial declustering, to return result sets, to receive results, to estimate selectivities, and to remove duplicates. With the $S^3J$ code, approximately 800 lines were added to measure skew, estimate selectivity, decluster, return results, receive results, and remove duplicate join-pairs.

4.8.1 Message Passing

Once the decision is made regarding how to perform the declustering, and the times for logical-I/O operations and communication are known, we must decide the specific
implementation details of how data is sent among nodes, how queries are sent to the other nodes, and how result sets are reconciled.

For this thesis, we send messages of up to 5000 objects when returning result sets from range queries, messages of up to 2500 pairs of objects when returning result sets from join queries, and messages of up to 5000 objects when re-declustering for single joins.

The Filter Tree and $S^3J$ both store objects as four integers and a double. The four integers represent the $x$ and $y$ coordinates of the lower-left and upper-right corners of the rectangle, and the double represents the Hilbert value. Each object also has a level. With the Filter Tree this is maintained by the B-tree structure, and with $S^3J$ this is effectively stored by keeping each object in the appropriate level file. When data is re-declustered for a single join, we must send at least the coordinates of each object. We may also send the level and Hilbert value, or can allow the remote node to recalculate these. In the implementation developed here, each message sent to the remote nodes contains the level of the objects, which requires that all objects within any message must be of the same level. Within the messages, each object is sent as five integers, representing the $x$ and $y$ coordinates and the Hilbert value. Because Hilbert values may be too large for an integer, a flag is sent when the value is too large, and the remote node will calculate the Hilbert values for these objects. Experiments showed that this occurred relatively rarely.

When returning result sets, we send only four integers for each object, representing the $x$ and $y$ coordinates of the objects.

When re-declustering for a single join, the minimum amount of data that must be sent is four integers per object. To calculate the optimum sending time, we consider all the data sent as one message, at a rate of 0.00018 ms per byte, with sixteen bytes per object (integers are four bytes each). Experiments below will compare the actual declustering time to this optimal time. Similarly, when returning result sets, the minimum time to send the results will occur if four integers per object are sent, and the results are sent as one message at a rate of 0.00018 ms per byte.

### 4.8.2 Estimating Selectivities

For both range and join queries, the selectivity is a key consideration in predicting the response time. It is therefore important, when performing multi-node queries, to be able to predict both the total selectivity for each query, and the selectivity at each node.

**Estimating the Selectivity for a Range Query on a Single Node**

To estimate the selectivity for a range query, a two-dimensional histogram is created representing the data space at Level 9, which is stored as a 512 by 512 matrix. All objects are then projected onto the histogram by calculating which cell at Level 9 the object's centre is within.

When a query window is given, we do not need to calculate the Hilbert values that it covers; we can simply find the row and column numbers at Level 9 that the window covers, and add the occupancies of each of the relevant cells in the histogram. With cells...
that are partially covered by the query window, we calculate what fraction of the cell is covered by the window, and multiply the occupancy of the cell by this fraction.

**Estimating the Selectivities for a Range Query on Multiple Nodes**

This is done in the same manner as estimating the selectivity on a single node, except, as we consider each cell at Level 9, we calculate which node will receive the cell. The workload for each node can be estimated then by finding the number of objects within a query window expected to be found at each node. An example is shown in Figure 4.6, where the data is declustered at Level 3, and the data is projected onto Level 3 (since this is easier to show than Level 9). To calculate the selectivities at the nodes, we calculate which cells are covered by the window, and which node these are sent to, as shown in Figure 4.6 (b). We then refer to the histogram to find the occupancies of these cells, as shown in Figure 4.6 (a). The predicted selectivities at each node are: Node 0: 16+12+38+35 = 101; Node 1: 73+49+65+56 = 243; Node 2: 36+85+36+54 = 211; and Node 3: 37+75+81+88 = 243.

![Figure 4.6: Estimating Selectivities for Range Query Using Four Nodes](image)

**Estimating the Selectivity for a Join Operation on a Single Node**

To estimate the selectivity of a join query, a histogram is created for each file, in the same manner as is done for range queries. The selectivity is then estimated by multiplying the occupancies of each pair of cells in the histogram, and by summing the multiplied occupancies. This sum is then divided by 1.3 since it was found experimentally that this is the fraction of objects within two corresponding cells at Level 9 that actually overlap, on average. An example is given in Figure 4.7. Again, we use a level higher than Level 9 for the figure, since this is easier to show. In this case we use Level 1. In this case the predicted selectivity will be \(((30 \times 14) / 1.3) + ((50 \times 28) / 1.3) + ((17 \times 80) / 1.3) + ((41 \times 62) / 1.3) = 4401.\)

**Estimating the Selectivities for a Join Operation on Multiple Nodes**

This is done in the same manner as estimating the workload for a join operation on a single node, with the same difference as between single and multiple nodes for range
queries: for each pair of cells at Level 9, we consider which node will receive these cells, and increment the estimated workload for this node.

As well, since the input sizes are also important to predict join query times, the histograms are also used to estimate how many rows will be sent to each node using a given declustering level.
Chapter 5

Experimental Results For Range Queries

Using Brute Force Division to determine, for a given set of queries, which strategy and initial declustering level are preferable requires predicting the range query response time at each node, the selectivities of the range queries, and the time required to return range query results to the primary node. This chapter presents formulas to predict each of these, and experiments to show the accuracy of these formulas.

Experiments 4, 5 and 6 demonstrate that the response time can be estimated from the output size. Experiment 6 does this for large output sizes; Experiments 4 and 5 do this with small result sizes, by using small input sets and small query windows respectively. Experiment 7 then demonstrates that histograms are effective to predict the selectivity of a range query. Experiments 4 to 7, then, demonstrate that we are able to predict the time required to perform a range query on a single node. To predict the time when using multiple nodes, we must also consider the time to return the results to the primary node. Experiment 8 demonstrates that the time to return the range query results can be predicted by predicting the size of the largest result set at any node.

This demonstrates that the multi-node range query’s time is minimized by assigning the nodes the min-max workload, which implies that to decluster efficiently, we must be able to estimate the workloads at the remote nodes. We therefore must determine how the workload at each node is affected by factors such as the number of nodes and the declustering level. Experiment 9 demonstrates that the workloads become more balanced as lower declustering levels are used, and with sufficiently low declustering levels, the number of nodes does not affect the load balance. Experiment 9 also demonstrates that by using the lowest level, Level 19, an excellent balance can be achieved for even the most skewed sets. This proves that declustering at Level 19 is an effective way to decluster for the Range-based Strategy. Since the Join-based Strategy requires replication, and replication also affects the workloads, it is therefore necessary to predict the increased workloads at the nodes given replication. Experiment 10 demonstrates that this can be done by considering the distribution of data across the levels. Experiment 11 then concludes this chapter, by demonstrating that the results found in Experiments 4 - 10 can be used to accurately predict the time for multi-node range queries.
5.1 Experiment 4. Varying the Size of the Dataset

Experiments 4 and 5 determine how the response times for range queries are affected by varying the size of the datasets and the size of the query windows. Experiment 4 involved range queries using Q2 on Diagonal1, Diagonal5, Fluid and Islands. Table 5.1 gives the response times for Diagonal1 and Diagonal5 using four different sizes of each dataset. Times are in milliseconds.

Table 5.1 also gives the number of Buffer References and Buffer Reads to perform the range query, and the ratio of the two. Buffer References count the number of times a data page is searched for some range of Hilbert values. Buffer Reads count the number of times a logical read is performed to read pages into the buffer. The implementation of the Filter Tree studied in this thesis uses a buffer of size 64 pages, where index pages are kept in with a higher priority than data pages. The datasets used here require either one or two levels of index pages for each level, and usually only one level of index pages. The index pages, therefore, normally fill approximately half the buffer, leaving the remaining half for data pages to be swapped in and out of memory.

When performing the range query, each page may be referred to many times. This is due to the fact that the implementation studied here searches the data by searching the levels one at a time, and within each level, searching the Hilbert ranges that cover the query window one at a time. When a page is referred to a second time, it may still be in the buffer, or it may have been swapped out. With smaller datasets, it is more likely that pages will remain in memory, since the total number of pages brought into the buffer may be less than or only slightly larger than the buffer size. Therefore, smaller sets tend to require fewer reads per reference.

Table 5.1 shows that the number of Buffer Reads per Buffer Reference increases with the size of the dataset. It was also found that for Islands, which is quite small, there were very few Buffer Reads per Buffer Reference, and with Fluid, which is quite large, the number of Buffer Reads was almost equal to the number of Buffer References. It is anticipated that eventually the number of data pages would become large enough that the buffer is comparatively very small and, therefore, only the index pages would tend to
remain in the buffer. Tests are done to discover whether this in fact occurs in Experiment 6.

Though smaller datasets tend to require fewer Buffer Reads per Buffer Reference, generally more data pages have to be referenced for each page of output. For this reason, smaller datasets tend to require more time per output page than large datasets. This experiment indicates that there is a general trend to require less time per output page for larger output sizes. However, in some cases there may be lower times per page for smaller output sizes, due to the lower rate of Buffer Reads per Buffer Reference.

5.2 Experiment 5. Varying the Query Window Size

<table>
<thead>
<tr>
<th>File</th>
<th>Window</th>
<th>#Pages Output</th>
<th>Total Time</th>
<th>Time per Output Page (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>Q1</td>
<td>4</td>
<td>35</td>
<td>8.7</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q2</td>
<td>49</td>
<td>297</td>
<td>6.1</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q3</td>
<td>82</td>
<td>473</td>
<td>5.7</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q4</td>
<td>111</td>
<td>649</td>
<td>5.8</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q5</td>
<td>143</td>
<td>821</td>
<td>5.7</td>
</tr>
<tr>
<td>MG</td>
<td>Q1</td>
<td>1</td>
<td>18</td>
<td>n.a.</td>
</tr>
<tr>
<td>MG</td>
<td>Q2</td>
<td>21</td>
<td>139</td>
<td>6.6</td>
</tr>
<tr>
<td>MG</td>
<td>Q3</td>
<td>49</td>
<td>299</td>
<td>6.1</td>
</tr>
<tr>
<td>MG</td>
<td>Q4</td>
<td>86</td>
<td>512</td>
<td>5.9</td>
</tr>
<tr>
<td>MG</td>
<td>Q5</td>
<td>118</td>
<td>678</td>
<td>5.7</td>
</tr>
<tr>
<td>Fluid</td>
<td>Q1</td>
<td>161</td>
<td>915</td>
<td>5.6</td>
</tr>
<tr>
<td>Fluid</td>
<td>Q2</td>
<td>682</td>
<td>4018</td>
<td>5.8</td>
</tr>
<tr>
<td>Fluid</td>
<td>Q3</td>
<td>692</td>
<td>4024</td>
<td>5.8</td>
</tr>
<tr>
<td>Fluid</td>
<td>Q4</td>
<td>696</td>
<td>4036</td>
<td>5.8</td>
</tr>
<tr>
<td>Fluid</td>
<td>Q5</td>
<td>699</td>
<td>4048</td>
<td>5.8</td>
</tr>
<tr>
<td>Islands</td>
<td>Q1</td>
<td>1</td>
<td>8</td>
<td>n.a.</td>
</tr>
<tr>
<td>Islands</td>
<td>Q2</td>
<td>1</td>
<td>20</td>
<td>n.a.</td>
</tr>
<tr>
<td>Islands</td>
<td>Q3</td>
<td>2</td>
<td>20</td>
<td>n.a.</td>
</tr>
<tr>
<td>Islands</td>
<td>Q4</td>
<td>3</td>
<td>32</td>
<td>10.6</td>
</tr>
<tr>
<td>Islands</td>
<td>Q5</td>
<td>4</td>
<td>46</td>
<td>11.5</td>
</tr>
</tbody>
</table>

Table 5.2: Varying the Query Window Size

In Experiment 5, we demonstrate the effect of varying the query window size on the response time. Table 5.2 gives the total times for queries using Diagonal1, MG, Fluid and Islands, for five different query windows.

Table 5.2 also shows the time per output page. (The notation "n.a." denotes the trials where the result sets were only 1 or 2 pages. In these cases, the page, or second page, may contain only a small portion of the potential capacity of a data page, and so
the time per page is not meaningful.) Table 5.2 shows that the time per output page tends to decrease as the query window is made larger and the output size becomes larger, though this isn't perfectly consistent.

Table 5.2 also indicates that the times per output page for Islands is more than for the other sets. This is due to the fact that the query window is located away from where the data is densest, and the output sets are small.

Experiments 4 and 5 indicate that, on average, the times per output page are smaller with larger result sets, which occur with larger datasets, larger query windows, and with query windows being located where the data is densest. They also indicate that, while this is the general trend, the times do fluctuate noticeably.

### 5.3 Experiment 6. Large Datasets

In Experiment 6, we demonstrate the response time for range queries when using large datasets. Table 5.3 shows the results of tests with large datasets using Q2. The experiment involved three different sizes of the sets Fluid, SuperSkew, Modified Fluid and Modified Super Skew. The SuperSkew set consisted of 20,000, 60,000 or 120,000 identical objects all located in the exact centre of the data space. This set is very artificial, and was created purely to study the case where no data pages are examined more than once, and all data pages that are examined have all their objects within the window, since these are characteristics of range queries on large datasets. The Fluid set had objects randomly removed in order to create files of the same three sizes. Modified sets were also created, in order to examine the effect of having data with different distributions both inside and outside of the query window. The modified sets were created by taking a random sample of either the Fluid or SuperSkew sets, and combining it with a random

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>#Pages</th>
<th>Total Time (ms)</th>
<th>Time per Output Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluid</td>
<td>20,000</td>
<td>114</td>
<td>672.6</td>
<td>5.9</td>
</tr>
<tr>
<td>Fluid</td>
<td>60,000</td>
<td>341</td>
<td>1875.5</td>
<td>5.5</td>
</tr>
<tr>
<td>Fluid</td>
<td>120,000</td>
<td>681</td>
<td>3541.2</td>
<td>5.2</td>
</tr>
<tr>
<td>SuperSkew</td>
<td>20,000</td>
<td>118</td>
<td>613.6</td>
<td>5.2</td>
</tr>
<tr>
<td>SuperSkew</td>
<td>60,000</td>
<td>353</td>
<td>1835.6</td>
<td>5.2</td>
</tr>
<tr>
<td>SuperSkew</td>
<td>120,000</td>
<td>706</td>
<td>3671.2</td>
<td>5.2</td>
</tr>
<tr>
<td>Mod. Fluid</td>
<td>20,000</td>
<td>88</td>
<td>572.0</td>
<td>6.5</td>
</tr>
<tr>
<td>Mod. Fluid</td>
<td>60,000</td>
<td>261</td>
<td>1513.8</td>
<td>5.8</td>
</tr>
<tr>
<td>Mod. Fluid</td>
<td>120,000</td>
<td>521</td>
<td>2709.2</td>
<td>5.2</td>
</tr>
<tr>
<td>Mod. SuperSkew</td>
<td>20,000</td>
<td>91</td>
<td>573.3</td>
<td>6.3</td>
</tr>
<tr>
<td>Mod. SuperSkew</td>
<td>60,000</td>
<td>270</td>
<td>1485.0</td>
<td>5.5</td>
</tr>
<tr>
<td>Mod. SuperSkew</td>
<td>120,000</td>
<td>539</td>
<td>2802.8</td>
<td>5.2</td>
</tr>
</tbody>
</table>

Table 5.3: Large Datasets (Times in ms)
sample from the Diagonal5 set, the most uniform of the Diagonal sets. This was done such that the objects from Diagonal5 composed 25% of the modified sets.

With small sets, each data page may contain only a small number of objects that are within the query window, which may require many data pages to be searched in order to find the objects belonging to the result set. This has the effect of increasing the time per output page for small datasets. As well, with small sets, as pages are checked multiple times for different ranges of Hilbert values, the pages tend to remain in the buffer, while with large datasets they tend to require a logical read for each reference. This has the effect of decreasing the time per output page for small datasets. Since there are factors both increasing and decreasing the time per output page for small datasets, relative to large datasets, it is expected that with small sets, the time per output page will vary depending on the distribution of the data.

With large datasets, however, each data page contains a relatively small range of Hilbert values, and for each range of Hilbert values (that cover the query window) that a page is checked against, generally more rows will be found within this range. When the dataset is sufficiently large, the number of Buffer References equals the number of Buffer Reads, and the number of Buffer References approaches the number of pages output. It is therefore expected that with large output sets, the time per output page will be smaller and more consistent than with small output sets.

