Development of a Big Spatial Data Management System

a diploma Thesis by

Thanasis Georgiadis

Advisor:

Nikolaos Mamoulis, Professor

Deparatment of Computer Science & Engineering

University of Ioannina

June 2021

Dedication

To my brother and sister, Konstandinos and Eleni, of whom I am infinitely proud.

Acknowledgements

Special thanks to Dimitris Tsitsigkos and Dinos Lampropoulos for their unlimited help and patience.

Also, to my supervising professor Nikos Mamoulis, for his mentoring and guidance.

Table of Contents

Abstract

Thanasis Georgiadis, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, June 2021.
Development of a Big Spatial Data
Management System.
Advisor: Nikolaos Mamoulis, Professor.

The purpose of this thesis is the development of a big spatial data management system in a distributed environment, using hybrid programming techniques. More specifically, the system performs a parallel partitioning on the big data, by assigning the records to cells of a fictional two-dimensional grid, based on their position in it. It then distributes the data accordingly to independent computers, with each receiving the contents of specific cells exclusively. The computer nodes can then perform actions on their received data independently of each other, applying different range queries depending on the user's needs. The system is implemented in C++ using the libraries OpenMP and MPI for parallel programming and inter-process communication respectively.

Περιληψη

Θανάσης Γεωργιάδης, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2021.
Ανάπτυξη Συστήματος Διαχείρισης Μεγάλων Χωρικών Δεδομένων.
Επιβλέπων: Νικόλαος Μαμουλής, Καθηγητής.

Ο σκοπός της συγκεκριμένης διπλωματικής εργασίας είναι η ανάπτυξη ενός συστήματος διαχείρισης μεγάλόυ όγκου χωρικών δεδομένων, παράλληλα και σε κατανεμημένο περιβάλλον. Πιο συγκεκριμένα, το σύστημα πραγματοποιεί παράλληλο διαχωρισμό πάνω στα δεδομένα, αναθέτοντας τις εγγραφές στα κελιά ενός πλασματικού δισδιάστατου πλέγματος, ανάλογα με τις θέσεις τους μέσα σε αυτό. Έπειτα διαμοιράζει τα δεδομένα στους υπολογιστές του κατανεμημένου περιβάλλοντος, με τον καθένα να λαμβάνει αποκλειστικά τα περιεχόμενα συγκεκριμένων μόνο κελιών του πλέγματος. Στην συνέχεια, οι υπολογιστές-κόμβοι του περιβάλλοντος μπορούν να απαντήσουν σε χωρικά ευρετήρια εύρους ανάλογα με τις ανάγκες του χρήστη. Το σύστημα υλοποιείται σε C++ κάνοντας χρήση των βιβλιοθηκών OpenMP και MPI για τις παράλληλες λειτουργίες και την διαδιεργασιακή επικοινωνία αντίστοιχα.

Chapter 1

Introduction

This first chapter consists of an introduction to the subject of the thesis as well as its structure and chapter description.

1.1 Parallelism on Big Data

1.1.1 Distributed Environments & Parallel Programming

Efficient big data management is becoming more vital in modern data processing every day. Humans generate massive amounts of information daily and some data sets are too large or complex to be processed with traditional data-processing software applications. Spatial data is one of the complex types of big data, including map coordinates, GPS signals, communication ranges, medical application data and many more.

Serial processing of big data is extremely time consuming and inefficient due to its vast size, making parallel programming and distributed environments much more preferable options. The database in such systems is divided into the distributed environment's nodes, an operation called **partitioning**. Thus, each node may search its corresponding sub-database to answer queries, effectively reducing the total time and parallelizing the whole process, especially when communication between nodes is unnecessary or minimal.

Additionally, every computer in the environment may utilize their available processor threads to further parallelize costly functions such as data record iteration and large amounts of comparisons. Implementations that make use of both distributed environment techniques and parallel programming are usually being referred to as **hybrid programs**.

1.1.2 Spatial Range Queries

Effective and efficient management of large data volumes is necessary in virtually all computer applications, from business data processing to library information retrieval systems, multimedia applications with images and sound, computer-aided design and manufacturing, real-time process control, and scientific computation [1].

Databases that contain spatial data, have to provide different types of query evaluations depending on the represented objects' qualities. However, since spatial data represents entities that can be positioned in 2D or 3D space, the most important service the database needs to provide is usually **range queries**. This means being able to return a group of objects that are contained within specified borders, that intersect some line or other shape, or which include a set of specific coordinates. In this thesis we will focus on **window queries**, which are represented as rectangles in two-dimensional space. Any object that is contained inside the specified window or that intersects its borders is considered a result to that window query.

The total number of comparisons between the database records and the query's values needed to evaluate it may vary and there are many techniques and algorithms that manage to reduce this volume, effectively reducing the total cost. However, some amount of comparisons is always required to retrieve the results and thus, in large data sets, some significant computational cost always exists.

By using hybrid programming techniques, the query evaluation process can be parallelized up to a degree so that the inevitable costly operations may at least be performed in a parallel way.

1.2 Related Work

The need for efficient big spatial data management has led the research community to develop multiple modern spatial data analytics systems. Each one with its own unique implementation and features, these systems are being used by academia and industry worldwide.

Most of the spatial data analytics systems run on large-scale data processing and distributed computing engines such as Apache Spark [2] and Hadoop [3], with the first prevailing in performance and thus being preferred by the community.

Some of these systems such as Apache Sedona [4] (formerly GeoSpark), Magellan [5] and SIMBA [6] along with their performance for various data types and datasets are analyzed in depth and thoroughly discussed in [7] but due to resource and time limitations, we do not compare the implemented program with any of them. In the following subsections we briefly describe the systems' characteristics, strengths and weaknesses.

Apache Sedona

Apache Sedona [4], formerly 'GeoSpark' before being acquired by ASF, is a cluster computing system for processing large-scale spatial data that extends Apache Spark / SparkSQL. With a set of out-of-the-box Spatial Resilient Distributed Datasets (RDD) / SpatialSQL that efficiently load, process, and analyze large-scale spatial data across machines.

It supports spatial range, kNN and join queries but not kNN joins. It runs on Java/Scala and supports different types of spatial data such as Points, Polygons, Rectangles and LineStrings. It offers multiple spatial proximity partitioning schemes such as KDB-Tree, R-Tree, Quad-Tree and more. The indexing can be performed using either R-Tree or Quad-Tree.

Sedona currently beats the rest of the spatial data analytics systems in terms of performance and speed. Along with providing a wide variety of query and data types, it dominates the query evaluation benchmarks and is considered the prevailing system at this moment.

However, Sedona is an in-memory cluster computing framework, meaning that for big datasets it has high memory consumption, even reaching almost 3x more memory for every dataset [7] (p. 1667).

Magellan

Magellan [5] is a distributed execution engine for spatial analytics on big data. It is implemented on top of Apache Spark and deeply leverages modern database techniques like efficient data layout, code generation and query optimization in order to optimize geospatial queries.

It supports a great variety of data types such as points, rectangles, polygons, linestrings, multipoints and multipolygons. It supports range queries and spatial joins but does not support kNN queries, distance joins and kNN joins. It uniquely adds geometric predicates such as intersects, within and contains. It indexes the data using Z-curve but can also leverage the indices if they were persisted earlier. It uses the Z-curve in a way that acts as a spatial partitioning technique, by inner joining and then filtering the datasets.

Magellan has lower memory footprint than Sedona, however the indexing costs, especially for linestring, are huge. It scales well but its query evaluation remains worse than Sedona's in most cases. It does not have an optimization option, ending up scanning all partitions for all datasets.

SIMBA

SIMBA (Spatial In-Memory Big Data Analytics) [6] is a distributed in-memory spatial analytics engine based on Apache Spark. It extends the Spark SQL engine across the system stack to support rich spatial queries and analytics through both SQL and DataFrame query interfaces. Besides, Simba introduces native indexing support over RDDs in order to develop efficient spatial operators. It also extends Spark SQL's query optimizer with spatial-aware and cost-based optimizations to make the best use of existing indexes and statistics.

To partition the data, Simba uses a method where an R-tree is constructed by sampling the input dataset and filled using the STR algorithm [8] to get the first level of the tree that represents the partition boundaries. It provides flexibility to the user to specify its own partitioning scheme. It uses an R-tree for indexing by default. It supports range (rectangle and circle) queries, kNN queries (points), distance joins (points) and kNN joins (points) but not spatial joins. One of Simba's greatest features is that it optimizes the index so that queries can be executed in parallel, increasing analytical throughput.

Simba is one of the best frameworks for point datasets regarding indexing costs. However, in evaluating range queries, Sedona still performs better than Simba.

1.3 Objectives

This thesis' goal is to create a system that effectively partitions big sets of data into a distributed environment and efficiently evaluates range queries regarding objects' MBRs. It is based on the Two-Level Partitioning [9] method, attempting to improve its performance by utilizing the hybrid programming's benefits.

We aim to create a system that efficiently partitions and indexes big spatial data to any number of machines, in a way that enables fast range query evaluation. Our goal is to achieve better overall performance, less memory usage and faster query evaluation than the currently existing systems. We focus on non-point spatial data (rectangles) and window range queries, instead of a grand variety of data types and queries. By the end of this thesis, we aim to have a functioning first version of the system that performs all of the above and works for computers with different specifications and limitations, with respect to memory usage and network bandwidth. Finally, it must provide scalability and be open to future extensions and improvements.

In summary, this thesis will show the methodology and line of thought that lead to the system's current version. The following chapters:

- Examine ways to optimize the parallel partitioning
- Analyze hybrid methods for the range query evaluation
- Evaluate the system's performance
- Suggest possible future extensions and modifications

1.4 Structure

This thesis is organized in 7 chapters. Their contents are briefly explained below.

Chapter 1 contains a brief introduction to the thesis' subject and goals.

Chapter 2 describes the tools, programming language, libraries and compilers used to implement the system as well as the technical specifications it needs to be able to run.

Chapter 3 gives an overview of the system. It also describes the environment in which the system was tested in as well as the resources used during research and development.

Chapter 4 covers an in depth analysis of the parallel partitioning. It compares different implementations for its individual operations and backs the final version with real time experiments.

Chapter 5 thoroughly describes the query evaluation part of the system. It tests multiple parallel versions and attempts to determine the optimal way for it to be implemented.

Chapter 6 describes extensively how the user may setup the system and use it to fit their needs. It also provides examples of how the system's configuration may affect its performance.

Chapter 7 contains the thesis' conclusion and suggestions for future work.

Chapter 2

OpenMP & MPI

2.1 OpenMP

The OpenMP [10] API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. It defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

OpenMP is fully supported by almost all modern compilers but this thesis will cover its usage with the C++ MPI compiler *mpicxx*, which is needed for compiling and linking MPI programs, as explained in Section 2.2.

Like most shared memory programs, OpenMP is usually used with the **fork/join model** meaning that only one thread is active at the beginning of the program until more are created by the code. Usually, threads are generated for specific purposes and are then destroyed or *joined* when they finish their task [11]. These threads are usually referred to as **workers** while the creator thread is typically called **master**.

2.1.1 OpenMP Overview

OpenMP offers a collection of *#pragma omp* commands that parallelize sections of code in various ways, without altering them. Standard serial compilers ignore these commands, preserving the program's ability to be compiled and run serially.

The most common and arguably primitive of the aforementioned type of commands is `#pragma omp parallel`. It creates a new set of threads and then each one, along with the creator thread, executes the code that is encapsulated in the command's curly brackets. The moment that all of the threads reach the end of the parallel region, the generated threads are destroyed leaving the creator thread to be the only one executing the next lines of code, if any.

The `#pragma omp` commands take arguments that specify various aspects of their functionality. For example, `#pragma omp parallel num_threads(N)` instructs the compiler to create exactly $N - 1$ more threads, so that there will be N threads in total executing the specified area's code. This option combined with OpenMP's default function `omp_get_max_threads()`, which is explained below, allows the developer to make sure that the program uses all of the system's available resources, in terms of thread usage.

2.1.2 Sharing Attributes

OpenMP is based on shared memory, so it is crucial that the program handles variable scopes with caution. When a group of new threads is created, they all share the same address space for shared variables as the creator thread. It is the programmer's responsibility to make sure that changes to these variables are carried out properly in order to ensure consistency. By adding one of the following arguments in the `#pragma omp` command, the developer can specify which of the creator thread's variables need to be shared among the threads and which do not:

- shared(x,y,...): The specified variables will be shared by the threads. Value updates by the threads must be properly protected by the programmer.
- private(x,y,...): The threads will each receive a copy of the specified variables, private to them. Only the creator thread's instance of the variable will continue to exist after the parallel region.

- firstprivate(x,y,...): Works like the private() argument, but each copy is initialized to match the creator thread's variable current value.

Moreover, any variables declared by a thread inside the parallel area in which it exists, will be private to them. This makes it easier for the developer to implement complex operations and algorithms by individual threads, without having to declare all variables beforehand and adding them to the `#pragma` command's private field. All worker threads' private address spaces are freed when they join and any data not saved in shared space known to the master thread will be lost. Any variables declared inside the parallel area, including the master thread's, are also deleted when all the threads join.

2.1.3 Useful Commands

OpenMP provides many commands that give flexibility to the developer to approach problems from different angles. Some of the following commands were used in the implemented system while others are being mentioned simply for being noteworthy.

One very important issue in shared memory systems is mutual exclusion, meaning that there may be code that needs to be accessed serially by the threads and not simultaneously. For instance, when a thread is attempting to alter the value of a variable shared by the rest of the group. It is a classic example where mutual exclusion is needed, due to the fact that parallel changes to it will probably lead to random results. The `#pragma omp critical` command specifies a **critical area**, so that any code enclosed inside will be executed by just one thread at a time. More specifically, when a thread reaches the beginning of the critical area, it continues inside only if no other thread is currently executing the protected code, otherwise it waits until that code becomes available. The order in which the threads enter the critical area is random.

In case there is code that must be executed *only once*, OpenMP provides the command `#pragma omp single` which allows only one thread to execute the enclosed code while the rest ignore it. The first thread that encounters the command is the one which will execute it and upon semi-simultaneous arrival of multiple threads, it is random which will be the one that enters the area.

Utility

OpenMP offers some utility commands that can be used by any thread, anywhere, to collect valuable information about the program's state or the system in general.

- `int omp_get_max_threads()`: Returns an upper bound on the number of threads that could be used to form a new team. If only one thread is in use, this routine effectively informs the program of the maximum available threads in the system.
- `int omp_get_thread_num()`: Returns the number or *ID* of the executing thread inside the group. It allows threads to identify themselves and to help the developer further customize parallelization. Outside a parallel area or in serial portions of code, it always returns 0.
- `int omp_get_num_threads()`: Returns the number of active threads in the team currently executing the parallel region from which it is called. Outside a parallel area or in serial portions of code, it always returns 1.
- `int omp_set_num_threads(int N)`: Sets the number of threads to be used in the next parallel area to be *N*. This can help pre-define the number of threads needed, to avoid using the `num_threads()` clause in subsequent parallel commands.

2.2 MPI

The Message Passing Interface (MPI) [12] is a standardized message-passing prototype designed to function in a wide variety of parallel and distributed computer architectures. It allows inter-process communication between processors of the same or different computers and supports a variety of high-speed networks through which they can communicate. Both point-to-point and collective communications are supported, rendering MPI the go-to standard for process communication among the parallel programmers and shared memory system developers.

Two major MPI implementations along with their respective compilers are currently dominating the community, being preferred as the de facto message passing interfaces in most shared memory systems of any kind. OpenMPI [13] and MPICH [14] are both developed by an international group of collaborators, originating from both academic backgrounds and the private sector. This thesis uses the MPICH implementation and the *mpicxx.mpich* compiler version 3.3.

2.2.1 MPI Overview

Usually, at the beginning of an MPI program, the user selects how many processes or *nodes* are needed. The creating process called *parent* will always be in the local processor of the computer executing the program, but the worker processes can be either in the same system or in an another computer, connected through a network. MPI programs are intended to run on homogeneous networks such as clusters, but can also run in a long distance heterogeneous network if needed.

Each process is assigned an integer identifier called **rank** beginning from 0, so that there is a way that the user/system can distinguish the nodes of the environment. The parent process is always assigned rank 0, so that it can be easily identified.

The processes execute the program in its entirety and it is the programmer's responsibility to organize them depending on their needs. When they need to communicate with each other, the processes may do so by a collection of communication commands offered by the MPI. These functions can be point-to-point or collective and will be thoroughly discussed in the following subsections.

Each process has its own private memory, along with copies of the program's defined variables which are completely private. Since MPI does not follow the shared memory model, there is no need for mutual exclusion mechanisms. All the processes run in parallel without interfering with each other and when needed, the communicate with messages.

