

Compilation-based Spatial Query Processing

by

Rahul Sahni

Bachelor of Technology in Computer Science, Guru Gobind Singh
Indraprastha University, 2019

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF**

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Suprio Ray, Ph.D., Computer Science
Examining Board: Georgiy Krylov, Ph.D., Computer Science (Chair)
Mohammad Mamun, Ph.D., Computer Science
Shabnam Jabari, Ph.D., Geodesy and Geomatics
Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2024

© Rahul Sahni, 2024

Abstract

The proliferation of spatial data applications and rising spatial data volumes demand efficient processing capabilities. Although most relational databases support spatial extensions of SQL, they offer limited scalability. Traditional relational database follows a pull-based model of query processing. This is inefficient for processing large volumes of data. Specialized systems, such as those extending Hadoop and Spark, improve scalability but often lack comprehensive SQL support or suffer from the overheads of the pull-based model.

This thesis introduces a distributed spatial query processing system using the Push-based query compilation approach, generating C++/UPC++-based query plans for both single node and distributed execution on a high-performance framework using the Partitioned Global Address Space paradigm. It also proposes two new morsel-driven parallelism algorithms for scalable spatial query execution. Experiments on real-world datasets show significant performance gains over leading systems, including Apache Sedona, Citus - a distributed database based on PostgreSQL, and PostgreSQL in single-node configurations.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Suprio Ray. His guidance and support was monumental and made it possible for me to carry out this research work. My deepest thanks to Xiaozheng Zhang, for helping me running some of the experiments, he made it possible for me to finish my work in time. I am also grateful to my talented lab mates, Sudip Chatterjee, Avinaba Mistry, Ronnit Peter, Suvam Das and Saumya Verma for their help and support throughout my research. Special thanks to my roommates, Sai Shashank and Mohammed Tabrez for cooking those delicious meals while I working late at the lab.

Above all, I would like to extend my heartfelt thanks to my family for always supporting and cheering me from India, throughout my academic journey. I am also grateful to Hayley Currie for always believing in me and encouraging me. Finally, I am thankful to my sweet cat Izzy, for always entertaining me with her insane parkour skills.

Table of Contents

Abstract	ii
Acknowledgments	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
Abbreviations	xi
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Statement	3
1.3 Research Questions	3
1.4 Thesis Contributions	4
2 Background and Related Work	5
2.1 Background	5
2.1.1 Query Processing	5
2.1.2 Query Compilation	7
2.1.3 Morsel-driven Parallelism	8
2.1.4 Apache Calcite	9
2.1.5 Spatial Data Processing	10

2.1.6	Spatial Data Types	12
2.1.7	Bounding boxes and Spatial Index	16
2.1.8	Spatial Functions	18
2.1.9	Spatial Partitioning	18
2.1.10	Spatial Joins	21
2.1.11	PGAS and UPC++	22
2.1.12	Distributed Spatial Query Processing	24
2.2	Related Work	24
2.3	Summary	25
3	Design and Implementation	27
3.1	CasaDB	27
3.2	Architecture	28
3.2.1	Query Plan Generator Module	29
3.2.2	Code Generation Module	29
3.2.3	Machine Code Generation Module	32
3.2.4	Data Partitioning Module	32
3.2.5	Data Assignment Module	33
3.2.6	Spatial Morsel Parallelism Module	33
3.3	Tile Assignment	33
3.4	Index Organization	34
3.5	Spatial Morsel-Driven Parallelism	35
3.5.1	Monolithic Tile-based Morsel Parallelism (MTMP)	36
3.5.2	Granular Tile-based Morsel Parallelism (GTMP)	37
3.5.3	Distributed MTMP/GTMP	41
3.6	Query Processing	42
3.6.1	Spatial Join processing	42
3.6.2	Spatial Range Join processing	43

3.6.3	Spatial Distance Join processing	44
3.7	Summary	44
4	Evaluation	46
4.1	Experimental Setup	46
4.2	Dataset	47
4.2.1	TIGER Dataset	47
4.2.2	OSM Dataset	47
4.2.3	Queries	48
4.3	Code Generation and Compilation	49
4.4	Execution Time Breakdown	50
4.5	CasaDB - Single Node Evaluation	51
4.5.1	Performance analysis of GTMP and MTMP	51
4.5.2	Comparison with PostgreSQL	52
4.6	CasaDB - Distributed Evaluation	55
4.6.1	Partitioning granularity analysis	55
4.6.2	Performance analysis of GTMP and MTMP	55
4.6.3	Scalability of our system	57
4.6.4	Comparison with Apache Sedona and Citus	59
4.7	Summary	62
5	Conclusion and Future Work	64
5.1	Conclusion	64
5.2	Future Works	65
5.2.1	Code Compilation Time	65
5.2.2	Spatial Distance Join Processing	65

Vita

List of Tables

- 2.1 Geometries represented in WKT 13
- 2.2 Spatial Functions 19

- 3.1 Calcite and CasaDB operators 31

- 4.1 TIGER Dataset 47
- 4.2 OSM Dataset for UK 47
- 4.3 TIGER dataset queries 48
- 4.4 OSM dataset queries 48

List of Figures

2.1	Steps in query processing [52]	6
2.2	(a) Relation algebra plan (b) plan with pipeline boundaries [40]	8
2.3	Apache Calcite architecture [11]	9
2.4	Evolution of GIS [19]	11
2.5	Spatial data types [53]	12
2.6	(a) MBR of a line, (b) MBR of a polygon, (c) MBRs of line and polygon intersecting each other	16
2.7	R-tree based indexing using MBRs [46]	17
2.8	Quadtree-based spatial partitioning [21]	20
2.9	Spatial Join	21
2.10	Spatial Range Join	22
2.11	Spatial Distance Join	22
2.12	PGAS memory organization	23
3.1	CasaDB - Single Node Architecture	28
3.2	CasaDB - Distributed Architecture	28
3.3	Produce-Consume Model	30
3.4	Code generation	31
3.5	Index organization	34
3.6	Monolithic Tile-based Morsel Parallelism	38
3.7	Granular Tile-based Morsel Parallelism	38
4.1	Code generation time for TIGER queries	49

4.2	Code compilation time for CasaDB - Single Node	49
4.3	Code compilation time for CasaDB - Distributed	50
4.4	Execution time breakdown for TIGER_Q3	50
4.5	MTMP vs. GTMP for TIGER_Q1, TIGER_Q2 with different parti- tion granularity for TIGER dataset	52
4.6	MTMP vs. GTMP for TIGER_Q3, TIGER_Q4 with different parti- tion granularity for TIGER dataset	52
4.7	MTMP vs. GTMP for TIGER_Q5, TIGER_Q6 with different parti- tion granularity for TIGER dataset	53
4.8	MTMP vs. GTMP for TIGER_Q7, TIGER_Q8 with different parti- tion granularity for TIGER dataset	53
4.9	CasaDB - Single Node vs. PostgreSQL for TIGER	54
4.10	CasaDB - Single Node vs. PostgreSQL for OSM	54
4.11	GTMP performance for TIGER_Q1, TIGER_Q2 with different parti- tion granularity for TIGER dataset	56
4.12	GTMP performance for TIGER_Q3, TIGER_Q4 with different parti- tion granularity for TIGER dataset	56
4.13	GTMP performance for TIGER_Q5, TIGER_Q6 with different parti- tion granularity for TIGER dataset	56
4.14	GTMP performance for TIGER_Q7, TIGER_Q8 with different parti- tion granularity for TIGER dataset	57
4.15	GTMP vs. MTMP for TIGER_Q1 and TIGER_Q2 queries	58
4.16	GTMP vs. MTMP for TIGER_Q3 and TIGER_Q4 queries	58
4.17	GTMP vs. MTMP for TIGER_Q5 and TIGER_Q6 queries	58
4.18	GTMP vs. MTMP for TIGER_Q7 and TIGER_Q8 queries	59
4.19	CasaDB - Distributed with TIGER Dataset	60
4.20	CasaDB - Distributed with OSM Dataset	60

4.21 CasaDB vs. Citus vs. Apache Sedona for TIGER	61
4.22 CasaDB vs. Citus vs. Apache Sedona for OSM	62

Abbreviations

API	Application Programming Interface
BLOB	Binary Large Objects
DBA	Database Administrator
GIS	Geographic Information System
GTMP	Granular Tile-based Morsel-driven Parallelism
JIT	Just-in-Time
JSON	JavaScript Object Notation
MBR	Minimum Bounding Rectangle
MTMP	Monolithic Tile-based Morsel-driven Parallelism
OGC	Open Geospatial Consortium
OSM	OpenStreetMap
ORDBMS	Object Relational Database Management System
PGAS	Partitioned Global Address Space
RDBMS	Relational Database Management System
RMA	Remote Memory Access
RPC	Remote Procedure Call
SQL	Structured Query Language
TIGER	Topologically Integrated Geographic Encoding and Referencing
QEP	Query Execution Plan
WKB	Well-Known Binary
WKT	Well-Known Text

Chapter 1

Introduction

Relational database management systems (RDBMS) are widely used for enterprise data management, partly due to the popularity of SQL. RDBMS engines follow an iterator-based “tuple-at-a-time” model, which is also known as the Volcano model [25]. However, their main focus was to minimize disk I/O and CPU utilization was less important, so they could not take full advantage of the improvements in processor technology and show poor performance on modern CPUs [40] [2]. Hence, they suffer from performance bottlenecks on modern CPUs. They are inherently inefficient in terms of performance due to processing each tuple by making repeated calls to the *next* function call for each relational operator in the query plan from an input tuple stream [40].

Driven by advances in computer architecture, modern machines are equipped with large main memory and several processing cores. Since larger amounts of data can fit in the main memory than was previously possible, optimizing code for memory usage and adopting CPU-efficient techniques have become more important. In recent years, query compilation-based execution model has attracted considerable attention from the research community [47, 40, 31].

Although query compilation can offer significant performance benefits, re-architecting

SQL query engine to incorporate data-centric compilation is challenging because of the associated complexity. Due to the wide adoption of traditional and emerging geo-spatial applications, providing support for spatial operations became a necessity for information management systems. Furthermore, adapting query compilation techniques for spatial workloads entails additional complexities. This thesis presents a compilation-based spatial query processing, which combines high-performance computing with parallel and distributed data processing.

1.1 Motivation

The volume of spatial data is rising due to many factors, including the spread of GPS-enabled mobile devices and sensors, geo-social media, advances in remote sensing and satellite imaging, and improving storage capacity. In response to the challenges of spatial big data, several specialized systems were proposed that extended cluster computing frameworks Hadoop and Spark. These systems enabled practitioners to process spatial data by leveraging a cluster of machines. Although they provide good scalability, they were still slow in executing the queries [61]

Query compilation generates query-specific and data-centric code and is different from traditional interpretation-based query processing. A lot of research is focused on query compilation techniques for single node and on non-spatial workloads. Although query compilation can offer significant performance benefits, re-architecting an SQL query engine to incorporate data-centric compilation is challenging because of the associated complexity. Consequently, existing RDBMSs have either not adopted it at all, or they support a very limited form of query compilation. For example, currently, PostgreSQL supports just-in-time (JIT) query compilation for tuple materialization and expression evaluation only. This requires building PostgreSQL from source with the flag `--with-llvm` or building it with OMR JitBuilder support [18]. Adapting

query compilation techniques for spatial workloads entails additional complexities. To our knowledge, only one previous research made an endeavour [54], where they identify why existing query compilation techniques are not quite effective for spatial queries. They propose a generative query compilation approach, LB2-Spatial, which transpiles a spatial SQL query into a source program (in Scala or C) and it then gets compiled into native code and executed. To our understanding, LB2-Spatial focuses on MBR-based spatial query execution (essentially, the Filter step of the 2-step Filter-Refinement process). Moreover, LB2-Spatial is based on a single node. Such an approach is not scalable, particularly, in view of rapidly growing data volume.

1.2 Thesis Statement

Execution time of spatial queries can be significantly reduced using query compilation techniques like Push-based model [40]. This technique generates code that is data-centric and then further improves the query execution speed using dynamic parallelism and can easily handle volumes of data using distributed computing.

1.3 Research Questions

The following research questions (RQ) have been pursued to support the thesis statement:

- **RQ 1:** How can we create a query execution engine for spatial queries using Push-based model and support dynamic parallelism and distributed computing?
- **RQ 2:** How does the system created in RQ1 perform against single node RDMS and distributed RDBMS?

1.4 Thesis Contributions

The work presented in this thesis has the following contributions:

- We propose a query compilation-based spatial SQL query processing system, that can generate code for a single node or a distributed runtime based on PGAS paradigm.
- We introduce two algorithms for morsel-driven parallel processing of spatial queries: Monolithic Tile-based Morsel-driven Parallelism (MTMP) and Granular Tile-based Morsel-driven Parallelism (GTMP).
- We present two index organization techniques, Global Index and Tile Index and show how they can be used with different kinds of spatial joins.
- We present extensive experimental results involving two real-world datasets.

Chapter 2

Background and Related Work

This chapter explains the concepts related to query processing, query compilation, spatial data and spatial query processing. It also discusses the distributed aspect of query processing and discusses libraries and frameworks used to build the system presented in this thesis. Finally, it explores the related work in the field and how our system tackles their shortcomings.

2.1 Background

This section gives an overview of how query processing works and the query execution techniques used by different query engines. It also discusses spatial query processing and its components.

2.1.1 Query Processing

A declarative language such as Structured Query Language (SQL) is used to define a query, which is processed by the query engine to extract data from the database as per the constraints defined in the query. This whole process is called query processing. Query processing has the following stages [52]:

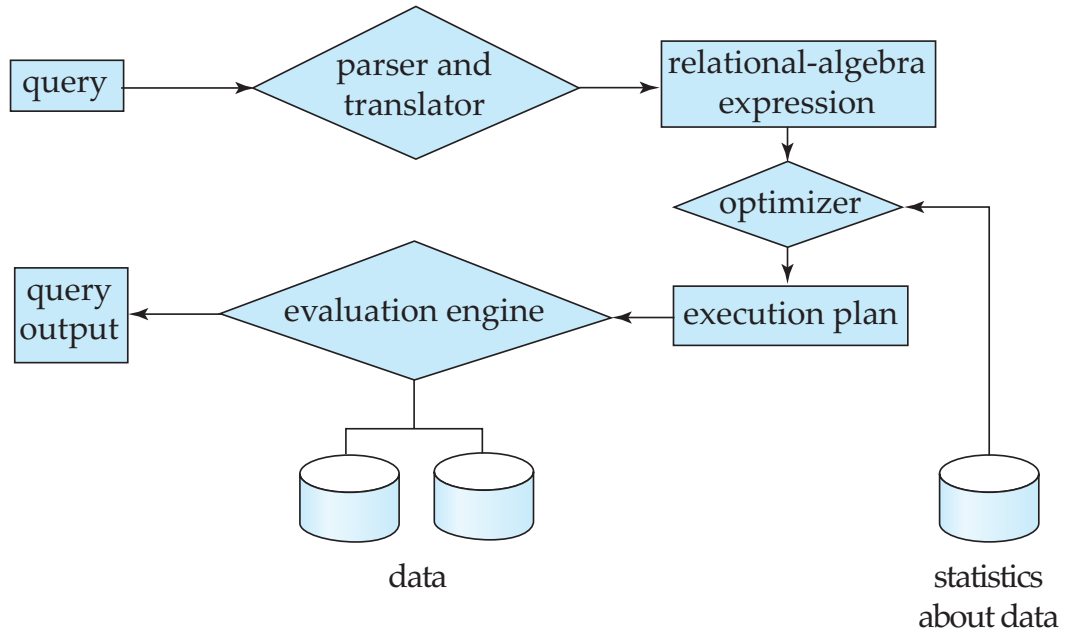


Figure 2.1: Steps in query processing [52]

- *Parsing and Translation:* When the query engine first receives an SQL query, the Lexical Analyzer part of the query engine breaks it into tokens having distinct meanings. These tokens can represent keywords, operations, constants or identifiers. The Syntax Analyzer receives the tokens and makes sure that the SQL is following the defined grammar and checks that all the tokens are valid. Finally, the Translator converts the appropriate tokens into Relational Algebra operators.
- *Optimization:* A Relational Algebra expression can be represented in many forms, and each of the forms has an associated cost related to it. In this stage, the optimizer calculates the cost of expressions and annotates them with an evaluation strategy, also known as an evaluation plan. The optimizer then chooses the most optimal evaluation plan using statistics like tuple count, index available on the table and system factors like disk access, CPU, and network communication. Finally, it creates a physical query plan.

- *Evaluation*: The physical query plans inform the engine about the exact operation to perform on the Relational Algebra operator and finally, they are executed to get the final result.

2.1.2 Query Compilation

Many of the database engines were developed when I/O was dominating the overall processing time and the main memory capacity was low. The popular way of query processing is to generate a query execution plan (QEP) tree, where each node corresponds to a relational operator. Each operator calls *next* to get a tuple from its child operator until it reaches the leaf node of the tree. This *tuple-at-a-time*/Volcano model [25] is a bottleneck for modern CPUs as the processing power has grown significantly in the last few decades. In addition, the capacity of the main memory has increased significantly. Therefore, database researchers are trying to mitigate the performance bottleneck by using query compilation techniques [47, 40, 31].

The main concept behind query compilation is to process the data in a tight loop and perform operator fusion. A tight loop is a loop having little to no function call, minimal memory allocations, efficiently uses hardware resources and overall has minimal overhead. In this way, the code and data reside in the CPU cache, thereby, producing better performance. Instead of iterating over every operator in the QEP tree for each tuple, the query compilation technique generates high-level/low-level code. The query compilation approach, proposed by Krikellas, Viglas and Cintra [31], generates code for every operator. Hence the operator boundaries were explicit. Later, Neumann et al. [40] formally define *pipeline breaker* as an operation that takes a tuple out of the CPU register and spills it to the memory during query processing. They introduced *Push-based model* and showed how operator fusion can improve performance significantly by combining multiple operators in a tight loop until there is a *pipeline breaker*, such as join. Fig. 2.2 shows a relational algebra

plan with pipeline boundaries. In this thesis, this approach of query compilation for spatial data is chosen. In addition, the generated code can be executed in distributed environments.