The experiment showed that, with sufficiently large output sets, the time required per output page is quite consistent. In this case, it does not appear to matter how much data is outside the query window, whether the data inside the window is in one very dense cluster (as with SuperSkew) or three less-dense clusters (as with Fluid), or whether there is or is not a large amount of relatively uniformly distributed data mixed with the clusters (as with the modified sets).

Table 5.3 shows that for each of the four datasets, when the data size is 120,000, the time per output page is 5.2 ms. In general, Experiments 4, 5 and 6 indicate that, where the result set is large, about 5.2 ms per output page is required for the range query and where the result set is small, the time required is less predictable but is at most about 12 ms per output page. This is, then, approximately the range of times that may be required for a range query. For sufficiently large output sizes, we may define the response time as Formula (1):

\[
\text{Single-node Range Query Time} = (\text{size of output in pages}) \times 5.2 \text{ ms.}
\]  

(1)

Formula (1) depends on both the CPU and logical-I/O times of the environment. Regardless of the environment, though, the same amount of CPU and logical-I/O will be performed for each page of output. Therefore each environment will have a fixed time per output page depending on the local CPU and logical-I/O times.
5.4 Experiment 7. Estimating Output Size on a Single Node

Since the response time for range queries is dependent on the output size, it is necessary to predict the output size to predict the time for range queries. Experiment 7 demonstrates that this can be done accurately using a multi-dimensional histogram, and the technique described in Section 4.8.2.

Table 5.4 shows the results of range queries using real and modified real datasets. For each, the output size is shown, measured in number of pages. The table also gives the expected output size using the uniform assumption, and using a histogram. The output estimate based on the uniform assumption is found by multiplying the size of the dataset by the fraction of the data space covered by the query window. These fractions are given in Table 4.4.

The datasets with the largest standard deviations (as measured using the technique described in Section 3.5.3), the most skewed sets, are those where the uniform assumption gives the worst estimate of the output size. For example, the estimate based on the uniform assumption is closer for LB than LB.shift75 or Fluid. The estimates based on the histogram were quite accurate, with the larger query windows generally associated with the greatest accuracy.

The experiment was also performed with MG, MG.shift75, Islands, and the five Diagonal sets. Tests with the Diagonal sets used the sizes 10,000, 25,000, 50,000 and 100,000. The uniform assumption was again more accurate for MG than MG.shift75, and was more accurate for the less-skewed than the more-skewed Diagonal sets. In the cases where the data is densest where the query window is centred, the diagonal sets, modified real sets, LB, MG and Fluid, the uniform assumption gives an underestimate of the output size. For Citadel and Islands, the data is densest in areas outside the query window, and, therefore, the uniform assumption overestimates the output size.

In all cases, the histograms gave a good estimate of the output size. We can conclude from this experiment that the uniform assumption gives a poor estimate of the output size, and that histograms are a good means to estimate output size, regardless of data size, skew or window size.

5.5 Experiment 8. Performing Range Queries Using Multiple Nodes

Experiment 8 determines the time necessary to perform a range query using multiple nodes. This requires determining the time for communication and for the primary node to process the results received from the remote nodes. To do this, Experiment 8 performs a multi-node range query using Diagonal1, declustered to four nodes using HCAM at Level 5 without replication. The query window was Q5.

Figure 5.1 shows the results of Experiment 8. The first part shows the times taken by the primary node. Initially it reads in the query window, and sends this to the remote nodes, after which it begins to wait for the responses from the remote nodes. The primary
Table 5.4: Estimating Output Size on a Single Node Using Real and Modified Real Data

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Std. Dev.</th>
<th>Window</th>
<th>Uniform Estimate</th>
<th>Histogram Estimate</th>
<th>Actual Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>3.91</td>
<td>Q2</td>
<td>12.50</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>LB</td>
<td></td>
<td>Q3</td>
<td>28.13</td>
<td>76</td>
<td>74</td>
</tr>
<tr>
<td>LB</td>
<td></td>
<td>Q4</td>
<td>50.02</td>
<td>122</td>
<td>121</td>
</tr>
<tr>
<td>LB</td>
<td></td>
<td>Q5</td>
<td>78.15</td>
<td>168</td>
<td>169</td>
</tr>
<tr>
<td>LB.shift75</td>
<td>7.57</td>
<td>Q2</td>
<td>12.50</td>
<td>112</td>
<td>106</td>
</tr>
<tr>
<td>LB.shift75</td>
<td></td>
<td>Q3</td>
<td>28.13</td>
<td>136</td>
<td>134</td>
</tr>
<tr>
<td>LB.shift75</td>
<td></td>
<td>Q4</td>
<td>50.02</td>
<td>170</td>
<td>166</td>
</tr>
<tr>
<td>LB.shift75</td>
<td></td>
<td>Q5</td>
<td>78.15</td>
<td>171</td>
<td>169</td>
</tr>
<tr>
<td>Fluid</td>
<td>70.50</td>
<td>Q1</td>
<td>1.13</td>
<td>170</td>
<td>161</td>
</tr>
<tr>
<td>Fluid</td>
<td></td>
<td>Q2</td>
<td>28.28</td>
<td>690</td>
<td>682</td>
</tr>
<tr>
<td>Fluid</td>
<td></td>
<td>Q3</td>
<td>63.64</td>
<td>696</td>
<td>692</td>
</tr>
<tr>
<td>Fluid</td>
<td></td>
<td>Q4</td>
<td>113.13</td>
<td>698</td>
<td>696</td>
</tr>
<tr>
<td>Fluid</td>
<td></td>
<td>Q5</td>
<td>176.77</td>
<td>699</td>
<td>699</td>
</tr>
<tr>
<td>Citadel1</td>
<td>24.17</td>
<td>Q2</td>
<td>15.73</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Citadel1</td>
<td></td>
<td>Q3</td>
<td>35.38</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>Citadel1</td>
<td></td>
<td>Q4</td>
<td>62.90</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>Citadel1</td>
<td></td>
<td>Q5</td>
<td>98.28</td>
<td>43</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 5.4: Estimating Output Size on a Single Node Using Real and Modified Real Data

node is, after it sends the window, either waiting for a message or processing a message. To process a message, the data is put into data pages, which are logically written to disk.

The other nodes send the primary node arrays of up to 5000 rows. Here, all four nodes sent one array of 5000 rows followed by another array of less than 5000 rows. The first array, therefore, takes longer for each node to create and send than the second. Figure 5.1 shows that the primary node receives the messages approximately 45 ms after the remote nodes have finished sending them.

An output page contains 170 rows, while a message contains 5000 rows, 29.4 times as many rows. Since logical writes on the output pages requires approximately .43 ms, the primary node requires $29.4 \times .43 \text{ ms} = 12.6 \text{ ms}$ to process each message. The observed times here were close to this: 9.3, 10.7, 9.1 and 10.7 ms.

After processing a message, how long the primary node waits to receive the next message depends on how long the next node takes to create and send the array to the primary node. If the result set at each node is large, requiring many messages of 5000 rows to be sent, then the time between sends, for each node, should be fairly consistent, as should the time spent waiting for receives to complete for the primary node. Assuming each remote node has a sufficiently large result set that it would normally take approximately 5.2 ms per output page if it were performing as a single node, then the remote nodes should take approximately $5.2 \times 29.4 = 153 \text{ ms}$ to create each message they send to the primary node. In this case the result sets were too small to average 5.2 ms per page, and the nodes averaged 210 ms to create their first messages, which is equivalent to about
Primary Node

<table>
<thead>
<tr>
<th>Event</th>
<th>Time (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>send query window to remote nodes</td>
<td>2.7</td>
<td>2.7</td>
</tr>
<tr>
<td>receive first message from node 1</td>
<td>257.8</td>
<td>260.5</td>
</tr>
<tr>
<td>process message</td>
<td>9.3</td>
<td>269.8</td>
</tr>
<tr>
<td>receive first message from node 2</td>
<td>24.7</td>
<td>294.5</td>
</tr>
<tr>
<td>process message</td>
<td>10.7</td>
<td>305.2</td>
</tr>
<tr>
<td>receive first message from node 3</td>
<td>5.7</td>
<td>310.9</td>
</tr>
<tr>
<td>process message</td>
<td>9.1</td>
<td>330.0</td>
</tr>
<tr>
<td>receive first message from node 4</td>
<td>4.9</td>
<td>324.9</td>
</tr>
<tr>
<td>process message</td>
<td>10.7</td>
<td>335.6</td>
</tr>
<tr>
<td>receive second message from node 1</td>
<td>1.9</td>
<td>337.5</td>
</tr>
<tr>
<td>process message</td>
<td>2.6</td>
<td>340.1</td>
</tr>
<tr>
<td>receive second message from node 2</td>
<td>2.2</td>
<td>342.3</td>
</tr>
<tr>
<td>process message</td>
<td>1.9</td>
<td>344.2</td>
</tr>
<tr>
<td>receive second message from node 3</td>
<td>1.8</td>
<td>346.0</td>
</tr>
<tr>
<td>process message</td>
<td>2.8</td>
<td>348.8</td>
</tr>
<tr>
<td>receive second message from node 4</td>
<td>1.8</td>
<td>350.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Time until sends first message</th>
<th>Time to send first message</th>
<th>Time until sends second message</th>
<th>Time to send second message</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td>199 ms</td>
<td>17</td>
<td>43</td>
<td>10</td>
<td>269</td>
</tr>
<tr>
<td>Node 2</td>
<td>233 ms</td>
<td>16</td>
<td>27</td>
<td>10</td>
<td>286</td>
</tr>
<tr>
<td>Node 3</td>
<td>193 ms</td>
<td>15</td>
<td>46</td>
<td>7</td>
<td>261</td>
</tr>
<tr>
<td>Node 4</td>
<td>214 ms</td>
<td>17</td>
<td>33</td>
<td>5</td>
<td>269</td>
</tr>
</tbody>
</table>

Figure 5.1: Time Requirements for Multi-node Range Queries

7.1 ms per output page.

Since each message is 5,000 rows, which is $5,000 \times 16 = 80,000$ bytes, we can assume a transfer rate of $0.00018$ ms per byte, which implies that messages require $14.4$ ms to be sent. The times observed here are quite close to this: 17 ms, 16 ms, 15 ms and 17 ms. Assuming, then 14.4 ms, and rounding to 15 ms, we conclude: each message for large sets should then require $153 + 15 = 168$ ms.

The ideal sending time for data is $(#\text{rows}) \times 16 \times 0.00018$ ms. In this case, the second message is small so we achieve a total communication time which is far from ideal. However, the sending times for the messages that are 5,000 rows are close to the optimal 14.4 ms. For larger result sets, where there are many 5,000-row messages, the communication time can therefore be expected to be close to optimal.

From this experiment, we can conclude that, for large datasets, the time to perform a multi-node query is approximately:

Multi-node Range Query Time = $45 \text{ ms} + 168 \text{ ms} \times (#\text{messages}) + 12 \text{ ms}$.  

(2)
That is, 45 ms to receive the first message, followed by receiving a message every 168 ms from the node that sends the most messages, followed by 12 ms to process the last message.

The 45 ms is dependent only on the communication time. The 168, is $153 + 15$ ms. The 153 ms is dependent on the CPU and logical-I/O time, and can be derived from Formula (1). The 15 ms depends only on the communication time. The final 12 ms depends on the logical-I/O time to write 29.4 pages, and the CPU time to process 5000 rows. This can also be derived from Formula (1).

This experiment also shows that the primary node can potentially determine how long the operation takes, since, if there are sufficiently many nodes, it cannot process the messages as fast as they arrive, and therefore may continue reading and processing messages after the other nodes are finished sending the messages. Since, in this environment, messages take 12 ms to process, and each node can create and send a message every 153 ms, the limit is about 13 nodes. Formula (2) holds where this situation does not occur. In the sequel we will assume that there is not a bottleneck at the primary node, and the time for the range query depends on how well balanced the nodes are. The query will complete when the last node sends its last message, and, if the nodes are balanced well, they will each send roughly the same number of messages.

### 5.6 Experiment 9. Varying the Number of Nodes and Declustering Level

<table>
<thead>
<tr>
<th># Nodes</th>
<th>Decl. Level</th>
<th>Diagonal1</th>
<th>Diagonal5</th>
<th>LB</th>
<th>MG</th>
<th>Fluid</th>
<th>Citadel</th>
<th>Islands</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>1.16</td>
<td>0.10</td>
<td>0.87</td>
<td>0.77</td>
<td>3.22</td>
<td>1.66</td>
<td>0.57</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.13</td>
<td>0.06</td>
<td>0.25</td>
<td>0.16</td>
<td>2.04</td>
<td>0.86</td>
<td>0.33</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0.13</td>
<td>0.06</td>
<td>0.17</td>
<td>0.11</td>
<td>1.78</td>
<td>0.46</td>
<td>0.29</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>0.09</td>
<td>0.08</td>
<td>0.16</td>
<td>0.09</td>
<td>0.89</td>
<td>0.18</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>0.15</td>
<td>0.06</td>
<td>0.07</td>
<td>0.05</td>
<td>0.09</td>
<td>0.08</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3.26</td>
<td>2.14</td>
<td>1.65</td>
<td>1.32</td>
<td>3.76</td>
<td>2.16</td>
<td>1.54</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>0.52</td>
<td>0.19</td>
<td>0.32</td>
<td>0.32</td>
<td>2.45</td>
<td>0.64</td>
<td>0.47</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>0.27</td>
<td>0.10</td>
<td>0.21</td>
<td>0.14</td>
<td>1.34</td>
<td>0.24</td>
<td>0.38</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>0.08</td>
<td>0.12</td>
<td>0.15</td>
<td>0.06</td>
<td>0.65</td>
<td>0.16</td>
<td>0.28</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>0.08</td>
<td>0.07</td>
<td>0.10</td>
<td>0.05</td>
<td>0.05</td>
<td>0.08</td>
<td>0.06</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>4.02</td>
<td>2.87</td>
<td>2.44</td>
<td>2.66</td>
<td>4.12</td>
<td>2.48</td>
<td>1.69</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>0.28</td>
<td>0.85</td>
<td>0.77</td>
<td>0.64</td>
<td>3.11</td>
<td>0.78</td>
<td>0.65</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>0.23</td>
<td>0.20</td>
<td>0.28</td>
<td>0.36</td>
<td>2.34</td>
<td>0.34</td>
<td>0.29</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>0.13</td>
<td>0.17</td>
<td>0.14</td>
<td>0.14</td>
<td>1.45</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>12</td>
<td>19</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
<td>0.06</td>
<td>0.08</td>
<td>0.09</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 5.5: Load Balance Varying the Number of Nodes and Declustering Level
Since small and large datasets behave differently, and many of the datasets used here do not behave in the same way as significantly larger sets would, it is useful to consider the load balancing achieved by declustering in different ways, in terms of the number of rows found by the nodes for given queries. These results can then be applied to different sizes of datasets, or environments where the I/O, CPU or communication costs are different, to estimate the specific time requirements.

Experiment 9 reveals how varying the declustering level and the number of nodes affects the load balance. The load balance is measured by the calculating the maximum number of rows found by one node minus the minimum number, over the average number. The smaller this number is, the better balanced the nodes are. This measure of load balance will also be used throughout the sequel. Table 5.5 shows the load balance for 4, 8 and 12 nodes, declustered at Levels 4 to 7, and 19, using Q2 and no replication. This table indicates that the load balances vary greatly between using 4, 8 or 12 nodes when declustering at high levels, such as 4 or 5, but the loads are generally very similar when lower levels, such as 6 or 7, are used.

Table 5.5 also indicates that good load balances can often be achieved by Level 5 or 6, often close to as good as the load balance achieved at Level 19. Relatively uniform sets, such as Diagonal5 can achieve fairly good declustering even at high levels. Though, at sufficiently high levels, the query window often will not intersect any fragments for certain nodes, and the load is very imbalanced. An example of this is shown in Figure 5.2, where the data is declustered at Level 2 for 16 nodes, and 12 of the 16 nodes receive no cells that intersect the window. Also, in some cases, as lower levels are used, the nodes receive the same data even though the data space is divided into smaller fragments. An example of this is shown in Figure 5.3, where data is declustered to four nodes, and the data is declustered in essentially the same way at Levels 1 and 2. This occurred at Levels 1 to 4 in this experiment, and therefore the balances at these levels are the same as for Level 4, and are not shown.