2.2.2 MPI Setup

At the beginning of every MPI program, the execution environment must be initialized. This is done with the command

```
int MPI_Init(
      int *argc,
      char ***arv,
)
```

which is called by all the processes specified by the arguments. However, this command initializes the environment for single thread execution, meaning that individual threads created by OpenMP will not be able to make MPI calls. Since the implemented system needs every thread to be able to communicate with the environment's processes independently, the initialization must be carried out with the command

```
int MPI_Init_thread(
      int *argc,
      char ***arv,
      int required,
      int *provided,
)
```

It works in the same way as the serial initialization, with the main difference being that the value of the field *required* will set the environment's thread-process communication behavior accordingly, depending on the program's needs. From the official MPI documentation, the field's value may be:

- `MPI_THREAD_SINGLE` - Only a single thread in the program will execute.
- `MPI_THREAD_FUNNELED` - The process may contain multiple threads, but the thread that called `MPI_Init_thread` is the only one that makes MPI function calls.
- `MPI_THREAD_SERIALIZED` - The process may contain multiple threads, and all of those threads may make MPI function calls, but only one at a time.
- `MPI_THREAD_MULTIPLE` - Multiple application threads may call MPI functions with no restrictions. This value is currently only supported on MS-MPI V6 running on Windows Server 2012, Windows Server 2012 R2, Windows 8, and Windows 8.1.

Since the system needs complete freedom regarding threads making MPI calls, the environment will be initialized by passing `MPI_THREAD_MULTIPLE` at the required field's value, which the function stores in the *provided* local variable.

Moreover, the program must properly finalize the MPI environment before finishing. This is done by simply calling the `MPI_Finalize()` routine which must be placed at the program's very end and must be called by every process's main thread before exiting. After this routine is called, no other MPI function calls may be made, including the MPI initialization methods.

2.2.3 Utility Commands

Various organization models and parallel implementations require the participating nodes to know the total size of the environment as well as their position in it. MPI provides two commands that can inform the nodes of their own rank and the total of active processes in the environment.

```
int MPI_Comm_size(
      MPI_Comm comm,
      int *size,
)
```

saves the total number of available processes in the communicator into the *size* local variable and can be called any moment during runtime. The same goes for

```
int MPI_Comm_rank(
      MPI_Comm comm,
      int *rank,
)
```

which saves the rank of the caller process into the *rank* local variable.

MPI also offers a high resolution timing function to properly measure time during a parallel program:

```
MPI_Wtime ()
```

which returns an elapsed time on the calling processor, avoiding any possible confusion between the environment's processes.

2.2.4 Inter-Process Communication

The most important part of a non-shared memory environment of multiple processes is their cooperation, which is based solely on communication and information exchanging. The nodes must be able to communicate with each other both point-to-point privately between them and collectively all together.

The standard point-to-point information exchanging methods for sending and receiving messages respectively are

```
int MPI_Send(
      void *buf,
      int count,
      MPI_Datatype datatype,
      int dest,
      int tag,
      MPI_Comm commm,
)
```

and

```
int MPI_Recv(
      void *buf,
      int count,
      MPI_Datatype datatype,
      int source,
      int tag,
      MPI_Comm commm,
      MPI_Status *status,
)
```

Processes refer to each other using their ranks, passing them as arguments to the *dest* and *source* fields informing the communicator of the message's destination or to filter incoming messages accepting only those originating from a specific source node. The receiver node may specify the `MPI_ANY_SOURCE` constant into the source field to accept all incoming messages, regardless of the sender node's rank.

The message's size is mandatory for the functions to work and is passed as an integer number of bytes into the *count* field. The receiving node needs to have allocated enough memory for its buffer before accepting a message, otherwise the method will fail. The communicator also needs to know the type of the data being passed through the channel. This is indicated by the *datatype* field's value which can be any one of the MPI's supported data types, containing mainly primitive types such as *char*, *int* or *long*. Complex data structures such as C++ vectors, class objects or even standard strings must be transformed into one of the supported data types before being sent.

Messages may also contain a specific numeric tag, which can be used to further filter or categorize incoming messages. Receivers may disregard tag filtering by specifying the `MPI_ANY_TAG` constant. If a tag is specified by the receiver, then only messages with the exact same tag will be accepted and the rest will be disregarded.

These functions are **blocking**, meaning that they perform an operation and do not return until it is completed. For example, `MPI_Recv()` does not return until a message matching the specified conditions is received in its entirety, otherwise the calling process is waiting indefinitely. On the other hand, `MPI_Send()` does not return until the buffer has been completely transmitted through the channel. Regardless of the data being received by a process or not, it returns the moment that the sending buffer can be safely reused again.

The receiving function has one extra field pointing to a special `MPI_Status` structure that contains useful information about incoming messages, including their size in bytes. This structure is needed in cases that the receiver does not know the actual size of the incoming message beforehand. MPI provides a way to handle such cases, by offering a function that inspects the channel for incoming messages without actually receiving them:

```
int MPI_Probe(
      int source,
      int tag,
      MPI_Comm comm,
      MPI_Status *status,
)
```

The `MPI_Probe ()` function observes the channel until an incoming message with the specified source and tag is noticed. Then, instead of receiving the message, it simply stores its status information without taking any further action and returns. After that, the system may retrieve the message's size with the method

```
int MPI_Get_count(
      MPI_Status *status,
      MPI_Datatype datatype,
      int *count,
)
```

which stores into the *count* variable the number of bytes it contains depending on the message's data type. Thus, the system knows exactly how much memory it must allocate for the buffer before calling the `MPI_Recv()` function and actually receive the message. This useful mechanism is vital for proper memory handling during data distribution, since the occasions where a receiver knows the size of the incoming data packages in advance are rare.

There are multiple options for different collective communication mechanisms provided by the MPI. A few basic methods are described below, though some of them were used in the implemented system only as a way of synchronization and not data distribution.

```
int MPI_Bcast(
      void *buffer,
      int count,
      MPI_Datatype datatype,
      int root,
      MPI_Comm comm
)
```

broadcasts data from one member to all others in a group of processes. When a process calls this function, its rank is compared to the value specified in the *root* parameter to decide whether it is the sender of the message about to be broadcast or one of the receivers. Only the root process transmits the message while the rest receive and save it into their local buffer.

```
int MPI_Scatter(
      void *sendbuf,
      int sendcount,
      MPI_Datatype sendtype,
      void *recvbuf,
      int recvcount,
      MPI_Datatype recvtype,
      int root,
      MPI_Comm comm,
)
```

`MPI_Scatter()` scatters a collection of data by splitting it into equal segments, one for each receiving process. The parameter *sendcount* specifies the number of *sendtype* continuous elements in the *sendbuf* that will be sent to each process in the group. Both sendcount and *recvcount* must be the same value as well as sendtype and *recvtype*.

2.3 Compiling & Combined Usage

2.3.1 MPI program

In order to compile and run an MPI program in multiple computers, they must be able to connect to each other using *SSH* without asking for a password because by default, when an MPI program begins, it automatically SSH's to all other computers. This is done when the environment is a small cluster containing up to 64 computers. In case there are more than 64 nodes, MPI handles connection spawning automatically using a tree spawn algorithm, in which nodes might ssh to other nodes.

To compile a program, the user must use an MPI-supporting compiler such as `mpiCC` or `mpicxx`, depending on the installed prototype as well as the programming language in which the source code is written.

The user may also define the exact number of nodes in the environment upon execution by adding the **-np «number»** argument after the `mpirun` command. Moreover, a list of the computers and their addresses must be existent on the disk. When executing the program with the `mpirun` command, the user may specify the node file in which the required node-related information is stored by passing the **-hostfile «file name»** argument. The file may contain the computers' IP addresses, their specified names -if any- in the network, their maximum available *slots* (processes) and how many of these slots to be initialized as nodes of the environment. Each line must contain an individual entry for a node, with the arguments separated with single spaces but the file's required format may vary depending on the installed MPI implementation.

For additional portability, the user may select the executing computer as rank 0 in an environment without needing to change the host file every time, by writing *localhost* as the first line of the file. If no file is specified on execution, then all the processes are created in the same local computer.

2.3.2 Hybrid program

OpenMP is supported by most standard compilers. The developer has to pass an argument to notify the compiler that OpenMP commands are being used in the source code. In C/C++ this argument is **-fopenmp** and it instructs the compiler to take the `#pragma` commands into account.

Thus, compiling a hybrid program that uses both OpenMP and MPI in C++ can be compiled simply by using a proper MPI compiler along with the `-fopenmp` argument in the following manner: **mpicxx -fopenmp program.cpp**.

As mentioned in subsection 2.2.2, in order for the threads to be able to make MPI calls, the environment must be initialized using `MPI_Init_thread()` with the argument `MPI_THREAD_MULTIPLE`. However, this is not the only peculiarity in developing hybrid programs using both these tools.

Processes in the environment's computers may create their own threads, limited only by that system's specifications. Work load may be distributed to the nodes to be handled simultaneously and each node will further parallelize it using its own threads, improving productivity and saving valuable time. The implemented system makes use of this hybrid ability to a full extent, trying to be as both distributed and parallel as possible.

2.3.3 Further References

OpenMP & MPI exist since the 1990s and are being developed continuously, with many international books and publications using and expanding them. Honorary mention to the book ”Parallel Systems & Programming” [15] which was studied thoroughly before and during this thesis' research to accumulate an adequate understanding of the tools.

Chapter 3

System Overview & Evaluation Methodology

3.1 System Overview

The system performs query evaluation on spatial data, so it must load the records and build the database first. It does so by reading the data from the disk, using the computer's available threads to parallelize the whole process. The data records are not saved in memory. Instead, they are distributed to the cluster's individual computers and stored in their respective local hard drives. The computers send data to each other by messages through their network.

The system then, can perform the query evaluation on the distributed data. During this process, the system reads range queries from a data file on disk and uses the distributed environment's computers to evaluate them. All of the individual machines possess different segments of the database and use their threads to parallelize the evaluation even more.

3.2 Development Environment

During development, all experiments were conducted in the CSEcluster at the University of Ioannina. The cluster consists of 12 Dell PowerEdge R430 with 8 Intel Xeon E5-2620 v4 CPUs each and 16 GB memory. Each processor has one thread, meaning that for every parallel operation in a computer, up to 8 threads can be utilized without over-exhausting the system's resources. The computers ran Ubuntu OS version 18.04.5 LTS.

The network through which the computers communicate with each other and with external systems is 1 Gbps Ethernet. All computers can SSH to each other without password request and know each other's names/addresses.

Along with a shared filesystem between them, each computer has a local SSD and HDD disk for private storing.

3.3 Data Sets

A total of 3 different datasets were used, which represent real world geo-spatial objects located in the United States. They are publicly available Tiger 2015 datasets which were downloaded by the official site of the open source extension **SpatialHadoop** [16].

Their details are listed below:

Name	Total Records	Size
T1	129,179	4.2 MB
T4	70,380,191	2.3 GB
T8	19,349,214	634 MB

3.4 Queries

The queries used in each experiment are the same as those used in [9], which were generated by the authors and apply on non-empty areas of the map (i.e., they always return results). Their relative area is varied as a percentage of the entire data space, inside the {0.01, 0.05, 0.1, 0.5, 1} value range (default value 0.1% of the area of the map). The experiments run the first 100 queries of the list, instead of all 10,000.

Chapter 4

Parallel Partitioning

The main issue with distributed databases is the record loading and their indexing. The data distribution cannot be random and must follow some patterns, in order for the system to support efficient query evaluation. This is achieved through the partitioning, that manages to distribute the data to the computers in a way that enables faster query evaluation.

There are many different partitioning strategies. In this chapter we examine a certain type of data partitioning in order to optimize the database for range query evaluation. We focus on the parallelization of the Two-Layer Partitioning method [9] as well as efficient memory handling and fast inter-process communications but with overall low program complexity to favor scalability.

4.1 Partitioning

The time needed to answer a single query depends heavily on the database's size; namely, on the number of records the system has to iterate through and check whether they meet the query's specifications or not. In large quantities of data, checking every single record in the database can be very time-consuming and counterproductive.

This creates a need for some kind of separation between the records or a way to selectively ignore portions of the data, reducing execution time.

4.1.1 The Two-Layer Partitioning

The objective of queries is to quickly and accurately gather all relevant records from the database. This requires some pre-processing of the data, so that it can be easily iterated through and examined based on each record's qualities. The data is modified and stored in a special data structure on disk called **index**, that improves the speed of data retrieval operations by reducing the amount of record accessing.

Non-point spatial data objects are approximated and indexed by their MBRs (Minimum Bounding Rectangles). Then, the queries are processed using a filter-and-refinement method. The first step, the filtering, uses the index to reduce the amount of record groups that have to be checked. The refinement step minimizes the total number of comparisons needed to be made for each group, based on their qualities.

The Two-Layer index [9] is a recently proposed partitioning approach which is based on an $N \times N$ grid and assigns each object MBR to all tiles that are intersected by it. However, objects' MBRs may cross multiple cells of the grid, being assigned to more than one partitions and thus creating duplicates of the record. The system needs to be able to recognize which of the records contained in a cell appear also in other cells as well, in order to disregard the replicas. This problem is called **deduplication** and can be faced with the reference point technique by Dittrich and Seeger [17]. Basically, this method defines a reference point for each object and based on that, assigns it to only one of the partitions, instead of all the intersecting ones.

The Two-Layer Partitioning is a better approach that does not apply the reference point technique but instead accesses only the relevant partitions to a query. By classifying the objects locally in every cell, the system can instantly determine the record's state in the the grid and choose to ignore certain groups of records. However, since the objects are approximated by their MBRs (Minimum Bounding Rectangles), a reference point is needed to efficiently perform positional checks and classification. Each rectangle's reference point (begin point) is its bottom left corner's coordinates and with that in mind, the method classifies the objects into the following four categories, for a partition T that they overlap:

- Class A: Objects begin inside partition T in both dimensions.

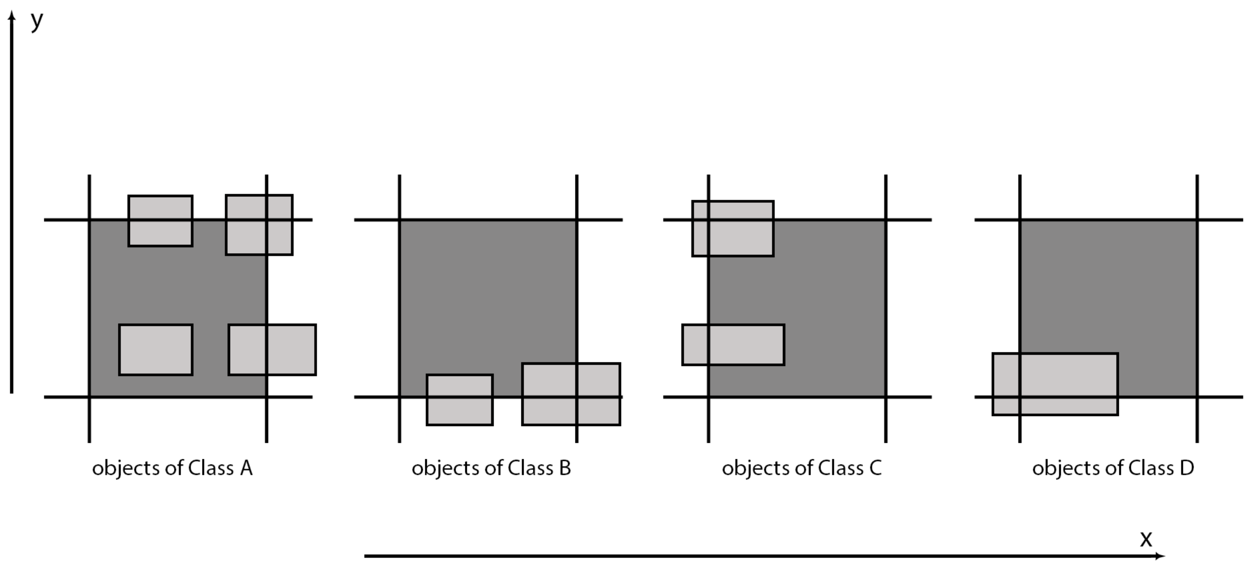


Figure 4.1: Examples of objects classified as each one of the four classes in a tile T.

- Class B: Objects begin inside partition T only in dimension x.
- Class C: Objects begin inside partition T only in dimension y.
- Class D: Objects begin before partition T in both dimensions.

Figure 4.1 illustrates the four different cases of rectangles in a cell T, one for each class.

Mathematically, the classification of an object r assigned to a tile T can be done with algebraic checks [9]. Considering $r.x_l, r.x_u$ to be the rectangle's projection on the x axis and $r.y_l, r.y_u$ its projection on the y axis:

- r belongs to class A, if for every dimension $d \in \{x, y\}$, the begin value of $r.d_l$ of r falls into projection $T.d$, i.e., if $T.d_l \leq r.d_l$.
- r belongs to class B if $r.x$ begins inside $T.x$ and $r.y$ begins before $T.y$, i.e. if $T.x_l \leq r.x_u$ and $T.y_l \geq r.y_l$.
- r belongs to class C if $r.x$ begins before $T.x$ and $r.y$ begins inside $T.y$, i.e., $T.x_l \geq r.x_l$ and $T.y_l \leq r.y_l$.
- r belongs to class D if both its $x-$ and $y-$projections begin before T, i.e., if $T.x_l > r.x_l$ and $T.y_l > r.y_l$.