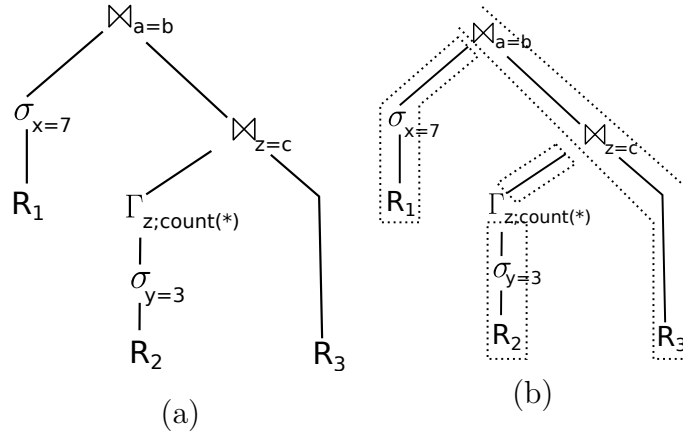


Figure 2.2: (a) Relation algebra plan (b) plan with pipeline boundaries [40]

2.1.3 Morsel-driven Parallelism

Parallelism is achieved in the Volcano model using *exchange* operators [24]. This operator encapsulates the parallelism such that the other operators are not aware of the parallelism. It acts as a bridge between the operators running on different threads by routing the tuple streams between them. The degree of parallelism is statically determined by the optimizer so, the parallelism is plan-driven and heavily influenced by the partitioning scheme of the data. While exchange operators simplify the support for parallelism in new operators, they introduce performance bottlenecks due to context switching, data transfer between threads, and static parallelism. Additionally, load-balancing challenges arising from the chosen partitioning scheme can further impact performance.

To overcome these limitations, Morsel-driven parallelism [32] was introduced. It achieves parallelism by running the operator pipelines in parallel on separate threads and these threads are bounded by the number of hardware threads of the processor.

The data is divided into small fixed-sized chunks, called “morsels” and then assigned to the threads working on a pipeline. Once a morsel is processed, the worker is assigned another morsel and keeps executing the same until the next pipeline breaker. A key feature of this approach is that task distribution occurs at run-time, enabling effective load balancing. It supports dynamic parallelism, with the degree of parallelism determined independently of the query plan.

2.1.4 Apache Calcite

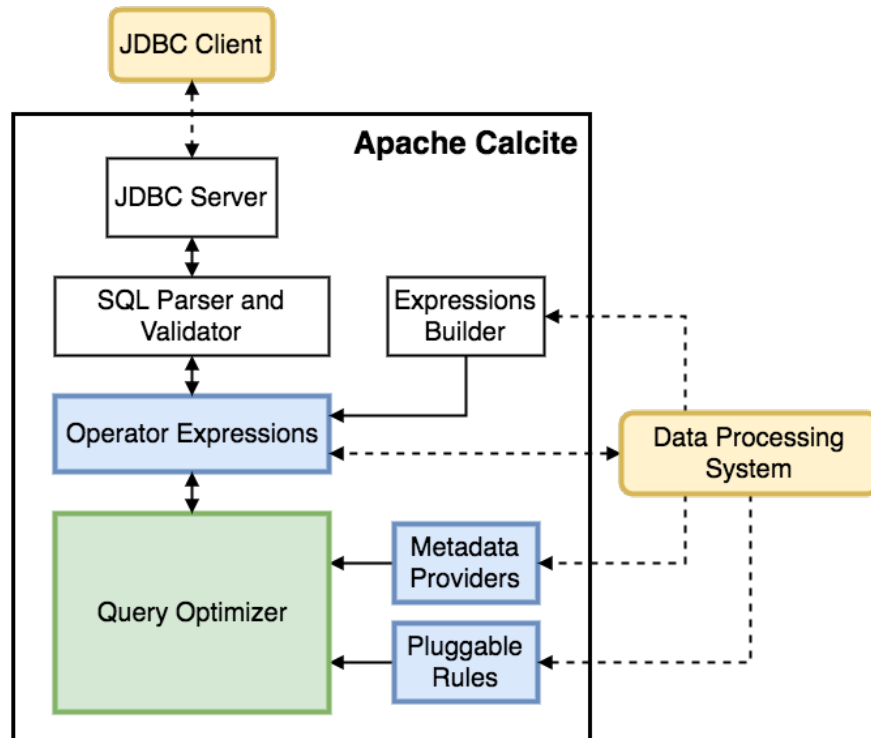


Figure 2.3: Apache Calcite architecture [11]

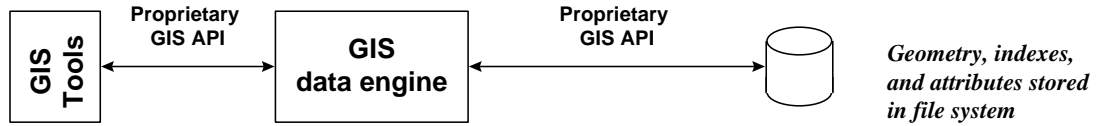
Apache Calcite [11][8] is an open-source dynamic data management framework. Calcite offers a great foundation for building a database engine, some key features include SQL Parser and Validator to parse and validate SQL queries, and query optimization which offers rule-based and cost-based optimization techniques. It provides Relational Algebra support, which can be used to represent the SQL queries as a

relational algebra operator and can further transform and optimize them. Calcite optimizes queries by repeatedly applying planner rules to a relational expression. A cost model guides the transformation process and the planner engine generates an alternative expression that has the same semantics as the original but at a lower cost. Calcite can connect to various data sources including relational databases, NoSQL data stores and streaming data sources using the adapter architecture. It provides schema adapters to read data from sources like Apache Arrow [27], Apache Cassandra [13], MongoDB [38], Redis [50] and many others including CSV files. It is widely used by open-source data processing systems like Apache Hive [30], Apache Flink [23], Alibaba MaxCompute [36] and various others for SQL parsing, query optimization, data virtualization/federation, and materialized view rewrite [11]. Calcite also supports building database drivers through Avatica [10], which is a sub-project of Calcite. Apache Calcite is widely adopted in the open-source community and supports various data sources, extensible and pluggable query optimizer with relational algebra support.

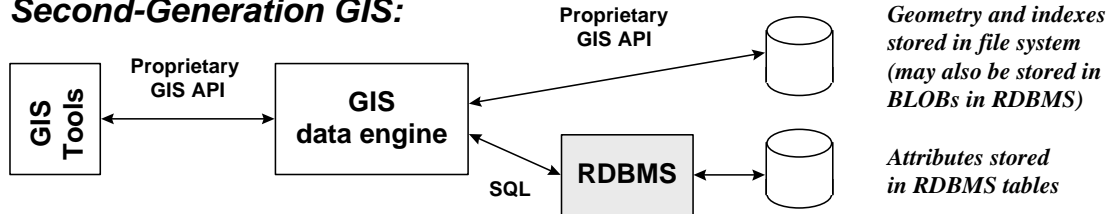
2.1.5 Spatial Data Processing

Spatial data represents location, shape, and relationship with other objects in a space. Spatial data can be broadly categorized into two types, Raster data and Vector data. Raster data represents data as a grid of cells or pixels. Each of these cells depicts a value corresponding to a particular feature or attribute, like, temperature or elevation of ground. Vector data represents data in the form of geometrical shapes like points, lines and polygons. We can use a point to represent a landmark, or a line to represent a road and a building or a park with polygons. The system developed in this thesis currently only handles Vector data. Spatial data is analyzed and visualized in Geographic Information Systems (GIS) and is vital for them. The first-generation GIS were proprietary systems and only dealt with spatial data [19]. All

First-Generation GIS:



Second-Generation GIS:



Third-Generation GIS:



Figure 2.4: Evolution of GIS [19]

GIS data and indexes were stored in flat files in the system and the vendor's GIS used to run as an application on top of the stored files. They had several disadvantages, they only dealt with spatial data and required "glue code" if a query had to interact with spatial and other business-related data. Moreover, they used proprietary data models and had to manage the security, integrity, backup and recovery of their data in the file system. The second-generation GIS took advantage of some of the built-in capabilities of the existing RDBMS systems. It allowed GIS to save spatial data into RDBMS as Binary Large Objects (BLOBs) and transfer some of the data management burden to the RDBMS. It enabled support for complex queries involving both spatial and other business-related data. Second-generation GIS were still not able to directly integrate with the underlying RDBMS as GIS control the RDBMS schema and use the proprietary API to access and analyze the spatial data. GIS maintained its own indexes and could not use the performance benefits offered by

the RDBMS system. It also added additional responsibility of maintaining both the GIS and RDBMS system to the database administrator (DBA). Object Relational Database Management Systems (ORDBMS) paved the way for third-generation GIS. It treated spatial data as first class database objects and changed the GIS from being a GIS-centric to DBMS-centric. It implemented the spatial data and spatial indexes as abstract data types, user-defined functions and user-defined indexes and treated them as another datatype within the ORDBMS. This ensured that the performance benefits offered to ORDBMS also applied to the spatial data and removed the additional burden introduced by second-generation GIS on the DBA. Fig 2.4 shows the evolution of GIS, and shows various generations of GIS.

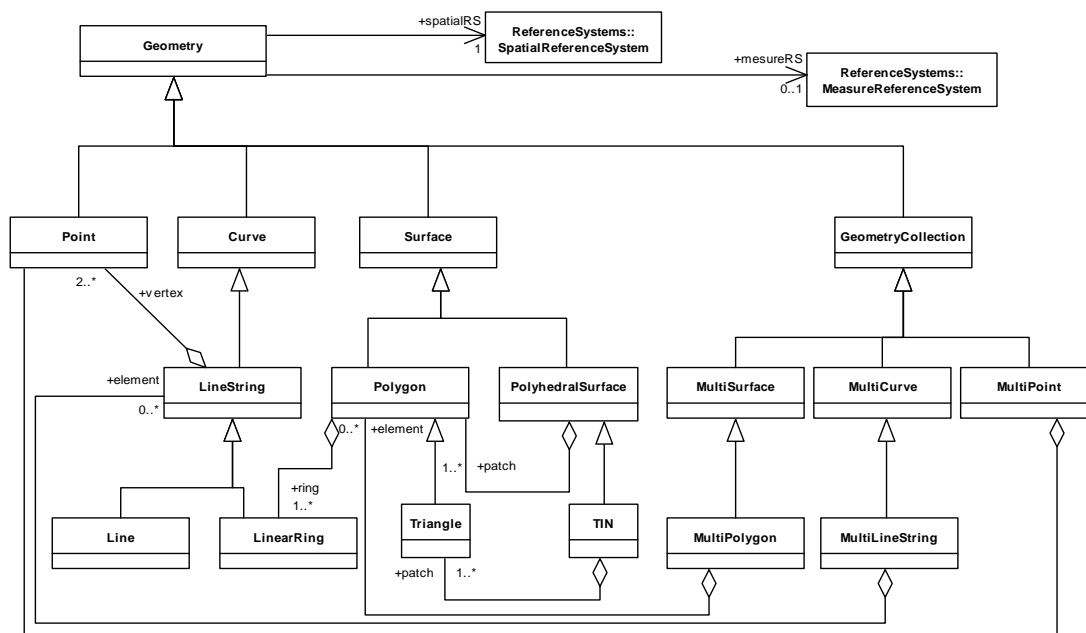


Figure 2.5: Spatial data types [53]

2.1.6 Spatial Data Types

A regular database supports numbers, strings and dates data types. A spatial database includes all of these data types and adds an additional type, called Spatial data type to represent vector spatial data. Open Geospatial Consortium (OGC) has

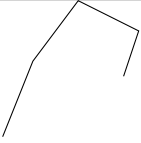
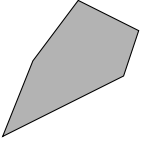
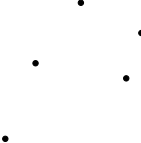
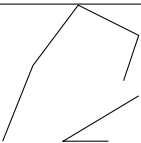
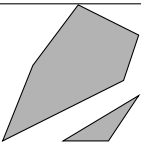
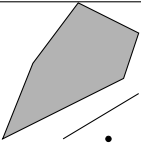
Geometry Type	WKT representation	Geometry
Point	Point(10,10)	•
LineString	LineString(10 10, 20 35, 35 55, 55 45, 50 30)	
Polygon	Polygon((10 10, 20 35, 35 55, 55 45, 50 30, 10 10))	
MultiPoint	MultiPoint((10 10),(20 35),(35 55), (55 45),(50 30))	
MultiLineString	MultiLineString((10 10, 20 35, 35 55, 55 45, 55 30),(45 30, 30 10))	
MultiPolygon	MultiPolygon(((10 10, 20 35, 35 55, 55 45, 55 30, 10 10)), ((55 25, 30 10, 45 10))	
Geometry Collection	GeometeryCollection(Point(45, 10), Linestring(55 25, 30, 10), Polygon(10 10, 20 35, 35 55, 55 45, 55 30, 10 10))	

Table 2.1: Geometries represented in WKT

standardized the geometrical storage models that are used to represent 2-D geometric objects like Point, Line, Polygon, Multi-Point, Multi-Line and many more. This standard is called Simple Feature Access [53] and Fig 2.5 depicts how geometrical shapes are categorised in this standard. Vector spatial data can be represented in the following ways:

- **Well-Know Text (WKT):** This form uses a standard syntax to represent geometrical objects. This way of representing spatial data is human-readable and easy-to-use. WKT can represent geometries in 0-, 1- and 2- dimensional objects. Geometries represented by WKT usually can have x , y , z dimensions depending on the shape they are representing, and m coordinates where, m is a measurement. Table 2.1 shows a few geometries represented in WKT.
- **Well-Know Binary (WKB):** Binary format for representing the WKT and is designed for efficient storage and processing.
- **GeoJSON:** This format use JavaScript Object Notation (JSON) object to represent geometrical objects and its associated attributes. It uses a geographic coordinate reference system, World Geodetic System 1984, and units of decimal degrees.

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates": [102.0,
0.5]},
      "properties": {"name": "Point example"}
    },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
```