![Figure 5.2: Declustering to 16 Nodes, Where Only 4 Nodes Receive Data](image)

With Diagonal5, we can see that the balance sometimes gets worse as we use lower declustering levels. In general, the balance will improve as we go lower, but between any two successive levels, once there is a reasonably good balance, this may not be the case. This pattern of having consistent improvement until a certain level, and then general but inconsistent improvement, occurs with all datasets, though, since we only included to Levels 4 to 7, and 19, in the table, the second part of the pattern cannot be seen for all sets.
This experiment also demonstrates that once a sufficiently low level is used to decluster, the number of nodes does not affect the appropriate declustering level. Once we decluster low enough that the problems indicated in Figures 5.2 and 5.3 do not occur, then for each dataset, there is roughly the same balance for 4, 8 or 12 nodes.

In general, when declustering is done at lower levels, the load balance is fairly similar for different sets. An exception is Fluid, but Table 5.5 indicates that even Fluid becomes as well balanced as other sets at a sufficiently low level. When declustering is done at higher levels, though, the load balance varies significantly from one set to another.

This test demonstrates that a very good balance can be achieved when using Level 19, even for very skewed sets such as Fluid. We can conclude, based on this, that when declustering only for range queries, and replication is not necessary, HCAM can give a very good balance if we use a sufficiently low level. Consequently, load-balancing techniques such as List Scheduling are, in fact, not necessary to achieve good balancing.

5.7 Experiment 10. Comparing Declustering With and Without Replication

In Experiment 10, we demonstrate the accuracy of a formula that predicts the additional number of rows found at each remote node due to replication. Table 5.6 gives, for two synthetic sets and five real sets, the percentage of data at each of the levels. This can be used to estimate the number of extra rows that will be found at the nodes due to replication.

The number of extra rows found at the nodes, where there are between 4 and 15 nodes, can be estimated by the following formula:

Let $l$ be the declustering level.
Let $r$ be the number of rows found at the nodes if there is no replication. ($r$ is the actual total number of rows within the query window.)

\[
\# \text{ of extra rows} = (\text{fraction of data at Levels } 0 \text{ to } l - 2) \times r \times (#\text{nodes}) \\
+ (\text{fraction of data at Level } l - 1) \times r \times 4
\] (3)
In general, the number of extra rows is:

\[
e = \sum_{j=1}^{q-1} F_{i-j} r^4 j + \sum_{i=0}^{l-q} F_i r (#\text{nodes})
\]

where 4^{q-1} < #\text{nodes} \leq 4^q - 1

\(e\) is the number of extra rows

\(F_j\) is the fraction of data at level \(j\)

Experiment 10 tests Formula (3) using 12 nodes, declustering at Level 7. Various datasets and query windows are used. The results are shown in Table 5.7. One of the cases in Table 5.7 is Diagonal1, where there are 12 nodes and the data is declustered at Level 7, so \(l\) is 7. The number of rows without replication, \(r\), is 607. Table 5.6 indicates that Levels 0 through \(l - 2\) contain 0.32\% of the data, and level \(l - 1\) contains 0.29\%. This gives us:

\[
\# \text{ of extra rows} = 0.0032 \times 607 \times 12 \\
+ 0.0029 \times 607 \times 4 \\
= 30.34
\]
We would, therefore, expect there to be \(607 + 30 = 637\) rows found by the nodes, while Table 5.7 indicates there was actually 646.

The ratios given in Table 5.7 are the ratio of the number of rows actually found with replication, over the number of rows found without replication. This indicates the total amount of extra work experienced by the nodes due to replication. This extra work would generally be distributed to the nodes proportional to their loads without replication. For example, the case with LB and Q1 shows there are 260 rows without and 410 rows with replication. The ratio of these is 1.58. With good load balancing, we would expect each of the twelve nodes to find \(260/12\) rows without replication, and \(410/12\) with, giving the same ratio of 1.58.

The experiments also showed that the ratio of time spent performing the range query with replication, over time spent performing the range query without replication, was, in all cases and on all nodes, approximately the same as the ratio of the total number of rows found with and without replication. Since the time required by a range query is determined by the maximum time taken by any node, and since the time spent by this node will increase by this same ratio, the total time for the range query also increases by this ratio. The experiments verified that this was, in fact, the case.

The experiment was repeated with MG, LB.shift75, Fluid and Islands, using 4, 8 and 10 nodes, and declustering at Levels 5 and 9, with the same results.

Because MG's distribution of data across its level files is different than that of the other datasets considered here, the effect of replication is shown separately for MG, in Table 5.8. MG has much more data at higher levels than the other sets, and so replication causes the increase in time required for range queries to be much larger than it is for other sets. With MG, the extra cost due to replication is significantly greater with lower declustering levels and greater numbers of nodes. Table 5.8 also shows that estimates are more accurate with larger query windows than with smaller windows.

In general, Formula (3) gives a fairly accurate estimate of the number of extra rows that will be within any given window due to replication. The larger the query window,
### Table 5.8: Comparing Declustering for MG With and Without Replication

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2</td>
<td>4</td>
<td>161</td>
<td>172</td>
<td>191</td>
<td>1.2</td>
</tr>
<tr>
<td>Q1</td>
<td>8</td>
<td>4</td>
<td>161</td>
<td>592</td>
<td>473</td>
<td>2.9</td>
</tr>
<tr>
<td>Q1</td>
<td>3</td>
<td>6</td>
<td>161</td>
<td>192</td>
<td>211</td>
<td>1.3</td>
</tr>
<tr>
<td>Q1</td>
<td>8</td>
<td>6</td>
<td>161</td>
<td>734</td>
<td>549</td>
<td>3.4</td>
</tr>
<tr>
<td>Q1</td>
<td>3</td>
<td>8</td>
<td>161</td>
<td>198</td>
<td>231</td>
<td>1.4</td>
</tr>
<tr>
<td>Q1</td>
<td>8</td>
<td>8</td>
<td>161</td>
<td>877</td>
<td>745</td>
<td>4.6</td>
</tr>
<tr>
<td>Q4</td>
<td>2</td>
<td>4</td>
<td>14,535</td>
<td>15,564</td>
<td>16,042</td>
<td>1.1</td>
</tr>
<tr>
<td>Q4</td>
<td>8</td>
<td>4</td>
<td>14,535</td>
<td>53,430</td>
<td>53,876</td>
<td>3.7</td>
</tr>
<tr>
<td>Q4</td>
<td>3</td>
<td>6</td>
<td>14,535</td>
<td>17,352</td>
<td>17,102</td>
<td>1.2</td>
</tr>
<tr>
<td>Q4</td>
<td>8</td>
<td>6</td>
<td>14,535</td>
<td>66,305</td>
<td>65,986</td>
<td>4.5</td>
</tr>
<tr>
<td>Q4</td>
<td>3</td>
<td>8</td>
<td>14,535</td>
<td>17,866</td>
<td>17,996</td>
<td>1.2</td>
</tr>
<tr>
<td>Q4</td>
<td>8</td>
<td>8</td>
<td>14,535</td>
<td>79,180</td>
<td>79,371</td>
<td>5.5</td>
</tr>
</tbody>
</table>

The histogram allows us to estimate how many rows will be found at each node. To estimate the total response time, we divide this by 5000, to predict how many messages will be sent from each node. We can then predict which node will send the most messages to the primary node and may then estimate the total response time, using Formula (2).

Table 5.9 indicates that the estimates using this formula are consistently quite accurate, though they would be still more precise with large datasets, where we can predict the time to generate the messages much more accurately.

We also found that, when declustering at Level 19, a good estimate of the workload at each node can be found by estimating the workload on a single node and dividing by the number of nodes. Often this gave as good an estimate as using the histogram, due to the fact that the loads were balanced almost perfectly when using Level 19. For higher
levels, such as Level 5, dividing the workload by the number of nodes is not as accurate, and histograms are necessary.

This experiment indicates that it is possible to accurately predict the time required for range queries when using the Range-based Strategy, where no replication is used. Since Experiment 10 demonstrates that we can also estimate the extra time with replication, we can also, therefore, estimate how long the range queries will take using the Join-based Strategy. This indicates that it is possible for the Brute-Force Division method to exhaustively calculate the time for range queries for both strategies, and for all levels for the Join-based Strategy, and by doing so, can accurately determine the time necessary for range queries for each strategy and declustering level.

| Dataset | # Nodes | Level 5 | | | Level 19 | | |
|---------|---------|---------|----------------------|----------------------|
|         | Estimated Total Time | Actual Total Time | Estimated Total Time | Actual Total Time |
| MG      | 4       | 259.7   | 280.4               |                      |
| MG      | 8       | 161.4   | 170.2               | 146.3               |
| MG      | 12      | 94.2   | 104.3               | 83.7               |
| LB      | 4       | 360.2 | 340.1               | 351.3               |
| LB      | 8       | 185.0 | 167.3               | 177.1               |
| LB      | 12      | 124.2 | 120.0               | 119.1               |
| Fluid   | 4       | 1,566.2 | 1,610.4           | 1,443.8             |
| Fluid   | 8       | 731.7 | 693.8               | 723.4               |
| Fluid   | 12      | 568.5 | 590.4               | 483.2               |
| Citadel1| 4       | 98.1 | 95.2               | 92.6               |
| Citadel1| 8       | 55.0 | 56.3               | 48.3               |
| Citadel1| 12      | 40.6 | 38.2               | 34.5               |
| Islands | 4       | 15.9 | 16.4               | 12.2               |
| Islands | 8       | 10.8 | 11.4               | 6.6               |
| Islands | 12      | 7.6 | 9.0               | 5.4               |

Table 5.9: Accuracy of Estimating the Total Time (Times in ms)
Chapter 6

Experimental Results For Joins

Using Brute Force Division to determine which of the Range-based and Join-based Strategies is preferable, and what the best initial declustering level is, requires predicting the times taken by the Partition, Sort and Join Steps for the join queries. This chapter presents formulas to predict each of these, and experiments to prove their accuracy.

Experiments 12 to 16 demonstrate that the times for each of the three steps, other than logical output times, can be predicted from the input sizes. Experiment 12 shows the relative times taken by each of the steps. Experiment 13 demonstrates that the response time for joins increases predicableably with the size of the input sets. Experiment 14 shows that formulas we developed to estimate response times given the input sizes are accurate. Experiment 15 demonstrates that output sizes can be predicted using histograms. Experiment 16 demonstrates that, using the results of Experiments 12 to 15, we can accurately estimate the time required by join queries on single nodes. To estimate response times for multi-node queries, it is also necessary to predict the communication times for the joins. When using the Range-based Strategy, this requires re-declustering the datasets and receiving the result sets. When using the Join-based Strategy, this requires only receiving the result sets.

Experiment 17 to 19 determine the time requirements to re-decluster the datasets and to receive the result sets. Experiment 17 also demonstrates that, as with range queries, the minimum time for join queries is achieved when the nodes receive the min-max workload.

Since Experiment 17 indicates that the minimum response time is realised when the nodes receive the min-max number of rows, we must determine how to decluster to achieve this. Experiments 20 and 21 show the relationship between the declustering level and the workloads. Experiment 20 demonstrates that the min-max workload is achieved where the goals of load-balancing and minimizing replication are best balanced. Experiment 21 also demonstrates that this level can be predicted by measuring the skew of the two sets, and can be predicted more accurately using histograms. Experiment 22 then demonstrates that Division Based on Skew can efficiently re-decluster to perform a single join query.
6.1 Experiment 12. Time Required By Individual Steps of the Join Operation

Experiment 12 measures the time required by the Partition Step, Sort Step, Join Step without output, and the logical writing of the output to perform joins on various pairs of datasets on a single node. Outputting the result set is normally part of the join step, but this test includes the times with and without actually outputting the result set, in order to reveal how much time is required for logical disk output.

The experiment used the six synthetic sets Data1 to Data6. All 21 pairs (including the self-joins) of the six sets were used. The datasets varied greatly in: size of objects, variation in size of objects, overlap of objects, the amount of the data space covered by objects, and the output size. Table 6.1 shows the times (in milliseconds) for the 10 combinations of the first four sets, and shows that the Partition and Sort Steps take similar times for all datasets. The Join Step, even without output, requires the largest amount of time of the three steps, but fluctuates the most.

The time to perform the logical write on the output file is, in most cases, a small portion of the overall time for the join. Though in some cases, where the output is exceptionally large, the time to output the results set is quite significant. The time to perform the logical write on the output file is about 0.43 ms per page, which is consistent with Experiment 1. Where there is a large output, the logical-I/O to write the output file requires by far the greatest portion of the total time.

The tests with Data5 and Data6 found similar times for the Partition and Sort steps, and similar savings per page where the output was not written. The times for the Join Step varied, but were also between 4,000 and 14,000 ms.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Partition</th>
<th>Sort</th>
<th>Join With Output</th>
<th>Total Time</th>
<th>Pages Output</th>
<th>Join Without Output</th>
<th>Savings</th>
<th>Savings Per Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data1</td>
<td>Data1</td>
<td>1,277</td>
<td>2,290</td>
<td>14,686</td>
<td>18,253</td>
<td>588</td>
<td>14,411</td>
<td>275</td>
<td>.468</td>
</tr>
<tr>
<td>Data1</td>
<td>Data2</td>
<td>1,355</td>
<td>2,543</td>
<td>8,700</td>
<td>12,598</td>
<td>620</td>
<td>8,443</td>
<td>257</td>
<td>.414</td>
</tr>
<tr>
<td>Data1</td>
<td>Data3</td>
<td>1,648</td>
<td>1,945</td>
<td>27,572</td>
<td>31,165</td>
<td>39,887</td>
<td>9,224</td>
<td>18,348</td>
<td>.460</td>
</tr>
<tr>
<td>Data1</td>
<td>Data4</td>
<td>1,619</td>
<td>2,136</td>
<td>12,761</td>
<td>16,516</td>
<td>338</td>
<td>12,571</td>
<td>190</td>
<td>.562</td>
</tr>
<tr>
<td>Data2</td>
<td>Data2</td>
<td>1,305</td>
<td>2,877</td>
<td>13,046</td>
<td>17,228</td>
<td>588</td>
<td>12,779</td>
<td>267</td>
<td>.455</td>
</tr>
<tr>
<td>Data2</td>
<td>Data3</td>
<td>1,123</td>
<td>2,238</td>
<td>29,548</td>
<td>32,909</td>
<td>49,463</td>
<td>7,735</td>
<td>21,813</td>
<td>.441</td>
</tr>
<tr>
<td>Data2</td>
<td>Data4</td>
<td>1,266</td>
<td>2,176</td>
<td>8,285</td>
<td>11,727</td>
<td>865</td>
<td>7,943</td>
<td>342</td>
<td>.395</td>
</tr>
<tr>
<td>Data3</td>
<td>Data3</td>
<td>1,198</td>
<td>1,921</td>
<td>62,068</td>
<td>65,187</td>
<td>93,662</td>
<td>4,560</td>
<td>57,508</td>
<td>.614</td>
</tr>
<tr>
<td>Data3</td>
<td>Data4</td>
<td>1,521</td>
<td>1,892</td>
<td>8,512</td>
<td>11,925</td>
<td>725</td>
<td>8,229</td>
<td>283</td>
<td>.390</td>
</tr>
<tr>
<td>Data4</td>
<td>Data4</td>
<td>1,526</td>
<td>2,161</td>
<td>9,061</td>
<td>12,748</td>
<td>1,158</td>
<td>8,473</td>
<td>588</td>
<td>.508</td>
</tr>
</tbody>
</table>

Table 6.1: Time Required By Individual Steps of the Join Operation (Times in ms)
6.2 Experiment 13. The Effect of Dataset Size

Experiment 13 demonstrates the relation between dataset size and the response time for join queries. The experiment uses a series of files that have the same characteristics as the six files used in the previous experiment. Each file used in Experiment 12 corresponds to six different input files here, each with a different size. For this experiment, no output files were written to, so that the logical-I/O to output the results can be considered separately. The experiment determines if the CPU time varies significantly for datasets with different distributions, or if the CPU time depends closely enough on the size of the input files, that this is adequate to estimate the Join Step time.

This experiment considered the Partition, Sort and Join Steps. Since the Join Step requires the most time and fluctuates the most, only the Join Step and Total times are shown here, in Table 6.2 and Table 6.3. Again, the ten combinations of the sets Data1 to Data4 are shown. In all cases both sets that are joined have the same size.