The actual query evaluation that uses the above partitioning mechanism along with its distributed implementation is thoroughly explained in Chapter 5.2 .

4.1.2 Data Distribution

The system follows the **Master/Slave** model, meaning that one of the nodes will be called the *Master* and the rest will be the *Slaves*. The node executing the program is automatically assigned the Master role and is solely responsible for properly reading and distributing the data to the slaves. The master node will neither save any of the classified data locally nor load large chunks of it in memory. A summary of the master node's responsibilities regarding the partitioning:

- Data read from disk, with caution in memory handling.
- Data classification based on the records' position in the grid.
- Classified data distribution to the slaves.

The master node needs a way to determine which records will be delivered to each slave node. Since the data classification is based on the grid, one efficient way would be to assign cells to nodes, meaning that the contents of a tile are sent exclusively to a single slave. It would be wise to assign **consecutive tiles to different nodes**, since the system deals with rectangles that may overlap multiple neighboring tiles. This way the distributed environment's benefits are preserved, engaging more of the system's nodes in queries that intersect more than one of the grid's tiles. Another advantage of this approach is that we achieve load balancing if the spatial distribution of the objects is not uniform.

The grid's tiles are each labeled with an integer identifier through which they can be referenced to, during the cell-node assignment. For an $N \times N$ grid, the identifiers range from *0* to $N^2 - 1$ and are produced using the following expression:

$$ID_{Cell} = y \times N + x$$

where x and y are the tile's position in the grid's x and y axes respectively and N is the number of tiles in a single dimension, as show in Figure 4.2.

Each cell represents actual coordinates related to the data. The system needs to know the minimum and maximum value for each dimension, to properly normalize and position the rectangles to their corresponding tiles. This information can be gathered by pre-processing the data, though this option adds overhead delays to the program's total execution. The system considers these values as well as the total

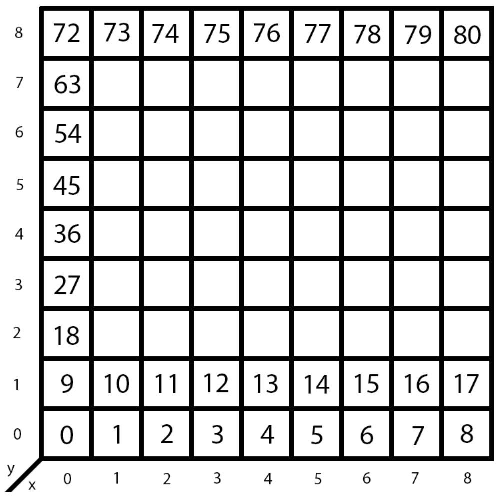


Figure 4.2: An example of a 9x9 grid and its identification

number of records in the data known, since the pre-processing can be done once before the partitioning and later be updated through changes in the database.

The tile-to-slave allocation is performed using a basic algebraic function that assigns consecutive cells to different slave nodes based on both of their IDs. It has to work for all cluster and grid sizes. In an environment with S total slave nodes, the contents of a cell with ID_{Cell} will be sent to the node with ID produced by the following function:

$$ID_{Node} = ID_{Cell} \mod S$$

So for example, in figure 4.2 and for an environment with 10 slave nodes, the contents of the cell with $ID_{Cell} = 13$ will be sent to the node with $ID_{Node} = 13 \mod 10 \rightarrow ID_{Node} = 3$.

The slaves do not have access to the raw data file. Instead, they receive messages containing records that have been already classified by the master and save them locally into their disk. Each message sent by the master is consisted by multiple rectangles. For each rectangle, the message contains information regarding the record's ID (which is the line number in the input data file), the cell ID which it overlaps, its class in that cell and the rectangle's actual coordinates.

The way in which they store the classified data is very important, since they must be able to iterate through it and collect the answers to queries efficiently. In order to quickly locate all records belonging to a specific tile and class, the slaves create files named after both these attributes using the following name format **cell_<cellID>_<classType>**. So, if for example a node is responsible for a tile T and receives records with classes A and C, it will create two data files named `cell_T_A` and `cell_T_C` with each one containing all records agreeing to the name's specifications.

4.1.3 Program Flow

Upon beginning, the system sets up the directories in which the data will be stored in all of the slave nodes' local disks. Moreover, as mentioned in the previous subsection, the program needs to know how many records are there in total as well as the min and max values of the coordinates in both dimensions. This is the point in which the program will perform the pre-processing in order to obtain that information, given that it does not already have it.

Then, the master node begins the partitioning while the slave nodes wait for incoming messages. The master reads chunks of data into memory, creating packets targeted to each slave node based on the aforementioned criteria. Each packet will contain no more than a specific quantity of records called **threadChunk**, specified by the user. This value exists so that the user can have control over how much memory the system uses while performing the partitioning. Thus, each slave receives packets containing `threadChunk` records at most and writes them to its disk as they arrive.

When all of the records have been read, classified and distributed to the slaves, the master sends a specific termination message to the nodes so that they will stop waiting for more partitioned data and move on to the next phase. This type of synchronization in the system is important, to avoid data loss and properly separate the system's functionalities. The nodes then, can move on to the query evaluation section all together.

4.2 Parallelization

This section examines the many ways in which the partitioning can be parallelized and analyzes the pros and cons of each case. The main principle followed during this process is the identification of individually parallelizable sections and the evaluation of their contribution in the attempted time saving.

4.2.1 Parallel Read

As mentioned before, the master node is solely responsible for the data handling. For each record, the master must read it from the disk, format it properly so that it can be classified, actually classify it, put it into a message package and store it before sending it through the channel to the corresponding slave. This is too much work to be done serially for each record and can be very time-consuming for large data sets.

Moreover, actions that are applied directly on the hard disk are usually slow and must be handled with caution, especially when multiple processes attempt to perform them. However, at this moment, the master performs exclusively reading actions, which can be done by the threads simultaneously with safety. So this is the perfect case to utilize the master node's threads to parallely read the data from the disk, allegedly saving time.

During development, the following two methods for parallel data reading stood out as the most efficient and were thoroughly examined and compared with each other before one being included in the system.

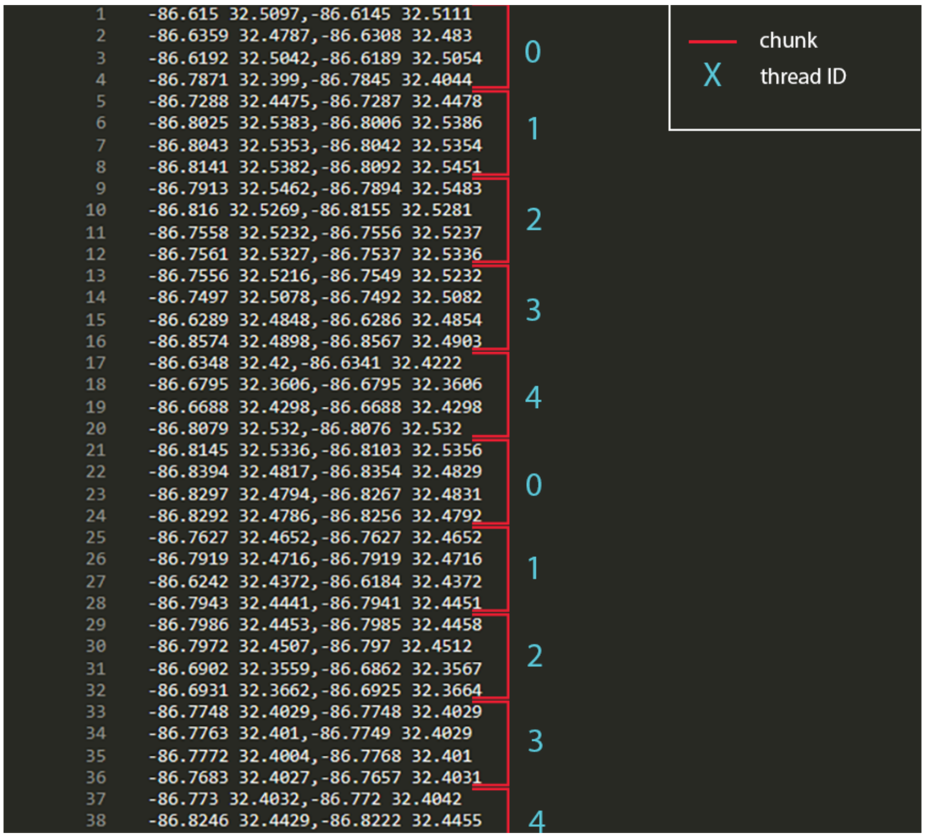
Thread Indices Method

This method uses threads that read different sections directly from the input data file. The main process creates a group of threads before opening the file and uses pointers to guide each thread to its assigned data segment.

More specifically, each thread owns a private copy of the file reader and iterates the data file independently from start to finish. It only actually loads into memory the data that is pointed to by its local indices, ignoring the rest of the records. These segments will be from now on referred to as **chunks**, with each one containing `threadChunk` records.

An example of the data iteration by the threads is shown in Figure 4.3. In this instance, thread 0 will load into memory the first four lines of the data file (the first chunk) and then read through lines 5-20 without loading any data, until it reaches line 21 where its next chunk begins. The thread recognizes its chunks by their integer IDs, using a pointer which is at first initialized to match the thread's ID and is later increased by the total number N of threads in the group, each time the thread finishes loading a chunk.

Obviously, there is a huge drawback in this method. In a group containing N



```
-86.615 32.5097,-86.6145 32.5111
-86.6359 32.4787,-86.6308 32.483
-86.6192 32.5042,-86.6189 32.5054
-86.7871 32.399,-86.7845 32.4044
-86.7288 32.4475,-86.7287 32.4478
-86.8025 32.5383,-86.8006 32.5386
-86.8043 32.5353,-86.8042 32.5354
-86.8141 32.5382,-86.8092 32.5451
-86.7913 32.5462,-86.7894 32.5483
-86.816 32.5269,-86.8155 32.5281
-86.7558 32.5232,-86.7556 32.5237
-86.7561 32.5327,-86.7537 32.5336
-86.7556 32.5216,-86.7549 32.5232
-86.7497 32.5078,-86.7492 32.5082
-86.6289 32.4848,-86.6286 32.4854
-86.8574 32.4898,-86.8567 32.4903
-86.6348 32.42,-86.6341 32.4222
-86.6795 32.3606,-86.6795 32.3606
-86.6688 32.4298,-86.6688 32.4298
-86.8079 32.532,-86.8076 32.532
-86.8145 32.5336,-86.8103 32.5356
-86.8394 32.4817,-86.8354 32.4829
-86.8297 32.4794,-86.8267 32.4831
-86.8292 32.4786,-86.8256 32.4792
-86.7627 32.4652,-86.7627 32.4652
-86.7919 32.4716,-86.7919 32.4716
-86.6242 32.4372,-86.6184 32.4372
-86.7943 32.4441,-86.7941 32.4451
-86.7986 32.4453,-86.7985 32.4458
-86.7972 32.4507,-86.797 32.4512
-86.6902 32.3559,-86.6862 32.3567
-86.6931 32.3662,-86.6925 32.3664
-86.7748 32.4029,-86.7748 32.4029
-86.7763 32.401,-86.7749 32.4029
-86.7772 32.4004,-86.7768 32.401
-86.7683 32.4027,-86.7657 32.4031
-86.773 32.4032,-86.772 32.4042
-86.8246 32.4429,-86.8222 32.4455
```

Figure 4.3: An example of the chunk segmentation process for the first 38 lines of a data file, with threadChunk=4 and N=5 active threads.

threads, each one will iterate through *N – 1* chunks useless to it, every time it loads a single chunk in memory. This is due to the fact that the threads iterate through every line, instead of 'jumping' right to the beginning of their next chunk.

This can fixed by using the C++ routine

```
seekg(streamoff off, ios_base::seekdir way)
```

which sets the position of the next character to be extracted from an input stream, based on the `off` parameter. This offset may specify a position counting from the beginning or the ending of the stream as well as its current position. By specifying only the offset value, it automatically assumes the user means the absolute position from the beginning of the stream.

The main issue here is that the offset parameter needs to be in characters and not lines, meaning that the system cannot use it to jump a specific number of lines without knowing their total size in characters (bytes). This information cannot be obtained during runtime efficiently, so some kind of pre-processing is needed.

Since a possible pre-processing of the data file has been already discussed in Section 4.1.2, it can be further enhanced to store character-related information about each line or even better, about each chunk. The idea is that if a thread knows exactly how many characters exist between its current and next chunk, it can use `seekg()` to move right to it, without having to iterate through useless data. So, during pre-processing, the system counts how many characters constitute each chunk and writes the result into a binary file on the disk referred to as **navigation file**. Then, when a thread needs to make a 'jump', it simply reads through the navigation file to find out how many bytes it needs to skip in order to reach its next chunk. Binary files are ideal for this kind of work, since they are quick to iterate through and small in size.

However, now the pre-processing is not optional for the system, since the binary file is mandatory for the `seekg()` routine. The most costly functionality during the pre-processing is the determination of the minimum and maximum record values in both axes. The system must format each string line into numerical values before performing the comparisons, adding too much overhead on the whole process. At this point, it is reasonable to wonder whether the benefits outweigh the costs, since the pre-processing can be quite time-consuming.

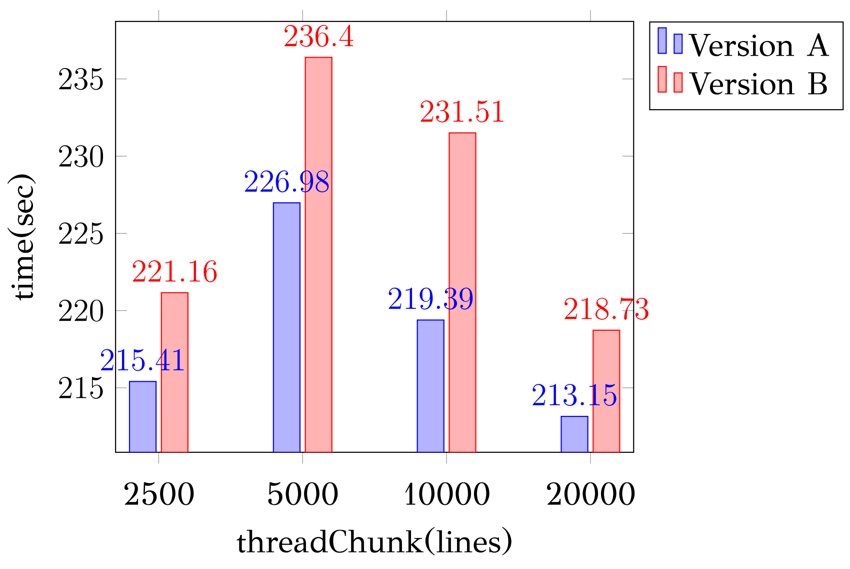


Figure 4.4: Comparison between the two versions for a few different threadChunk sizes.

This can be easily tested by performing a comparison of two different program versions: One that pre-processes the data, creating the binary file which is later used by the threads in combination with the `seekg()` method to perform the partitioning

(**Version A**) and one that does not use the method, forcing each thread to read the data file in its entirety but avoiding any form of pre-processing (**Version B**). For the experiment, the T4 dataset was used in the cluster utilizing all of the twelve nodes (one master, 11 slaves) as well as 8 threads in the master node. The results are shown in Figure 4.4.

Despite Version A's small but steady lead against Version B in execution time, it is important to consider the pre-processing costs. Figure 4.5 shows the total time elapsed during the pre-processing in Version A and how it is divided between the navigation file creation and the min/max values acquisition. It is obvious that formatting the data to properly determine the min/max values creates a very large overhead whilst the navigation file creation process adds a rather small delay.