```

        [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0,
1.0]
    ]
  },
  "properties": {
    "name": "LineString example"
  }
},
{ "type": "Feature",
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
        [100.0, 1.0], [100.0, 0.0] ]
    ]
  },
  "properties": {
    "name": "Polygon example"
  }
}
]
}

```

Listing 2.1: GeoJSON representing a point and a line and a polygon

- **Shapefile:** This format is a popular vector data format for GIS software and stores location, shape, and attributes of geographic features. It is developed and maintained by Environmental Systems Research Institute (ESRI). A shapefile contains collection of files with different extensions and must have at least a Shapefile shape format (.shp) file, Shapefile index format(.shx) file and

Shapefile attribute format (.dbf) files. A .shp file contains the geometry data and the geometry object is stored as a set of vector coordinates in binary. It also contains the positional index of the geometry objects, and can be used to seek backwards and forwards in the .shp file. The .dbf file contains the attributes related to the geometries.

2.1.7 Bounding boxes and Spatial Index

A bounding box, also known as Minimum Bounding Rectangle (MBR), describes the smallest rectangle that completely encloses a given geometric object and is parallel to the coordinate axes. MBRs are effectively used in spatial indexing and in optimizing spatial queries.

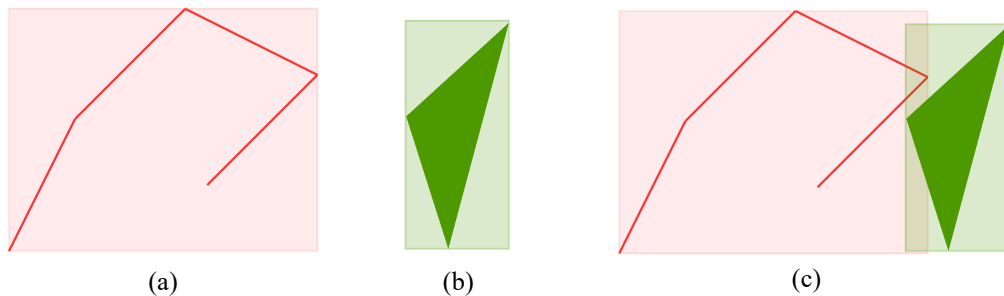


Figure 2.6: (a) MBR of a line, (b) MBR of a polygon, (c) MBRs of line and polygon intersecting each other

Indexing allows working with large datasets and it allows for fast access to subsets of data. For a regular workload (non-spatial), we use B+tree [39, 15, 32] for indexing the dataset. A B+tree partitions the data using the natural sort order of the regular workload, which includes numbers, strings and dates. Determining the natural sort order for all of these data types is straightforward, as each value is either less than, greater than or equal to others. In contrast, the spatial data is very different, and the natural sort order of the spatial objects is complex to determine, as they could be multi-dimensional and for each of the dimension we need to maintain a separate

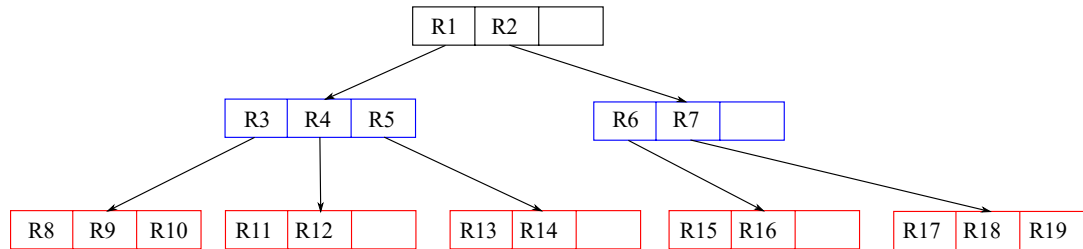
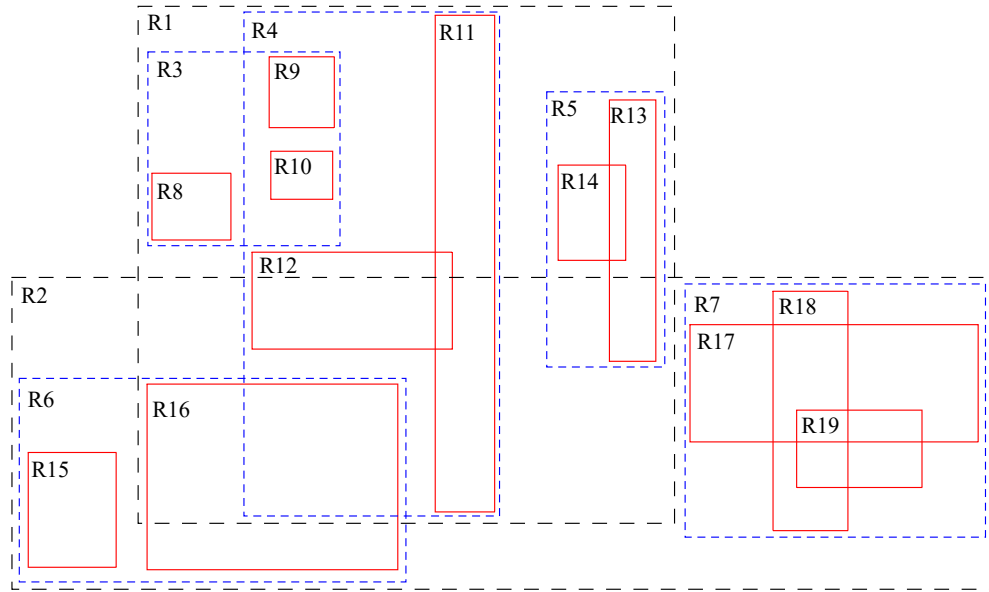


Figure 2.7: R-tree based indexing using MBRs [46]

index.

R-tree [26] is used commonly to index spatial data. It uses the MBR of spatial objects to partition the data and it organizes the data into a hierarchy of rectangles, sub-rectangles, and so on. Figure 2.7, shows how the whole dataset is divided into rectangles, each of which contains other smaller rectangles. An R-tree indexes these rectangles (MBR) and as groups of MBRs of spatial objects into a bigger MBR. In the Figure 2.6(c), to answer questions like “Check if the green triangle touches the red line”, we can use the R-tree index in the following way. We first find the biggest MBR that intersects the MBRs of both of the shapes. If it does not then we can safely say that the shapes do not touch each other, else, we try to find whether the

MBR of the red line touches the MBR of the triangle and if it does then only we evaluate whether the actual shapes touch each other or not. The first phase of using the index is called *Filtering Phase* and the final predicate evaluation phase is called *Refinement Phase*. This two-phase process helps in filtering out many spatial objects and effectively reducing our search space, especially if we are doing spatial joins.

2.1.8 Spatial Functions

For processing spatial data we need a set of functions, called Spatial Functions. These functions are defined and standardized by Open Geospatial Consortium (OGC) in their Simple Feature Access [53] standard. They can be roughly classified into the following categories.

- Conversion: Functions that convert spatial objects to different formats for representing them.
- Geometric: Functions that deal with geometric operations, such as finding the area of a geometric object, finding the centroid.
- Relational: Functions that deal with the relationship between two spatial objects, like if two objects are equal, or if they intersect each other.
- Transformation: Functions that change the structure of a spatial object and change it to create a new spatial object.

Table 2.2 list out some of these spatial functions with their description

2.1.9 Spatial Partitioning

Spatial partitioning is a technique that is used to divide the spatial data into smaller regions called *partitions* or *tiles*. They are vital for the analyzing, processing and querying of spatial data in applications like GIS and spatial databases. It enables

Function Type	Function Name	Description
Conversion	ST_GeomFromText	Converts a WKT to geometry
Conversion	ST_GeomFromWKB	Converts a binary object to geometry
Conversion	ST_GeomFromGeoJSON	Converts a GeoJSON to geometry
Geometric	ST_Length	Calculates length of a LineString or MultiLineString
Geometric	ST_Area	Calculates area of a geometry
Geometric	ST_Perimeter	Calculates 2D perimeter of a geometry
Geometric	ST_Distance(A,B)	Calculate the shortest distance between geometry A and B
Relational	ST_Equals(A,B)	Checks if geometry A is identical to geometry B
Relational	ST_Intersects(A,B)	Checks if geometry A and geometry B's boundary or interiors intersects
Relational	ST_Disjoint(A,B)	Checks if geometry A and geometry B are disjoint, i.e. if they have no point in common
Relational	ST_Crosses(A,B)	Checks if geometry A and geometry B's interior intersects with the interior of A at some but not all points
Relational	ST_Overlaps(A,B)	Checks if geometry A and B overlaps. Two geometries overlap if they have the same dimension, their interiors intersect in that dimension and each has at least one point inside the other
Relational	ST_Touches(A,B)	checks if geometry A and B touch their boundaries, but do not intersect in their interior
Relational	ST_Within(A,B)	checks if geometry A is completely within geometry B
Relational	ST_DWithin(A,B, x)	checks if geometry A is within x meters of geometry B

Table 2.2: Spatial Functions

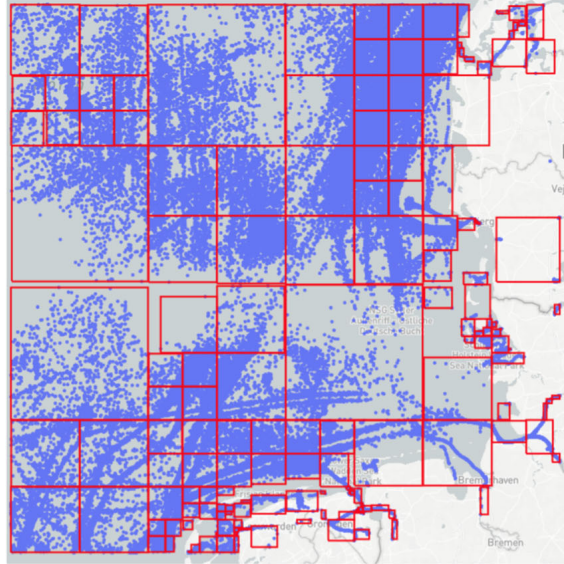


Figure 2.8: Quadtree-based spatial partitioning [21]

efficient processing of large data by dividing them into tiles and processing each tile or subset of tiles at once. Moreover, it allows for building efficient indexes like the Quadtree-based [22] and Grid-based [45, 51] spatial indexes. Quadtree-based partitioning scheme divides the data into four quadrants recursively. Fig. 2.8 shows a Quadtree partitioned dataset. It considers density and dense areas are subdivided more and helps in reducing *data skew*. Spatial data is characterized by *data skew* and *processing skew*. Data skew is caused by uneven distributions of tuples, and processing skew is caused by variations in computation time in the Refinement Phase due to the difference in object sizes and complexity. Grid-based spatial indexes divide the data uniformly so that each of the tiles has the same dimensions. This does not consider the density of spatial objects, and so it is more prone to data skew. Partitioning the data ensures that all the spatial objects whose MBRs can intersect each other are in the same tile. If any spatial object lies on the tile boundary, then it is replicated in all the tiles whose boundaries it is intersecting.

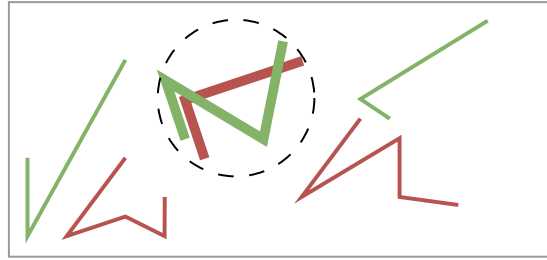


Figure 2.9: Spatial Join

2.1.10 Spatial Joins

In spatial databases, spatial joins combine two or more tables based on their spatial relationship, unlike joins in a non-spatial workload where joins are performed on a particular column. Spatial joins can be broadly categorized into the following.

- **Spatial Join:** It finds object pairs from tables satisfying a spatial predicate like `ST_TOUCHES`, `ST_OVERLAPS` and geometric other relational functions listed in Table 2.2. Fig. 2.9 shows an example of the Spatial Join, where join is performed when two lines intersect each other. In a spatial partitioned dataset, when performing Spatial Join, we only need to look at the corresponding tiles from all the tables involved in the join.
- **Spatial Range Join:** It finds object pairs from tables where the objects are within a drawn radius of the query object. `ST_DWITHIN` can be used for this join. Fig. 2.10 shows an example of the Spatial Range Join, where join is performed for all the spatial objects inside the drawn radius. In a spatial partitioned dataset, when performing Spatial Range Join, we need to look at all the tiles, because the radius around the query object could be greater than the tile boundaries and could extend to other tiles.
- **Spatial Distance Join:** It finds object pairs that satisfy a particular distance predicate. `ST_DISTANCE` can be used for this join. Fig. 2.11 shows an example

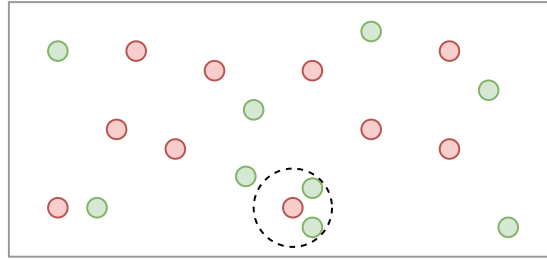


Figure 2.10: Spatial Range Join

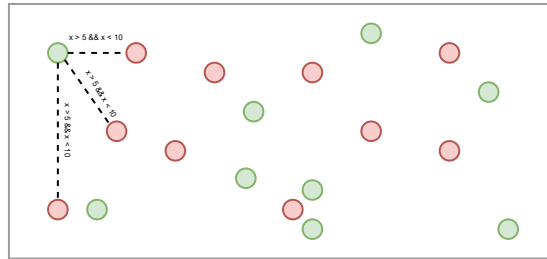


Figure 2.11: Spatial Distance Join

of the Spatial Distance Join, where join is performed when the distance between two spatial objects is less than 10 units but greater than 5 units. In a spatial partitioned dataset, when performing Spatial Distance Join, we also need to look at all the tiles, because the distance predicate between two objects could be greater than the tile dimension.

2.1.11 PGAS and UPC++

Partitioned global address space (PGAS) [5] is a parallel programming model for developing high-performance applications on computer clusters. It provides a global address space partitioned among the cluster nodes. One significant aspect of PGAS is its support for one-sided communication, allowing processes or threads to access memory on remote nodes without requiring explicit synchronization with processes on those nodes. One-sided communication improves performance by decoupling pro-

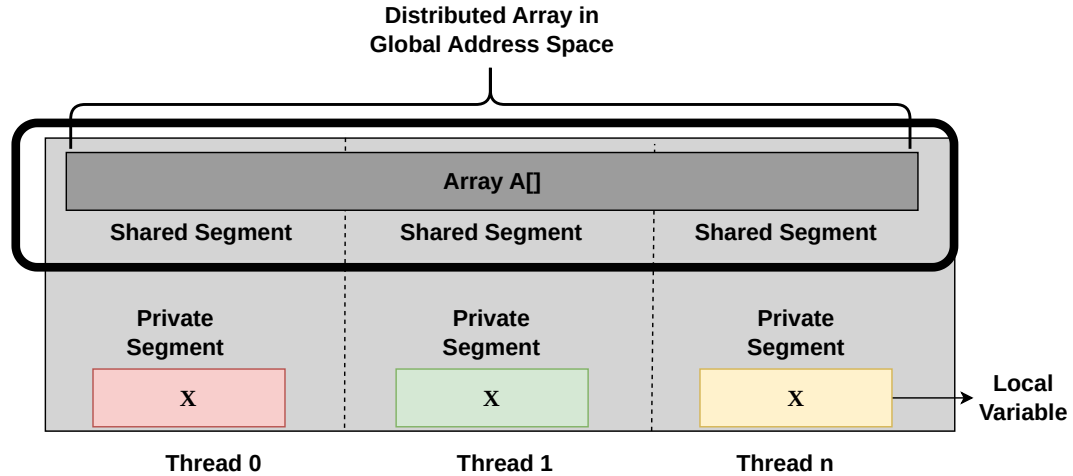


Figure 2.12: PGAS memory organization

cess synchronization from data transfer but requires the developer to think about appropriate synchronizations that will be needed between reads and writes. PGAS programs are based on a single program, multiple data (SPMD) [17] model running on a cluster. At run time, a PGAS program consists of multiple processes executing the same code on different nodes. Each process has a rank, which is the identifier of the node it runs on. The processes can access the global address space partitioned into local address spaces for each process. Local addresses can be accessed directly. Remote addresses belonging to different processes are accessed using API calls, also known as remote procedure calls (RPC). Figure 2.12 illustrates the memory organization of a PGAS system. PGAS contains two segments, a *shared segment* where the data is shared globally and a *private segment* where the data is private to each thread. The array $A[]$ is a shared distributed data structure. Each process can access it. GASNet [12], a PGAS network protocol, provides both synchronous and asynchronous versions of reads and writes.

UPC++ [63, 1] is a C++ library that supports the Partitioned Global Address Space (PGAS) programming model, and is designed to inter-operate smoothly and efficiently with MPI [58], OpenMP [16], CUDA [34] and other HPC frameworks. It

leverages GASNet-EX to deliver low-overhead, fine-grained communication, including Remote Memory Access (RMA) and Remote Procedure Call (RPC). UPC++ exposes a PGAS memory model, including one-sided communication (RMA and RPC).

2.1.12 Distributed Spatial Query Processing

The volume of spatial data is increasing rapidly due to advances in GPS-enabled devices, social media, remote sensing and satellite imaging, oceanographic data and many more. Processing this huge volume of data can be beyond the capabilities of a single machine. A distributed system can offer an effective solution to tackle large volumes of data, which can introduce scalability and enhance performance through parallel processing. Distributed spatial query processing involves analyzing the spatial data to extract the desired information with the help of a declarative language like SQL, across multiple nodes in a distributed system.

2.2 Related Work

The topic of scalable processing of spatial data has a long history. Paradise [44] is one of the earliest parallel databases developed for scalable, geo-spatial data storage and retrieval. Paradise is based on an object-relational data model that supports an extended version of SQL for spatial queries. The PostGIS spatial extension of PostgreSQL played a role in popularizing spatial SQL query processing.

The advent of MapReduce and an open-source implementation Hadoop [28], resulted in researchers introducing Hadoop-based spatial data systems, including SpatialHadoop [20] and Hadoop-GIS [3]. To take advantage of the aggregate memory pool of a compute cluster, Spark [62] has been introduced, which offers significantly better performance than Hadoop. Subsequently, several Spark-based spatial sys-

tems have been proposed. They include Apache Sedona [7] (previously, GeoSpark), SpatialSpark [60], Simba [59], Magellan [35], STARK [29], and LocationSpark [55]. As noted by Yu et al. [61], among the Spark-based systems only Sedona supports standard spatial extensions to SQL [9]. Another limitation of some of these systems is that they support only point objects or MBR-based filter-oriented spatial query processing. Similar findings were reported by a study [43].

The focus of the above-mentioned systems is long-running spatial queries involving complex operations, such as spatial join. A few spatial data systems have been proposed that aim at frequent updates and relatively short-running queries on points, such as spatial range and kNN queries. MD-HBase [41] is one of the earliest systems in this category, followed by systems such as DISTIL⁺ [37]. These systems do not support SQL.

Data and processing skew are significant issues with spatial query processing. Niharika [48], introduces a spatial partitioning scheme and several load-balancing algorithms for handling data skew. ARF [4] presents a one-dimensional range query filter approach for disk-resident data. AQWA [6] is designed to address spatial computation skew in MapReduce-based systems. LocationSpark provides an optimized query plan for managing spatial query skew with spatial range join and kNN join operators. The SPINOJA [49] approach focuses on processing skew caused by variation in computation time in the refinement step due to object (geometry) size. This involves a skew-resistant partitioning approach involving splitting each object along tile boundaries.

2.3 Summary

This chapter discusses query execution techniques like the Volcano model and its disadvantages and then talks about how researchers are looking into query compilation-

based query execution techniques, such as the Push-based model and how they overcome the disadvantages of the Volcano model. This chapter also mentions the work related to spatial query processing and how distributed systems based on Hadoop, Spark and other RDBMS are processing spatial data. Further, this chapter introduces the concepts, tools and frameworks that are going to be used to pursue RQ1.

Chapter 3

Design and Implementation

This chapter discusses the design and implementation of the spatial query processing engine of CasaDB [14, 57] developed in this thesis. It presents a detailed description of the architecture of CasaDB. It also describes two novel approaches for Spatial Joins, Monolithic Tile-based Morsel Driven parallelism (MTMP) and Granular Tile-based Morsel Driven parallelism (GTMP) used in the spatial query engine. It also discusses the different type of Spatial Joins and how MTMP and GTMP is used with them.

3.1 CasaDB