The Partition Step was found to require about 420 ms (to create the level files, and other overhead) plus about 0.01 ms for every row, for all pairs of datasets. The sort times initially increased rapidly as the input size increased from where most level files fit in buffer, to where most did not fit in buffer. Once the input size increased to where some level files require the merge step (described in Appendix A) to be performed, the sort time increased roughly linearly with the input size. For small datasets (10,000 or 20,000 rows) the sort step required about .007 ms per row. For larger datasets, it required about .019 ms per row. The Partition Step and Sort Step times are then as shown in Formulas (4) and (5):

\[
\text{Partition Time} = 420 \text{ ms} + (\text{sum of input sizes}) \times 0.01 \text{ ms.} \tag{4}
\]

\[
\text{Sort Time} = M \times (\text{sum of input sizes}), \text{ where } M \text{ is .007 for combined sizes under 40,000 rows, and .019 for combined sizes over 40,000 rows} \tag{5}
\]

For the Partition Step, the 420 ms is mostly to create the level files. The time to create these files is dependent on the local environment. The 0.1 ms is dependent on logical-I/O and CPU time. Each environment would then have its own value for partition time per row. For the Sort Step, as well, the time depends on the local CPU and logical-I/O times. A similar formula can be developed for each environment.

Join times vary a fair amount from one dataset to another, but the difference is usually no more than a factor of three between largest and smallest time.

This experiment shows that the times required for the Partition, Sort, and Join Steps increase roughly linearly with the dataset size, and that the time to perform a join, other than the logical writing of the output set, can be estimated fairly accurately from the size of the input sets. However, an accurate formula for the Join Step cannot be developed until we consider the case where the two datasets have different sizes.
### Table 6.2: The Effect of Dataset Size - Join Step (Times in ms)

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>Data1</td>
<td>Data1</td>
<td>1,590</td>
</tr>
<tr>
<td>Data1</td>
<td>Data2</td>
<td>1,408</td>
</tr>
<tr>
<td>Data1</td>
<td>Data3</td>
<td>1,509</td>
</tr>
<tr>
<td>Data1</td>
<td>Data4</td>
<td>940</td>
</tr>
<tr>
<td>Data2</td>
<td>Data2</td>
<td>1,412</td>
</tr>
<tr>
<td>Data2</td>
<td>Data3</td>
<td>1,661</td>
</tr>
<tr>
<td>Data2</td>
<td>Data4</td>
<td>1,224</td>
</tr>
<tr>
<td>Data3</td>
<td>Data3</td>
<td>1,289</td>
</tr>
<tr>
<td>Data3</td>
<td>Data4</td>
<td>2,183</td>
</tr>
<tr>
<td>Data4</td>
<td>Data4</td>
<td>1,613</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td>1,482</td>
</tr>
</tbody>
</table>

### Table 6.3: The Effect of Dataset Size - Total Times (Times in ms)

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>Data1</td>
<td>Data1</td>
<td>2,340</td>
</tr>
<tr>
<td>Data1</td>
<td>Data2</td>
<td>2,246</td>
</tr>
<tr>
<td>Data1</td>
<td>Data3</td>
<td>2,344</td>
</tr>
<tr>
<td>Data1</td>
<td>Data4</td>
<td>1,714</td>
</tr>
<tr>
<td>Data2</td>
<td>Data2</td>
<td>2,227</td>
</tr>
<tr>
<td>Data2</td>
<td>Data3</td>
<td>2,455</td>
</tr>
<tr>
<td>Data2</td>
<td>Data4</td>
<td>1,948</td>
</tr>
<tr>
<td>Data3</td>
<td>Data3</td>
<td>1,987</td>
</tr>
<tr>
<td>Data3</td>
<td>Data4</td>
<td>2,890</td>
</tr>
<tr>
<td>Data4</td>
<td>Data4</td>
<td>2,498</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td>2,265</td>
</tr>
</tbody>
</table>

### 6.3 Experiment 14. Time Required Where Datasets Have Different Sizes

In Experiment 14, we continue to focus on the relationship between the sizes of the input sets and the time required by the join, without writing the output. Experiment 14 tests formulas to estimate the times for each of the three steps.

For the Join Step, we found the time varies with the sum of the input sizes. Based on performing the join operation with several sizes of several pairs of the six synthetic sets, we developed the formula for join time:
Join Time = 1,000 ms + (sum of input sizes) \times .07 ms. \hspace{1cm} (6)

The values 1,000 ms and .07 ms are dependent on the local CPU and logical-I/O times. These figures will be larger or smaller in different environments depending on their CPU and logical-I/O times.

Table 6.4 shows the times required for the Partition, Sort and Join Steps, and the total time required by the join operation. This is done for various sizes of Data3 and Data4. Table 6.4 also shows the estimated response time. This was based on Formulas (4) and (5), to predict the times for the Partition and Sort Steps, and Formula (6) to predict the Join Step time. For example, with dataset sizes 20,000 and 60,000, we would expect the partition time to be 420 + 80,000 \times .01 = 1220, the sort time to be 80,000 \times .019 = 1520 ms, and the join time to be 1000 + 80,000 \times .07 = 6,600 ms. This gives an expected total time of 9,340 ms. The actual total time was 9,795 ms.

Table 6.4 demonstrates that Formulas (4), (5) and (6) are able to estimate the times for join queries accurately.

<table>
<thead>
<tr>
<th>Data3 Size</th>
<th>Data4 Size</th>
<th>Partition (ms)</th>
<th>Sort (ms)</th>
<th>Join (ms)</th>
<th>Est. Total (ms)</th>
<th>Act. Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>634</td>
<td>146</td>
<td>2,875</td>
<td>3,160</td>
<td>3,655</td>
</tr>
<tr>
<td>10,000</td>
<td>20,000</td>
<td>763</td>
<td>207</td>
<td>3,437</td>
<td>4,030</td>
<td>4,407</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>1,027</td>
<td>847</td>
<td>4,205</td>
<td>6,370</td>
<td>6,079</td>
</tr>
<tr>
<td>10,000</td>
<td>60,000</td>
<td>1,089</td>
<td>1,330</td>
<td>4,798</td>
<td>8,350</td>
<td>7,217</td>
</tr>
<tr>
<td>20,000</td>
<td>20,000</td>
<td>882</td>
<td>285</td>
<td>4,105</td>
<td>4,900</td>
<td>5,272</td>
</tr>
<tr>
<td>20,000</td>
<td>40,000</td>
<td>1,248</td>
<td>1,046</td>
<td>5,339</td>
<td>7,360</td>
<td>7,633</td>
</tr>
<tr>
<td>20,000</td>
<td>60,000</td>
<td>1,228</td>
<td>1,659</td>
<td>6,908</td>
<td>9,340</td>
<td>9,795</td>
</tr>
<tr>
<td>40,000</td>
<td>40,000</td>
<td>1,231</td>
<td>1,708</td>
<td>7,279</td>
<td>9,340</td>
<td>10,218</td>
</tr>
<tr>
<td>40,000</td>
<td>60,000</td>
<td>1,435</td>
<td>1,767</td>
<td>7,873</td>
<td>11,320</td>
<td>11,075</td>
</tr>
<tr>
<td>60,000</td>
<td>60,000</td>
<td>1,718</td>
<td>2,392</td>
<td>8,771</td>
<td>13,300</td>
<td>12,881</td>
</tr>
</tbody>
</table>

Table 6.4: Time Required Where Datasets Have Different Sizes

6.4 Experiment 15. Estimating the Output Size For Joins

For datasets with large output, the largest part of the overall join time is writing the output to file. Therefore, an accurate estimate of the join time must also include the logical-I/O time to write the result set. Estimating the time for this requires only estimating the size of the output, in pages, and multiplying this by the time per page to perform a logical write. Estimating result sizes is also important for multi-node queries, since it is necessary to estimate communication times.

In Experiment 15, we demonstrate that the technique to estimate the output size of join queries presented in Section 4.8.2 yields accurate estimates. In Experiment 15, we
perform joins of various pairs of datasets and record the estimate of the result sizes, using histograms, and the actual result sizes. The results are given in Table 6.5. Output sizes are given in pages.

This experiment demonstrates that the estimates are reasonably accurate. As well, estimating the output size was fast, and required, on average, about 3% the time required to perform the joins.

<table>
<thead>
<tr>
<th>File1</th>
<th>File2</th>
<th>Estimated Result Size</th>
<th>Actual Result Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>Diagonal2</td>
<td>232</td>
<td>275</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Diagonal5</td>
<td>110</td>
<td>142</td>
</tr>
<tr>
<td>LB</td>
<td>MG</td>
<td>166</td>
<td>146</td>
</tr>
<tr>
<td>Fluid</td>
<td>MG.shift50</td>
<td>340</td>
<td>322</td>
</tr>
<tr>
<td>Islands</td>
<td>LB.shift75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Citadel1</td>
<td>LB</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Citadel1</td>
<td>Islands</td>
<td>108</td>
<td>139</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>LB</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>MG.shift75</td>
<td>303</td>
<td>286</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>MG</td>
<td>103</td>
<td>81</td>
</tr>
<tr>
<td>LB</td>
<td>Fluid</td>
<td>261</td>
<td>240</td>
</tr>
<tr>
<td>Citadel1</td>
<td>MG</td>
<td>159</td>
<td>137</td>
</tr>
</tbody>
</table>

Table 6.5: The Accuracy of Output Size Estimation For Joins Using Histograms

6.5 Experiment 16. Estimating Join Times

Based on Experiments 12, 13 and 14, we can estimate the time for a join operation (without outputting the result set), and based on Experiment 15, we can estimate the output size for a given pair of datasets. From the size of the output set, we can estimate the time to perform the logical write on the output.

Experiment 16 tests how well Formulas (4), (5) and (6) and the histograms apply to the diagonal, real, and modified real sets. It also determines if using the formulas along with a histogram to predict output sizes is effective to predict the total response times for joins on single nodes. To test this, we perform a series of joins on various pairs of datasets and record the predicted and actual response times. These are shown in Table 6.6. The diagonal sets had four different sizes: 10,000, 25,000, 50,000 and 100,000 rows.

This experiment confirms that using Formulas (4), (5) and (6) in conjunction with a histogram is quite accurate for predicting the total times required by joins.
Table 6.6: Estimating Join Times (Times in ms)

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Size</th>
<th>Dataset 2</th>
<th>Size</th>
<th>Estimated Total Time</th>
<th>Actual Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>Diagonal2</td>
<td>100,000</td>
<td>8,799</td>
<td>9,776</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>MG</td>
<td>39,231</td>
<td>8,892</td>
<td>7,987</td>
</tr>
<tr>
<td>LB</td>
<td>53,145</td>
<td>MG</td>
<td>39,231</td>
<td>10,639</td>
<td>10,533</td>
</tr>
<tr>
<td>Fluid</td>
<td>120,206</td>
<td>MG</td>
<td>39,231</td>
<td>17,346</td>
<td>19,421</td>
</tr>
<tr>
<td>Islands</td>
<td>14,375</td>
<td>MG</td>
<td>39,231</td>
<td>8,092</td>
<td>9,690</td>
</tr>
<tr>
<td>Cidade1</td>
<td>66,837</td>
<td>LB</td>
<td>53,145</td>
<td>13,299</td>
<td>14,441</td>
</tr>
<tr>
<td>Cidade1</td>
<td>66,837</td>
<td>Islands</td>
<td>14,375</td>
<td>9,881</td>
<td>11,004</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>50,000</td>
<td>LB</td>
<td>53,145</td>
<td>11,632</td>
<td>11,980</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>100,000</td>
<td>MG</td>
<td>39,231</td>
<td>15,331</td>
<td>13,548</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>MG</td>
<td>39,231</td>
<td>6,335</td>
<td>6,330</td>
</tr>
<tr>
<td>LB</td>
<td>53,145</td>
<td>Fluid</td>
<td>120,206</td>
<td>18,688</td>
<td>20,009</td>
</tr>
<tr>
<td>Cidade1</td>
<td>66,837</td>
<td>MG</td>
<td>39,231</td>
<td>11,982</td>
<td>12,332</td>
</tr>
</tbody>
</table>

6.6 Experiment 17. Performing Join Queries Using Multiple Nodes

Experiment 17 verifies the feasibility of declustering data for a single join query, and demonstrates the efficiency of declustering data in the manner described in Section 4.8.1. This is done by recording the response times using both a single node and four nodes for given queries. Experiment 17 also shows the times necessary for individual steps of a multi-node join query, since this knowledge is necessary to predict response times for multi-node queries. The file Uniform is used for this experiment, since it is understood how best to decluster uniform data to four nodes: we decluster at Level 1 and replicate only Level 0. This experiment first performed a self-join on Uniform using a single node. This required 13,666 ms, broken down as follows:

- partition: 1,800 ms
- sort: 2,845 ms
- join: 9,021 ms
- total: 13,666 ms

Given the algorithm to send data, described in Section 4.8.1, declustering the two datasets to the remote nodes requires sending at least one message for each dataset, each level, and each receiver, or $2 \times 20 \times 4 = 160$ sends. Since Uniform has 49,952 rows, and each row requires 5 integers, a total of $2 \times 49,952 \times 5 = 499,520$ integers, or $4 \times 499,520 = 1,998,080$ bytes must be sent. Each send would then be $1,998,080 / 160 = 12,488$ bytes on average, without considering replication. Based on this and on Experiment 2, we would expect that the time required to send these messages should be approximately $1,998,080 \times 0.00018 \text{ ms/byte} = 360 \text{ ms}$.
The logical-I/O to decluster the data consists of reading in each page once. Uniform requires 294 pages. Therefore, we would expect $2 \times 294 \times 0.49 \text{ ms/page} = 288 \text{ ms}$. Communication and logical-I/O time for declustering should then be approximately $360 + 288 = 648 \text{ ms}$. The actual time observed was 789 ms, which also included CPU time.

The actual declustering time, 789 ms, included 368 ms for communication. Section 4.8.1 indicates that the minimum communication time for declustering is the total number of rows times sixteen times $0.00018 \text{ ms/byte}$. In this case, then, the minimum time is $2 \times 49,952 \times 16 \times 0.00018 \text{ ms/byte} = 287 \text{ ms}$. The actual declustering time is, therefore, approximately $5/4$ the minimum declustering time, since five, and not four integers are sent for every row. Nevertheless, experiments indicated that sending five integers for each row is faster than sending four and requiring the remote nodes to recalculate each Hilbert value.

The times for the multi-node join are broken down as shown in Figure 6.1.

<table>
<thead>
<tr>
<th>Primary Node</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition:</td>
<td>1,832</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort:</td>
<td>1,835</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decluster Data:</td>
<td>789</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive Results:</td>
<td>3,268</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>7,724</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receive Data:</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4,498</td>
<td>4,506</td>
<td>4,510</td>
<td>4,520</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Join Step:</th>
<th>2,742</th>
<th>2,699</th>
<th>2,776</th>
<th>2,792</th>
</tr>
</thead>
</table>

| Send Results:         | 75     | 69     | 81     | 79     |

Figure 6.1: Times Required for Self-join on Uniform Using Four Nodes

The primary node required $1,832 + 1,835 + 789 = 4,456 \text{ ms}$ to finish sending the data to the four other nodes. As with range queries, there is about 45 ms between the primary node completing the sending of the data, and the remote nodes completing the receiving of the data.

The Join Step at each node should be, based on Formula (6), $1000 + ((49,952 + 49,952)/4) \times .07 = 2748 \text{ ms}$. Figure 6.1 shows that the actual times are very close: 2,742, 2,699, 2,776 and 2,792 ms.

As indicated in Section 4.8.1, the communication costs to return the result sets are ideally equal to the total output size divided by the number of nodes, multiplied by 32 (since each join-pair requires 32 bytes), multiplied by $0.00018 \text{ bytes/ms}$. This is $48,306 / 4 \times 32 \times 0.00018 \text{ ms/byte} = 68 \text{ ms}$. Figure 6.1 shows that the actual times, 75, 69, 81 and 79 ms are close to the ideal time to return the results.

The time for the primary node to receive the results is just slightly greater than the largest time required by any of the remote nodes to join and send the results, as would be expected. The last node finishes sending its data after $2,792 + 79 = 2871 \text{ ms}$, and the primary node completes processing the messages after 3,268 ms, 397 ms later.
Since the Partition, Sort and logical disk output for the Join Step are done by the primary node, the same as in the single-node case, only the CPU and logical disk input of the Join Step are done in parallel. Nevertheless, a savings in time was observed. Join operations where there is a very large output, and the logical output is much more significant than the CPU and logical disk input work for the join step, would see the same reduction in time as shown here, but this would be a relatively insignificant reduction in time, since the logical disk output is done by the primary node whether single or multiple nodes are used. Further improvements in join time could therefore be found by allowing the remote nodes to write their result sets to their local disks.