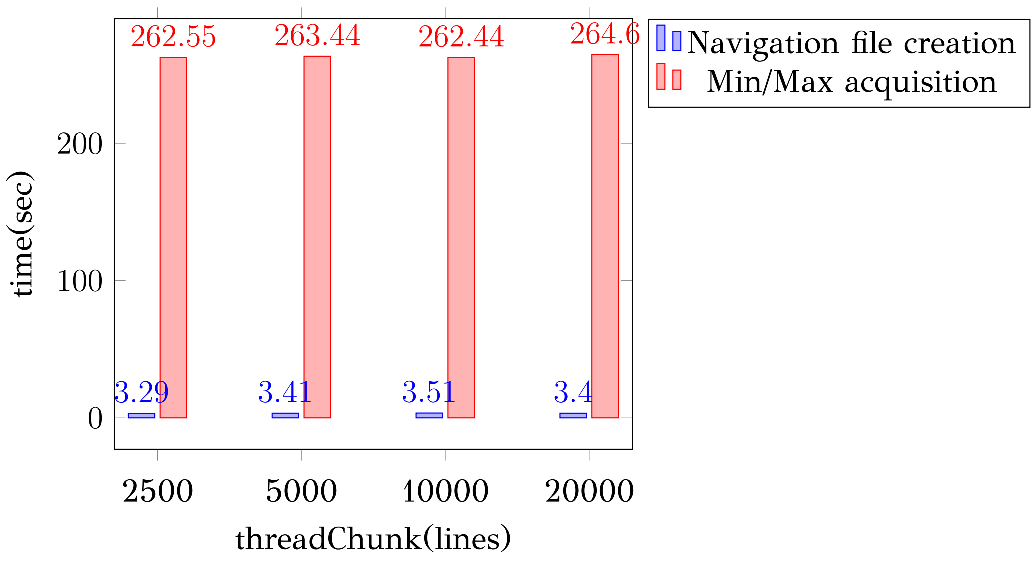


Figure 4.5: Pre-processing time analysis for various threadChunk values. The Min/Max value acquisition includes the time needed to format the strings as well as the actual comparisons.

Considering all this along with the fact that the pre-processing can be done once per database, Version A seems to work adequately so far. This is the final version of the Thread Indices Method and will be later compared to the Part Files Method to determine which one will be used in the system.

Part Files Method

This method uses a different kind of pre-processing. It requires the input data file to be split into multiple **part files** before running the program, so that each thread can take over groups of data without having to iterate the entire file or use pointers to be guided through it. The part files are similar to the chunks in the Thread Indices Method with the difference that they exist separately on the hard disk.

Part files are created beforehand using a separate program and they are sorted properly, to keep the actual order of the records intact. Each part file has a name with unique identifier so that they can be easily identified position-wise. The part-file-to-thread assignment is handled during runtime. The threads request part files from the system one at a time and when a part file is assigned to a thread, it cannot be given to another one as well. Each time a thread finishes partitioning a part file, it requests the next available until all of the files and their contents have been read through.

The system must be built in a way that handles the part file assignment to threads carefully. This can be implemented easily using basic mutual exclusions techniques such as the aforementioned **critical area**. Basically, the code responsible for the part file requests and assignments is enclosed in a critical area, ensuring that it will be executed one thread at a time, avoiding confusion.

This method does not use navigation files or complicated pointers, managing to lower overall program complexity by encumbering a little more the pre-processing of the data.

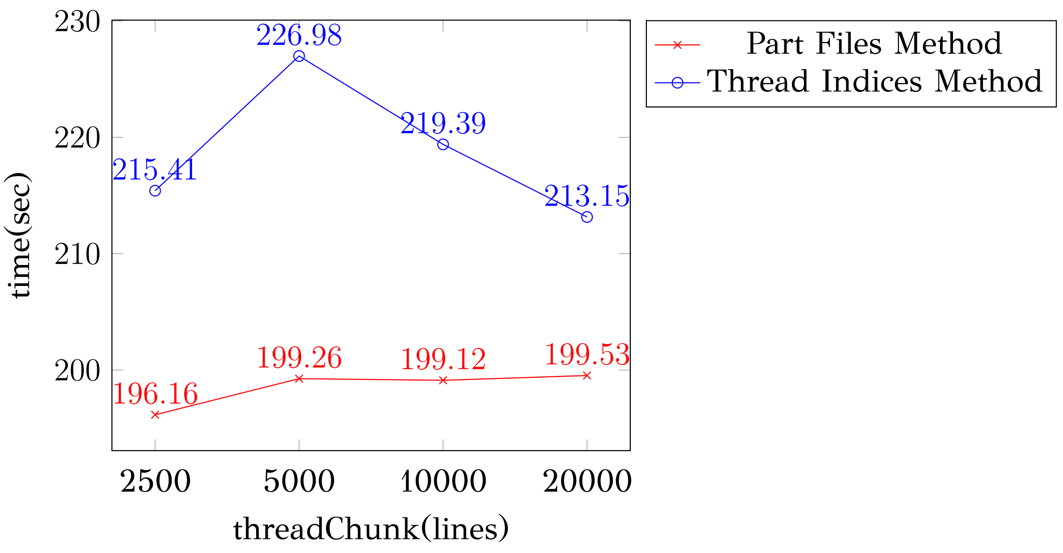


Figure 4.6: Comparison between the two parallel partitioning methods for various threadChunk sizes.

Final Comparison

Figure 4.6 shows the comparison between the Thread Indices Method and the Part Files Method, executed in the cluster. It is obvious that the Part Files Method works faster and along with its low complexity, makes a fine solution for the Parallel Read problem. At this point we accept the pre-processing cost during the build-up of the database, since it allows the usage of the part files mechanism along with the needed total line count and min/max evaluation.

4.2.2 Distribution Mechanism

The way in which the data is actually distributed to the slaves is of major importance and must be done with respect to efficient memory handling. The *threadChunk* size is a key parameter to this operation, since it limits the amount of records each thread can load into memory simultaneously. However, inter-process communications can be costly if handled poorly, since the records are being distributed in packs that may overload the channel.

On top of that, each thread may send packets to multiple slave nodes, so it needs to keep the loaded records stored separately, creating different message packets for each node. This is solved by allowing the threads to keep private lists that contain this data. For S slave nodes in an environment, each thread stores a list with size S in which it saves the message packets directed at each node. It also keeps a list with each pack's record counters, which are used to properly contain the amount of data it loads into memory. When a list reaches `threadChunk` records, the thread sends its contents to the respective slave node and then empties it before continuing the data file iteration.

In this way, the user/developer has control over how big message packets are and by extension how often information is being sent through the channel. In an attempt to determine the actual memory used by the master node and its threads, we must consider the parameters affecting memory usage:

- Number of threads N
- Number of slave nodes S
- threadChunk size

Using the above, we can safely assume that the master node will never store more than

$$R = N \times S \times (threadChunk - 1)$$

records in memory at the same time, since each list is emptied when filled with `threadChunk` records. Moreover, it is possible to make a crude approximation regarding the expected memory usage in bytes by assuming an average record size in bytes B and multiplying it by R.

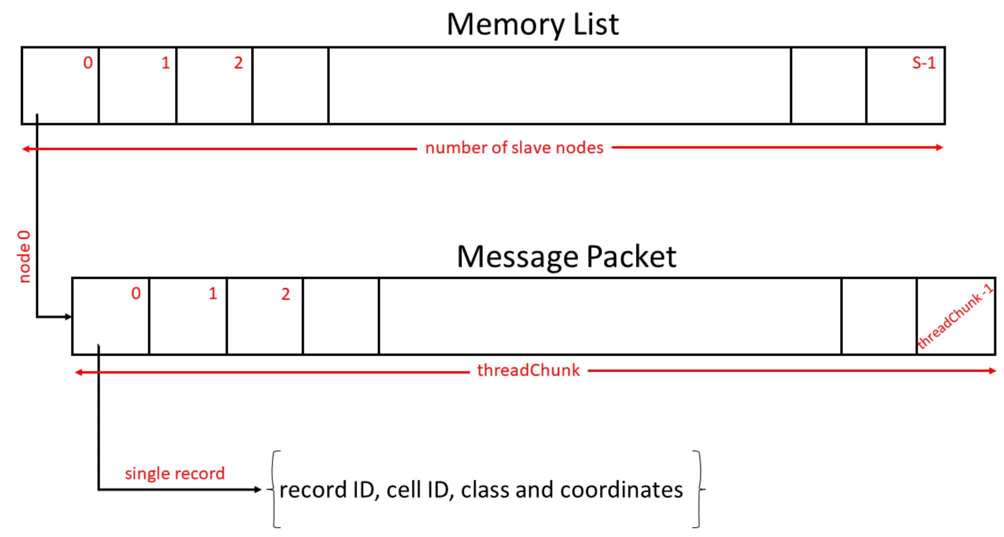


Figure 4.7: A visualization of each thread's memory usage in the master node.

Each thread owns a private memory list and keeps record of the saved rectangles, before sending the filled packets. Figure 4.7 shows how the threads place the records in the appropriate message packets, depending on the destination slave node.

Parallel distribution can be handled efficiently by the way MPI is structured. Slave nodes will ignore packets that are not directed at them and simultaneous `MPI_Send()` operations by the threads are performed without interfering with each other.

Moreover for each rectangle, the receiving node needs to possess the record's ID, the cell ID which it overlaps with, the rectangle's class in that cell and the actual coordinates from the data file. So, each message contains the above information separated with a special delimiter called *messageDelimiter* for easy deconstruction. Different records inside the message packet are separated by the `newline character \n`. Each message is first formatted in this way before being placed in the message packet.

If a rectangle overlaps with more than one cell, multiple messages are sent to probably different slave nodes creating duplicates. This will be further discussed and resolved in Chapter 5, during the range query evaluation.

4.2.3 Data Receipt

The slave nodes await for message packets containing the records assigned to them. They need to retrieve the message from the channel by saving into a temporary buffer and then deconstruct it to determine the records' save location on the disk.

For each received packet, the slave nodes iterate through the records and use the message delimiter to collect the needed information about each record. Then, they determine the save file name for each rectangle based on the cell's ID and record's class as mentioned in 4.1.2 and write the rectangle's coordinates and ID into that file on their local disk.

This process needs to be performed for each record contained in the message packets and since write operations on the hard disk are slow, it can be very costly time-wise. However, this can be parallelized by utilizing the slave nodes' threads which were unneeded until now. The main idea is to use the node's threads for parallel writing, by making them write different parts of the message pack simultaneously. But, due to the way writing to the disk works, when two or more threads attempt to write to the same file at the same time then it is probable that some data will be lost.

The optimal way to efficiently parallelize this operation, would be to assign different save files to threads, making sure that each thread writes to a file exclusively. However, the number of save files in which the records contained in a message pack will be saved to, is unknown without first iterating through the pack in its entirety. So, the slave node's threads read through the pack's records and keep in shared memory the different save file names that will be used for this message pack. They also store each record group destined to a save file in a different location in memory, separating them without writing them. Then, the thread group is destroyed, having fulfilled its purpose.

Now, the slave node has two lists in memory containing the save files that are going to be used for this message pack and their respective new contents. At this point it is possible to utilize the node's threads once again, to actually write the records safely. Each thread handles exclusively different save files, effectively parallelizing the whole process.

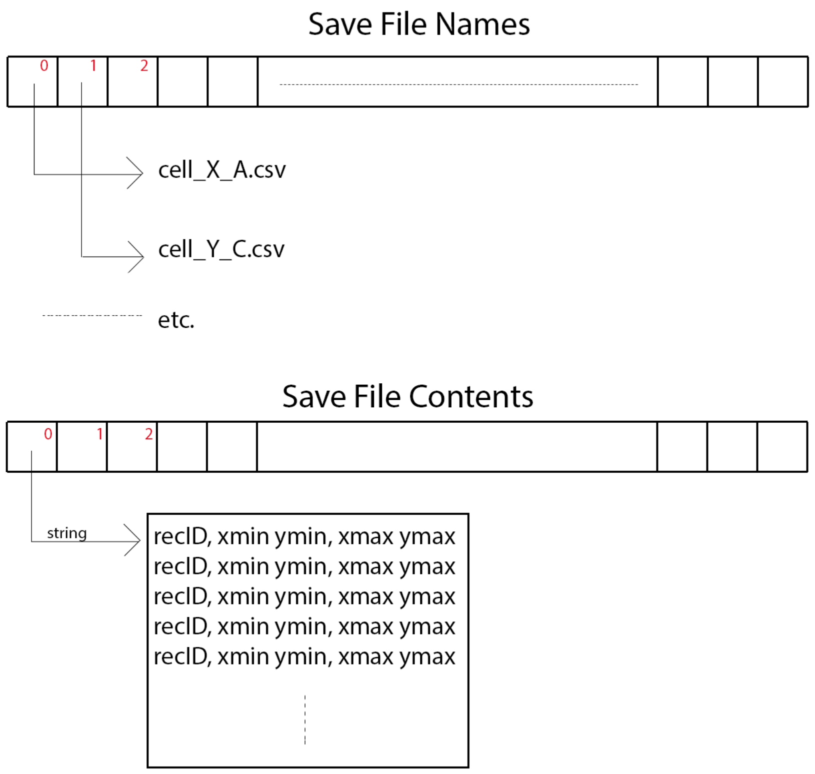

Save File Names
0
1
2
cell_X_A.csv
cell_Y_C.csv
etc.
Save File Contents
0
1
2
string
recID, xmin ymin, xmax ymax
recID, xmin ymin, xmax ymax
recID, xmin ymin, xmax ymax
recID, xmin ymin, xmax ymax
recID, xmin ymin, xmax ymax

Figure 4.8: The two shared lists in a slave node, containing the save file names and their contents before being written to the disk.

Figure 4.8 shows an example of the two lists in a slave node. They are shared by the threads and are being reset for each received message pack. These lists are formed by the first group of threads during the message pack iteration, using mutual exclusion since they are shared memory.

After being formed, a second group of threads handles the writing. Each thread writes to different save files, represented in the example by the lists' cells. As it is shown, the contents of the memory block in position 0 of the Save File Contents list or **fileLines** belong to the save file in position 0 of the Save File Names list or **fileNames** named `cell_X_A.csv`, where X is the cell ID and A the rectangles' class in that cell.

The number of threads to be created for the second group is based on the number of save files that need to be accessed for that message package. For example, if the `fileNames` list's elements are less than the available or defined by the user number of threads in the system called **NUMBER_OF_THREADS**, then the slave node creates an amount of threads equal to the list's size. On the other hand, if the data files are more, then NUMBER_OF_THREADS threads are created and they are being assigned data files to write into repetitively and exclusively.

Finally, after the partitioning is over and all data has been properly saved to the appropriate locations on the disk, the slave nodes receive a special termination message by the master and halt the receiving process.

Chapter 5

Query Evaluation

This chapter focuses on the query evaluation part of the system and the ways it can be parallelized to maximize efficiency. It uses the partitioned data stored in the slave nodes to quickly answer range queries by making the most of the distributed environment's benefits as well as the nodes' threads.

5.1 Query Evaluation Overview

Processing queries and efficiently evaluating them is probably one of the most important features any type of database must provide. It fills the gap between database programming languages and file systems [1] or in other words between the user and the actual data.

Range queries are extremely useful, since spatial data can represent many types of information. Being able to determine and retrieve the objects contained in a specific range quickly is a feature that can be used almost in any data-management system for a lot of purposes.

The system answers rectangular range queries called **window queries W**. Any object's MBR intersecting W is considered an answer to the query and must be returned in the result list. There are many ways to evaluate range queries such as the R-tree algorithm and its variants, with all of them having better efficiency than the basic serial checking of the records in the database. This thesis attempts to parallelize the Two-Layer Query Evaluation [9] and implement it in a distributed environment, potentially optimizing its runtime.

5.2 Two-Layer Query Evaluation

First, as the method suggests, we have to consider the way we can determine which rectangles intersect W. It can be found in $O(1)$ time with algebraic operations. More specifically, for the tile $T_{i,j}$ that is positioned at the i-th row and j-th column in the grid, the tiles that intersect W are all those for which $\lfloor W.x_l/N \rfloor \leq i \leq \lfloor W.x_u/N \rfloor$ and $\lfloor W.y_l/N \rfloor \leq j \leq \lfloor W.y_u/N \rfloor$.

Now, we must consider which of the rectangles contained in the above tiles must be actually checked during the evaluation. Since the system has classified the records based on their position in the grid, we can use this information to minimize the number of accesses the system makes, effectively reducing the total cost.

Figure 5.1 shows the object classes needed to be checked based on a window query W. It is obvious that by avoiding accessing objects of classes not included in a tile's required checks, the system reduces its total workload and by extension, its execution time. Moreover, the actual amount of comparisons for the rectangles' coordinates is also minimized. For example, in a tile where classes A and B are the only ones that must be checked, the system can completely ignore the x axis comparisons, since the tile is completely covered by the window in that dimension. With similar logic, the system performs only the necessary checks in the rest of the tiles, optimizing the required computation cost.