CasaDB is an in-memory, relational database named after our research group Center for Advanced Studies-Atlantic (CASA). CasaDB generates query-specific and data-centric C++ Code which is then compiled to machine code. Recent work [57] in CasaDB, introduced a new intermediate representation code (IR), called CasaIR, which effectively reduced the compilation time of generated C++ code significantly. This thesis extends CasaDB to support efficient spatial query processing for single node and also for distributed computing by generating data-centric C++/UPC++ code.

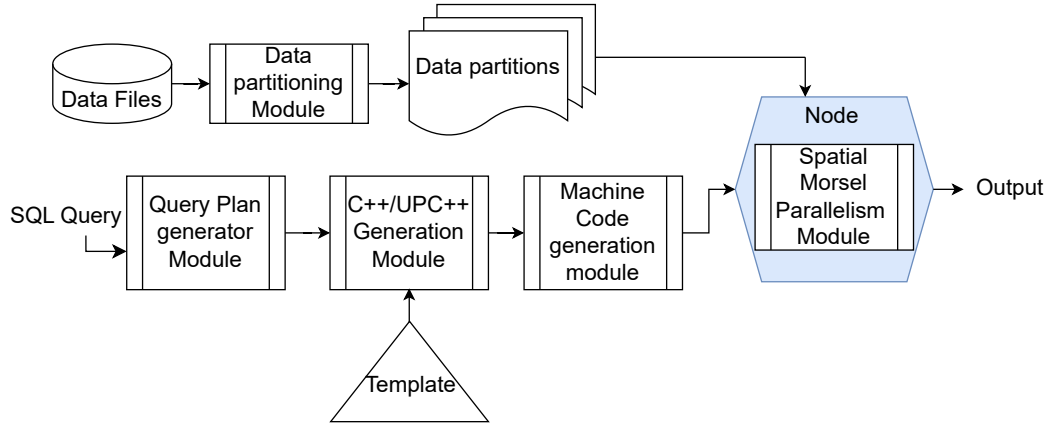


Figure 3.1: CasaDB - Single Node Architecture

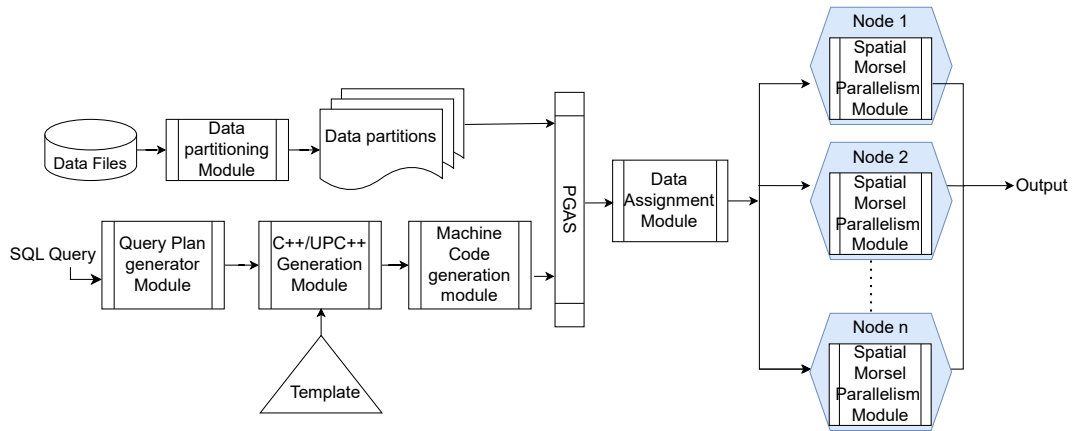


Figure 3.2: CasaDB - Distributed Architecture

3.2 Architecture

The high-level architecture of CasaDB - Single Node is shown in Figure 3.1, and the CasaDB - Distributed architecture is shown in Figure 3.2. The query compilation engine of CasaDB consists of several modules: Query Plan Generator, Code Generation module, Machine Code Generation module, Data Partitioning module, and Data Assignment module. The Data Partitioning module and Data Assignment module are only available in CasaDB Distributed. In the case of Single Node, there is only one Spatial Morsel-driven Parallelism module and in the Distributed architecture, each node has an independent Spatial Morsel-driven Parallelism module. This module

supports spatial morsel-driven parallelism (Section 3.5) in each of the nodes, which helps them to efficiently process each pipeline independently. Finally, the generated C++/UPC++ code is compiled into native code, which runs on the PGAS runtime for Distributed CasaDB, to produce the query output. We will discuss more about these modules in the next few sections.

3.2.1 Query Plan Generator Module

A SQL query is passed through various phases in query execution. In the query parsing phase, the SQL query is passed through the Lexer to divide the query text into tokens, which are then fed to the Syntax Analyzer to make sure that the query adheres to the SQL grammar. In the next phase, it is passed through the Query Optimizer, which first takes the tokens from the Syntax Analyzer and transforms them into a Logical Plan using logical operators. Then the Logical Plan is transformed into a Physical Plan after choosing the best plan based on a cost model. All these phases are taken care by Apache Calcite [11] and is part of the Query Plan Generator module. We have implemented a wrapper on top of Apache Calcite, which converts the Physical plan generated by Calcite to a CasaDB-specific Physical plan. This CasaDB-specific Physical plan helps us to easily interact with the code generation module.

3.2.2 Code Generation Module

We use the Push-based model [40] to generate the C++ / UPC++ code for an input SQL query. This model introduces two separate functions, *produce()* and *consume()*, to generate the code, as shown in Fig. 3.3. The generated code is data-centric and keeps the tuples in the CPU registers as long as possible, and only materializes the tuples at pipeline breakers. It pushes the data towards the operators, which results in more efficient code and data locality. One thing to note about this technique is that

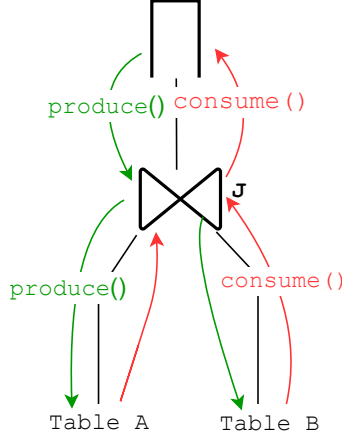


Figure 3.3: Produce-Consume Model

the *produce()* and *consume()* function calls on the operators are not present in the final generated C++/UPC++ code but instead we use these functions to generate parts of the final generated code. The output of the Query Plan Generator module is then ingested by the Code Generation module, which generates query-specific, data-centric C++ code for CasaDB-Single Node and UPC++ code for CasaDB-Distributed. A physical query plan generated by CasaDB for a *Spatial Join query* is shown in Figure 3.4(b). CasaDB has a set of physical operators and they all correspond to Relational Algebra (RA) operators. Each of these physical operators has *produce()* and *consume()* defined on them. Calling *produce()* on any of these operators will call *produce()* on its child and so on, and calling *consume()* will emit the operator-specific code. The *CTableScan* operator listed in Table 3.1, emits code related to reading tuples from a table, whereas *CIndexScan* is responsible for emitting code related to index scan. The operator *CSpatialJoin* emits code that performs the actual spatial join operation and spatial predicate evaluation. The operator *CApply* performs the projection operation and emits the code related to that. Finally, the *CStore* operator is responsible for materializing the results and it emits code related to that. All of these operators have there pre-defined C++/UPC++ templates,

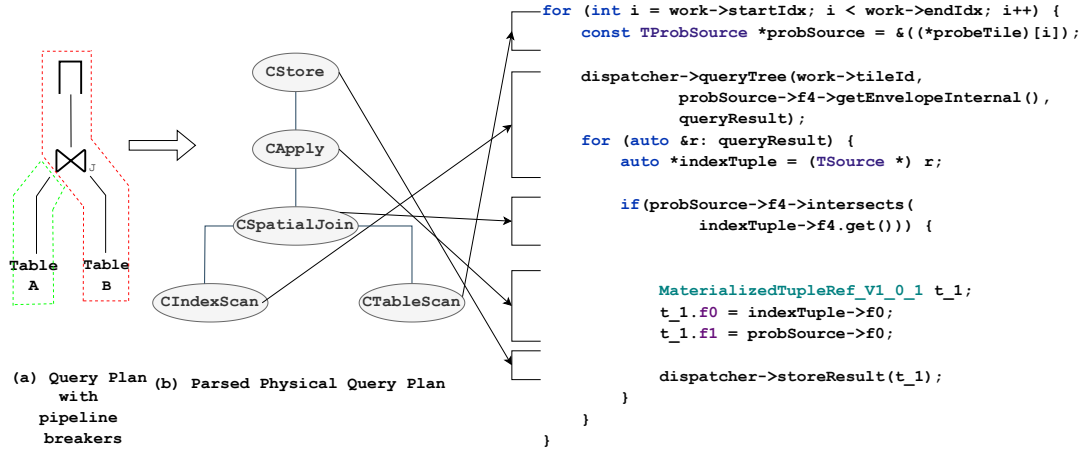


Figure 3.4: Code generation

which are used to generate data-centric code. As can be seen in Figure 3.4(b), the root of the tree *CStore* calls *produce()* on its child *CApply*, and the *CApply* calls *produce()* on its child, and this continues until we reach the leaf node. When *CTableScan* determines that it has no child to call *produce* on, it will call *consume()* on its parent with the emitted code, which is highlighted in the Figure 3.4. Then the parent of *CTableScan* will call *consume()* on its parent, and it will continue until we reach the root node. There is a *CompilerState* object in the Code generation module that knows where to place the emitted block of code in the final generated code.

Calcite Operators	CasaDB Operators	Description
EnumerableTableScan	CTableScan, CIndexScan	Read tuples from a table or index
EnumerableNestedLoopJoin	CSpatialJoin	Performs spatial join
EnumerableProject	CApply	Projection operator
-	CStore	Stores the final result in a data structure

Table 3.1: Calcite and CasaDB operators

3.2.3 Machine Code Generation Module

The Machine Code Generation module takes the code generated from the Code Generation module and then compiles it using g++ in CasaDB - Single Node and for CasaDB - Distributed it uses UPCXX to compile. Then the compiled code is sent to other nodes in the cluster and they are processed by the UPCXX runtime.

3.2.4 Data Partitioning Module

In a typical relational distributed query execution system, regular (non-spatial) data is statically partitioned using Range or Hash partitioning so that each node in the cluster gets equal relevant data. In a spatial partitioned dataset, Spatial Joins use the spatial boundary of spatial partitions or tiles to perform computation. So, such type of partitioning scheme will not work with spatial data. We also need to efficiently deal with the data skew, which is caused by the variation in the number of tuples in each partition and also need to deal with processing skew, which is due to the complexity and difference in the size of spatial objects. We use a spatial partitioning scheme that tries to create balanced spatial partitions and to deal with the inherent skew in spatial data. This is based on a recursive partitioning of the spatial domain, resembling Quadtree [22] partitioning. In each round of the algorithm, our partitioning approach finds the tile with the highest number of objects and partitions it into four quadrants of equal size. This process continues until the total number of generated tiles reaches a threshold. This threshold can be configured to generate partitions with a configurable number of tiles, that is specified by a parameter. The results of the performance evaluation of our runtime with the parameter set to 512, 1024, 2048 and 4096 are available in Chapter 4.

3.2.5 Data Assignment Module

This module is only present in CasaDB - Distributed. Once the data is partitioned, it is the responsibility of the Data Assignment module to assign data partitions to the nodes in the cluster. It makes sure that nodes read only the data they will be processing. Section 3.3 discusses how data partitions are assigned to the nodes in detail.

3.2.6 Spatial Morsel Parallelism Module

The Machine Code Generator module produces optimized machine code, which is then sent to all of the nodes in the cluster. Each of these nodes supports an independent Spatial Morsel-driven Parallelism Module. This module supports two novel parallelism approaches Monolithic Tile-based Morsel Parallelism (MTMP) and Granular Tile-based Morsel Parallelism (GTMP) in each of the nodes, which helps them to efficiently process each pipeline independently. Section 3.5 discusses both MTMP and GTMP in detail.

3.3 Tile Assignment

For CasaDB - Distributed, after partitioning the data, we need to make sure that we only read data that is required by the node. We achieve this by range-partitioning the number of tiles and then assigning a range of tiles to each of the nodes. This ensures that each node only deals with the tiles it is assigned. This assignment is handled by the Data Assignment Module. Each node is assigned less than or equal to nt tiles. nt is defined as:

$$nt \leq \left\lceil \frac{\text{number of tiles}}{\text{number of nodes}} \right\rceil \quad (3.1)$$

This tile distribution plays a major role in the final join operation in the Spatial Morsel-driven Parallelism module. When a tuple from the probe table is processed, we only examine the tiles it belongs to within the indexed table, eliminating the need to search through irrelevant tiles. This significantly reduces our search space.

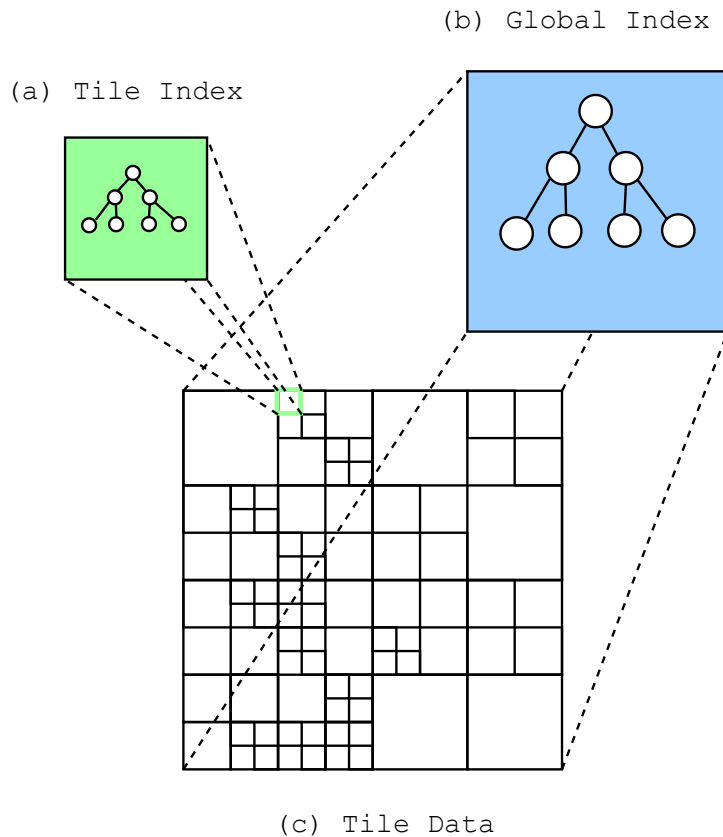


Figure 3.5: Index organization

3.4 Index Organization

Indexing can speed up the join operation. Ideally, we build the index on the smaller table and then use the same index on all of the nodes, but in our system index

creation depends on the type of spatial join (section 2.1.10) being performed. We formally define two different types of indexes. In the first one, called *Tile Index*, for each tile, we build a separate R-tree index and for the *Global Index*, we create an index on the whole table. We use the Sort-tile-recursive (STR) packing algorithm [33] for both types of indexes. For *Spatial Join Query*, CasaDB uses the Tile Index to perform the join, where in the Filtering Phase, objects belonging to the same tile are processed together. In the case of *Distance Join Query*, employing Tile Index or Global Index is not useful, because we need to calculate the distance between each object pair, and then check if they satisfy the specified distance or not. For *Spatial Range Join Query*, Tile Index does not prove useful, because we could be checking whether an object is within the radius of a query object located in some other tile. So for this case, we use the Global Index. Our system does not support *kNN Join Query* yet, but it could also benefit from the tile-wise index to find top-K nearest neighbours satisfying a spatial predicate. Building an index dynamically enables CasaDB to fully utilize the degree of parallelism (DoP) through our spatial Morsel-driven parallelism approach.

3.5 Spatial Morsel-Driven Parallelism

The original morsel-driven parallelism [32] works well for regular workloads, but for tile-based/partitioned spatial workloads, using this approach presents additional challenges. Especially when it comes to “defining” a morsel and also dealing with inherent processing skew in spatial workloads. We propose two different algorithms based on morsel-driven parallelism for parallel spatial query processing: *Monolithic Tile-based Morsel-driven Parallelism (MTMP)* and *Granular Tile-based Morsel-driven Parallelism (GTMP)*. In both of these algorithms, each tile is treated as a morsel, but in the *GTMP* we introduce granularity at a sub-morsel level, these are called “gran-

ules” and instead of processing a morsel, we process its granules. The scheduling of these work units involves a dispatcher assigning a morsel (in MTMP) or granule (in GTMP) to a worker thread. Once a thread is done processing its assigned entity (morsel or granule), it receives the next one until all of the entities are processed. In our experimental evaluation, we found that sometimes MTMP is affected by data skew, especially when both of the tiles in a spatial join operation have a lot of tuples (i.e. a large morsel). The worker assigned to this morsel takes a lot of time, while the other workers are done with their morsels and the overall process is waiting for this worker to finish. GTMP handles this scenario by employing job-decomposition technique. It breaks each morsel into granules and processes them efficiently such that the other waiting workers can take up the granules from the big morsel for processing. By doing so, we ensure that even if a thread is busy processing a morsel, which may contain complex geometric shapes, and takes much longer time to process compared to other morsels, other threads can process other morsels in the meantime. In this way, we ensure that no thread is in an idle state and they are always busy doing useful work.

3.5.1 Monolithic Tile-based Morsel Parallelism (MTMP)

In this algorithm, all the tuples in a tile make one morsel, - essentially we are processing one tile at a time. So, if we have used a partitioning scheme that divides the data into 512 tiles, for example, then the number of morsels will be 512. The number of tuples in a morsel (or the morsel size) in this approach is dynamic, compared to the original morsel-driven approach, where they set the morsel size to a fixed number, for instance, 10,000 tuples. The morsel size in our approach is equal to the number of tuples in a particular tile. So, the morsel size is dynamic but the number of morsels is static. This approach was our first attempt at introducing morsel-driven parallelism in our spatial query engine. It has three components, *MTMPWork*, *MTMPWorker*

and *MTMPDispatcher*. *MTMPWork* defines the work that needs to be done. It contains the morsel, the id of the tile the morsel belongs to and the kind of processing that needs to be done on the morsel, which is defined by *WorkState*. This *WorkState* indicates the *MTMPWorker* the kind of work that needs to be done on the morsel. The work could involve processing the tiles, or waiting for all the other workers to finish their processing when all the morsels are assigned to the workers, or the final state when all the processing is done. *MTMPWorker* does the actual processing of the morsel. As soon as such a *MTMPWorker* is spawned it asks for *MTMPWork*, and based on the *WorkState* of the *MTMPWork*, it performs the actual work. A worker is alive as long as there is some work to process. As can be seen in Algorithm 1, *MTMPDispatcher* is responsible for managing the *MTMPWorker* and assigning the *MTMPWork* to them. It divides the dataset into morsels and finally gives them to the workers to perform the actual work when they ask for it. It also keeps track of the overall state of the actual processing we are doing and stores the final result. One key thing about this approach is that it does not need an index to process *Spatial Join Operations* since the *MTMPWorker* knows which tile it is processing and it has access to the morsel in that particular tile from both of the tables. Fig. 3.6 shows how this works and Algorithm 1 shows how work is assigned to the workers. In Section 4 we will show how this approach works in different kinds of joins.

3.5.2 Granular Tile-based Morsel Parallelism (GTMP)

This algorithm introduces the concept of a *granule*, which is the basic unit of processing. In MTMP, the basic unit of processing is a morsel. In GTMP, a morsel is composed of granules. The number of granules depends on the morsel size in the corresponding tile. So, the number of granules can be a dynamic entity, in the same way the morsel size can be dynamic in MTMP. GTMP uses index for different join

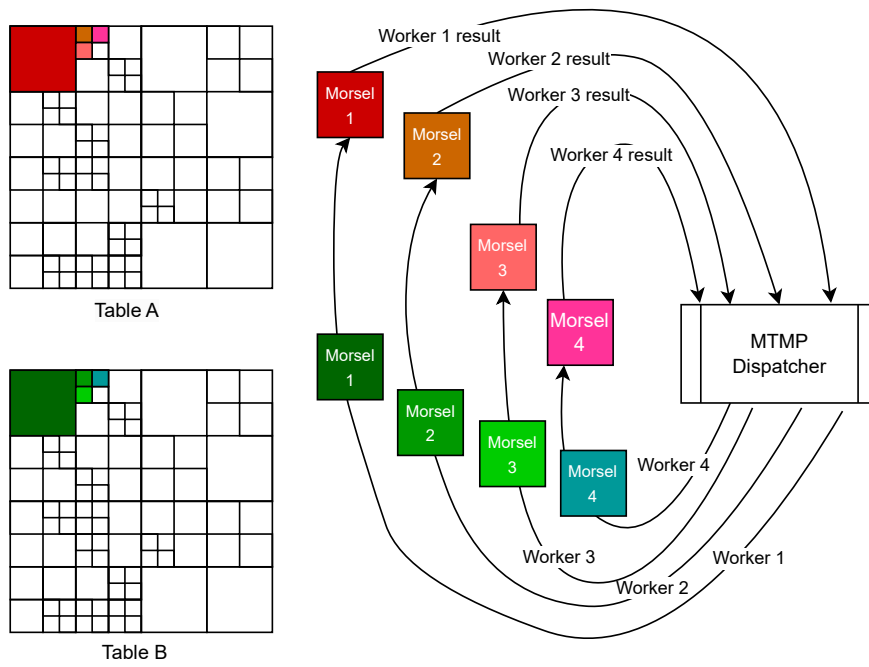


Figure 3.6: Monolithic Tile-based Morsel Parallelism

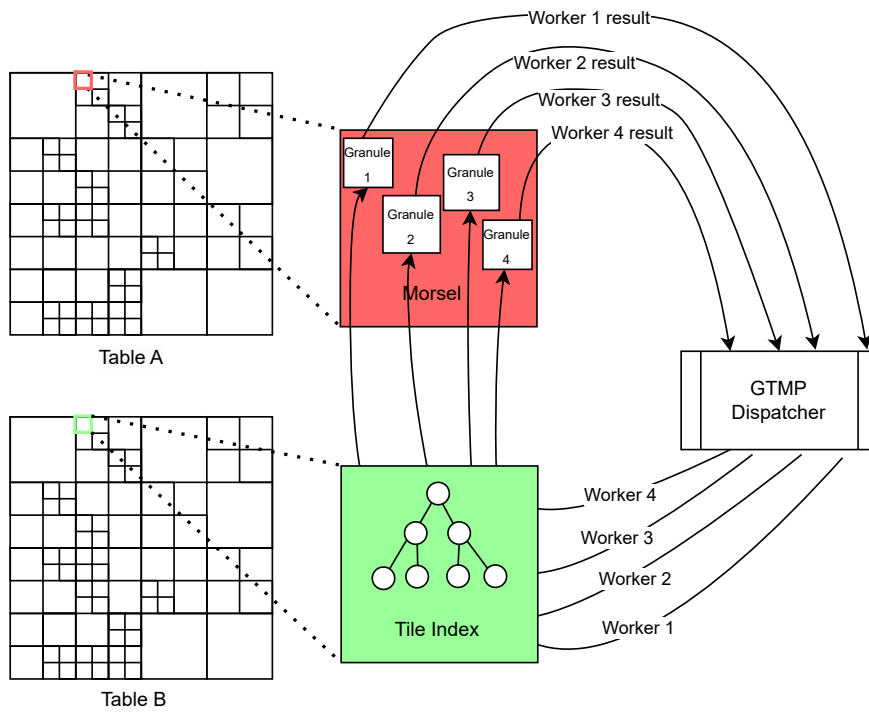


Figure 3.7: Granular Tile-based Morsel Parallelism

Algorithm 1: Pseudo-Code for getWork() MTMPDispatcher

Input: dispatcherState, tileIdQueue
Output: MTMPWork