Table 6.7 shows the times, for various sizes of the Uniform dataset, to perform a self-join using four nodes. This includes the times to decluster the data, for the remote nodes to calculate the join results, and for the primary node to receive the results. The total time, using four nodes, is the sum of the Partition, Sort and Decluster column and the Receive Results column, since these represent the total work for the primary node. Table 6.7 also shows the times, for the same datasets, to perform a self-join using a single node. This indicates that using four nodes becomes preferable to using one node somewhere between 1,000 and 5,000 rows.

Table 6.7 also suggests that with small sets, the time for the Join Step does not include the 1000 ms start-up that is normally assumed. Experiments with the Uniform and other sets indicated that 1000 ms is normal, though, for larger sets.

Results are sent to the primary node as messages of up to 2,500 join-pairs, which is the equivalent of 29.4 output pages. The logical-I/O time to write each results message is therefore approximately 29.4 × .43 ms/page = 12.6 ms. The time for the remote nodes to produce messages, though, varies, since the time to perform the join step depends on the input sizes, and not the output sizes. Therefore, there is no fixed maximum number of processors given CPU, logical-I/O and MPI speeds to perform join queries as there is for range queries.

From Experiment 17, we may conclude that it is feasible to decluster for single join operations, though not for datasets that are sufficiently small, and that the methods to decluster data and return the results given in Section 4.8.1 are efficient, as compared to the optimal declustering time, also described in Section 4.8.1.

<table>
<thead>
<tr>
<th>Size</th>
<th>Partition Sort &amp; Decluster</th>
<th>Node #</th>
<th>Node #</th>
<th>Node #</th>
<th>Receive Time</th>
<th>Total Time (4 Nodes)</th>
<th>Total Time (1 Node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000</td>
<td>4,456</td>
<td>2,817</td>
<td>2,768</td>
<td>2,857</td>
<td>2,871</td>
<td>3,268</td>
<td>7,724</td>
</tr>
<tr>
<td>25,000</td>
<td>2,544</td>
<td>924</td>
<td>1,145</td>
<td>1,15</td>
<td>956</td>
<td>1,206</td>
<td>3,742</td>
</tr>
<tr>
<td>15,000</td>
<td>2,078</td>
<td>392</td>
<td>582</td>
<td>509</td>
<td>653</td>
<td>704</td>
<td>2,764</td>
</tr>
<tr>
<td>10,000</td>
<td>1,061</td>
<td>332</td>
<td>255</td>
<td>374</td>
<td>441</td>
<td>524</td>
<td>1,589</td>
</tr>
<tr>
<td>5,000</td>
<td>0,700</td>
<td>254</td>
<td>213</td>
<td>237</td>
<td>247</td>
<td>410</td>
<td>1,070</td>
</tr>
<tr>
<td>1,000</td>
<td>0,263</td>
<td>86</td>
<td>91</td>
<td>106</td>
<td>79</td>
<td>142</td>
<td>408</td>
</tr>
</tbody>
</table>

Table 6.7: Varying the Input Size for Self-Joins on Uniform Data (Times in ms)
6.7 Experiment 18. Time Required to Decluster Data

Experiment 17 demonstrated that the time required to perform multi-node join queries consists of the time to perform the Partition and Sort Steps, to decluster the data, to perform the Join Step in parallel, and to return the results. The times required for the Partition, Sort and Join Steps are the same as in the single-node case. We must, though, develop formulas to predict the time to decluster the data and to return the results. Formulas to predict these times are presented and tested in Experiments 18 and 19. Experiment 18 investigates the time necessary to decluster data.

Table 6.8 gives the times required to decluster LB.shift25 and MG.shift25 to 4, 8, and 12 nodes using declustering levels from 3 to 8. The results indicate that the time to decluster the data is determined only by the amount of data sent, and not by the number of nodes, which is consistent with Experiment 3. The amount of data sent is determined by the size of the datasets, and by the degree of replication.

As was indicated in Experiment 17, the declustering time can be estimated from the logical-I/O time and MPI time, since the CPU time is relatively small. The I/O time is determined only by the input sizes, regardless of the replication. The MPI time is determined by both the input sizes and the replication. Nevertheless, communication time per row sent is fairly consistent, regardless of the difference between the number of rows read in, and the number sent out. The time to decluster may be predicted using the following formula:

\[
\text{Declustering Time} = (\text{sum of input sizes}) \times 0.0084 \text{ ms.} \quad (7)
\]

Although in this environment the communication time is dominant, in other environments, the CPU or logical-I/O necessary may be more significant, and the three times may need to be considered individually.

Table 6.8 indicates that Formula (7) is consistently accurate. This experiment was repeated using Islands and Citadel, Diagonal4 and Diagonal5, Fluid and MG, and LB and MG.shift75. The same results were revealed: the time to decluster per byte was consistent, and could be predicted accurately using Formula (7).

6.8 Experiment 19. Time Required to Return Result Sets

Experiment 19 determines the time to return result sets to the primary node, since estimating this is necessary to estimate the time required by multi-node join queries.

When join-pairs are sent to the primary node, we require eight integers per join-pair, or 32 bytes per join-pair. We can therefore estimate the time to return the result sets as \((\# \text{ join-pairs}) \times 32 \times 0.00018 \text{ ms/byte}\), which is \((\# \text{ join-pairs}) \times 0.00576 \text{ ms/byte}\), yielding Formula (8):

\[
\text{Return Join-pairs Time} = (\# \text{ join-pairs}) \times 0.00576 \text{ ms} \quad (8)
\]
This formula depends only on the communication times for the environment. Other environments would, then, have a different value as opposed to 0.00576 ms.

Experiment 19 demonstrates the accuracy of Formula (8) by measuring the time required to return the result sets for joins using each combination of LB, MB and the modified LB and MG sets, and using four, eight and twelve nodes, declustering at Level 5. The results for LB and MG using four nodes are given in Table 6.9. The times do not include messages with fewer than 2500 join-pairs, since these require communication times greater than 0.00018 ms/byte, but consume a relatively small part of the the total time to return the results, especially for large sets.

Table 6.9 shows that Formula (8) accurately predicts the time to return the result sets. Formula (8) was equally accurate for the other datasets and numbers of nodes. As well, since Formula (8) depends only on the number of join-pairs returned, it is not relevant that the remote nodes take different amounts of time to generate each message of results. Though the results are returned as the Join Step is performed, the time to return the results can be predicted by assuming that the complete Join Step is performed,

<table>
<thead>
<tr>
<th>Decl. Level</th>
<th># Nodes</th>
<th>Rows Sent</th>
<th>Time</th>
<th>Time Per Row Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>107,012</td>
<td>920</td>
<td>.00859</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>108,309</td>
<td>939</td>
<td>.00866</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>111,555</td>
<td>947</td>
<td>.00849</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>127,497</td>
<td>988</td>
<td>.00775</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>134,643</td>
<td>1,121</td>
<td>.00833</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>143,264</td>
<td>1,145</td>
<td>.00799</td>
</tr>
<tr>
<td>avg</td>
<td>4</td>
<td></td>
<td></td>
<td>.00830</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>108,108</td>
<td>940</td>
<td>.00870</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>113,434</td>
<td>964</td>
<td>.00850</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>121,265</td>
<td>982</td>
<td>.00810</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>129,843</td>
<td>1,070</td>
<td>.00824</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>142,285</td>
<td>1,221</td>
<td>.00858</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>163,988</td>
<td>1,430</td>
<td>.00872</td>
</tr>
<tr>
<td>avg</td>
<td>8</td>
<td></td>
<td></td>
<td>.00847</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>111,423</td>
<td>947</td>
<td>.00885</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>121,412</td>
<td>1,025</td>
<td>.00844</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>138,774</td>
<td>1,049</td>
<td>.00756</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>152,992</td>
<td>1,328</td>
<td>.00868</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>160,612</td>
<td>1,275</td>
<td>.00794</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>184,396</td>
<td>1,624</td>
<td>.00881</td>
</tr>
<tr>
<td>avg</td>
<td>12</td>
<td></td>
<td></td>
<td>.00838</td>
</tr>
</tbody>
</table>
followed by returning the entire result set.

<table>
<thead>
<tr>
<th></th>
<th>Remote Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td># Join-pairs</td>
<td>3,151</td>
</tr>
<tr>
<td>Time To Return</td>
<td>18.6</td>
</tr>
<tr>
<td>Time / Join-pair</td>
<td>0.00592</td>
</tr>
</tbody>
</table>

Table 6.9: Time Required to Return Results Sets (Times in ms)

6.9 Experiment 20. Determining the Best Declustering Level

Formula (6) indicates that the time spent by each node is proportional to the number of rows sent to each node. From this we may conclude that the min-max time required by the nodes occurs when the nodes receive the min-max number of rows, and therefore, the minimum total time taken for the join query occurs when the nodes receive the min-max number of rows. Experiment 20 tests the accuracy of that hypothesis, and confirms that the best declustering level is where the goals of load balancing and minimizing replication are best balanced.

Table 6.10 shows the relationship between the number of rows sent to each node and the total response time for the query, for Diagonal1 and LB using six nodes. This demonstrates that, as lower declustering levels are used, the balance improves until we reach the level with the min-max number of rows. The total times improve until we reach this level, then increase as we continue to use lower levels. The total times are given in milliseconds.

This experiment was repeated for various other pairs of datasets, and numbers of nodes. The same result was found consistently: the minimum total time occurs at the declustering level where the remote nodes receive the min-max number of rows. Therefore, any scheme that declusters data for join operations should seek to decluster at this level. In most cases, the levels near the best declustering level tend to have similar load balances, and similar total times. This is shown for Diagonal1 and LB in Table 6.10; the best declustering level is Level 5, but Levels 4, 6 and 7 are quite similar.

6.10 Experiment 21. The Relationship Between Data Skew and the Best Declustering Level

Experiment 21 determines the relationship between data skew and the best declustering level, and evaluates methods to determine the best declustering level. We tested two methods to determine the best declustering level: histograms, and a heuristic based on the standard deviations of the two sets.
The method using histograms considers each level, one at a time, and estimates first the declustering time given the size of the input sets, and the replication at the current level using Formula (7). It then uses the histogram to predict the number of rows given to each node, using the method given in Section 4.8.1. From this, it applies Formula (6) to predict the Join Step time at each node. Using the technique for histograms, also given in Section 4.8.1, an estimate of the result size at each node is predicted, and Formula (8) is used to predict the time for each node to return their results. Since the times for the Partition Step, Sort Step and writing the output file are the same regardless of the declustering level used, only these three times must be estimated, and the best level is where their sum is the minimum.

The method using the standard deviations, calculates the standard deviation using the method described in Section 3.7.4, and uses the heuristic given in Table 6.11.

<table>
<thead>
<tr>
<th>Largest Skew</th>
<th>Best Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 - 1.5</td>
<td>4</td>
</tr>
<tr>
<td>1.5 - 2.5</td>
<td>5</td>
</tr>
<tr>
<td>2.5 - 7.0</td>
<td>6</td>
</tr>
<tr>
<td>7.0 - 20.0</td>
<td>7</td>
</tr>
<tr>
<td>20.0 - 60.0</td>
<td>8</td>
</tr>
<tr>
<td>60.0 -</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.11: Heuristic for Best Declustering Level Based on Skew

This method uses only the larger of the two skews, and seeks to use the minimum level such that this set is balanced well among the nodes.

Table 6.12 gives the calculated best declustering level using both methods, and gives the actual best declustering level, found by performing the join on four nodes using each possible declustering level. Experiment 21 tests the two methods using the diagonal sets and the LB sets paired with Islands. In the first case the sets are skewed in the same manner; in the second case the sets are skewed in different ways. Whether or not the sets...
are skewed in the same manner can potentially affect whether or not load imbalances are exacerbated by assigning the densest parts of both sets to the same nodes, since the Join Product Skew may be very large.

We can see that the more skewed the data is, the lower the best declustering level is. Histograms give a better estimate of the best declustering level than using the standard deviations. However, the histograms require significantly more time to calculate than the standard deviations, and are therefore not as suitable when declustering for a single join operation. Calculating the standard deviation requires approximately .7% of the time to perform the join on a single node; estimating the best declustering level using the histogram method required approximately 13% the time to perform the join on a single node. Therefore, using histograms is more appropriate to perform the initial declustering using Brute Force Division, and the standard deviations are more appropriate to decluster for a single join using Division Based on Skew.

Tests were also done with the MG sets and showed that, though the histograms still performed well, the heuristic based on the standard deviations did not. This is because the histogram method considers the distribution of the data across the levels, while standard deviation method does not, and therefore allows excessive replication. The heuristic is, therefore, useful for most datasets, but does lead to poor performance in some cases.

The experiment was repeated with several pairs of datasets. This indicated that histograms generally picked the best level within one or two levels, and that the time required at the calculated best level was usually within 5% of the time required at the actual best declustering level. This indicates that histograms do, in fact, allow Brute Force Division to accurately calculate the best declustering levels for joins.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Std. Dev.</th>
<th>Dataset 2</th>
<th>Std. Dev.</th>
<th>(Std.Dev.) Est. Best Level</th>
<th>(Histogram) Est. Best Level</th>
<th>Actual Best Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>4.12</td>
<td>Diagonal2</td>
<td>3.13</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>4.12</td>
<td>Diagonal5</td>
<td>1.15</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>3.13</td>
<td>Diagonal3</td>
<td>2.72</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>3.13</td>
<td>Diagonal5</td>
<td>1.15</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>2.72</td>
<td>Diagonal4</td>
<td>1.89</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>1.89</td>
<td>Diagonal5</td>
<td>1.15</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LB.</td>
<td>3.91</td>
<td>Islands</td>
<td>6.27</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>LB.shift25</td>
<td>4.59</td>
<td>Islands</td>
<td>6.27</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LB.shift50</td>
<td>5.57</td>
<td>Islands</td>
<td>6.27</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LB.shift75</td>
<td>7.57</td>
<td>Islands</td>
<td>6.27</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6.12: The Relationship Between Data Skew and the Best Declustering Level
6.11 Expirement 22. Efficiency of Division Based on Skew

In Experiment 22, we calculate the skew, measured by standard deviation, as described in Section 3.7.4, for various pairs of datasets. The times to calculate the standard deviations and to perform the joins on a single node and using twelve nodes (using the level calculated by the heuristic) are given in Table 6.13. Times are in milliseconds. This experiment demonstrates that the time to calculate the standard deviations is quite small compared to the overall join time, and that the time using twelve nodes represents a good speedup as compared to using one node. Therefore, multi-node joins using Division Based on Skew can be performed quite efficiently.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Time to Calculate Skew</th>
<th>Join Time Using 1 Node</th>
<th>Join Time Using 12 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB.shift50</td>
<td>Citadel</td>
<td>83</td>
<td>13,607</td>
<td>5,941</td>
</tr>
<tr>
<td>LB.shift75</td>
<td>Islands</td>
<td>61</td>
<td>9,152</td>
<td>4,872</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>LB.shift25</td>
<td>78</td>
<td>12,681</td>
<td>5,655</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>LB.shift75</td>
<td>77</td>
<td>12,705</td>
<td>5,602</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Diagonal5</td>
<td>76</td>
<td>8,010</td>
<td>4,005</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Diagonal5</td>
<td>77</td>
<td>8,120</td>
<td>4,010</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>LB</td>
<td>80</td>
<td>12,091</td>
<td>5,876</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>LB</td>
<td>79</td>
<td>11,876</td>
<td>5,754</td>
</tr>
<tr>
<td>Diagonal5</td>
<td>LB</td>
<td>78</td>
<td>11,764</td>
<td>5,721</td>
</tr>
</tbody>
</table>

Table 6.13: Time Required to Estimate Skew (Times in ms)
Chapter 7

Experimental Results For Combined Range and Join Queries

This chapter describes experiments that determine if the results found for just range queries and just joins apply to the case where we perform combined range and join queries. Experiment 23 tests if the results found when performing joins on complete sets apply equally to joining range query results. Experiment 24 tests if the response time for multi-node range queries can be predicted when using the Join-based Strategy. This uses the results found in previous chapters and applies them to the particular steps required by the range query in this case.

Experiment 25, then, concludes the experiments by determining if the Brute Force Division algorithm is able to, using histograms and the formulas developed here, accurately predict the response times for sets of combined queries, using each possible declustering strategy and initial declustering level, and if, by doing so, it is able to accurately predict the best declustering strategy and level.