5.2.1 Distributed Implementation

Before attempting to parallelize the query evaluation process, we must consider the master's and slaves' roles in this part of the program. Since the master node does not own any classified records, it will act as coordinator between the user and the

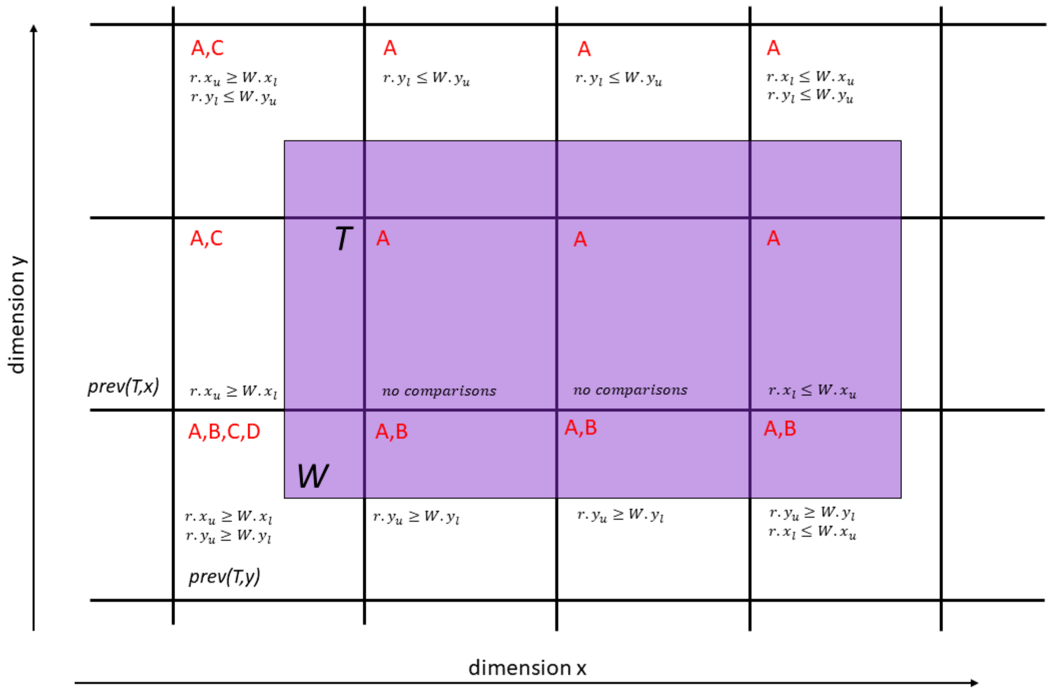


Figure 5.1: Examples of object classes and comparisons in a window query.

system's back end.

At this point, the system assumes that the user's queries exist in a text file on the master node's local disk. Each query is written on a different line, so the master reads one line at a time, loading it into its memory. The master has to consider which nodes are needed to answer each query, based on which tiles it overlaps. For example, if a window query intersects just one tile, then the system will utilize only the slave node responsible for that tile, since all the records contained in the rest of the nodes will be definitely outside the query's borders.

Figure 5.2 shows an example of how the master node determines which slave nodes will be utilized to answer a window query, using the grid's tile-to-slave assignment as mentioned in section 4.1.2. The window W intersects cells 16, 17, 22 and 23 of the 6x6 grid and based on the assignment expression, the slaves that possess the rectangles to be checked are nodes 4, 5, 10 and 11.

This process takes advantage of the distributed environment's benefits by not only reducing the execution time because of the irrelevant records' fast disregarding, but also by effectively parallelizing the whole process using the slave nodes to perform the comparisons simultaneously on their respective data.

The master node sends the query's information to the slave nodes and then waits

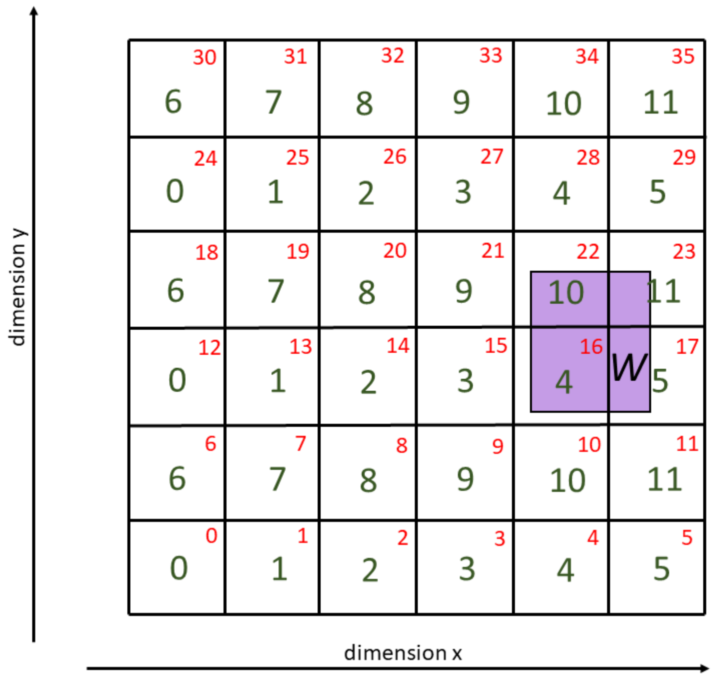


Figure 5.2: Example of a query window W on a 6x6 grid and 12 slave nodes.

for the results before moving on to the next query. For each one, it sends both its coordinates and the corner tiles it intersects in the grid, assisting the slave nodes in locating the rest of the overlapping tiles. It saves the incoming results to its local disk for later use to avoid memory overflow.

On the other side of the communication channel, the slave nodes iterate through their locally stored records using the Two-Layer Query Evaluation method. They all execute the same piece of code that performs the aforementioned comparisons and send the results back to the master.

Since a single query may have too many records as an answer, the slave nodes must be careful with their memory handling. For this reason, while iterating through their database performing the comparisons, they store only up to `threadChunk` results in a private temporary buffer.

At this point, we must examine the optimal way to return the results to the user. The slaves could send message packets with the results back to the master node or they could simply dump them on their local disk or a shared filesystem. Whilst the second option saves time by removing any communication overhead, the results still have to be joined into a single file, which is a costly operation.

Due to time limitations we didn't perform thorough investigation on which option is ultimately the best. By sending the results back to the master, the system runs fast and since the simple join operations on disk performed poorly, we decided to go with the first option. At this point, when the slave node's buffer is full, it sends the its contents to the master as a single message pack, empty it and then continue looking for results from where they had stopped. Once again, the `threadChunk` value comes into play to control the system's memory usage and protect the channel from congestion.

When finished, each participating slave sends a special termination message to notify the master that they are done checking every record needed to answer that specific query. When all termination messages are received by the master, it finishes and moves on to the next query, if any.

5.3 Parallel Query Evaluation

The query evaluation process is performed by all the slave nodes in the same way, but on their respective partitioned data sets. They initially receive the query's information from the master node and then determine the window query's situation in the grid, before accessing any records.

The grid's tiles which intersect the window query determine the actual file accesses each slave node will perform. The participating tiles for a query may be positioned in a horizontal or vertical line in the grid, in a rectangle form or be just a single tile containing the window W in its entirety. Figure 5.3 illustrates these four cases. The slave nodes receive from the master the top left, top right, bottom left and bottom right tiles that are being intersected by the window query, so that they can easily determine which case is present and iterate through the tiles.

5.3.1 Possible Parallelization

The process is already parallelized since it utilizes the distributed environment's nodes to perform the evaluation on parts of the data simultaneously. However, each node has threads available that may expedite the process even better. There are more than a few ways this can be done, so research must be thorough. At this point we must consider the optimal way to effectively parallelize the evaluation process since it can be applied in various parts of the code.

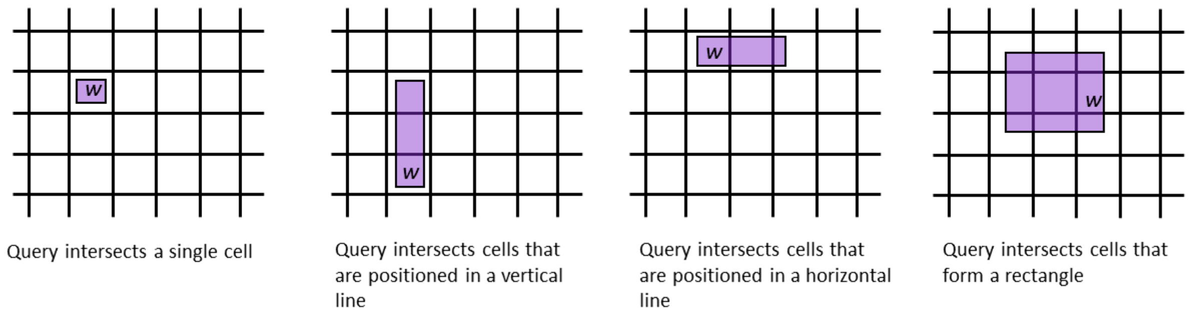

w
w
w
w
Query intersects a single cell
Query intersects cells that are positioned in a vertical line
Query intersects cells that are positioned in a horizontal line
Query intersects cells that form a rectangle

Figure 5.3: Example of all possible query cases in the grid.

Each slave node iterates firstly through the tiles indicated by the query and for each one that it is responsible for, it accesses the corresponding file on its local disk. It then iterates through the file's contents, performing comparisons to finally determine which records constitute the query's results. In the case that the tiles needed to be checked form a rectangle, the program iterates each row by beginning from the top left corner and moving upwards.

Essentially, we must determine if the parallelization of the tile iteration works faster than the parallelization of the records' comparisons. Moreover, in the case the system needs to iterate through a rectangle of the grid's tiles, we must decide if it is preferable to parallelize the row or column iteration (inner or outer loops). So, it comes down to the following 3 cases of different parallelizations:

- Record Iteration Parallelization
- Inner Loop Parallelization
- Outer Loop Parallelization

Record Iteration Parallelization

The slave nodes iterate the grid's intersected-by-the-window tiles serially, but when opening a data file on the disk to check the records, it creates a group of threads to parallely read the records and perform the comparisons.

It follows the exact same logic as the master node's parallel data file iteration *Thread Indices Method* discussed in subsection 4.2.1. Each thread uses local pointers to read exclusively different parts of the data file. It stores in its local buffer the records that satisfy the query's conditions and when the buffer is filled, it sends a message packet containing the results back to the master node. To preserve the safe memory handling, each thread is assigned a buffer with size $threadChunk/N$ records, where N is the total number of threads created by the slave node.

This method utilizes the parallel and distributed environment's benefits to a maximum, since the slaves' threads make MPI calls and send messages to the master, the opposite of what was happening during the partitioning. This logic resembles the broadcast and gather functionalities which offer scalability and flexibility to later modifications.

Inner & Outer Loop Parallelization

Before accessing the records saved on disk, the slaves determine the tiles in which they belong to. This action begins from the bottom left tile of the tiles-rectangle and reads rows moving upwards. In the example of Figure 5.4, the slaves know the bottom left, bottom right, top left and top right cell IDs that intersect the window query W. The iteration begins from cell 9 moving in the x axis until cell 11 is checked, then the iteration resumes from cell 15 and so on.

The cell iteration process is implemented using two for-loops, one for the rows and one for the columns. This method parallelizes the inner or outer loop, assigning all the tiles in a row to a group of threads to be read simultaneously. By extension, each thread then opens a different data file containing records and performs all the comparisons needed to determine the query's results.

The main problem with this method is that it depends heavily on the number of tiles intersected by the window query. For example, if the number of tiles needed to be checked are fewer than the available threads, then we have an under-utilization of the system's resources. Moreover, each thread iterates through all records contained in a tile assigned to it, regardless of class. This is a costly action and since it is performed by a single thread every time, it is probably not the optimal time-saving solution.

Method Comparison

By comparing the above methods, we can determine which one is the best to be used in the final version of the system. Figure 5.5 shows the average time it took

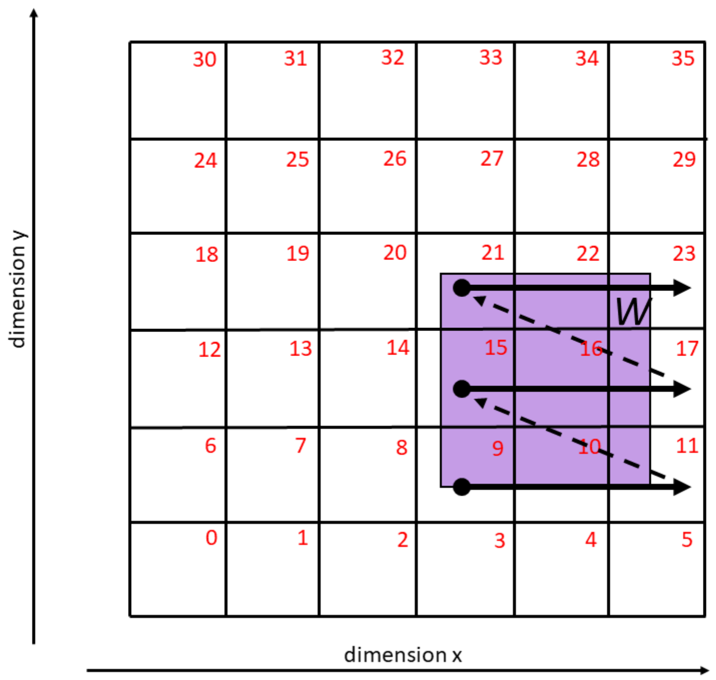


Figure 5.4: Example of the cell iteration in the slave nodes, for a window query W in a 6x6 grid.

the program to answer the same 100 window queries, using all the aforementioned methods as well as a serial query evaluation implementation that does not parallelize any loop nor the file iteration.

The experiments were executed in the cluster, using 12 nodes (1 master, 11 slaves) and 8 threads per node. The partitioning and pre-processing times were not taken into account in the shown values. The T4 dataset was used.

As expected, the method that parallelizes the file iteration and the actual record coordinates' comparisons is the fastest one. The inner and outer loop parallelizations do not even surpass the serial implementation in efficiency, even though they are situational and strongly depend on the type of the queries.

Thus, the current version of the system uses the slaves' threads to parallely read their respective data to answer each query, utilizing the system's resources to a great extend and taking advantage of the distributed environment's benefits.

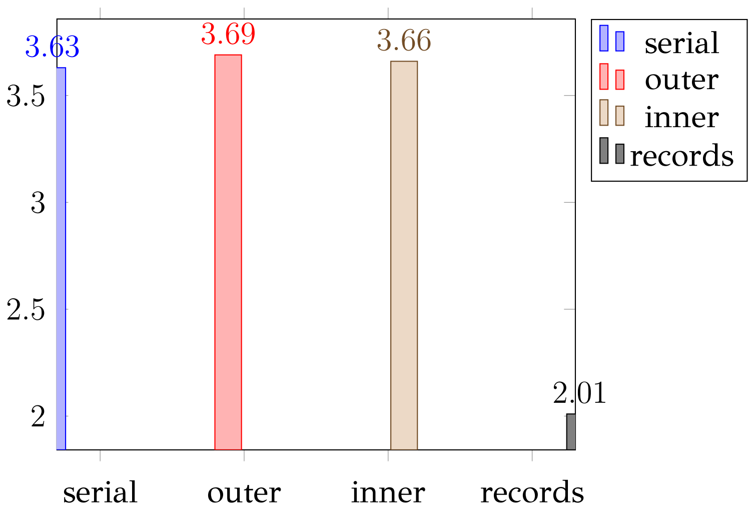


Figure 5.5: Average query evaluation time per method.

Chapter 6

System Usage & Experimental Analysis

This chapter describes the ways in which the user can use the system to its full potential and how they can customize it to fit their needs.

6.1 Setup

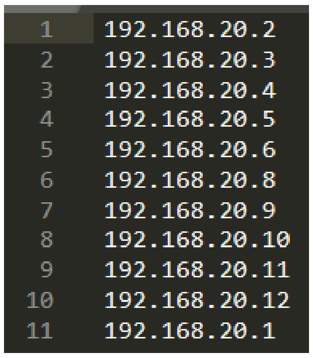
During the partitioning, the system sets up a modified database on the disk which originates from the raw input data. It is specifically formed to utilize the distributed and parallel Two-Layer Query Evaluation.

The system's final version is able to work for any number of nodes in a distributed environment, though it was tested for a maximum of 11 computers due to resource limitations. For the parallel actions, the system by default utilizes the maximum number of threads offered by each node. It works both serially (1 thread) and parallely for any number of threads but was tested for a maximum of 8 per computer.

The distributed environment's computers must be connected with each other through a local Ethernet network and know each other's names. They must be able to connect to each other using SSH without password requests and to have RSA encryption public and private keys properly set up beforehand.

The data file must be present on the master node's hard disk, since it is the only node reading through it. Any one of the nodes can be the master, by compiling and executing the program in that computer, given that they possess the input data file. However, the data file containing the nodes through which MPI creates the distributed environment, must be in the correct format:

- When compiled with MPICH, the node file passed as an argument in the execution `mpirun` command must contain all the computer nodes IP addresses as shown in Figure 6.1.



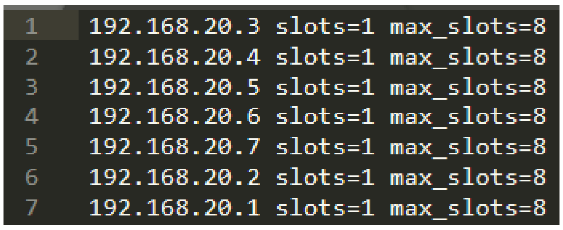
```
192.168.20.2
192.168.20.3
192.168.20.4
192.168.20.5
192.168.20.6
192.168.20.8
192.168.20.9
192.168.20.10
192.168.20.11
192.168.20.12
192.168.20.1
```

Figure 6.1: Node file containing the distributed environment's computer IP addresses when compiled using MPICH.