```
1 if dispatcherState == probingMorsels then
2   | if tileIdQueue.isEmpty() then
3   |   | dispatcherState = doneProbingMorsel;
4   |   | return MTMPWork with JobState::waitingProbe;
5   | end
6   | currentTileId, gets current processing tile from tileIdQueue.front();
7   | probeMorsel, get morsel for currentTileId;
8   | assign probeMorsel to MTMPWork with JobState::probe and also
   |   | include the currentTileId;
9   | return MTMPWork
10 else if dispatcherState == doneProbingMorsels then
11   | if all workers are done probing then
12   |   | return MTMPWork with JobState::done;
13   | else
14   |   | return MTMPWork with JobState::waitingProbe;
15   | end
16 else
17   | return MTMPWork with JobState::done;
18 end
```

Algorithm 2: GTPMWorker/MTMPWorker

```
1 isAlive = true;
2 while isAlive do
3   | get next available Work from Dispatcher, by calling getWork() ;
4   | get WorkState from Work;
5   | if WorkState == build then
6   |   | noOp;
7   | else if WorkState == waitingBuild then
8   |   | noOp;
9   | else if WorkState == probe then
10  |   | Iterate the morsel/granule and do the processing according to the
   |   | spatial predicate. Store result in global data structure using
   |   | Dispatcher;
11  | else if WorkState == waitingProbe then
12  |   | Wait for all of the workers to finish the probing
13  | else
14  |   | isAlive = false
15  | end
16 end
```

Algorithm 3: Pseudo-Code for `getWork()` GTMPDispatcher

Input: `dispatcherState`, `tileIdQueue`, `granuleSize`,
`granuleStartIndex`, `granuleEndIndex`

Output: GTMPWork