7.1 Experiment 23. Comparing Declustering Complete and Partial Datasets for Join Queries

Experiment 23 determines whether the methods to decluster sets for join queries found in Chapter 6 apply equally to portions of the same sets that result from range queries. That is, if we know the best level at which to decluster a pair of datasets to perform a single join operation, is this also the best level at which to decluster, if we instead wish to join only portions of these two sets.

Since we assume that both range query results are necessary and are not performed only for the purpose of performing the join, we cannot optimize the range queries to use only the overlapping area of the two query windows. For example, if it were known that two range queries, using Q1 and Q5, were performed only to join their results, then the range queries could be optimized by having both use only Q1.

Table 7.1 indicates that the best declustering level is generally the same or almost the same with complete and with partial sets. Therefore, when given a set of range and join queries, it is possible to estimate the time taken by the join queries, whether the range
query results are re-declustered for each join or not.

If we are using the Range-based Strategy, then for each range query we can estimate the output size, and from the standard deviations, we can calculate the declustering level that Division Based on Skew will choose. From the declustering level and the distribution of data across the levels, we can estimate the amount of extra work required due to replication. The method to estimate the output size of join operation at each node using a histogram, described in Section 4.8.1, can be applied to the portions of the data space covered by the query window, and we can therefore estimate the workload at the remote nodes to perform the Join Step. From this information, it is possible to estimate the time taken by the join query.

If we are using the Join-based Strategy, we can calculate the best declustering level for the joins on the range query results by finding the best declustering level for the complete datasets. Balancing this with the need to perform the range queries efficiently, we can choose the best level for the initial declustering. By predicting the distribution of range query results across the nodes, we can predict the Join Step time at each node, and by predicting the distribution of join query results across the nodes, we can predict the time for each node to return the results. We can, therefore, predict the time required for the joins using the Join-based Strategy as well.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Window 1</th>
<th>Window 2</th>
<th>Complete Sets</th>
<th>Partial Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Best Level</td>
<td>Best Level</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>MG</td>
<td>Q1</td>
<td>Q5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>MG.shift75</td>
<td>Q2</td>
<td>Q4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Fluid</td>
<td>Q3</td>
<td>Q3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Diagonal4</td>
<td>Cidade1</td>
<td>Q4</td>
<td>Q2</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Diagonal5</td>
<td>Islands</td>
<td>Q5</td>
<td>Q1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>LB</td>
<td>Diagonal1</td>
<td>Q1</td>
<td>Q5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LB.shift75</td>
<td>Diagonal2</td>
<td>Q2</td>
<td>Q4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>MG</td>
<td>Diagonal3</td>
<td>Q3</td>
<td>Q3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>MG.shift75</td>
<td>Diagonal4</td>
<td>Q4</td>
<td>Q2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Fluid</td>
<td>Diagonal5</td>
<td>Q5</td>
<td>Q1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Fluid</td>
<td>LB</td>
<td>Q1</td>
<td>Q5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Cidade1</td>
<td>LB.shift75</td>
<td>Q2</td>
<td>Q4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Cidade1</td>
<td>MG</td>
<td>Q3</td>
<td>Q3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Islands</td>
<td>MG.shift75</td>
<td>Q4</td>
<td>Q2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Islands</td>
<td>Cidade1</td>
<td>Q5</td>
<td>Q1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 7.1: Best Declustering Levels for Complete and Partial Datasets for Join Queries
7.2 Experiment 24. Predicting Response Times for Multi-node Range Queries When Using the Join-based Strategy

We have presented Formula (2) to predict the response time for multi-node range queries using the Range-based Strategy. With the Join-based Strategy, however, multi-node range queries require more time, because the data must be both returned to the primary node and logically written to the local disk. Since the data is replicated, the algorithm must also check to return only rows within cells for which the local node is responsible to the the primary node. This additional check requires approximately .4% more time for the range queries. They therefore require an additional .4% of 153 ms = 0.5 ms. As well, the range query times must be multiplied by the predicted replication factor and the result sets logically written to the local disk contain replicated rows. The size of the result sets at each node can be predicted by predicting the result set at each node, using a histogram as described in Section 4.8.1, and adding the predicted replication, using Formula (3). Given this, we can derive Formula (9):

Multi-node Replicated-Data Range Query Response Time =
\[
45 \text{ ms} + (\# \text{messages} \times P) \times (153 + .5) + (\# \text{messages}) \times 15 \text{ ms} \\
+ (\# \text{output pages} \times P) \times 0.43 \text{ ms} + 12 \text{ ms},
\]

where \( P \) is the ratio of the number of rows with and without replication \( (9) \)

Formula (9) is derived from the above formulas, and so any differences in CPU, logical-I/O or communication times in different environments would change those formulas, and Formula (9) would be changed accordingly.

Experiment 24 tests the accuracy of Formula (9) by recording the response time for range queries while using the Join-based Strategy. This is done for LB, declustered to 4, 8 or 12 nodes, using Q3. The results are given in Table 7.2. Table 7.2 indicates that the estimate for the range query response time based on Formula (9) is quite accurate.

7.3 Experiment 25. Accuracy of the Brute Force Division Method For Sets of Range and Join Queries

Experiment 25 tests the use of Brute Force Division to determine the best strategy and declustering level for a set of queries. For each set, the Brute Force Division algorithm estimates the time to perform the queries using both the Range-based and Join-based Strategies. For the Range-based Strategy, it uses Level 19 for the initial declustering, and assumes Division Based on Skew is used to re-decluster for each join. For the Join-based Strategy, it considers each possible declustering level for the initial declustering. Table 7.3 shows the selected strategy and selected declustering level based on this calculation. With the Range-based Strategy, “n.a.” is entered for Best Level.
Table 7.3 also gives the times to perform the query sets, both on a single node and using multiple nodes. The time for multiple nodes uses the strategy and declustering level determined by the Brute Force algorithm. This indicates that there is a noticeable speed-up as compared to the single-node case.

In this experiment, the set of queries were performed using both strategies, and each possible declustering level for the Join-based Strategy. This determined what was actually the best strategy and declustering level for the set of queries. These results are also shown in Table 7.3.

For the Range-based Strategy, we use the following calculation to estimate the total response time for the query set. For the range queries, we estimate the selectivity using a histogram. We then assume each node will receive an equal workload, and calculate the number of 5000-row messages the nodes will return. We then use Formula (2) to predict the range query response times. For each join, we use Formulas (4) and (5) to predict the Partition and Sort Step times on a single node. The standard deviations are calculated to determine the declustering levels that will be used to re-decluster for the joins. We then use Formula (3) to estimate the replication, and based on this, Formula (7) to predict the time to re-decluster for each join. Using a histogram and the declustering level, we predict both the number of rows sent to each node (without replication), and the selectivity at each node. From the number of rows sent to each node, and the replication, we use Formula (6) to predict the maximum time to perform the Join Step at the nodes. We then, using the predicted selectivity at each node, predict the maximum time for any node to send its result set to the primary node, using Formula (8). We assume that there are few enough nodes that the primary node is able to perform logical writes of the result sets as it receives them. Therefore, the time for the joins are: the time to re-decluster plus the maximum Join Step time of any of the nodes plus the maximum communication time of any of the nodes.

For the Join-based Strategy, for each possible declustering level, we use the following
calculation to estimate the total response time for the query set. We use Formula (3) to estimate the replication for the two data sets, given the declustering level. We then, for each range query, using a histogram, estimate the selectivity at each node. This is then adjusted, given the predicted replication. The range results are both returned to the primary node, and written to disk. The time to perform the range query and perform the logical write of its results is predicted by Formula (9). Each node may begin the Partition, Sort and Join Steps once the two range query results are logically written to disk. The time for the Partition, Sort and Joins Steps are predicted using Formulas (4), (5), and (6). The join query results must then be returned to the primary node. To predict the time for this we use Formula (8).

The difference between the two strategies is that, with the Join-based Strategy, range queries take longer, but joins require less time. The range queries take longer due to the replication, the poorer load balancing, and the necessity of removing the duplicates before the range query results are returned to the primary nodes. The joins require less time because, we do not have to re-decluster (though we do have the additional step of performing a logical write of the range query results to the local disk), and the Partition and Sort Steps are parallelized.

Table 7.3 indicates that the Range-based Strategy is best for Query Sets 1, 3, 4, 5, 6 and 9; the Join-based Strategy is best for Query Sets 2, 7, 8 and 10.

The query sets were chosen in order to capture the general characteristics of query sets where the Range-based and Join-based strategies are preferable. The query sets each contain a certain number of range and join queries. For example, Query Set 1 used Diagonal1 and Diagonal5. On Diagonal1, there were 10 range queries with Q1 and 50 with Q2. On Diagonal5, there were 50 queries using Q1, 70 using Q2 and 30 using Q3. The results of Q1 on Diagonal1 were joined once with the results of Q2 of Diagonal5.
CHAPTER 7. EXPERIMENTAL RESULTS FOR COMBINED RANGE AND JOIN QUERIES

Query Set 1

Range Queries on Diagonal1:  
10 × Q1
50 × Q2

Range Queries on Diagonal5:  
50 × Q1
70 × Q2
30 × Q3

Join Queries:  
1 × Diagonal1(Q1),Diagonal5(Q2)

With Query Set 1, since there were few joins, it was preferable to optimize the range queries.

Query Set 2

Range Queries on MG:  
40 × Q1
40 × Q2
45 × Q3

Range Queries on LB:  
55 × Q1
50 × Q2
35 × Q3

Join Queries:  
1 × MG(Q1),LB(Q2)
1 × MG(Q2),LB(Q2)
2 × MG(Q1),LB(Q3)

With Query Set 2, there are many joins, and even though MG has much data at high levels, and therefore requires significant replication, join queries are more expensive than range queries. Where there are a sufficient number of joins, optimizing for the joins is preferable.

Query Set 3

Range Queries on MG:  
40 × Q1
40 × Q2
45 × Q3

Range Queries on LB:  
55 × Q1
50 × Q2
35 × Q3

Join Queries:  
1 × MG(Q1),LB(Q2)
Query Set 3 was the same as Query Set 2, except there were fewer joins. Instead of four joins, the query set contained only one join. MG has a large amount of data at high levels, and there were not sufficient join queries to justify the replication necessary to decluster using the Join-based method.

**Query Set 4**

Range Queries on Diagonal4:  
$25 \times Q1$

Range Queries on Islands:  
$50 \times Q1$

Join Queries:  
$1 \times \text{Diagonal4}(Q1),\text{Islands}(Q2)$

With Query Set 4, the range results were small, and so the joins were fairly small operations. Given that, there were not a sufficient number of joins to use the Join-based Strategy.

**Query Set 5**

Range Queries on Fluid:  
$45 \times Q1$

Range Queries on Citadel:  
$50 \times Q1$

Join Queries:  
$1 \times \text{Fluid}(Q1),\text{Citadel}(Q1)$

With Query Set 5, the datasets are highly skewed, and the Join-based Strategy would require using a relatively low declustering level, which would cause a significant amount of replication. Therefore, the Range-based Strategy was best for this query set.

**Query Set 6**

Range Queries on LB.shift75:  
$100 \times Q1$

Range Queries on Diagonal1:  
$100 \times Q1$

Join Queries:  
$1 \times \text{LB.shift75}(Q1),\text{Diagonal1}(Q1)$
CHAPTER 7. EXPERIMENTAL RESULTS FOR COMBINED RANGE AND JOIN QUERIES

With Query Set 6, since the query windows were small, even though they were centred where the data is densest, the range query results were small, and the joins were fairly small. This caused the Range-based Strategy to be the best choice.

**Query Set 7**

Range Queries on LB.shift75: \(100 \times Q5\)

Range Queries on Diagonal1: \(100 \times Q5\)

Join Queries: \(1 \times \text{LB.shift75}(Q5),\text{Diagonal1}(Q5)\)

Query Set 7 was the same as Query Set 6 except the range queries used Q5 instead of Q1. With Query Set 7, the query windows were large, and therefore the result sets were large. This made the joins more significant operations, and made the Join-based Strategy preferable.

**Query Set 8**

Range Queries on Diagonal4: \(50 \times Q1\)

\(50 \times Q2\)

Range Queries on Diagonal5: \(50 \times Q1\)

\(50 \times Q2\)

\(30 \times Q3\)

Join Queries: \(1 \times \text{Diagonal4}(Q1),\text{Diagonal5}(Q3)\)

With Query Set 8, the two datasets are relatively uniform, and can therefore be declustered without much replication. Since the replication added relatively little overhead to the range queries, the Join-based Strategy was preferable.

**Query Set 9**

Range Queries on Fluid: \(5 \times Q1\)

\(5 \times Q2\)

Range Queries on Diagonal5: \(60 \times Q1\)

\(60 \times Q2\)

\(60 \times Q3\)

Join Queries: \(1 \times \text{Fluid}(Q1),\text{Diagonal5}(Q2)\)

With Query Set 9, most of the range queries were on the relatively non-skewed dataset. It was, therefore, undesirable to decluster this with replication, at the low level necessary to balance the heavily-skewed set. Therefore, the Range-based strategy was preferable.
Query Set 10 was identical to Query Set 9, but was performed using twelve, instead of four, nodes. With Query Set 10, the large number of nodes allows the Partition and Sort Steps to be performed much faster, since these are parallelized with the Join-based Strategy, but not with the Range-based Strategy. The Join-based strategy was therefore preferable.

The experiments were repeated using randomly-placed query windows, as opposed to centred query windows. It was found that, though randomly-placed windows had different selectivities than centred windows (except for with the relatively uniform sets), the range query response time per output page was the same. In deciding which of the Range-based or Join-based Strategies is preferable, the important factor is the output size of each range query, regardless of whether the window was centred or not.

Experiment 25 demonstrates that the formulas presented in this thesis to estimate the various steps of multi-node range and join queries, Formulas (2) through (9), are accurate, and that using them, we are able to select the best declustering strategy and level using the Brute Force Division technique. In all cases, the Brute Force algorithm was able to determine the best declustering strategy, and for the Join-Based Strategy, was able to estimate the best declustering level within two levels. The experiment also indicated that the total times at the actual best and predicted best declustering levels were generally quite close. Therefore, despite the best level not being chosen, the difference in response time was small. It is expected that the results will be still more accurate with larger datasets.

Experiment 25 also demonstrated that whether the Range-based or Join-based strategy is preferable depends roughly on the number of range queries versus the number of join queries. However, the size of the range query results, the distribution of data across the levels, the data skew and the number of nodes are also relevant.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis considered the problem of declustering rectangular spatial data to multiple nodes, in order to efficiently perform range and join queries using the Filter Tree and $S^3J$. To do this we considered two strategies: the Range-based Strategy, where the initial declustering optimized the range queries and the data was re-declustered to perform the joins; and the Join-based Strategy, where the data was declustered with replication such that it was not necessary to re-decluster to perform the joins.

We have developed four new techniques to divide a data space into fragments: Brute Force Division, Division Based on Skew, Variable Fragment-Size Division and Slicing. All four of these apply only to grid-based data structures, and the first two apply only to hierarchical grid-based structures. We examined the applicability of each of these to the problem of performing a large number of range and join queries on multiple nodes, and concluded that Brute Force Division is the most appropriate for the initial declustering, and Division Based on Skew is the most appropriate for re-declustering for single join operations. We also compared a number of techniques to distribute fragments to nodes, and of these concluded that HCAM is the most suitable to decluster data for the Filter Tree and $S^3J$.

Experiments were performed for range and join queries using single and multiple nodes. Formulas were presented to measure the times required by the various steps of range and join queries. The estimates provided by these formulas were individually shown to be accurate by the experiments. As well, it was finally shown that the Brute Force Division algorithm could apply the formulas to accurately determine the best declustering strategy and level for sets of queries.

With range queries on single nodes we found that, with sufficiently large datasets and query windows, the time required can be predicted from the size of the output set, and the output size can be estimated accurately using histograms, as is shown in Table 5.4. This implied that, when using multiple nodes, the time for each node can be estimated by predicting the number of objects it finds within the query window. Experiments showed this to be correct. With multi-node range queries, the remote nodes calculated the result set, and sent this in messages to the primary node. Experiments showed that the primary
node requires a certain amount of time to process each message it receives. Therefore, if there are sufficiently many remote nodes, the primary node will not be able to process the messages as fast as it receives them, and therefore will become a bottleneck. In this case, the time required for the range query depends only the number of messages sent to the primary node, regardless of how many of these messages are sent by each node. However, if there are few enough nodes that the primary node is able to process the messages faster then the rate at which they are received, the time required for the range query is determined by how well balanced the remote nodes are.