- When compiled with OpenMPI, the node file must be in the format shown in Figure 6.2, because OpenMPI needs to know exactly how many processes to create in each node.

In both cases, the file's first line defines the computer that will be the master node, so it is crucial to be the same computer which the program is being compiled and executed in. This is because node rank is assigned to the computers in the same order they appear inside the node file. The system always regards the master node to be rank 0, so its IP address must be located in the first line of the node file. In case the master node is not assigned rank 0, then the system's behavior can not be predicted but will most likely be incorrect.

Moreover, the system assumes that MPI creates a single process in each node. The environment's size X in nodes, as defined by the `-np X` argument in the `mpirun`



```
192.168.20.3 slots=1 max_slots=8
192.168.20.4 slots=1 max_slots=8
192.168.20.5 slots=1 max_slots=8
192.168.20.6 slots=1 max_slots=8
192.168.20.7 slots=1 max_slots=8
192.168.20.2 slots=1 max_slots=8
192.168.20.1 slots=1 max_slots=8
```

Figure 6.2: Node file containing the distributed environment's computer IP addresses when compiled using OpenMPI. Argument slots=X indicates the number of processes to be created in each node and max_slots=X the total number of available processes in that computer.

command, must not surpass the number of nodes contained in the node file. If so, then any leftover processes remaining by the defined number will be created in a node that has been already activated. In this way, the program is guaranteed to fail since multiple processes will battle for the same computer's resources and will perform disk actions simultaneously without caution.

6.1.1 Parameters

There are multiple parameters that can be set accordingly to adjust the program based on the available resources. In different systems and use cases, the user may need less or more memory to be used at any given time, depending on the available RAM, communication channel bandwidth or other limitations.

Number of Threads

Arguably, one of the parameters with the greatest impact is the number of threads in each node. For the master, utilizing more threads means faster data iteration and distribution during the partitioning whilst for the slaves, means faster data saving on the disk and query evaluation.

The parameter controlling the number of threads per parallel group created in all of the nodes is named `NUMBER_OF_THREADS` and it is global. This way, the user can manually set the number of threads they want their nodes to utilize for every parallel action. However, unless all of the environment's nodes offer the same amount of threads, this can lead to errors. If they want to utilize the system's resources to its maximum they can set the variable to its highest possible *per computer*, using OpenMP's method `omp_get_max_threads()`.

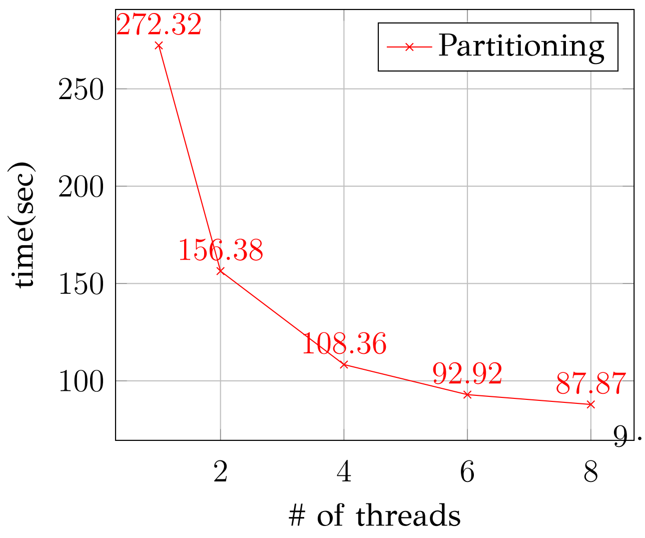


Figure 6.3: Effect of the total amount of threads used on the partitioning time.

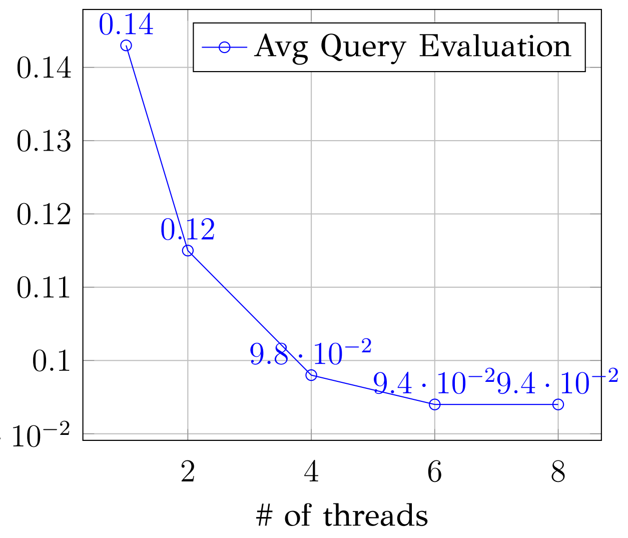


Figure 6.4: Effect of the total amount of threads used on the average query evaluation time.

The threads play one of the most important roles in the system's implementation and directly affect its performance. Clusters and computers with increased number of threads per processor can efficiently use the program for extremely fast query evaluations. However, hyper-threading is not the same as a distributed environment, since it simply enables each processor to reach its full processing potential and not necessarily double its performance.

Figures 6.3 and 6.4 illustrate the threads' importance. The experiment was performed on a 12-node environment with the T4 dataset and as it is shown, the partitioning greatly was greatly improved by a higher thread count. This is highly logical, since the master node uses threads to parallely read the input data. Moreover, the slaves use threads to both parallely save the data on their local disk as they receive the packets and evaluate the queries by performing the comparisons. Both functionalities' execution time is improved by increasing the amount of threads but it is probable that there is a bottleneck value on the thread count since multi-threading usually benefits less complex operations. The experiments did not exceed the 8 threads per computer due to resource limitations.

A better way to understand the system's performance on the query evaluation, is to measure it in queries-per-minute instead of real time, since we deal with fragments of the second. Figure 6.5 shows the effect that the number of threads have on the query evaluation, measured in queries per minute. The user has the option to run batches of queries in a single run, so having an estimation of how many queries would run by the minute would help them expect the total runtime.

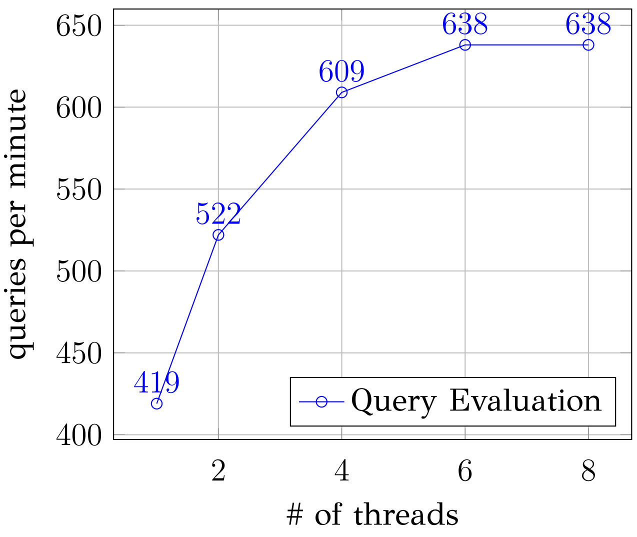


Figure 6.5: Effect of the total amount of threads used on the query evaluation time, measured in queries per minute.

Number of Nodes

Additionally, the total number of nodes in the distributed environment affects the system's performance greatly, since the parallel query evaluation is depended on the environment's size. More nodes means less individual computer load and disk space, since the data gets separated even more with larger environment size. Figures 6.6 and 6.7 illustrates the effect that the total number of nodes has on the partitioning and average query evaluation times.

As expected, the parallel partitioning is not particularly affected by the distributed environment's size since the nodes mainly await message packets by the master node and save them as they come. Besides the case of 1-master and 1-slave, the partitioning remains mainly unaffected by the node counts.

On the other hand, the query evaluation is directly affected, since more nodes means greater parallelism. For example, having 2 nodes in total (one master and a slave) is obviously slower than having 2 or more slaves to divide the workload into. It is important for the system to offer scalability, meaning that the user has to have the option to add more computers to the environment to further enhance the system's performance.

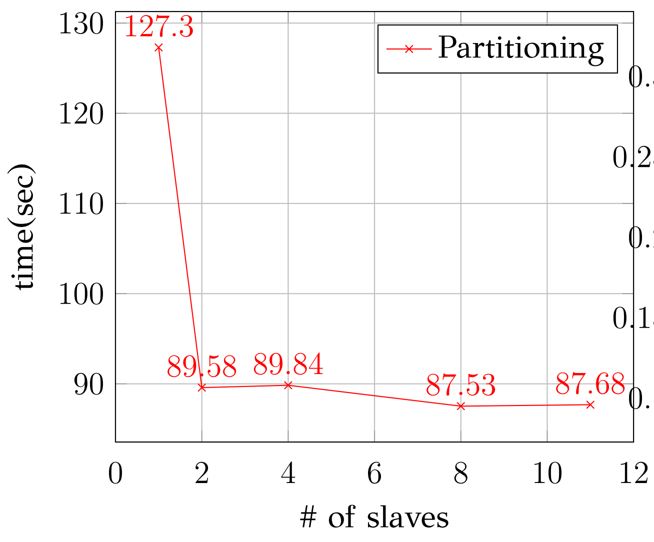


Figure 6.6: Effect of the total number of slave nodes on the partitioning time.

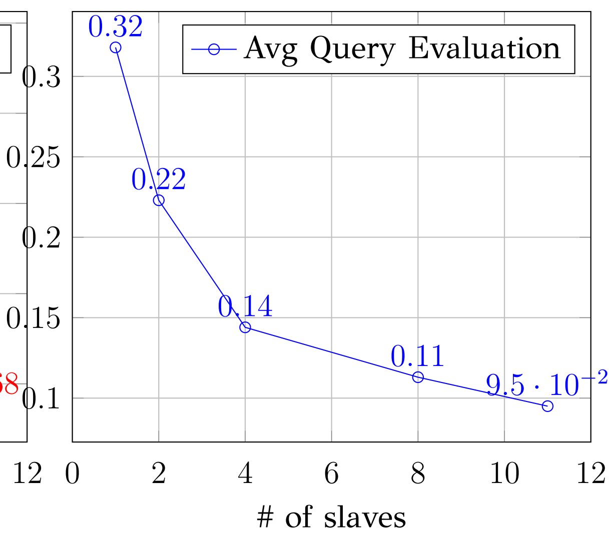


Figure 6.7: Effect of the total number of slave nodes on the average query evaluation time.

Once again, we see the query evaluation's performance in queries per minute in Figure 6.8 for better understanding. It seems that the system scales pretty well, but due to resource limitations we were unable to run it in a cluster containing more than 12 computers.

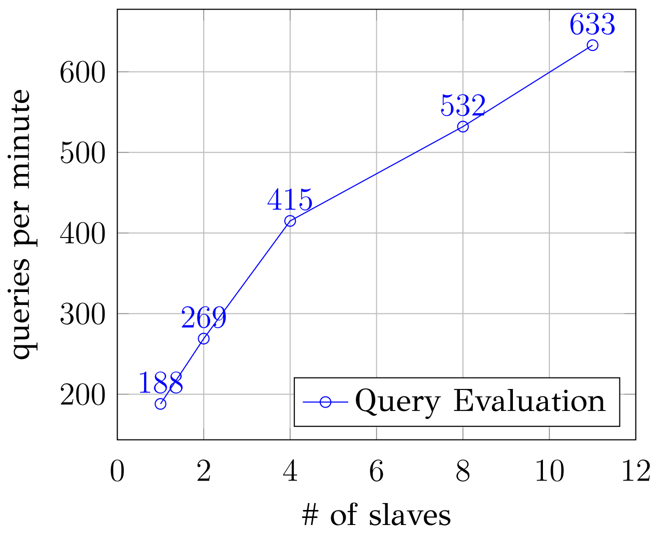


Figure 6.8: Effect of the total amount of nodes in the cluster on the query evaluation time, measured in queries per minute.

Grid Size

The actual size of the grid in which the objects are placed in and are being classified in accordance with, directly affects the program's performance.

Smaller grid size means bigger tiles and thus more records inside each one. In contrast, larger grid size means smaller tiles containing less records each, but with increased computational cost, since more tiles need to be checked for each query. Moreover, having queries that refer to a small section -intersecting the same few tiles- of the grid, means that most workload will probably fall on just a few of the environment's nodes, under-utilizing the distributed parallelism that was aimed at by the partitioning.

The grid is created based on the objects' minimum and maximum values in both axes, so the user may have control over how many tiles there will be in each grid's dimension without affecting the borders.

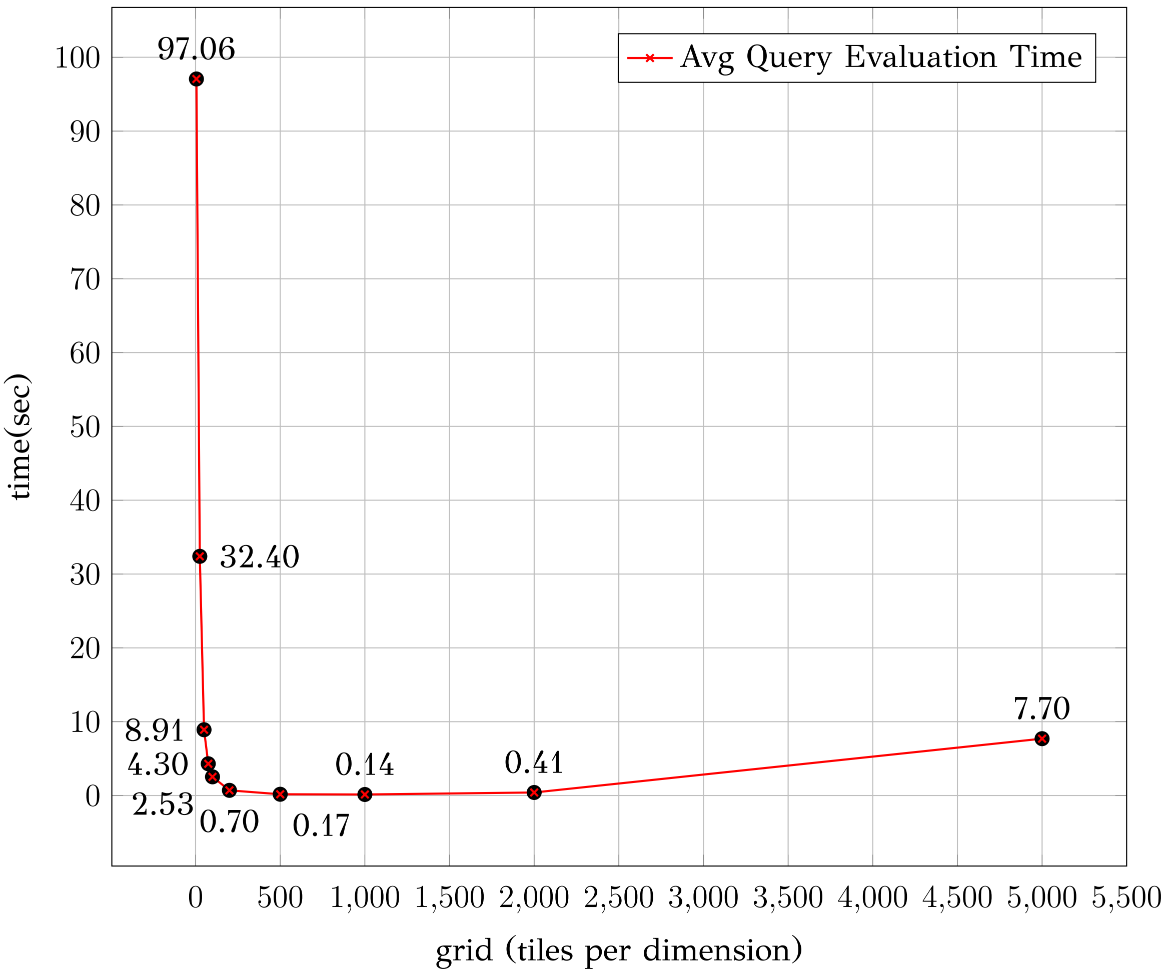


Figure 6.9: Effect of the grid's size on the average query evaluation time.

The parallel partitioning remains unaffected by the various tested grid sizes, but the query evaluation process is greatly influenced by the total number of tiles in the grid. Figure 6.9 displays the average query evaluation time for different grid sizes, in a 8-node environment with each one having 8 threads. The same data file and queries were used as the rest of this thesis' experiments so far.

It is obvious that with a larger grid size, the query evaluation time is significantly reduced. This is due to the fact that more tiles mean wider segmentation of the classified objects into data files and thus, faster evaluation by disregarding groups of records based on their corresponding intersecting tile. However, more data files on disk means higher I/O cost, since writing multiple files on a disk simultaneously using the computer's threads can be very heavy on the system.