```
1 if dispatcherState == probingMorsels then
2   if tileIdQueue.isEmpty() then
3     dispatcherState = doneProbingMorsel;
4     return GTMPWork with JobState::waitingProbe;
5   end
6   currentTileId, get current processing tile from tileIdQueue.front();
7   probeMorsel, get morsel for currentTileId;
8   granuleStartIndex = granuleEndIndex;
9   granuleEndIndex = granuleStartIndex + granuleSize;
10  if granuleEndIndex >= probeMorsel.length() then
11    granuleEndIndex = probeMorsel.length();
12    tileIdQueue.pop()
13  end
14  Slice a granule from probeMorsel using granuleStartIndex and
    granuleEndIndex and assign it to GTMPWork with JobState::probe
    and also include the currentTileId;
15  if granuleEndIndex >= probeTuples.length() then
16    granuleStartIndex = granuleEndIndex = 0;
17  end
18  return GTMPWork
19 else if dispatcherState == doneProbingMorsels then
20   if all workers are done probing then
21     return GTMPWork with JobState::done;
22   else
23     return GTMPWork with JobState::waitingProbe;
24   end
25 else
26   return GTMPWork with JobState::done;
27 end
```

operations. It needs an index because a morsel (i.e. a tile) is divided into granules, and each of these granules can be processed by different workers hence, we need a way to process the granules within a morsel with those in the other table in the same morsel. For *Spatial Join*, GTMP uses *Tile Index*, and for *Spatial Range Join* it uses the *Global Index* on the table because the query object could be present in some other tiles.

GTMPWork component, contains the tile id, granule and the *WorkState*. The worker here is called *GTMPWorker* and its function is different from its counterpart in the MTMP, as it uses index (Global or Tile Index Fig. 3.5). The dispatcher in GTMP is called *GTMPDispatcher* and it is responsible for handling the *GTMPWorker* and creating granules from each morsel and assigning them to the workers. Algorithm 3 shows how work is assigned to the workers and Figure 3.7 provides a holistic view of this approach. We show how this algorithm works with different kinds of join categories in section 3.6.

3.5.3 Distributed MTMP/GTMP

Each node in the cluster has its own MTMP or GTMP Dispatcher and Worker. The Dispatcher spawns as many workers as the machine-dependent threads are supported in a node. The main node in the cluster runs the Data Assignment module (Figure 3.2) and is responsible for dividing the tiles equally and assigning them to all the nodes including itself. Once each of the nodes loads its needed data, it processes the data in morsel/granule-driven fashion independently. Finally, when the Dispatchers of all the nodes complete their tasks, the main node aggregates the results and produces the final output.

3.6 Query Processing

In the following sections we discuss how Monolithic Tile-based Morsel-driven Parallelism (MTMP) and Granular Tile-based Morsel-driven Parallelism (GTMP) support different types of Spatial Join queries.

3.6.1 Spatial Join processing

Spatial relational functions (Table 2.2) like `ST_TOUCHES`, `ST_OVERLAPS` can be used to perform Spatial Join as discussed in section 2.1.10. In a non-indexed join operation on non-partitioned table, there are two nested loops, where the loops iterate over the two tables involved in the join. We compare the tuples from the outer (loop) table with each of the tuples in the inner (loop) table and then do spatial predicate evaluation (Refinement Phase) on both of the tuples. If the pair of tuples successfully passes the predicate evaluation, we add it to the final result. In the case of an index-based join operation, we have a pre-built index on the smaller table (less number of tuples). The outer loop is to iterate over the bigger table and then inside that loop, we do index lookup using the MBR of the outer-loop tuple (Filter Phase). We iterate over the objects returned by the index lookup and perform the Refinement to get the final output.

MTMP and GTMP both deal with spatial partitioned datasets. In the case of MTMP, each tile corresponds to a morsel, we just need to perform the Refinement Phase on the morsels from both of the tables belonging to the same tile. The Filter Phase is implicitly done by using the tiles, so using an index is not necessary. We can directly proceed to the Refinement Phase. *MTMPWorker* receives a tile id, and its associated morsels from both the tables and then it uses the non-indexed nested loop join approach to do the final predicate evaluation (Refinement). The main advantage of this algorithm is that it achieves faster Spatial Join processing without using an

index.

In GTMP, morsels are divided into granules and these granules are then assigned to the *GTMPWorker*. *GTMPWorker* worker could be processing a granule in a morsel, while another *GTMPWorker* could be processing another granule of the same morsel. Hence, if we do predicate evaluation on the granules of both the tables then we will get incomplete (or partial) results, as we are not comparing against the entire morsel. So, to get complete results, we pre-build the Tile Index and then perform the Filter Phase with the granules using the index. After the Filter step, we use the result of the index lookup and do the predicate evaluation between granules and the index query result to get the final output. MTMP and GTMP show similar performance in most of the cases, but GTMP outperforms MTMP whenever there is a significant data skew in morsels.

3.6.2 Spatial Range Join processing

As per the discussion in section 2.1.10 `ST_DWITHIN(geomA, geomB, 1)` can be used to implement queries involving radius. This spatial predicate checks if `geomA` is within 1 unit distance of `geomB`.

We can use the global index in the Filter Phase and can make our query execute faster by eliminating a lot of tuples. We can not use the Tile Index in this scenario because the query object and the object from the table could be outside of the tile. In MTMP, we create morsels from the query object table and we use the pre-built Global Index on the other table. We then create a buffer object on the query object using the `ST_BUFFER` and the radius mentioned in the `ST_DWITHIN` and then use the buffered object to query the global index. The objects returned by the index undergo actual predicate evaluation to obtain the final result. By using the global index, we can effectively filter a lot of tuples from the other table.

GTMP is very similar to MTMP in this join operation. It also uses the pre-built

global index and creates the granules from the query object morsels. It uses the same approach of creating buffer objects and then querying the buffer objects against the index to filter out tuples. Finally, the tuples from the index and the query granules undergo predicate evaluation to produce the final output.

3.6.3 Spatial Distance Join processing

As per the discussion in section 2.1.10 `ST_DISTANCE(geomA, geomB)` can be used to implement Spatial Distance Join. There is no easy way to use an index in this query and we have to find the distance between each tuple pair from both tables to check if it satisfies the distance condition.

In MTMP, we implement this join by replicating the smaller table across all of the nodes. The *MTMPWorker* receives the whole small table and the morsel it needs to process. It then uses the Nested Loop Join approach to iterate the small table and compare it with the tuples in the morsels from the other table. For each tuple in the small table and in the morsel we do the predicate evaluation and then store the final result if it satisfies the predicate.

In GTMP, we also replicate the small table across all of the nodes. The *GTMP-Worker* also receives the whole small table and the granule from the morsel assigned to the worker and uses the same approach of predicate evaluation as that of MTMP to get the final output.

3.7 Summary

This chapter discusses CasaDB and its architecture and how its various modules are used in both CasaDB-Single Node and CasaDB-Distributed. It then discusses how tiles are assigned to each node in CasaDB-Distributed and then talks about two different types of spatial indexes are used in CasaDB. It then describes two different

new algorithms MTMP and GTMP and how they are used in different kinds of Spatial Joins. Adding support for distributed spatial query processing with MTMP and GTMP, pursue RQ1 in this chapter.

Chapter 4

Evaluation

In this section, the experimental setup, datasets used and various queries used to test the system are explained. Finally, our system's performance is compared with PostgreSQL, Apache Sedona and Citus - distributed database based on PostgreSQL.

4.1 Experimental Setup

Two different hardware platforms both running Linux are used for the experiments. To evaluate CasaDB - Single Node's performance, a machine with Intel Xeon Gold 5120 Processor with 112 cores clocked at 2.2 GHz with 128GB main memory is used. CasaDB - Single Node's performance is compared with PostgreSQL v14.8. For evaluating the performance of CasaDB - Distributed, a cluster of 6 machines is used, each having a Dual-Core AMD Opteron Processor 2222 clocked at 3 GHz and 16 GB main memory and its performance is compared with Apache Sedona v1.4.0. JDK 11 is used for our code generator and UPC++ 2023.9.0 to run the distributed code. We compile generated UPC++ code using g++ version 9.4.0.

4.2 Dataset

Two different datasets are used for the experiments: TIGER [56] California and OpenStreetMap [42] (OSM) UK.

4.2.1 TIGER Dataset

TIGER stands for the Topologically Integrated Geographic Encoding and Referencing system and represents the U.S. Census Bureau’s geographic spatial data. It contains spatial information about various geographic features in the United States, such as roads, rivers, railroads, boundaries, landmarks, and more. Table 4.1 provides more details about the tables in the TIGER dataset.

Table Name	Geometric Shape	No. of tuples
Arealm	Polygon	6,708
Areawater	Polygon	40,799
Pointlm	Point	49,837
Edges	Line	4,251,911

Table 4.1: TIGER Dataset

4.2.2 OSM Dataset

OpenStreetMap (OSM) is a free map data provided by the website openstreetmap.org. It is one of the most popular geographic data created by volunteers. Many companies use the OSM map as a reference map, and they put an overlay showing their data. Table 4.2 provides more details about the tables in the OSM dataset.

Table Name	Geometric Shape	No. of tuples
poi_point_uk	Point	907,914
bld_poly_uk	Polygon	12,982,472
lwn_poly_uk	Polygon	2,742,757
rds_line_uk	Line	593,706

Table 4.2: OSM Dataset for UK

4.2.3 Queries

A set of Join queries are evaluated for each of the datasets, which involves spatial predicates, such as `ST_TOUCHES`, `ST_CROSSES`, `ST_OVERLAPS` and `ST_WITHIN`. The Queries for datasets are listed in Table 4.3 and Table 4.4.

Query Name	Query
TIGER_Q1	select * from arealm join areawater on ST_TOUCHES(arealm.geom, areawater.geom)
TIGER_Q2	select * from edges join arealm on ST_CROSSES(edges.geom, arealm.geom)
TIGER_Q3	select * from edges join edges as edges2 on ST_CROSSES(edges.geom, edges2.geom)
TIGER_Q4	select * from edges join areawater on ST_CROSSES(edges.geom, areawater.geom)
TIGER_Q5	select * from areawater join areawater as areawater2 on ST_OVERLAPS(areawater.geom, areawater2.geom)
TIGER_Q6	select * from pointlm join areawater on ST_WITHIN(pointlm.geom, areawater.geom)
TIGER_Q7	select * from pointlm join areawater on ST_DWITHIN(pointlm.geom, areawater.geom, 1)
TIGER_Q8	select * from pointlm join areawater on ST_DISTANCE(pointlm.geom, areawater.geom) >= 1

Table 4.3: TIGER dataset queries

Query Name	Query
OSM_Q1	select * from rds_lin_uk join bld_poly_uk on ST_TOUCHES(rds_lin_uk.geom, bld_poly_uk.geom)
OSM_Q2	select * from rds_lin_uk join lwn_poly_uk on ST_CROSSES(rds_lin_uk.geom, lwn_poly_uk.geom)
OSM_Q3	select * from poi_point_uk join lwn_poly_uk on ST_WITHIN(poi_point_uk.geom, lwn_poly_uk.geom)
OSM_Q4	select * from bld_poly_uk join lwn_poly_uk on ST_OVERLAPS(bld_poly_uk.geom, lwn_poly_uk.geom)

Table 4.4: OSM dataset queries