Experiments also showed that the load balance improves as lower declustering levels are used, but that a good load balance is often achieved at a fairly high level. This is shown in Table 5.5. As well, experiments showed that, since a good load balance can generally be achieved using HCAM at a low level, estimating the total query result size and assuming each node will return an equal portion of the result set is sufficient to estimate accurately the time required for a range query using multiple nodes without replication. However, it is only when using the Range-based Strategy that we necessarily use low declustering levels. With the Join-based Strategy, we generally use higher levels and cannot, therefore, assume there will be a good load balance across the nodes. In this case, to accurately estimate the response time, it is necessary to estimate the load balance. Histograms were shown to be useful to do this. Table 5.9 showed that using the derived formulas and a histogram, we can accurately estimate response times for multi-node range queries. Also, the additional costs for range queries when replication is used are straightforward to estimate, and therefore, the time to perform range queries using replicated data can be estimated equally accurately.

Experiments with join queries on single nodes demonstrated that the time required can be estimated from sizes of the two input datasets and the output set, and that the output size can be estimated accurately using a histogram. The accuracy of histograms for predicting join query output sizes is shown in Table 6.5. Experiments then examined performing single joins on multiple nodes, considering in particular the three main steps: declustering the data, performing the joins at the remote nodes, and receiving the result sets. The experiments showed that the time required by the declustering step was determined only by the size of the two datasets and the amount of replication, regardless of the number of nodes. The time required for each node to perform the join was, as in the single node case, determined by the sizes of the input sets. The time required to receive the results was, as with range queries, determined by the number of activated nodes, and the time required by the primary node to process each message it receives.

Further experiments indicated that the best level at which to decluster for joins is determined by the skew. This is shown in Table 6.12. Standard deviation was shown to be a satisfactory technique to measure the data skew. Based on this, we were able to develop a heuristic indicating the best level at which to decluster given the skews of two datasets. Experiments showed this heuristic to be quite accurate. We were therefore able to demonstrate that Division Based on Skew is able to quickly and accurately determine the best level at which to decluster given the skews of two datasets.

Experiments also showed it is possible to predict the time taken by a join query given the sizes of two datasets, histograms representing their data distributions, the number of nodes, and the declustering level, as shown in Table 6.6. Given this ability, it was shown
possible to, given two datasets and the number of nodes, calculate the best level at which to decluster for only join queries, by exhaustively considering each level.

We then showed it was possible to, given a set of queries, consider using both the Range-based and Join-based Strategies, and determine the best strategy, and, for the Join-based Strategy, the best declustering level for the initial declustering. The experiments, then, verified that Brute Force Division is, in fact, an effective technique to perform the initial declustering. This is shown in Table 7.3.

Given this, we found that the Range-based Strategy is most appropriate where: there are many more range queries than joins; the range query results are very small; where much of the data is at high levels, and performing the initial declustering with replication would cause excessive replication; and where one of the datasets is much more skewed than the other, but a large number of range queries are done on the less-skewed set. The Join-based Strategy is most appropriate where: there are many join queries; and where most of the data is at low levels, and therefore declustering for join queries results in little replication.

8.2 Future Work

Higher-dimensional Data Spaces

The algorithm developed for these experiments to calculate Hilbert values (described in Appendix C) works in two dimensions but is not easy to extend to higher dimensions. Some initial experiments we performed showed this algorithm to be approximately forty times faster than the algorithm to calculate Hilbert values used in Koudas's PhD work [Kou97]. However, that code can be used for any number of dimensions. Since no fast algorithm to calculate Hilbert values for high-dimensional data spaces has been developed, if Hilbert ordering is used, we are forced to use the slower algorithm, which will not affect I/O times but will significantly affect the CPU times to calculate the Hilbert ranges covered by the query windows for range queries, and to calculate the Hilbert values of all objects during the Partition Step of $S^3J$. In the case of range queries, for large sets, this may still require a relatively insignificant amount of time. For joins, this would be a significant increase in query time, though it would be expected to be less so as CPU performance continues to improve more rapidly than I/O performance.

Another alternative to this is to use z-order to order the objects within each level, and to perform the declustering. The z-order gives a clustering not as good as with Hilbert ordering, but it can be easily calculated for any number of dimensions. The z-order is found by assigning each cell within a grid a binary index for each dimension, where the index contains as many digits as are necessary to uniquely order each cell in a given dimension. The z-order of each cell is found by interleaving the digits of the indices. For example, if the data space were two-dimensional and were divided into four cells, two in each dimension, then each cell would have two one-digit indices and the cells would be ordered 00, 01, 10 and 11.
Reorganizing Data

One technique to improve the declustering described in Section 2.4.2, Incremental Techniques, could be applied to this work. It would be possible to, instead of having the data declustered in the same way indefinitely, have the processors analyse the distribution of queries they receive and determine if the data could be better organized. Dynamic Load Balancing, however, is not applicable to this work, since it involves moving portions of the complete datasets during single queries, and since the datasets are potentially very large, it is not productive to move data other than intermediate results.

Use of Non-Static Data

It would be possible, instead of working strictly with static data, to allow inserts, deletes and updates. Although most datasets, realistically, are not static, it is generally believed that the data distribution remains relatively stable over time [HS94]. It may, then, not be necessary, in general, to change the manner in which data is declustered. However, it would be necessary to implement concurrency control, and where there is replicated data, replication protocols would be required.

Multi-user Environments

In this thesis we have dealt only with single-user environments, which are environments where only one query is posed at a time, and where each node will perform only that query, until its execution is complete. In a multi-user environment, queries may arrive at any time, and, at any point in time, different processors may be executing different queries.

Future work could examine multi-user environments to determine how different strategies for declustering may be desirable. The level of multiprogramming, i.e. the rate at which queries are posed relative to the time to perform them, may affect the desired amount of intra-query parallelism, and may make declustering to minimize replication, as opposed to balancing workloads more important.

In a single-user environment, the goal, when performing any set of queries, is to decluster the data among the nodes such that the queries are done in the minimum amount of time, which is generally achieved by declustering such that the nodes have the min-max workload. Doing this, there is extra work in declustering the data, in communication costs, and in consolidating the results. In a single-user environment, this extra work may not be considered important, as long as the total execution time is reduced.

In multi-user environments, however, the extra work may be considered problematic, and we generally wish to decluster such that the total work is reduced. Often this is best achieved by eliminating intra-query parallelism and performing all queries on a single node. However, even in multi-user environments, we may still wish to reduce the response time for each individual query, and so eliminating intra-query parallelism is not always desirable.

In a multi-user environment, the amount of intra-query parallelism that is desired depends on how quickly the queries arrive. If the queries arrive slowly enough that each
query is completely executed before the following query arrives, then we have what is essentially a single-user system in any case. In this case, we want as much parallelism as is possible within each query, such that the communication costs do not overshadow the gains from parallelizing the queries. If the queries arrive very quickly, then likely the best solution is to simply maximize the overall throughput by eliminating intra-query parallelism and implementing only inter-query parallelism.

Where the queries come at a rate somewhere in between, it is not as clear what the desired goals should be. One approach is to maximize throughput such that no single query takes beyond a certain time limit. This means occasionally using intra-query parallelism, but not to the same degree as with a single-user environment. Another approach is to minimize the response time for each query, with a fixed limit on the amount of extra work performed.

In single-user environments, the only negative aspect of parallelism occurs when the overhead associated with declustering makes the operation more expensive than if the operation were performed on a single node, or on fewer nodes. With multi-user environments, it is also possible to experience resource contention. Important resources include the CPU, hard disk and buffer. Any future work dealing with multi-user environments must consider resource contention, and must organize the processes to minimize this.

Where the goal is to maximize throughput, we need a balance between having too little and too much intra-query parallelism. Too little intra-query parallelism may occur where queries are assigned to a single node and they execute entirely on that node. This can lead to some nodes being idle, while other nodes are unnecessarily performing large queries single-handedly. Too much intra-query parallelism can result in increased overhead, which will also cause the throughput to be reduced [CABK88].

Sending Partial Query Windows to Nodes

Another approach to performing range queries on multiple nodes is to decluster the complete sets to each node, and to send a fraction of the query window to each node. This would allow the primary node to maintain information about the occupancy of various portions of the data space, and to send the query windows to the nodes in such a way that each individual range query could be balanced, and would not depend on a single initial declustering. Remote nodes could calculate statistics about the time required to process the partial windows they receive, and the primary node could use this to improve its distribution of the partial windows. This may also be useful in multi-user environments, where we may wish to give certain nodes less work, if they are still calculating other queries. Using this method, the range results will be same, and therefore joins would be the unaffected. Nevertheless, the potential savings in range queries are worth exploring.

Declustering Using Slicing

Slicing appears to be a promising method to perform the re-declustering for single join operations. It appears to be quite fast and, in many cases, would require less replication than HCAM. It, therefore, seems useful for future work to experiment with Slicing, and
to determine specifically for which situations it is preferable.
Bibliography


Appendix A

The $S^3J$ Algorithm

The $S^3J$ consists of three main steps:

Partition: read in the input files, determine the level and Hilbert value for each object, and create the level files

Sort: sort the level files by their Hilbert values

Join: join the two sets of level files

Partition Step

The Partition Step divides each input file into a series of level files. As pages are read in from the input files, the algorithm loops through each object in the page. The Buffer contains a page representing each of the $L$ possible levels, where $L$ is the number of levels at which the data space is represented. Objects are placed in the appropriate page in the Buffer, and the Buffer pages are written to the corresponding level files as they become full.

create $L$ level files for each input dataset
loop for each input file
  loop until have read entire input file
    read in a page
    loop through each object in this page
      determine the level of the current object
      determine the Hilbert value of the current object
      write this object to appropriate page in Buffer
      if this page is now full, write it to the appropriate level file
    end loop
  end loop
write all non-empty pages in the Buffer to appropriate level files
end loop
delete all level files that received no objects
Sort Step

The Sort Step sorts each level file, one at a time. For each level file, it reads in as many pages as can fit in the Buffer at a time, and sorts these. If the entire level file can fit in the Buffer, then the records are simply sorted in memory and then written to disk. If the entire level file cannot fit in the Buffer, we must perform a merge sort; we read the level file in over a series of runs and during each run, we create a temporary file. After the last run we merge the temporary files, overwriting the level file created in the Partition Step. A structure, referred to as the MBuffer, has the same size as the Buffer, and is used to perform an in-memory quick sort on a set of records. It has a slightly different structure than the Buffer, which allows the Unix qsort operation to be performed on the MBuffer.

```plaintext
loop for each level file
  if the level file fits in the Buffer
    read all pages of the level file into the Buffer one at a time
    copy all records from the Buffer to MBuffer
    perform qsort on MBuffer
    copy sorted records from MBuffer back to Buffer
    write each page from the Buffer to the level file one at a time
  else // the level file does not fit in the buffer
    calculate number of runs = ⌈((size of level file) / (size of buffer))⌉
    loop for each run
      read in as many pages as will fit in Buffer, one page at a time
      copy all records from the Buffer to MBuffer
      perform qsort on MBuffer
      open a temporary file
      copy the sorted records from MBuffer back to Buffer
      write pages from Buffer to the temporary file
    end loop

  seek to beginning of each temporary file
  calculate cluster = (buffer size - 1) / (# temp files). This is how many pages are read from each temporary file at a time
  read cluster pages from each temporary file into the Buffer
  copy records into a structure called heapArray. This has an array for each temporary file. A set of pointers called heapCurrent keeps track of the current front of each array.
  compare the current front of each array and locate the array whose first element has the smallest Hilbert value
  copy the record to a buffer page
  if this page is now full, write to disk
  advance heapCurrent to the next record for this array
  if heapCurrent is at the end of the array, read next cluster pages
```
end if
end loop

Join Step

The join step reads each page of each level file in order, with the goals that each page is read only once, and that, at any time, each level file has at most one page in memory. These goals are usually attainable, but datasets can be constructed where the two goals are not simultaneously possible. The algorithm alternates between comparing a portion of a level file from the first dataset against the corresponding and higher level files in the second set, and comparing a portion of a level file in the second dataset against the higher levels in the first dataset. The range of objects considered each time are within the range minCurrent to minMax, and are placed in two data structures, called STable1 and STable2.

Assuming we are comparing one level of the first dataset to the corresponding and higher levels of the second dataset, the value minCurrent is found by locating the level of the first dataset with the smallest Hilbert value of all its objects that have not yet been compared against the second dataset. Figure A.1 gives an example of two datasets, where the data is in the first three levels, and data pages hold four objects. The numbers represent the Hilbert values of the objects in each level. Figure A.2 shows the same objects in terms of the data space. Where multiple levels have equivalent Hilbert values, the lowest such level is used. In this case minCurrent is found at Level 2 and is 0.

The value minMax is found by examining this level, the corresponding level in the opposite set, and all higher levels in the opposite set. We find the min-max value currently in memory for these four levels, which is 1, at Level 1. This can be seen by considering the largest Hilbert values for the four relevant levels, currently in memory, in terms of Level 2. These are Level 2, Cell 15: 15; Level 2, Cell 14: 14; Level 1, Cell 1: 7; Level 0 Cell 0: 15. Since, of these, 7 is the smallest, this is minMax. The value minMax is then calculated in terms of each level. At Level 0, it is 0, at Level 1, it is 1 and at Level 2 it is 7. We then copy the objects from these four levels in the range minCurrent to minMax into STable1 and STable2 and find the join-pairs. The objects at Level 2, in Cells 0 to 7, of the first set are then marked as having been compared to the second set.

After comparing this set of objects, the Join Step, in the same way, compares a set of objects within a certain range of Hilbert values from one level of the second dataset, with the objects within the same range of Hilbert values in the first dataset, for all levels higher than the corresponding level. We continue in this way until we have compared all the objects in each set to the other set. In this example, we would next compare Level 2 of the second set to Levels 0 and 1 of the first set. The first unprocessed cells at each level of the second dataset, in terms of Level 2 are Level 0: 0, Level 1: 0, Level 2: 0. Therefore, minCurrent would be Cell 0 of Level 2. The last cells in memory, in terms of Level 2, for the relevant levels are: Level 2, Cell 14 of the second set; Level 0, Cell 15 of the first set, and Level 1, Cell 15. Therefore, minMax is 14 at Level 2, and we search for join-pairs using the range 0 - 14 for Level 2 of the second set, the range 0 to 3 for Level 1 of the first set, and the range 0 to 0 for Level 0 of the first set. Cells 0 to 14 would be marked as having been compared to the first set, and, since all cells currently in memory
have now been compared to the other set, the next page for this level will be be read in.