Moreover, in the 5x5 grid case, the distributed environment was hugely under-utilized, since only 2 of the 7 slave nodes were used to evaluate the queries. As mentioned before, this is due to the fact that most of the objects were contained in 1 or 2 tiles, so the comparison workload fell on to at most two nodes for every query tested, significantly increasing the program's time.

Finally, as expected, there is a bottleneck to the amount of time improvement the grid size provides. From around 200 to 1000 tiles per dimension, the system's performance is very high, having an average query evaluation time of under 1 second. For grids bigger than 1000x1000, the performance begins to deteriorate due to the fact that there are too many hard disk operations being performed simultaneously at each slave node as well as higher free disk space requirement. The optimal grid size depends largely on the data set and may vary a lot, depending on the partitioned database.

6.1.2 Pre-Processing

In this subsection we analyze the pre-processing needed in setting up the partitioned database in depth. This operation along with its various features needs to be performed once before running any queries. After that, any changes to the database can happen without repeating the whole process all over again.

Data Part Files

As mentioned before, the master node needs to have the input data file separated into individual part files on its local disk. This must be performed before initiating the system and is done so through a separate C++ program called **file splitter**. It splits the data file into equal sized part files with each containing the same amount of lines (records) with the last one probably containing less. The initial order of the records is preserved throughout the data separation.

We examine the alleged effect that the number of part files may have on the system's performance. Each thread takes on part files in their entirety, so it is possible that for certain part file amounts the master may under-utilize its resources. This is because some threads may finish iterating their data files early and then have nothing to do but wait for the rest of the group to finish.

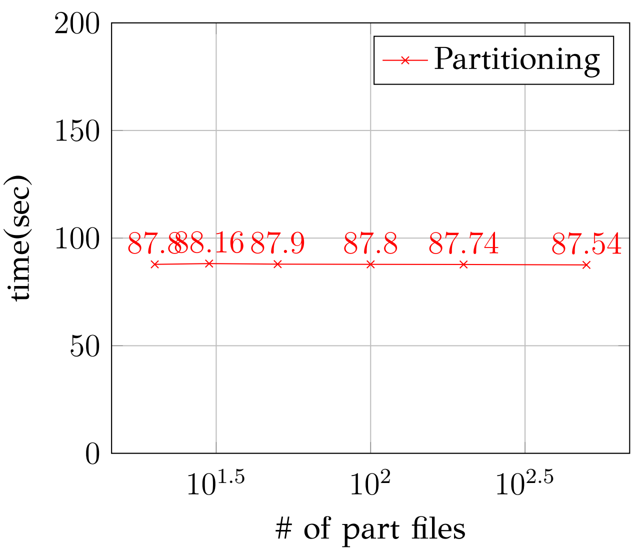


Figure 6.10: The effect of the amount of part files created by the pre-processing in the parallel partitioning.

Figure 6.10 illustrates the effect that the amount of part files created beforehand has on the parallel partitioning time. It is obvious that it does not affect the process at all, since the amount of records read by the threads remains unchanged. In fact, if there are at least that many part files as the available number of threads to be used in the parallel partitioning, then the program will work efficiently. However if there are less, then some threads will wait idle and thus the resources will be considered under-utilized.

Record Count & Border Values

These operations have been previously discussed and are a vital part of the system. Both the record count and the min/max values of the contained objects' coordinates are needed to properly perform the parallel partitioning. Specifically, the system needs the total number of records in the input data file and the $x_{min}, x_{max}, y_{min}, y_{max}$ values of the entire database.

Acquiring this information is time-expensive for the system since it repeatedly formats the string data and splits its contents to perform the comparisons, however if performed once then it is easily updated through changes in the database. If the user possesses this information, it is possible to have them provide it to the system without it having to performing these costly operations.

6.2 Runtime

Now that the partitioned database is properly set up in the distributed environment, the user may run any number of queries they wish. The master node reads each query from a text file saved on its local disk and initiates the parallel evaluation.

The user has the option to request the query results to be put into a data file on the master's disk or to be sent elsewhere for projection. The user must be careful though to not run queries that exceed the grid's borders (min/max coordinate values). A safeguarding mechanism can be implemented to prevent the user from going out of borders.

If the user wishes to add more nodes to the distributed environment, then the partitioning operation has to be run again from the beginning. On the other hand, adding, updating or deleting records can be done almost instantly through the master and then the corresponding slave nodes.

Chapter 7 discusses possible extensions for the system and future features.

6.3 Experimental Analysis

For each dataset, the experiments use different cluster sizes and the time values are the average of 4 executions per case. 8 threads were used in each computer and the partitioning was performed in a 1000x1000 grid.

The following experiments confirm the system's scalability and good performance for different types of datasets regarding the query evaluation. We have come to the conclusion that the partitioning remains mainly unaffected by the number of nodes in the distributed environment and is an unavoidable cost.

6.3.1 Dataset T1

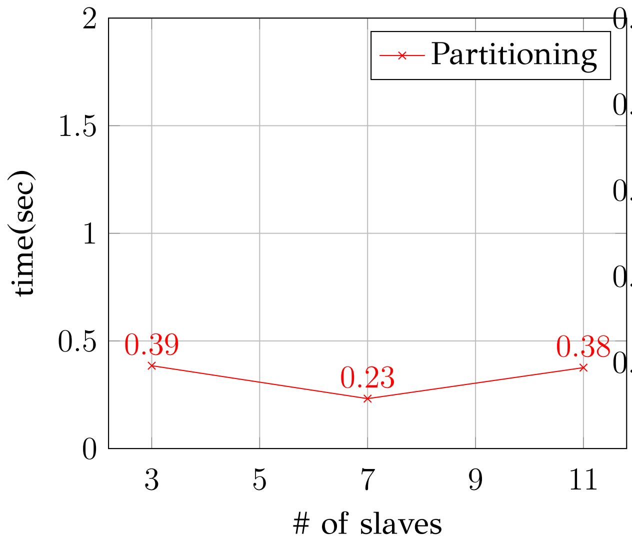


Figure 6.11: Effect of the total number of slave nodes on the partitioning time.

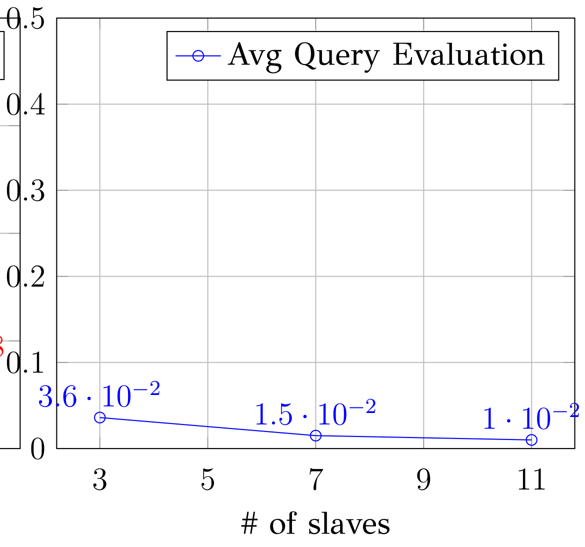


Figure 6.12: Effect of the total number of slave nodes on the average query evaluation time.

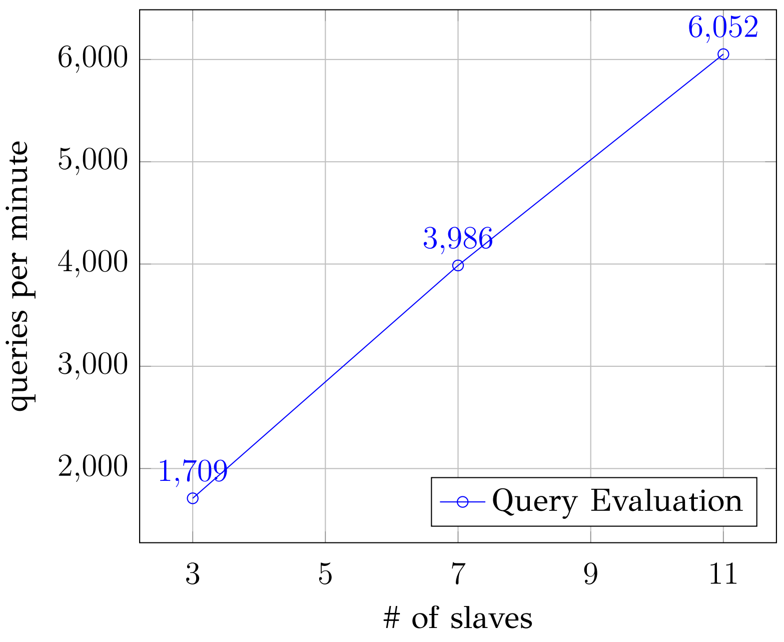


Figure 6.13: Effect of the total amount of nodes in the cluster on the query evaluation time, measured in queries per minute.

6.3.2 Dataset T4

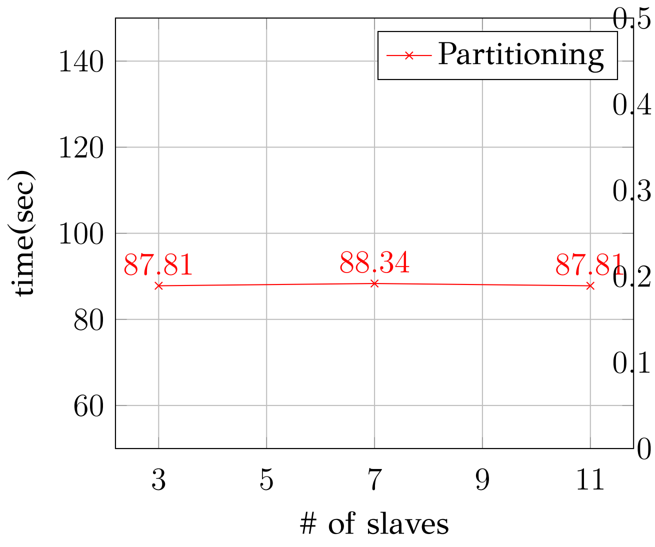


Figure 6.14: Effect of the total number of slave nodes on the partitioning time.

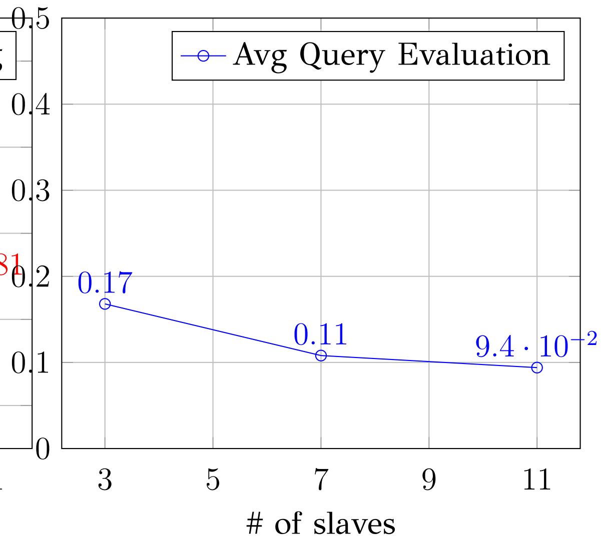


Figure 6.15: Effect of the total number of slave nodes on the average query evaluation time.

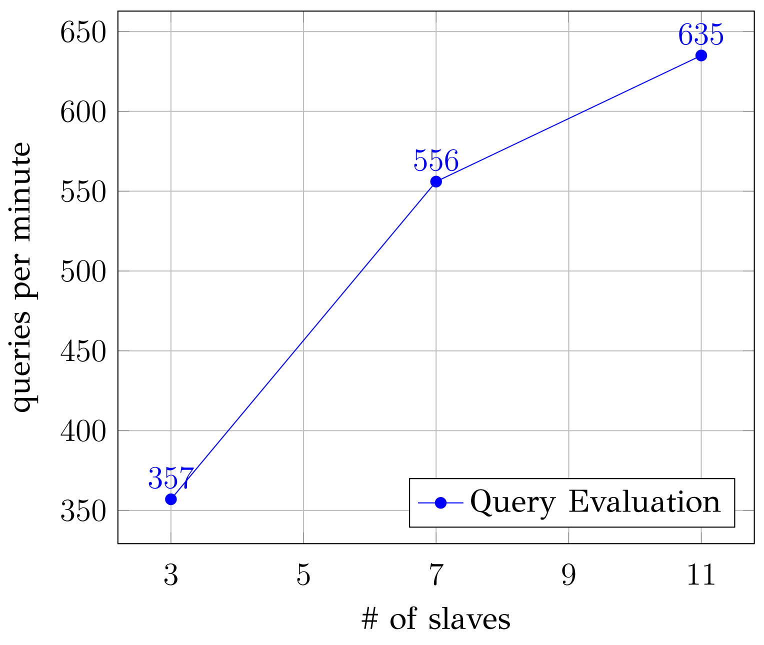


Figure 6.16: Effect of the total amount of nodes in the cluster on the query evaluation time, measured in queries per minute.

6.3.3 Dataset T8

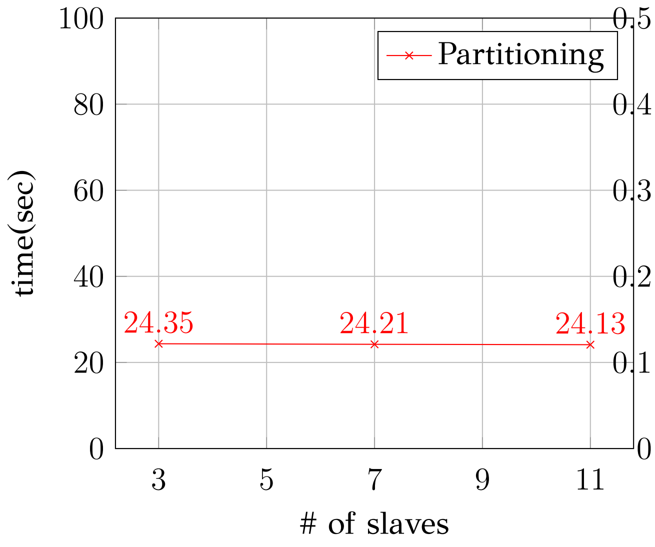


Figure 6.17: Effect of the total number of slave nodes on the partitioning time.

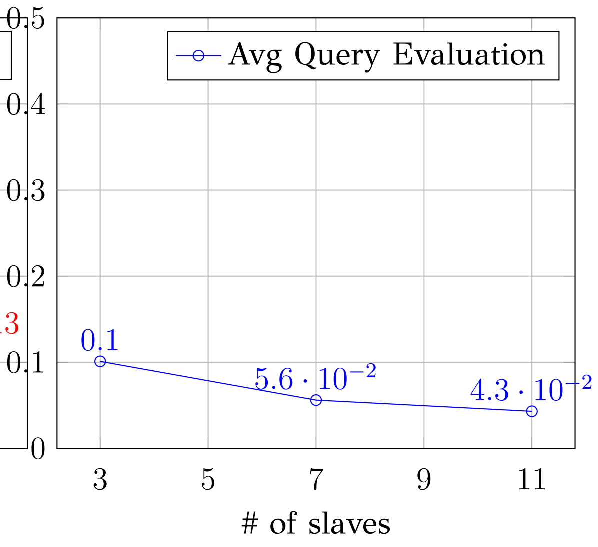


Figure 6.18: Effect of the total number of slave nodes on the average query evaluation time.

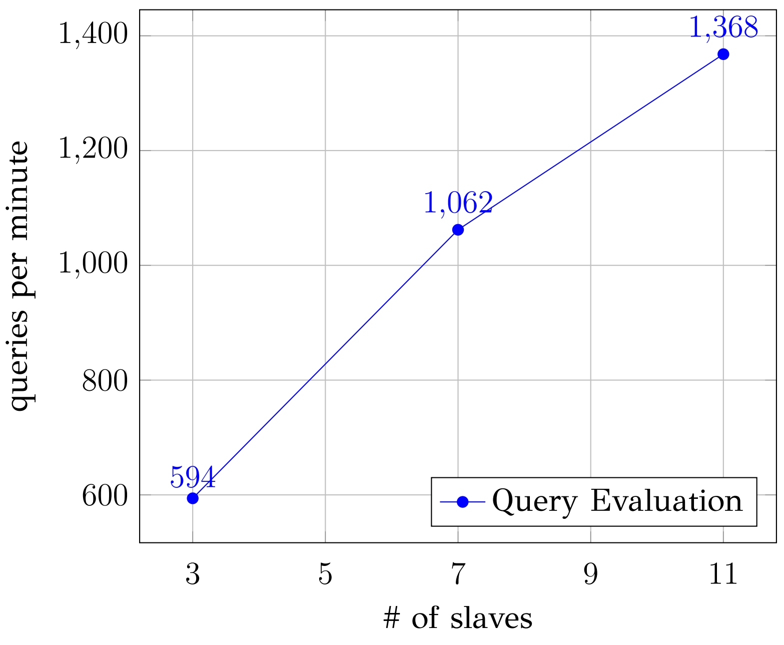


Figure 6.19: Effect of the total amount of nodes in the cluster on the query evaluation time, measured in queries per minute.