4.3 Code Generation and Compilation

Code generation times for TIGER queries are listed in Figure 4.4 and are roughly the same for both CasaDB - Single Node and CasaDB - Distributed. The code generation times are not significant compared to the actual spatial query execution times (as can be seen in the next few sections).

Code compilation times for TIGER queries for CasaDB - Single Node and CasaDB - Distributed are shown in Fig 4.2 and Fig 4.3. For CasaDB - Single Node, all of the queries take around 6 to 7 seconds to compile and for CasaDB - Distributed, it took around 17 seconds to compile. This extra overhead is due to the UPC++ libraries.

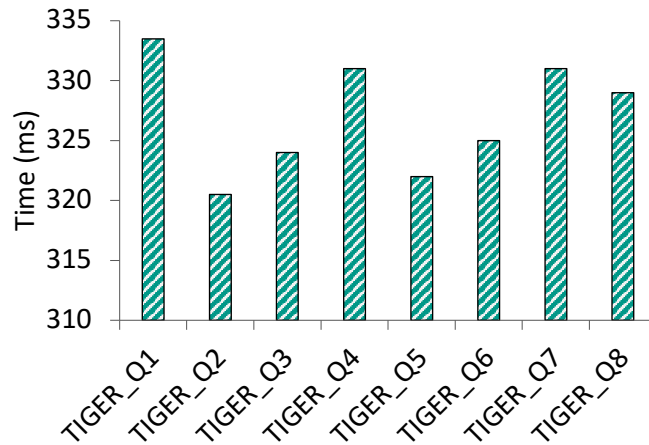


Figure 4.1: Code generation time for TIGER queries

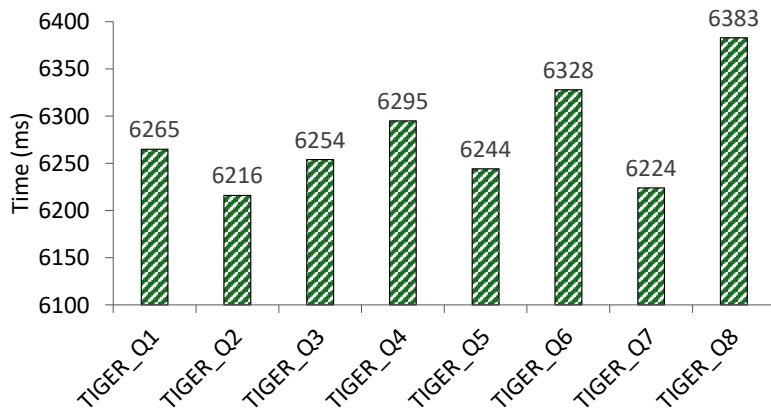


Figure 4.2: Code compilation time for CasaDB - Single Node

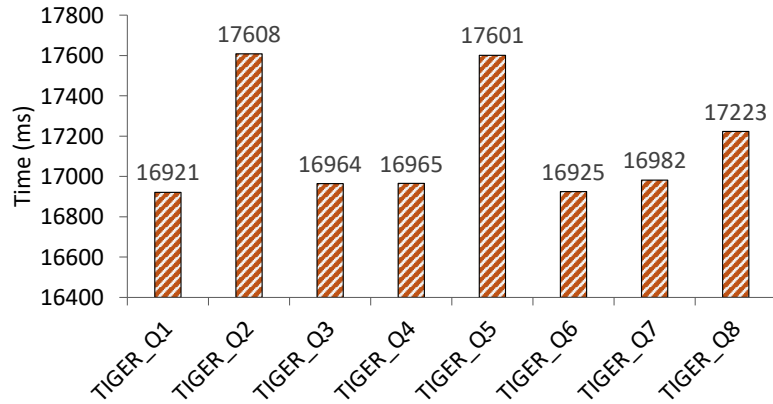


Figure 4.3: Code compilation time for CasaDB - Distributed

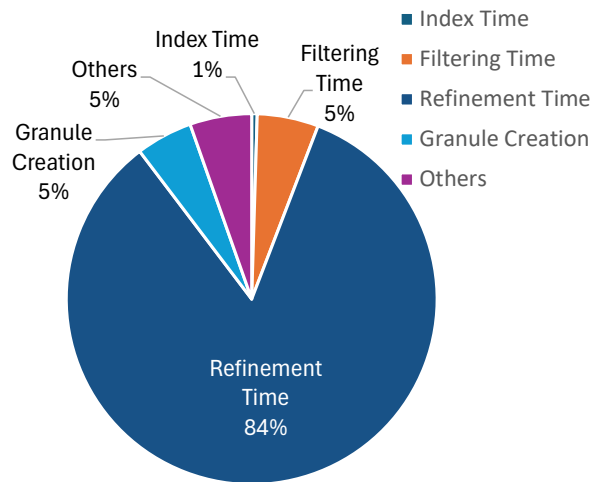


Figure 4.4: Execution time breakdown for TIGER_Q3

4.4 Execution Time Breakdown

Figure 4.4 shows the query execution time breakdown for TIGER_Q3 with GTMP algorithm. The Refinement Phase takes up 84% of the execution time, while filtering phase took only 5% of the time. Index creation took only 1%, and creation of granules for GTMP took 5% of the time. As expected the refinement phase is the most dominant phase in spatial query processing.

4.5 CasaDB - Single Node Evaluation

CasaDB - Single Node, generates C++ code for each of the queries, the execution time for each of the TIGER and OSM queries is measured and compared with PostgreSQL in this section.

4.5.1 Performance analysis of GTMP and MTMP

We evaluated the performance of MTMP and GTMP with varying tile sizes on a single node. Figures 4.5, 4.6, 4.7, 4.8 shows the execution times for various TIGER queries for both MTMP and GTMP algorithms. With increasing the tile sizes the query execution time is decreasing, and it is true for both MTMP and GTMP. GTMP performs significantly better than MTMP when the tile size is 512, whereas with increasing the tile sizes MTMP achieves better performance than GTMP. With a smaller number of tiles (512) there is a possibility of more skewness in data compared to a higher number of tiles (2024, 4096), which means some tiles could have more tuples and some tiles will have less. This shows that GTMP is more skew resilient compared to MTMP. This is more evident in TIGER_Q3 (Fig 4.6), where GTMP with 512 tiles is 3.8x faster than MTMP with 512 tiles. In contrast, MTMP with 4096 tiles is 2.3x faster than GTMP with 4096 tiles and it shows that MTMP has the potential of performing significantly better than GTMP when data has less skew and MTMP's Dispatcher can spawn a greater amount of workers (in this case, 128). Surprisingly, TIGER_Q5 (Fig 4.7) does not follow the trend of reduced execution time on increasing the number of tiles, this behaviour is due to the replication of data while declustering the data, so in case of more number of tiles, there is an increase of number of tuples to process and hence, this trend is observed. Another unusual result is shown in TIGER_Q7 (Fig 4.8), here, the results are also affected by the data replication due to partitioning but one more unusual thing is that MTMP is

always performing better in 512 tiles and 1024 tiles partitioning scheme. Although, the GTMP is more resilient for data skew, in this case, since one of the tables involves point data type, MTMP with a larger number of workers was able to process those point tuples with ease compared to GTMP, where there was some overhead in creating granules and processing them. In TIGER_Q8 (Fig 4.8), which is a Distance Join, GTMP outperformed MTMP.

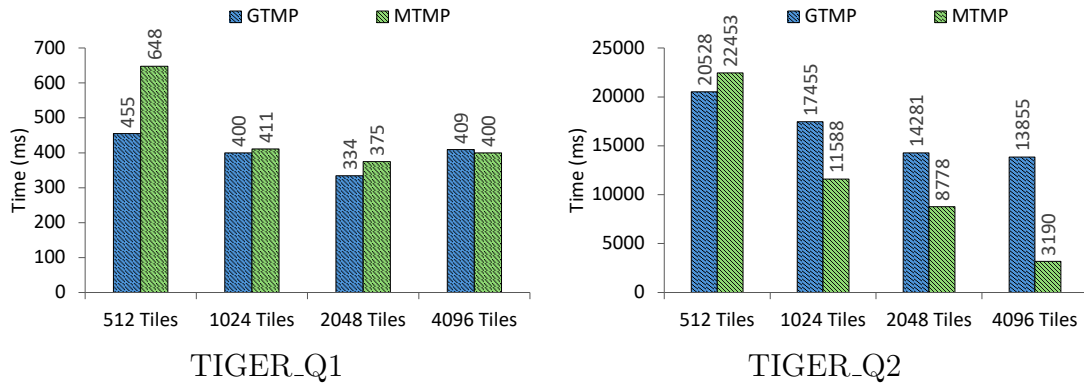


Figure 4.5: MTMP vs. GTMP for TIGER_Q1, TIGER_Q2 with different partition granularity for TIGER dataset

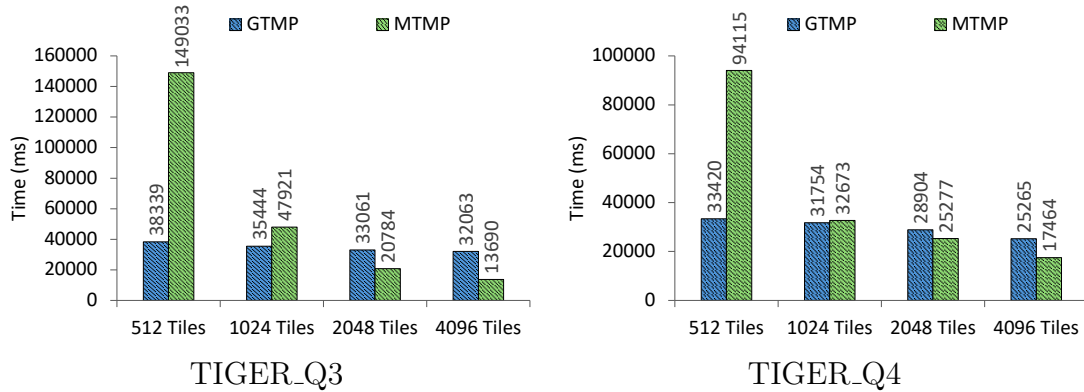


Figure 4.6: MTMP vs. GTMP for TIGER_Q3, TIGER_Q4 with different partition granularity for TIGER dataset

4.5.2 Comparison with PostgreSQL

The performance of CasaDB - Single Node with GTMP algorithm and 4096 partition granularity is evaluated in this section on TIGER and OSM datasets against

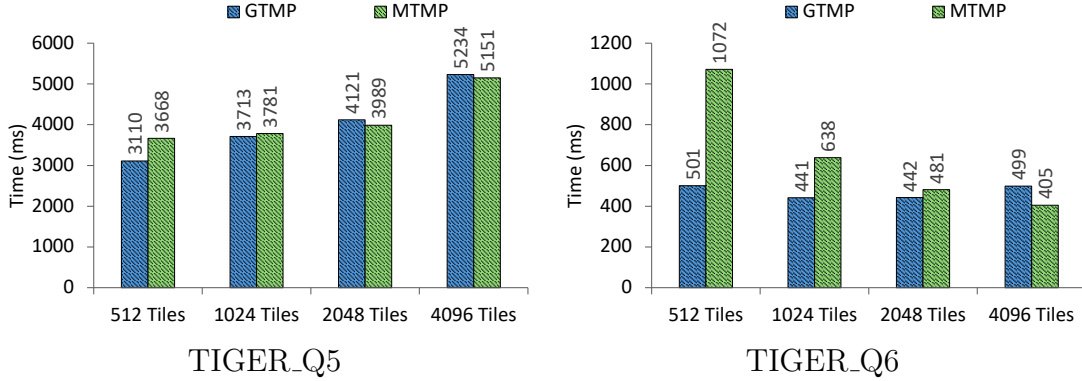


Figure 4.7: MTMP vs. GTMP for TIGER_Q5, TIGER_Q6 with different partition granularity for TIGER dataset

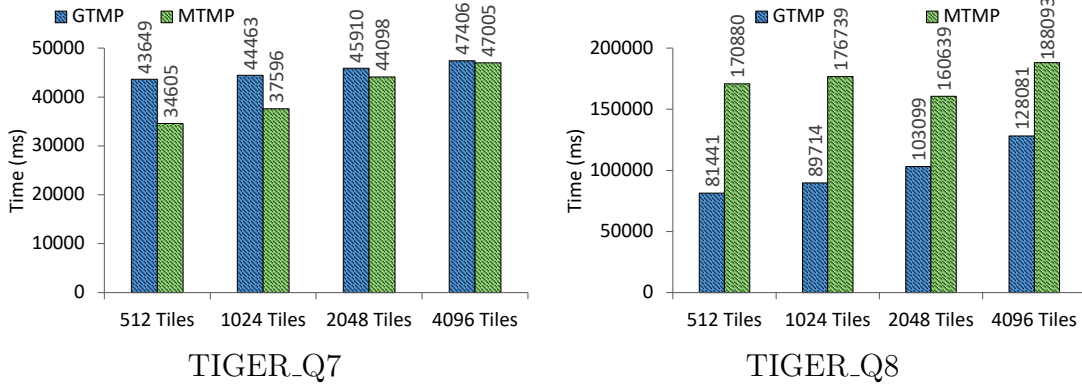


Figure 4.8: MTMP vs. GTMP for TIGER_Q7, TIGER_Q8 with different partition granularity for TIGER dataset

PostgreSQL. Figure 4.9 shows the execution time for all TIGER queries. TIGER_Q1 is 68x faster in CasaDB than PostgreSQL, TIGER_Q2 is 125x faster, TIGER_Q3 is 7x faster, TIGER_Q4 is 279x faster, TIGER_Q5 is 39x faster, TIGER_Q6 is roughly 2x faster, TIGER_Q7 is 3x faster and TIGER_Q8 is 14x faster. In conclusion, Spatial Join queries are 2x to 279x faster, Spatial Range Join is 3x faster, and Spatial Distance Join is 14x faster in CasaDB than in PostgreSQL on TIGER dataset.

Figure 4.10 shows the execution time for all OSM queries. OSM_Q1 is 146x times faster in CasaDB than PostgreSQL, OSM_Q2 is 89x faster, OSM_Q3 is 10x faster, and OSM_Q4 is 54x faster than PostgreSQL. For OSM dataset, CasaDB - Single Node is 10x-146x faster than PostgreSQL

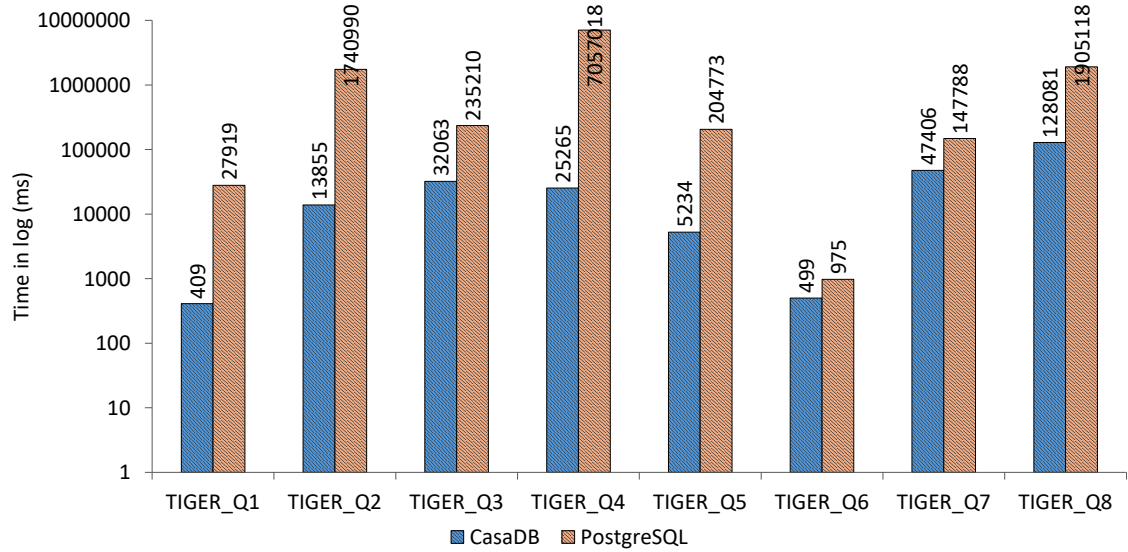


Figure 4.9: CasaDB - Single Node vs. PostgreSQL for TIGER

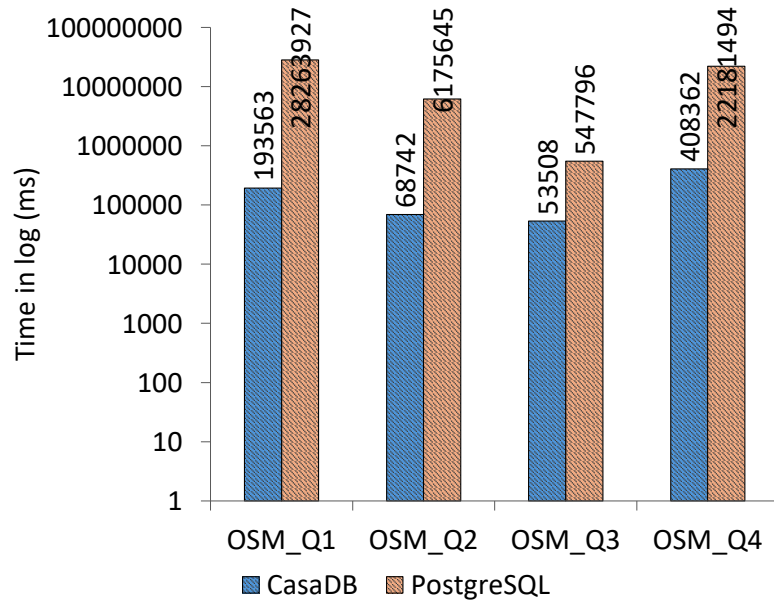


Figure 4.10: CasaDB - Single Node vs. PostgreSQL for OSM

4.6 CasaDB - Distributed Evaluation

CasaDB - Distributed, generates UPC++ code for each of the TIGER and OSM queries, the execution time for the queries on 2, 4 and 6 nodes cluster is measured and compared with Apache Sedona and Citus in this section.

4.6.1 Partitioning granularity analysis

We wanted to determine the optimal partition granularity for our algorithms in the cluster, so the datasets are partitioned with varying number of tiles: 512, 1024, 2048 and 4096. We conducted experiments involving all these sizes and using a combination of 2 nodes, 4 nodes and 6 nodes in the cluster. Figures 4.11, 4.12, 4.13, 4.14 present the execution time of TIGER queries on 2, 4 and 6 nodes cluster with 512, 1024, 2048 and 4096 tiles size. It can be seen that for queries TIGER_Q2 (Fig 4.11), TIGER_Q3 (Fig 4.12), TIGER_Q4 (Fig 4.12) and TIGER_Q6 (Fig 4.13) the setting of 4096 tiles performs better than all the others and it also scales well with increasing the number of nodes in the cluster, however, there are some interesting cases. In TIGER_Q1 (Fig 4.11), though the difference in execution time is insignificant, the setting of 1024 tiles performs better across all nodes. This behaviour is due to the additional overhead of processing more tiles in 2048 and 4096 configuration and also processing more tuples due to replication. Same behaviour can be seen in TIGER_Q5 (Fig 4.13) and TIGER_Q7 (Fig Figure. 4.14). In TIGER_Q8 (Fig 4.14), in 2 nodes, 4096 tile configuration is performing worst among all, but on 4 and 6 nodes, 4096 is performing the best.

4.6.2 Performance analysis of GTMP and MTMP

We evaluate the performance of GTMP and MTMP by running the TIGER queries listed in Table 4.3 on 2 nodes, 4 nodes and 6 nodes cluster and measured the exe-

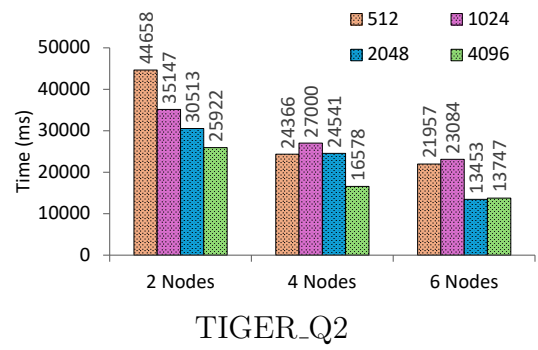
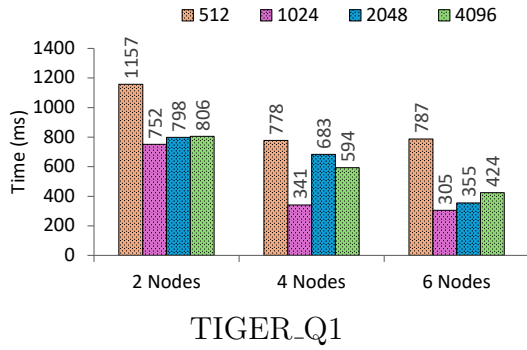


Figure 4.11: GTMP performance for TIGER_Q1, TIGER_Q2 with different partition granularity for TIGER dataset

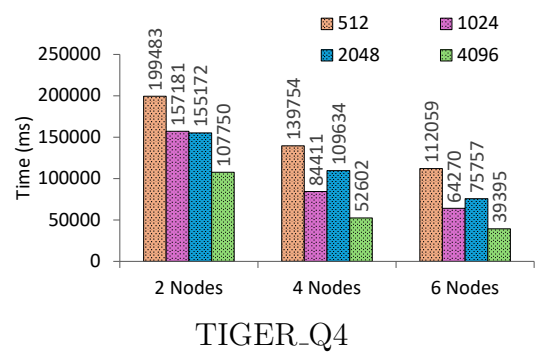
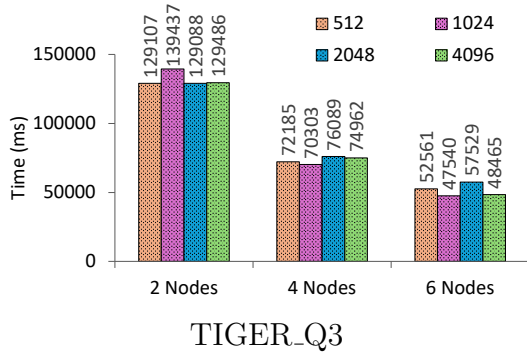


Figure 4.12: GTMP performance for TIGER_Q3, TIGER_Q4 with different partition granularity for TIGER dataset

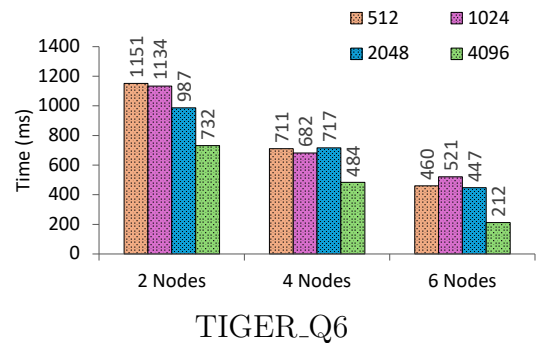
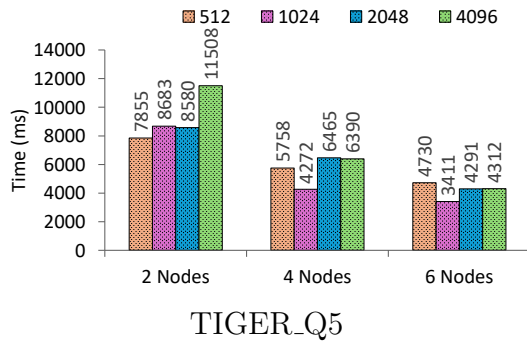


Figure 4.13: GTMP performance for TIGER_Q5, TIGER_Q6 with different partition granularity for TIGER dataset

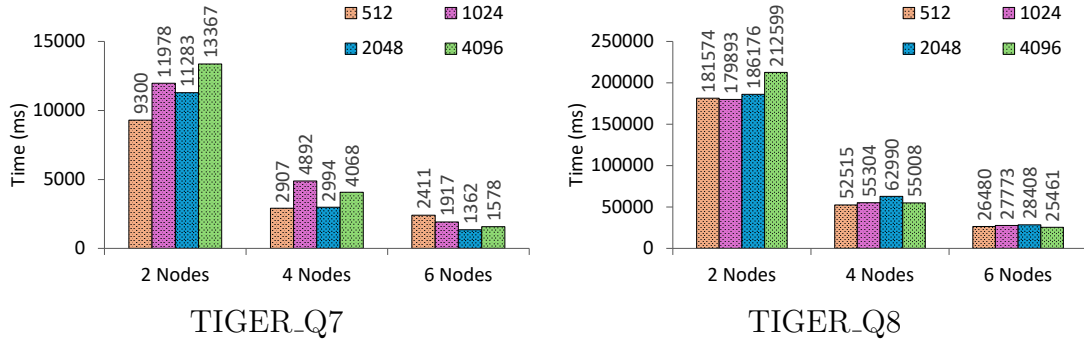


Figure 4.14: GTMP performance for TIGER_Q7, TIGER_Q8 with different partition granularity for TIGER dataset

cution time of all of the queries. As we found in the previous section 4096 partition granularity works best for our algorithms, so we used 4096 tiles setting for these experiments. For the majority of the queries both, GTMP and MTMP have similar execution times, apart from the Spatial Join queries involving larger tables (Edges). For TIGER_Q3 (Fig 4.16), GTMP is roughly 2x faster than MTMP. This is because of the high degree of data skew in the tables (Edges) involved in this query. While MTMP’s Dispatcher could sometimes provide large morsels to its workers, GTMP’s Dispatcher breaks the large morsels into granules and gives them to its workers. As a result, GTMP’s workers complete their work more quickly and then move on to their next task. GTMP shows much more resilience to data skew than MTMP, which is an important consideration when it comes to spatial query processing. GTMP also outperforms MTMP in Spatial Range Join (TIGER_Q7) and Spatial Distance Join (TIGER_Q8). TIGER_Q7 (Fig 4.18) is 2.1x - 2.6x faster than MTMP and TIGER_Q8 (Fig 4.18) is 5.4x - 5.6x faster than MTMP.

4.6.3 Scalability of our system

Figure 4.19 shows the scalability of CasaDB (4096 Tiles, GTMP) on the cluster using TIGER dataset. For long-running Spatial Join queries like TIGER_Q3, our system performs 1.7x to 2.6x better on 4 and 6 nodes respectively when compared

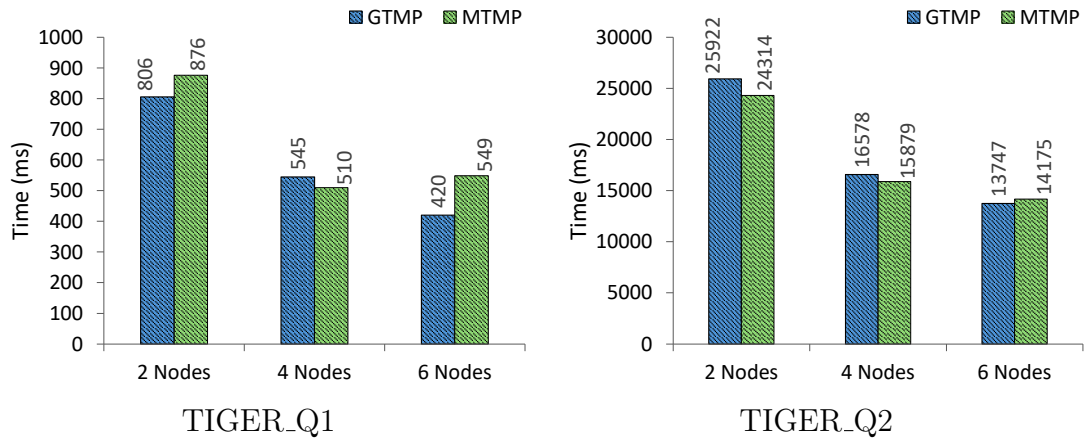


Figure 4.15: GTMP vs. MTMP for TIGER_Q1 and TIGER_Q2 queries

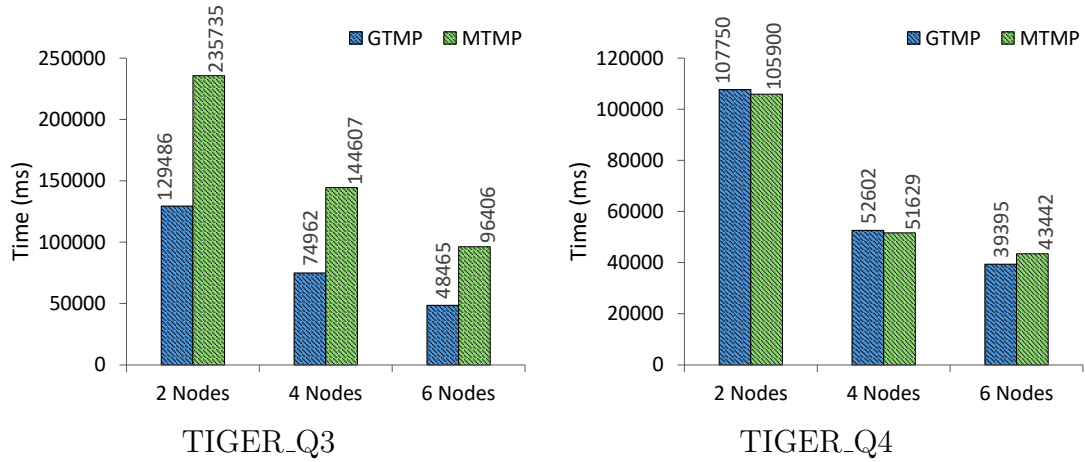


Figure 4.16: GTMP vs. MTMP for TIGER_Q3 and TIGER_Q4 queries

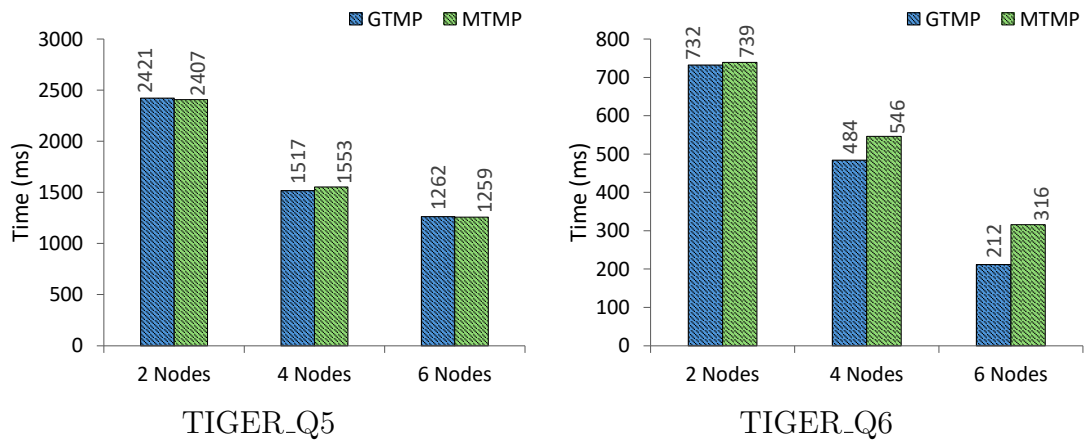


Figure 4.17: GTMP vs. MTMP for TIGER_Q5 and TIGER_Q6 queries

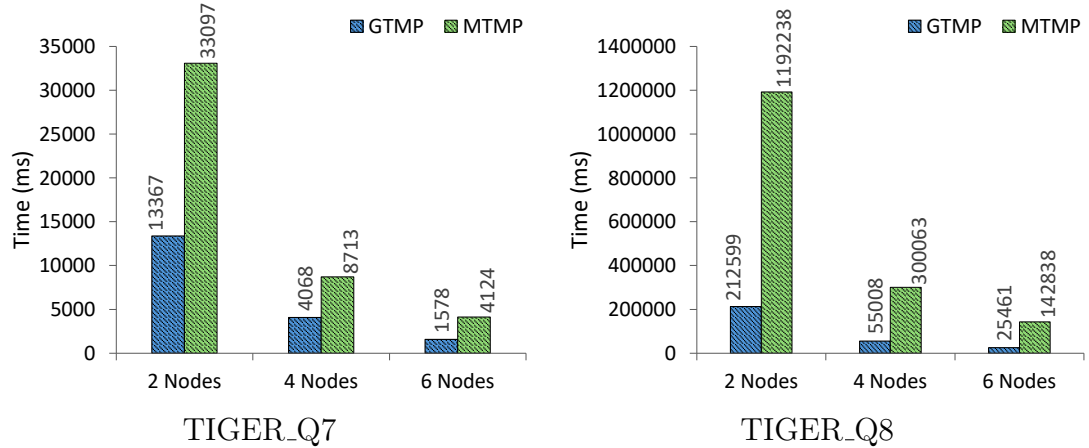


Figure 4.18: GTMP vs. MTMP for TIGER_Q7 and TIGER_Q8 queries

to 2 nodes, and also for TIGER_Q4, it scales well with 2x to 2.7x performance boost on the cluster. Short-running Spatial Join queries like TIGER_Q1 also scales 1.4x to 1.9x on 4 and 6 nodes, and TIGER_Q6 scales 1.5x to 3.45x. Spatial Range Join, TIGER_Q7 performs 3.3x to 8.5x faster on 4 and 6 nodes, when compared to 2 nodes. Spatial Distance Join, TIGER_Q8 shows performance improvement of 3.8x to 8.4x on 4 nodes and 6 nodes when compared to 2 nodes.

Figure 4.20 shows the scalability of CasaDB (4096 Tiles, GTMP) on the cluster using OSM dataset. OSM_Q1 scales well and shows a performance boost of 1.7x on 4 nodes and 2.4x on 6 nodes cluster when compared to 2 nodes. OSM_Q2 performs 1.6x to 2.2x faster on 4 and 6 nodes respectively. OSM_Q3 performs 1.1x to 1.6x faster on 4 and 6 nodes. Finally, OSM_Q4 shows 1.8x to 2x performance boost on 4 and 6 nodes when compared to 2 nodes.

Overall, the system shows significant scalability when increasing cluster size on both TIGER and OSM datasets.

4.6.4 Comparison with Apache Sedona and Citus

We evaluate CasaDB against Apache Sedona and Citus, Fig. 4.21 shows execution times for TIGER queries running on 6-node cluster. CasaDB’s fastest-running Spa-

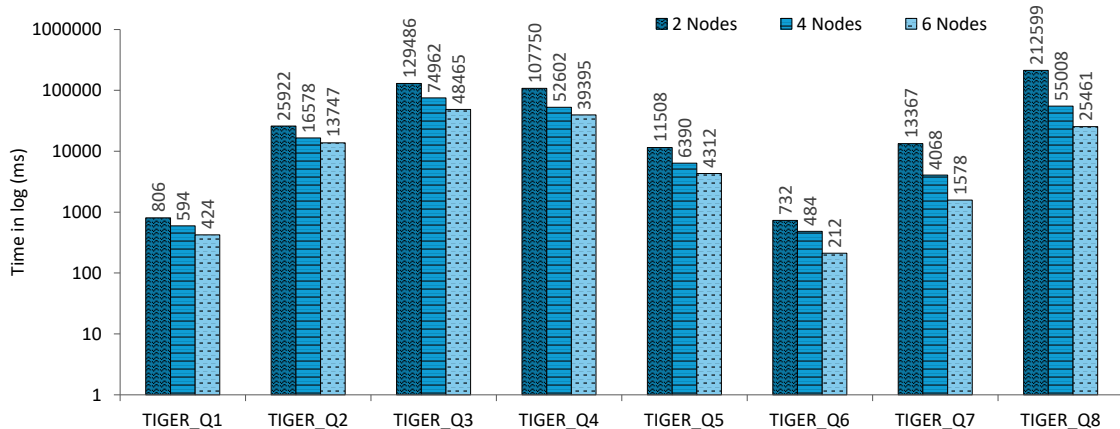


Figure 4.19: CasaDB - Distributed with TIGER Dataset

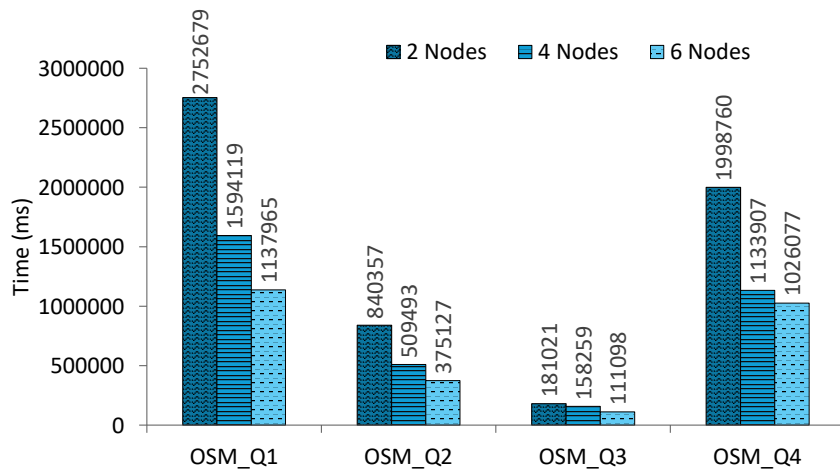


Figure 4.20: CasaDB - Distributed with OSM Dataset

tial Join query on TIGER dataset was TIGER_Q6 and it took 212 ms to finish, while the same query took 415 ms to execute on Citus and roughly 32 seconds on Sedona. Another short-running Spatial Join query, TIGER_Q1 took 424 ms on CasaDB, 912 ms on Citus and around 30 seconds on Sedona. CasaDB’s longest-running Spatial Join query on TIGER dataset was TIGER_Q3, and it took 48.4 seconds to execute, while on Citus it took 68.4 seconds to execute and Sedona took roughly 4 mins. Another long-running Spatial Join query, TIGER_Q4 took approximately 40 seconds on CasaDB, Citus took 4.2 minutes and Sedona took 94 minutes to execute. Short-running Spatial Join queries are 2x faster on CasaDB than Citus and 58x - 70x faster