The following time, we again compare a range of cells from one level of the first cell, to the corresponding a higher levels of the second dataset. The first unprocessed cells at each level of the first dataset, in terms of Level 2 are Level 0: 0, Level 1: 0, Level 2: 12. Therefore, minCurrent would be Cell 0 of Level 1. The last cells in memory, in terms of Level 2, for the relevant levels are: Level 1, Cell 15 from the first set; Level 1, Cell 7 from the second set; and Level 0, Cell 15 from the second set. Therefore minMax is 7 at Level 2, and we search for join-pairs using the range 0 - 3 for Level 1 of the first set, the range 0 to 3 for Level 1 of the second set, and the range 0 to 0 for Level 0 of the second set. Cells 0 to 3 would be marked as having been compared to the second set. This continues until all objects have been compared to the opposite set.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>0 0</td>
</tr>
<tr>
<td>Level 1</td>
<td>0 2 3</td>
</tr>
<tr>
<td>Level 2</td>
<td>0 1 12 15</td>
</tr>
</tbody>
</table>

Figure A.1: Join Operation with Three Levels

create JoinOutput file
read one page from each level file into Buffer (if there is a level file for this level)
loop until done performing the join (when all levels of both files are finished)
call getMinCurrent for first dataset (check each active level for lowest Hilbert value).
call getMinMax for the second dataset.
loop through each level of second dataset starting with the level of minCurrent and
going upwards to level 0
call SerialSearch to determine the locations of the set of records in this level of
the second dataset between minCurrent and minMax
copy these records from the Buffer into STable2.
end loop
copy records from Buffer into STable1 for the current level of the first file, in the range
minCurrent to minMax
call SerialSweep with STable1 and STable2

check if have finished all cells currently in memory for the current level for file 1
if have, read in another page
if there are no more pages, we’re done with this level of this file

   do the same, switching roles of file1 and file2
end loop

Algorithm for getMinCurrent

Each unfinished level of each dataset will have one page in memory at any time. Within
these pages, there is a set of objects at the beginning of the page that have been compared
to the other dataset, and after these there are the objects that have yet to be compared
to the other dataset. This function examines each level of one of the two datasets, to find
the level with the object with the smallest Hilbert value that has not yet been compared
to the other dataset. getMinCurrent also returns the last Hilbert value currently in
memory for this level of the input file.

set min to the Hilbert value of the next unprocessed object of the lowest unfinished level
loop through each level above this level
   if the next unprocessed object at this level has a Hilbert value less than min
      set this value to min
end loop
return min and the maximum Hilbert value currently in memory for min’s level

Algorithm for getMinMax

This examines the maximum Hilbert values for all objects currently in memory for a given
set of levels of one of the datasets. It considers the maximum Hilbert value currently in
memory for the level corresponding to minCurrent, and the maximum Hilbert value for
the relevant levels in the other dataset. It then returns the smallest of these maximum
values.

initialize minMax to maximum Hilbert value in memory at minCurrent’s level
loop through relevant levels of the dataset, starting with minCurrent’s level, going towards level 0
if Hilbert value at current level < \( \text{minMax} \)
    \( \text{minMax} = \) Hilbert value at current level
end if
end loop

Algorithm for SerialSweep

SerialSweep is given two data structures, STable1 and STable2, each containing a set of objects from one dataset. SerialSweep finds all join-pairs within STable1 and STable2.

loop for all objects in the first table
    loop for all objects in the other table
        if the two objects overlap
            write to an output page
            if the output page is full
                write output page to disk
            end if
        end if
    end loop
end loop

Algorithm for SerialSearch

set \( \text{begRangeIndex} = -1 \)
set \( \text{endRangeIndex} = -1 \)
loop through each object in the data page
    if the object’s Hilbert value is greater than \( \text{minCurrent} \) and \( \text{begRangeIndex} \) is -1
        set \( \text{begRangeIndex} \) to current object’s index
    end if
    if the object’s Hilbert value is greater than \( \text{minMax} \)
        set \( \text{endRangeIndex} \) to the current object’s index - 1
        return \( \text{begRangeIndex} \) and \( \text{endRangeIndex} \)
end loop

I/O for Partition Step

Read: each page of the two input datasets is read exactly once
Write: each page of the level files is written exactly once

CPU for Partition Step

The CPU work for the partition step consists of calculating the level file and Hilbert value for each object. Calculating the level is done by examining the bit representation of
the object's coordinates as described by Sevcik et al. [SK96]. The algorithm to calculate the Hilbert value is given in Appendix C. Both calculations are fairly fast. The time for the Partition Step depends only on the number of objects in the two datasets.

I/O for Sort Step

For levels where level file fits in the buffer:
- Read: each page of the level file is read exactly once
- Write: each page of the level file is written exactly once

For levels where level file does not fit in buffer:
- Read: each page of the level file is read exactly twice (once from the level file and once from the temp file)
- Write: each page of the level file is written exactly twice (once to the level file and once to the temp file)

CPU for Sort Step

For qsort, the worst case is $O(N^2)$ and the average case is $O(N\log N)$.

For levels where level file fits in buffer:
- $O(N)$ to copy from Buffer to MBuffer
- $+ O(N\log N)$ for qsort

$N$ is # objects in this level file.

For levels where the level file does not fit in the Buffer:
- $O(N)$ to copy from Buffer to MBuffer
- $+ O(b \log b \times N/b)$, to sort the Buffer $N/b$ times
- $+ O(N/b \times N)$, to compare the $N/b$ objects at the front of each of the temp files. Each time this is done, we remove one of the $N$ objects

$N$ is # objects in this level file.
b is the size of the Buffer, in # objects
$N/b$ is the number of temporary files created

I/O for Join Step

Read: each page of each level file is read exactly once
Write: each page of the output is written exactly once

CPU for Join Step

The main loop is called until all records in file1 and file2 have been checked. Therefore the main loop is called up to $N + M$ times, where $N$ is the number of objects in the
first set, and $M$ is the number in the second set. In general, each time through the loop will process several objects and not just one. That is, instead of just a single cell being compared to the corresponding cells in the other input file, a range of cells is compared to the corresponding ranges of cells in the other input file. Therefore, the main loop will actually execute less than $N + M$ times. Each time through the main loop getMinCurrent, getMinMax, SerialSearch and SerialSweep are called twice, once to compare file1 to file2, and once to compare file2 to file1. The times for getMinCurrent, getMinMax, and SerialSearch are bound by the number of levels and the size of the buffer, and can therefore be considered to take constant time. What affects the running time of the Join Step is how many times the main loop executes, and how many objects are compared by SerialSweep each time through the loop. Each time some number, $n \times m$, of objects are compared, but it is quite difficult to estimate how large $n$ and $m$ will be on average given any two datasets.
Appendix B

The Range Query Algorithm Using the Filter Tree

The algorithm first reads in the top-level index page, for each level, from the index file into the Buffer. It then determines the size of the input file and chooses the containment level based on this, using the heuristic presented by Koudas [Kou97]. It next calculates the ranges of Hilbert values that intersect the query window at the containment level. It then loops through each level and through this set of Hilbert ranges, and for each range reads in all data pages that include some part of this range. It compares each object in these data pages to both the current Hilbert range and the query window. As objects are found that overlap the query window, they are copied to a page in the Buffer, and when this page becomes full, it is written to the output file.

The procedure RangeSearch searches one level of the input for the objects overlapping the query window, within one Hilbert range. To do this, it first examines the first page of the index that represents this level. Each level may require either one or two levels of index pages; a dataset would have to be significantly larger than those examined in this thesis to require three or more levels of index pages. If only one level of index pages is required, then each cell within this index page contains pointers to data pages. If two levels are required, the first index page contains pointers to other index pages, which contain pointers to data pages.

As data pages are read in that contain some records that overlap the Hilbert range currently being considered, the procedure WriteIntersection is called to find all objects in this page, within the current Hilbert range, that overlap the query window. Only the current Hilbert range is considered, to ensure that records are not reported multiple times.

Algorithm for Range Query

calculate total size of the input file
look up containment level in in-memory table based on this
read index pages into the Buffer

loop through each cell along each border of the query window
  calculate the Hilbert value of both the cell just inside and just outside the query window
  subtract the Hilbert value of the cell just outside the query window from the Hilbert
  value of the cell just inside the query window.
  if the difference is 1
    this is the start of a Hilbert range; add this Hilbert value to a list of starting values
    of Hilbert ranges
  end if
  if the difference is -1
    this is the end of a Hilbert range; add this Hilbert value to a list of ending values
    of Hilbert ranges
  end if
end loop

sort the list of starting values of Hilbert ranges and the list of ending values of Hilbert ranges,
determining pairs of numbers of numbers that specify the Hilbert ranges of interest

loop through all levels from 0 to 19
  loop through each of the ranges of Hilbert values that intersect the query window
    call RangeSearch
  end loop
end loop

Algorithm for RangeSearch

read first page of the index file for this level
if there is only one index page for this level of the Filter Tree
  loop through every cell in the index page
    if this cell’s Hilbert Value is between the start and end of the range
      read the corresponding data page
      call WriteIntersection for this data page and the query window
    end if
  end loop
else
  loop through all pointers in the index page
    read index page pointed to by current pointer
    loop through every cell in the index page
      if this cell’s Hilbert Value is between the start and end of the range
        read the data page
        call WriteIntersection for this data page and the query window
      endif
  end loop
endif
Algorithm for WriteIntersection

loop through each object in the data page
  if the object’s Hilbert value is within the current range
    if the object intersects the query window
      write the coordinates of the object to an in-memory page
      if this page is now full, write to disk
    endif
  endif
end loop

I/O Analysis of Range Query

Each page of the index and data files whose Hilbert values overlap the Hilbert values of the query window must be read at least once. They must be read further times if they are checked again for another Hilbert range and are no longer in the buffer. With large datasets, we can assume that each level has a sufficient number of data pages that each time a page is referred to, it must be read in again. Therefore the number of times pages are referred to is equal to the number of disk reads. With an extremely large dataset, each cell of each level would require at least one data page, and therefore, the number of pages referred to would equal the number of pages of output, and so the number of disk reads would equal the number of output pages.

For smaller datasets, it is much more difficult to quantify the number of disk reads. The number of logical reads must be at least equal to the number of logical writes, so this may be considered the best case. The worst case would occur when the cells, at each level, that are within the window are sparsely populated, and the cells adjacent to the window are very densely populated. If this occurs to a sufficient extent, then one disk read may be necessary for each row of the output. This may be considered the worst case; however, this is a highly unusual case, and the average case is somewhere between the two extremes.

CPU Analysis of Range Query

The CPU work consists of calculating the ranges of Hilbert values covered by the query window at the containment level, and comparing the query window to objects read from the datafile. Calculating the ranges of Hilbert values at the containment level takes an amount of time that depends only on position of the window. The time is not affected by the data, once the containment level is chosen. As well, experiments (not included in this thesis) indicated that the time to calculate the Hilbert ranges was only a tiny portion of the overall time to perform the range query. With larger sets, this portion would become
much smaller, and therefore, we will consider here only the time to compare objects to the query window.

Each page that has at least one object that is within the query window may be considered relevant, and must be checked at least once. Relevant pages are, in fact, checked once for each of the query window’s Hilbert ranges with which they coincide. Each time a page is checked, every object is first compared to the current Hilbert range. This requires two checks: the Hilbert value of the object must be compared against the start and end of the current Hilbert range. If it is within this range, four additional compares are performed. These check the \( x \) and \( y \) coordinates of the object against the \( x \) and \( y \) coordinates of the query window. Each object in the relevant data pages are then checked either two or six times. Most data pages contain 170 objects, though the last data page of each level may have any number from 1 to 170.

The CPU time per referenced page is, then, fairly constant, and the key issue in determining the amount of CPU work is the number of data pages referenced, that is, the number of relevant pages, and the number of times these are referenced. As with I/O, for large datasets, the number of pages referenced will approach the number of pages output, and with smaller sets, the best case is equal to the number of pages output, and the worst case is the number of rows output.

**Improvement to Range Query Algorithm**

One improvement to the range query algorithm, which has not yet been implemented, addresses the problem of data pages being referenced more than once, and therefore often being brought into memory more than once. Instead of looping through each of the ranges of Hilbert values that intersect the query window, it would be possible to, for each level, examine the index pages and determine which data pages’ ranges of Hilbert values intersect the query window’s range of Hilbert values. We could then read in these pages once each, and for each compare each object to the query window. This would likely be only a small improvement for very small, or large datasets, but would be a noticeable improvement for sets where pages are referenced multiple times, but cannot be kept in the buffer.
Appendix C

Algorithm to Determine Hilbert Values

The algorithm to determine Hilbert Values, developed for this thesis, accepts the x and y coordinates of a point, and the level at which we wish to place this point, and returns the Hilbert value of the cell within the specified level in which the point is contained. To find the Hilbert value for a rectangle, we calculate first the level at which the rectangle belongs. We then calculate the centre of the rectangle, and calculate the Hilbert value of this point at this level.

The algorithm first finds the cell in which the point is contained at Level 1, then Level 2, and so on until we have reached the desired level. To find the cell in which the point is contained at Level 1, we simply calculate which quadrant contains the point. At Level 2, we will again have four choices, since there are four cells that are covered by each cell at Level 1. Based on which of these four cells the point is contained in, we can find the Hilbert value at Level 2, by multiplying the Hilbert value at Level 1 and adding either 0, 1, 2 or 3. The four possible cells in which the point may be contained at Level 2 may be ordered four different ways, depending on which of the four cells the point was contained in at Level 1. The four orderings are shown in Figure C.1, and are referred to as Normal (N), Left Rotate (L), Right Rotate (R) and Upside Down (U).

![Figure C.1: Four Orderings For Cells](image)

To determine how the cells are ordered, we need only consider how the four choices at the above level were ordered, and which of those four cells contains the point. Figure C.2 shows how the four cells at the next level will be ordered for any given current ordering.

To determine how the cells are ordered, we need only consider how the four choices at the above level were ordered, and which of those four cells contains the point. Figure C.2 shows how the four cells at the next level will be ordered for any given current ordering.
For example, at Level 1 the four cells are ordered Normal. If the point is contained in the lower-left cell, then the four cells in which it may be contained at Level 2 are ordered in a Left Rotate order.

<table>
<thead>
<tr>
<th>Normal</th>
<th>Left Rotate</th>
<th>Right Rotate</th>
<th>Upside Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>U</td>
<td>R</td>
<td>L</td>
</tr>
<tr>
<td>N</td>
<td>L</td>
<td>U</td>
<td>R</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>N</td>
<td>U</td>
</tr>
</tbody>
</table>

Figure C.2: Determining Ordering for Cells at the Next Lower Level

An example of calculating a Hilbert Value at Level 3 is given in Figure C.3. To find the Hilbert value of the point at Level 1, we first calculate that it is in the upper-left quadrant of the data space. This gives us a Hilbert value of 1 at Level 1. We then determine which of the four possible cells at Level 2 contains the point. It is located in the lower-right quadrant. We know from Figure C.2 that these four cells are ordered in a Right Rotate order, and from Figure C.1 that the lower-right cell in a Right Rotate order has value 3. Therefore we can calculate the Hilbert value at Level 2 by multiplying the Hilbert value at Level 1, which was 1, by 4 and adding 3. This gives $1 \times 4 + 3 = 7$. We then calculate which of the four possible cells at Level 3 contains the point. It is in the upper-right quadrant. From Figure C.2 we know that these cells are ordered in an Upside Down order, and from Figure C.1 that the upper-right quadrant in an Upside Down order has value 0. Therefore the Hilbert value at Level 3 is $4 \times 7 + 0 = 28$. 

Figure C.3: Calculating a Hilbert Value at Level 3
Appendix D

Algorithm to Generate Synthetic Data

D.1 Algorithm to Generate Diagonal Files

This algorithm generates rectangles randomly, and keeps or discards them based on the desired skew and their distance from the main diagonal. It continues to generate rectangles randomly until the desired number have been kept. The algorithm first generates a point representing the lower-left corner of the rectangle. To do this, it randomly generates three numbers, the x and y coordinates and a number called keeper, used to determine whether or not to keep the point. Keeper is then multiplied by a constant called skewConstant which is based on the skew. For the five skews, the value of this constant is: Diagonal1: .103, Diagonal2: .175, Diagonal3: .226, Diagonal4: .421, and Diagonal5: 1.00. If the algorithm chooses to keep the point, it then determines the size of the rectangle. The extents in the x and y dimensions are chosen randomly and independently such that the extents are distributed uniformly from 0.0000019 to 0.000019 of the data space. These sizes were chosen because this is approximately the range of sizes of the objects in the five real sets used in this thesis.

rectanglesCreated = 0
while rectanglesCreated < desiredNumber
    do
        create random x and y coordinates
        create random value for keeper
        while (abs(x - y) > skewConstant * keeper)
            choose random extent for rectangle in x dimension
            choose random extent for rectangle in y dimension
            write rectangle to output file
        end while
end while
D.2 Algorithm to Generate Second Synthetic Set

This creates a set of square rectangles which are placed at even intervals. Since the rectangles are square, only the calculations of the lower and upper x coordinates are shown; the y values are the same.

if dataset == Data1
    rectSize = .0000381
if dataset == Data2
    rectSize = .0002286
if dataset == Data3
    rectSize = .0000381
if dataset == Data4
    rectSize = .0000381
if dataset == Data5
    rectSize = .0002286
if dataset == Data6
    rectSize = .0000381
xl = 0
while rectanglesCreated < desiredNumber
    xh = xl + rectSize
    write rectangle to output file
    if (dataset == Data3 or Data6)
        rectSize = rectSize + 0.0000381
        if (xl + rectSize > MAX_DIM)
            rectSize = 0.0000381
    end if
    if dataset == Data1
        xl = xl + .0000954
    if dataset == Data2
        xl = xl + .0013356
    if dataset == Data3
        xl = xl + .0000954
    if dataset == Data4
        xl = xl + .0000286
    if dataset == Data5
        xl = xl + .0001509
    if dataset == Data6
        xl = xl + .0001509
        if (xl+rectSize > MAX_DIM)
            xl = 0
    end while