6.3.4 Cost Breakdown

The partitioning and the query evaluation are each constituted by individual operations that affect their total execution time. Coding techniques, data structures and I/O operations may have a direct effect on how fast the program is executed.

A cost breakdown of both phases' individual operations would help detect possible improvements to the system. We describe the steps of both phases and give an approximation of how much time they take, as a percentage of the total phase's time.

Both experiments were executed in the cluster with the T4 datafile. All 12 nodes were used as well as 8 threads per computer.

Partitioning Cost Breakdown

The partitioning process in the master node consists of the following individual functionalities:

- Part file assignment to threads
- I/O operations + string formatting per record
- Object classification
- Distribution related communications

As described in Chapter 4, each thread takes over different part files. The part file assignment is protected with mutual exclusion to guarantee exclusivity, so it is possible to cost time. After that, each record is loaded and formatted so that it can be classified based on the grid. Then, it is actually classified and placed on a message packet along with other records directed to the same node, before being sent to the appropriate slave node. The last 3 phases are repeated for each object in the dataset, so we must make sure that they do not cost much or they will hurt the system's scalability.

Due to the fact that threads are being used to parallelize each of these phases, every thread was timed separately and an average of their time was calculated for the experiment. Thus, the percentages are an approximation of these phases' times *per thread*, but they depict a fairly accurate image of the whole process since every thread performs the same tasks simultaneously.

Figure 6.20 shows how much each of the above operations costs as a percentage of the partitioning's total execution time. It is obvious that most time is spent on I/O operations and string formatting, to properly set up the data for classification. This task is performed for every object in the dataset and if we manage to reduce its cost, the partitioning would greatly benefit from it. However, I/O operations are a necessary cost in data management and sometimes not much can be done, since it comes down to the programming language's peculiarities.

One possible way of reducing the I/O cost would be to have the input data in a different format. Some data formats such as binary data, are loaded and handled faster than other. Exploring new data formats and ways to handle them could improve the system's performance even more.

The positive thing is that the data distribution (communications) and the objects' classification do not slow the whole operation too much and seem to scale well.

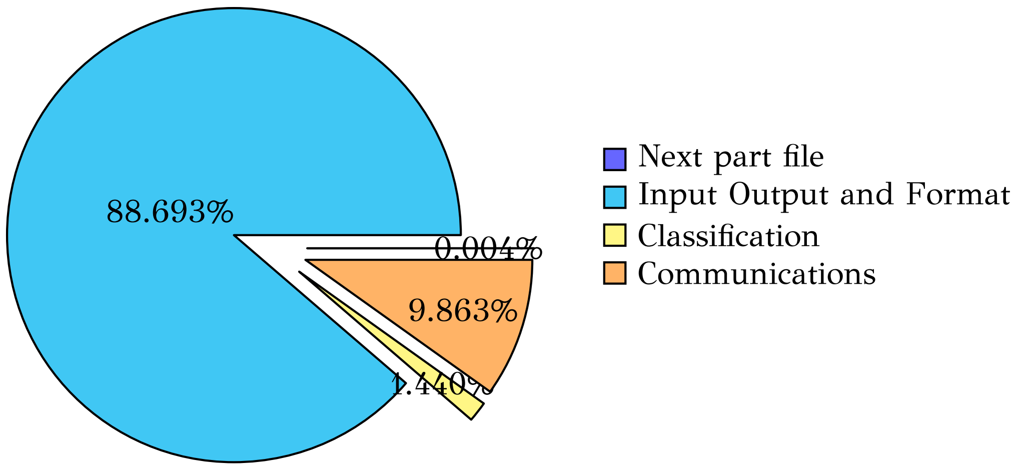


Figure 6.20: A cost breakdown of the partitioning process and its individual phases, as a percentage of the partitioning total time.

Query Evaluation Cost Breakdown

The query evaluation process in each slave node consists of the following individual functionalities:

- Query receipt + deconstruction
- Evaluation
- Communications

Slave nodes receive only those window queries that intersect tiles assigned to them, as described in Chapter 4. For every query, they must format it to extract the needed information such as query ID, the window's coordinates and the corner tiles that it intersects in the grid. Then, they can move on to actually evaluating the query using their threads. Each thread sends messages back to the master containing the results it found regarding that query.

Since workload is divided in the environment's computers, timing the individual operations is tricky. The experiment timed each slave node separately and averaged their execution time for each operation. The results shown in Figure 6.21 are an approximation of the operations' effect on the whole process.

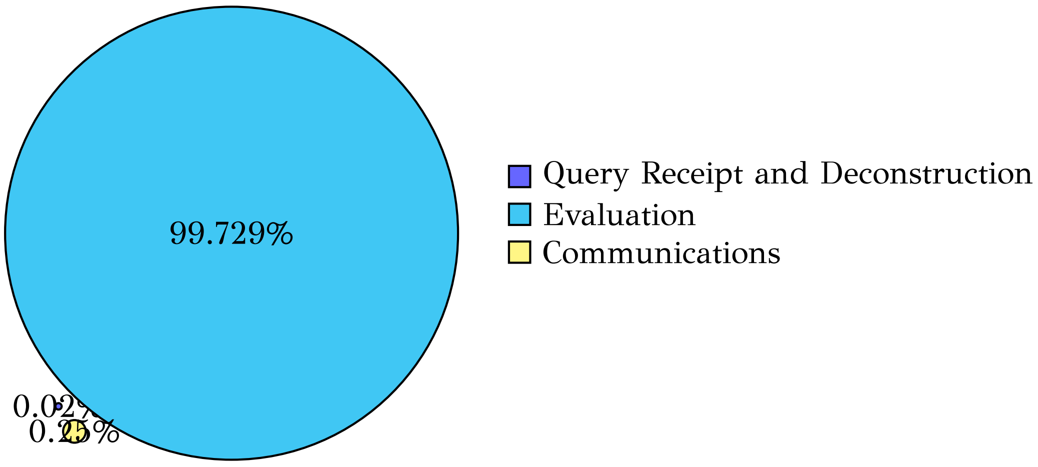


Figure 6.21: A cost breakdown of the partitioning process and its individual phases, as a percentage of the partitioning total time.

Most of the query evaluation's total time belongs to the actual query evaluation. It means that not much time is spent in string formatting or communications, but on the actual comparisons that perform the evaluation. By extension, this means that by adding more computers to the distributed environment, the cost due to their inter-process communication will probably not increase that much and thus, preserving the system's scalability.

However, since the evaluation method is optimized, it means that the overall query evaluation process may not be improved much more, since there is virtually no way to enhance it. We could examine the load balancing as well as the processor utilization, in an attempt to improve it by better distributing the workload. This could be done by timing the average idle time of each node and possibly creating a new way of data assignment to nodes, based on their distribution type in the axes.

Workload Distribution

To examine the amount of resource utilization in the system during the query evaluation, we have to find out how much of its resources are used per query. In both experiments we use the T4 dataset in the cluster using all of the 12 nodes (11 slaves). Only the first 100 queries from the query file were used. The grid size is 1000x1000.

First, we show how many nodes are used to evaluate each query. Figure 6.22 shows the actual number of nodes participating in each query. No more than 6 nodes, which is slightly more than half of the slave computers, are being used per query with the lowest being 4 computers. This amount depends largely on the grid size and even though we used a fairly large grid, half the system was sitting idle in each query.

This means that the system is under-utilized or that the queries never intersect more than a few of the grids tiles. Either way, workload distribution optimization will not be studied in the rest of this thesis, though possible workarounds are suggested for future work in Chapter 7.

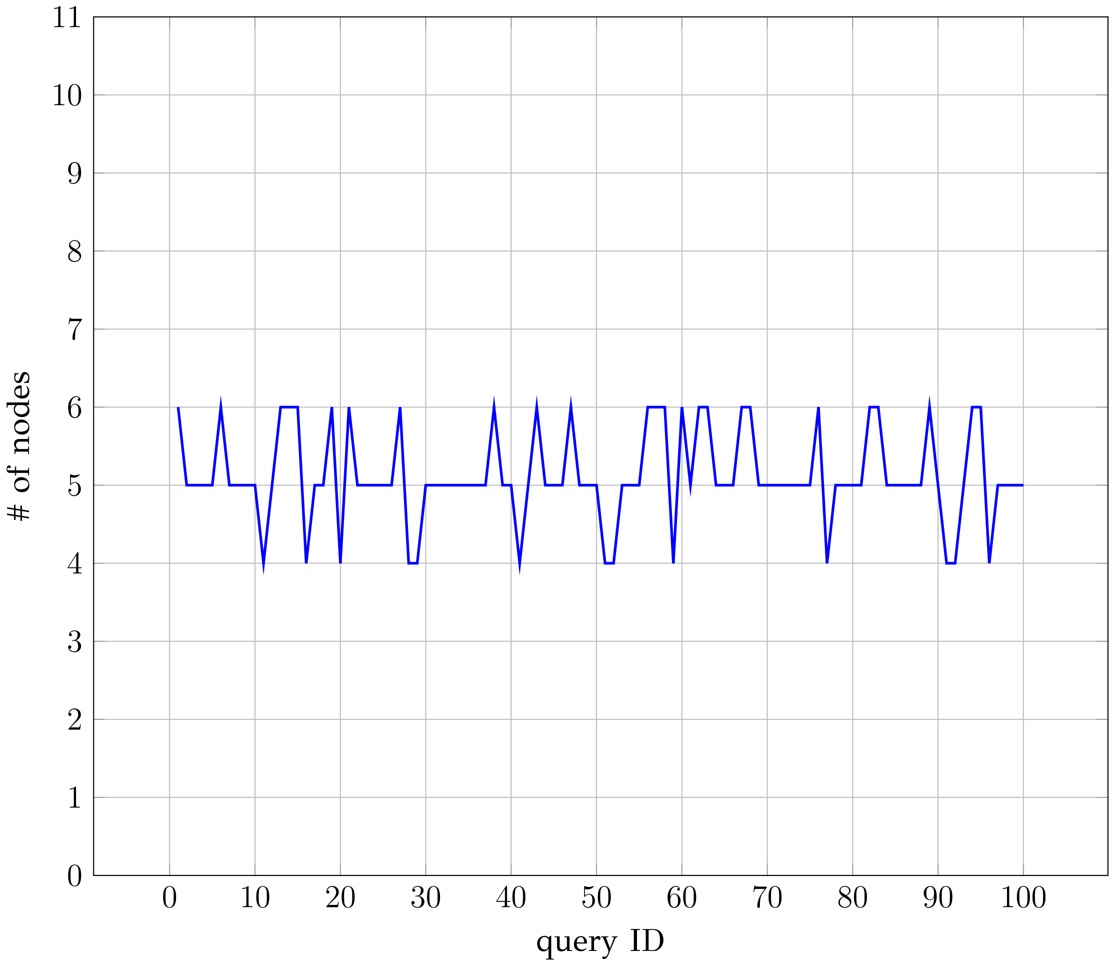


Figure 6.22: The total number of slave nodes used to evaluate each query.

Chapter 7

Conclusion

7.1 Summary

In this project, I have implemented an end-to-end distributed spatial data management system, which is based on a grid-based partitioning. As explained in this thesis, the implemented system consists of two individual functionalities: the partitioning and the query evaluation. Each one was examined thoroughly in an attempt to determine the optimal way in which they can be realized.

The parallel partitioning remains the most expensive operation of the two, since it heavily depends on the actual number of records inside the input data file. Even though it utilizes the master node's threads to parallelize the process, there is some serious workload regarding I/O operations and the benefits of hyper-threading for more than 8 threads remain untested. However, running the partitioning just once, will properly setup the partitioned database permanently on disk, allowing the user to run any number of range queries they wish at that moment or in the future.

Moreover, the user has control over the total memory usage by the system in both the master and slave nodes. Depending on the computers' specifications in the distributed environment, they may control how much RAM is being used and the maximum allowed size of each message packet being communicated through the network as well.

The system in its latest version, using the optimal parameters and setup as discussed in Section 6.1.1, manages to evaluate range queries very fast. The user has the option to run multiple range queries in a single execution by placing them on the appropriate text file on the master node's local disk. The system evaluates each query consecutively and either prints the results or stores them in a user-specified location on disk.

7.2 Future Work & Extensions

There are numerous possible expansions and improvements for future work that this system can benefit from, due to its unique and innovative nature. Enhancements and upgrades can be applied in most of the system's individual functionalities, improving its overall performance.

The partitioning is definitely capable of improvement, by utilizing binary files instead of text. Formatting and handling strings as well as converting them into numerical values when comparisons are needed can be very time-expensive and it is the main reason that the parallel partitioning is so slow. Moreover, the messages exchanged during the inter-process communication can be much smaller in size, thus transported more quickly, if their contents represent binary values and not characters. Avoiding the use of string structures and their replacement with binary values when possible, will certainly improve the system's overall performance and memory usage.

The query evaluation can be optimized to either send the results back to the master node or to dump them on disk and join them in a single file. Which method is best has yet to be determined and possible future work may improve even better the overall query evaluation performance. Moreover, the tile-to-node assignment can be implemented in a way that favors the dataset's distribution for optimal node utilization.

The system currently works for rectangle objects (MBRs) whilst the theory that it is based on regards non-point spatial data in general. Thus, a possible future extension may include linestring and polygon data types. On top of that, the system currently performs window queries exclusively. More query types such as kNN (k-nearest neighbors) and spatial join may be included in future versions.

Additionally, functions that can alter the database such as update, insert or delete, can be implemented to reduce the need for re-partitioning of the data. In this way and since the partitioning is largely correlated with the number of records being partitioned, the huge cost that comes with this operation will not appear that often, unless huge chunks of data are being inserted to the database frequently.

The query evaluation operation can be additionally parallelized by performing the evaluations simultaneously and asynchronously. The idea is that each slave node will move on the next query after sending the results for a previous one. The master makes sure to distribute all the relevant queries to the slave nodes, so that each node will continuously work without interruptions to evaluate all the queries that it has received. The master node is responsible for properly organizing and separating the query results it receives. This type of asynchronous query evaluation is expected to improve the system's performance for large batches of window queries.

Finally, we were unable to compare the system's performance with similar distributed spatial data management implementations such as GeoSpark (Sedona) [4], due to installation difficulties and restricted access to the cluster. An experimental comparison between the two would expose possible strengths and weaknesses in the system and assist in its future development.

Bibliography

[1] G. Graefe, *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, June 1993.

[2] *Apache Spark: a unified analytics engine for large-scale data processing.*, Apache Software Foundation. [Online]. Available: https://spark.apache.org/

[3] *Apache Hadoop*, Apache Software Foundation. [Online]. Available: https://hadoop.apache.org/

[4] *Apache Sedona*, Apache Software Foundation. [Online]. Available: http://sedona.apache.org/

[5] *Magellan: Geospatial Analytics Using Spark.* [Online]. Available: https://github.com/harsha2010/magellan

[6] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *2016 International Conference on Management of Data (SIGMOD Conference 2016)*, San Francrisco, CA, USA, July 2016, pp. 1071–1085.

[7] V. Pandey, A. Kipf, T. Neumann, and A. Kem, "How good are modern spatial analytics systems?" in *Proceedings of the VLDB Endowment, Volume 11, Issue 11*, July 2018, pp. 1661–1673.

[8] S. T. Leutenegger, J. Edgington, and M. A. Lopez, "Str: A simple and efficient algorithm for r-tree packing," in *13th International Conference on Data Engineering (ICDE)*, May 1997.

[9] D. Tsitsigkos, K. Lampropoulos, P. Bouros, N. Mamoulis, and M. Terrovitis, "A two-layer partitioning for non-point spatial data," in *37th International Conference on Data Engineering (ICDE)*, Chania, Greece, April 2021.

[10] *OpenMP application program interface version 4.5*, OpenMP Architecture Review Board, November 2015.

[11] M. J. Quin, *Parallel Programming in C with MPI and OpenMP*, international ed. Boston, USA: McGraw-Hill, 2003, pp. 406–407.

[12] *MPI: A Message Passing Interface Standard*, MPI Forum. [Online]. Available: https://www.mpi-forum.org/

[13] *Open MPI*, Open MPI Development Team. [Online]. Available: https://www.open-mpi.org/

[14] *MPICH Version 3.3*, MPICH, January 2019. [Online]. Available: https://www.mpich.org/

[15] V. V. Dimakopoulos, *Parallel Systems and Programming*. Athens, Greece: Hellenic Academic Libraries Link, 2015.

[16] *Spatial Hadoop*. [Online]. Available: http://spatialhadoop.cs.umn.edu/datasets.html

[17] J.-P. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *16th International Conference on Data Engineering (ICDE)*, 2000, pp. 535–546.