than Sedona. Long-running Spatial Join queries are 1.4x - 6.4x faster on CasaDB than Citus and 5x - 143x faster than Sedona. Other TIGER Spatial Join queries (TIGER_Q2, TIGER_Q5) are 1.3x to 7x faster than Citus and 20x - 58x faster than Sedona. Spatial Range Join Query, TIGER_Q7 on CasaDB is around 3x faster than Citus and 611x faster than Sedona. Spatial Distance Join Query, TIGER_Q8 on CasaDB is 2x faster than Citus and 60x faster than Sedona. Overall, CasaDB is 1.3x to 7x faster than Citus and 5x to 611x faster than Apache Sedona on TIGER dataset.

We also evaluated the performance of CasaDB on OSM dataset on 6 nodes. Fig. 4.22 shows execution times for OSM queries running on 6-nodes cluster. Longest-running query, OSM_Q1 took roughly 19 minutes to execute on CasaDB, Citus took 28.4 minutes and Sedona took roughly 96 minutes to execute, OSM_Q4 took 17 minutes to finish on CasaDB, while Citus took 26.7 minutes to finish and Sedona took 46 minutes to execute it. The short running query, OSM_Q3 performed best on Citus with just 10.3 seconds, while CasaDB took 1.8 minutes to finish and Sedona finished in 17 minutes. Overall, CasaDB is at least 5x to 9.5x faster than Sedona and at most 1.5x times faster than Citus DB on OSM dataset.

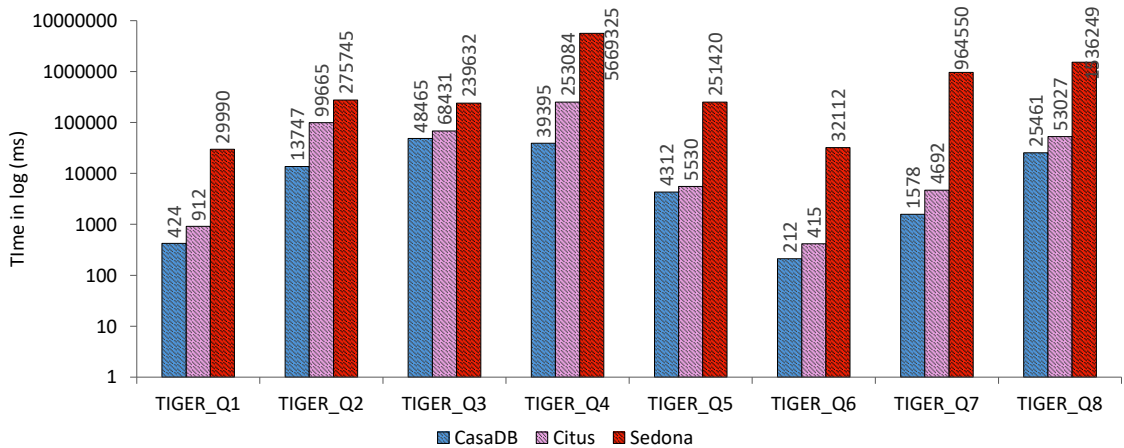


Figure 4.21: CasaDB vs. Citus vs. Apache Sedona for TIGER

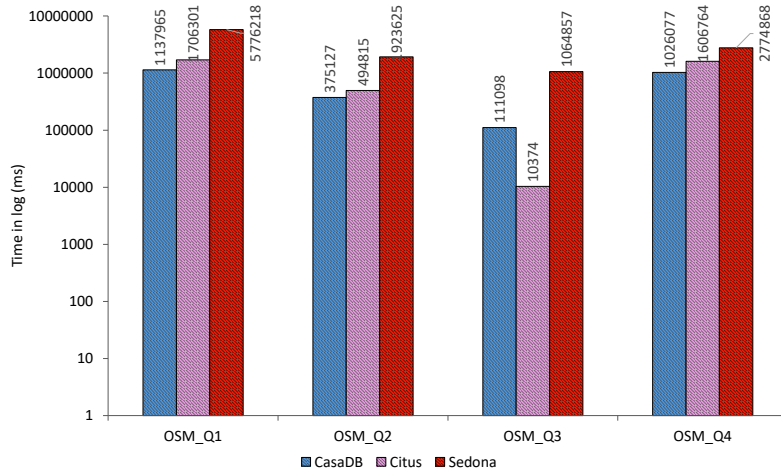


Figure 4.22: CasaDB vs. Citus vs. Apache Sedona for OSM

4.7 Summary

The Spatial query processing engine of CasaDB developed as part of the research in this thesis is evaluated in this chapter. We first evaluated the performance of CasaDB - Single Node and started by comparing the GTMP and MTMP algorithms and showed how GTMP performs better when data is skewed, and also showed the potential of MTMP when data is not skewed. We then compared the performance of CasaDB - Single Node with PostgreSQL and found it to be 2x to 279x faster on TIGER dataset and 10x to 146x faster on OSM dataset. We then evaluated the performance of CasaDB - Distributed and started by evaluating the system at various partitioning granularities and found that 4096 partitioning granularity performs best on the cluster of machines. We then evaluated the performance of MTMP and GTMP and showed that GTMP handles processing skew elegantly and overall performed better than MTMP. Then we evaluated the scalability of our system on 2, 4 and 6 node clusters. Finally, we compared the performance of CasaDB - Distributed against Apache Sedona on TIGER dataset and OSM dataset. The results show that our system scales well and is 1.3x to 7x faster than Citus and 4x to 611x faster than Apache Sedona on TIGER dataset. On OSM dataset, CasaDB is almost 1.5x faster

than Citus and 5x to 9.5x faster than Apache Sedona.

Chapter 5

Conclusion and Future Work

This chapter summarizes the key contributions of the thesis and propose future research directions.

5.1 Conclusion

This thesis presents a compilation-based single node and distributed spatial query processing engine for CasaDB. The system generates a query-specific and data-centric C++ code for single node and UPC++ code for a cluster of machines, which is then executed using PGAS runtime. To further improve the performance of the system and deal with the inherent processing skew present in spatial data, this thesis proposed two new morsel parallelism-based algorithms, Monolithic Tile-based Morsel Parallelism (MTMP) and Granular Tile-based Morsel Parallelism (GTMP). This thesis also proposed two index organization techniques, Global Index and Tile Index and how they can be used with different kinds of spatial joins. Finally, we evaluated the performance of our system on single-node configuration (CasaDB - Single Node) with PostgreSQL. We also evaluated our system on a cluster (CasaDB - Distributed) with a Spark-based distributed spatial processing system, Apache Sedona and Citus, a distributed database based on PostgreSQL. Experiment results

showed that the CasaDB Single Node is 2x to 279x faster on TIGER dataset and 10x to 146x faster on OSM dataset compared to PostgreSQL and CasaDB Distributed is 1.3x to 7x faster than Citus and 4x to 611x faster than Apache Sedona on TIGER dataset, and on OSM dataset CasaDB is almost 1.5x faster than Citus and 5x to 9.5x faster than Apache Sedona.

5.2 Future Works

This section discusses the potential for improvements to the proposed system.

5.2.1 Code Compilation Time

Compilation-based processing systems usually suffer from the code compilation time. Our system is also affected by the same, but recent work [57] proposed a new MLIR-based query processing dialect, called CasaIR for CasaDB, and it has helped to reduce the compilation time for non-spatial workload processing by a huge margin. CasaIR could be extended to support spatial data types and spatial operations in the future to reduce the compilation time for spatial workloads effectively.

5.2.2 Spatial Distance Join Processing

Our system supports Spatial Distance Join using *ST_DISTANCE*. Query execution time for such types of query is relatively higher than other Spatial Join and Spatial Range Join queries. This behaviour is expected as this kind of join can not utilize a spatial index, which other joins can. Note that for Spatial Distance Join, the execution times of Apache Sedona, PostgreSQL and Citus are also high compared to other joins, and our system still performs better than them. There is still scope for further optimizing the Spatial Distance Join processing using the metadata of partitioned tiles. We can use the tile dimensions and the distance constraint together

to filter out other tiles.

Bibliography

- [1] *UPC++ Wiki*, <https://upcxx.lbl.gov/docs/html/guide.html>, 2024, Accessed: 2024-10-25.
- [2] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood, *Dbmss on a modern processor: Where does time go?*, VLDB'99, UK, no. CONF, 1999, pp. 266–277.
- [3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz, *Hadoop-GIS: A High Performance Spatial Data Warehousing System over Mapreduce*, PVLDB (2013), 1009–1020.
- [4] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson, *Adaptive range filters for cold data: avoiding trips to siberia*, Proc. VLDB Endow. **6** (2013), no. 14, 1714–1725.
- [5] George Almasi, *Pgas (partitioned global address space) languages.*, 2011.
- [6] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah, *Aqwa: adaptive query workload aware partitioning of big spatial data*, Proc. VLDB Endow. **8** (2015), no. 13, 2062–2073.
- [7] *Apache Sedona*, <https://sedona.apache.org/>, 2024, Accessed: 2024-10-25.
- [8] *Apache Calcite*, <https://calcite.apache.org/>, 2024, Accessed: 2024-10-25.

- [9] M Ashworth, *Information technology – database languages – sql multimedia and application packages – part 3: Spatial, standard*, International organization for standardization (2016).
- [10] *Avatica*, <https://calcite.apache.org/avatica/>, 2024, Accessed: 2024-10-25.
- [11] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire, *Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources*, International Conference on Management of Data, 2018, p. 221–230.
- [12] Dan Bonachea and Jaein Jeong, *Gasnet: A portable high-performance communication layer for global address-space languages*, CS258 Parallel Computer Architecture Project, Spring (2002).
- [13] *Apache Cassandra*, <https://cassandra.apache.org/>, 2024, Accessed: 2024-10-25.
- [14] Sudip Chatterjee, Shubh Sharma, Nithin Ivan, Saumya Verma, Suprio Ray, Mark Stoodley, Calisto Zuzarte, and Ian Finlay, *Compilation of sql queries for efficient distributed in-memory processing*, International Conference on Computer Science and Software Engineering (CASCON), 2023, p. 149–154.
- [15] Douglas Comer, *Ubiquitous B-Tree*, ACM Comput. Surv. **11** (1979), no. 2, 121–137.
- [16] Leonardo Dagum and Ramesh Menon, *Openmp: an industry standard api for shared-memory programming*, IEEE computational science and engineering **5** (1998), no. 1, 46–55.
- [17] Frederica Darema, *The spmd model: Past, present and future*, European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting, Springer, 2001, pp. 1–1.

- [18] Debajyoti Datta, Mark Stoodley, and Suprio Ray, *Towards just-in-time compilation of sql queries with omr jitbuilder*, Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (USA), CASCON '21, IBM Corp., 2021, p. 256–261.
- [19] Judith R. J. Davis, *Ibm's db2 spatial extender: Managing geo-spatial information within the dbms*, 1998.
- [20] Ahmed Eldawy and Mohamed F Mokbel, *SpatialHadoop: A mapreduce framework for spatial data*, ICDE, 2015.
- [21] Dominik Filipiak, Krzysztof Weceł, Milena Stróżyna, Michał Michalak, and Witold Abramowicz, *Extracting maritime traffic networks from ais data using evolutionary algorithm*, Business Information Systems Engineering **62** (2020).
- [22] Raphael Finkel and Jon Bentley, *Quad trees: A data structure for retrieval on composite keys.*, Acta Inf. **4** (1974), 1–9.
- [23] *Apache Flink*, <https://flink.apache.org/>, 2024, Accessed: 2024-10-25.
- [24] Goetz Graefe, *Encapsulation of parallelism in the volcano query processing system*, Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '90, Association for Computing Machinery, 1990, p. 102–111.
- [25] Goetz Graefe and William J McKenna, *The volcano optimizer generator: Extensibility and efficient search*, Proceedings of IEEE 9th international conference on data engineering, IEEE, 1993, pp. 209–218.
- [26] Antonin Guttman, *R-trees: a dynamic index structure for spatial searching*, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '84, Association for Computing Machinery, 1984, p. 47–57.

- [27] *Apache Arrow*, <https://arrow.apache.org/>, 2024, Accessed: 2024-10-25.
- [28] *Apache Hadoop*, <http://hadoop.apache.org/>, 2024, Accessed: 2024-10-25.
- [29] Stefan Hagedorn, Philipp Gotze, and Kai-Uwe Sattler, *The STARK Framework for Spatio-Temporal Data Analytics on Spark*, Datenbanksysteme für Business, Technologie und Web (BTW) (2017).
- [30] *Apache Hive*, <https://hive.apache.org/>, 2024, Accessed: 2024-10-25.
- [31] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra, *Generating code for holistic query evaluation*, 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), IEEE, 2010, pp. 613–624.
- [32] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann, *Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age*, SIGMOD, 2014, p. 743–754.
- [33] Scott T. Leutenegger, Jeffrey M. Edgington, and Mario A. Lopez, *Str: A simple and efficient algorithm for r-tree packing*, Tech. report, 1997.
- [34] David Luebke, *Cuda: Scalable parallel programming for high-performance scientific computing*, 2008 5th IEEE international symposium on biomedical imaging: from nano to macro, IEEE, 2008, pp. 836–838.
- [35] *Magellan: Geospatial Analytics Using Spark*, <https://github.com/harsha2010/magellan>.
- [36] *Alibaba MaxCompute*, <https://www.alibabacloud.com/en/product/maxcompute>, 2024, Accessed: 2024-10-25.
- [37] Puya Memarzia, Maria Patrou, Md Mahbub Alam, Suprio Ray, Virendra C. Bhavsar, and Kenneth B. Kent, *Toward efficient processing of spatio-temporal*

- workloads in a distributed in-memory system*, International Conference on Mobile Data Management (MDM), 2019, pp. 118–127.
- [38] *MongoDB*, <https://www.mongodb.com/>, 2024, Accessed: 2024-10-25.
- [39] Shamkant B. (2010) Navathe, Ramez Elmasri, *Fundamentals of database systems (6th ed.)*, Pearson Education, Upper Saddle River, N.J., 2010.
- [40] Thomas Neumann, *Efficiently compiling efficient query plans for modern hardware*, PVLDB **4** (2011), no. 9, 539–550.
- [41] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi, *MD-Base: A scalable multi-dimensional data infrastructure for location aware services*, MDM, 2011.
- [42] *Openstreetmap*, <https://www.openstreetmap.org/>, 2024, Accessed: 2024-10-25.
- [43] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper, *How good are modern spatial analytics systems?*, Proc. VLDB Endow. **11** (2018), no. 11, 1661–1673.
- [44] Jignesh Patel, Jiebing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellmann, Jim Kupsch, Shelly Guo, Johan Larson, David Dewitt, and Jeffrey Naughton, *Building a scalable Geo-Spatial DBMS: technology, implementation, and evaluation*, SIGMOD, 1997, pp. 336–347.
- [45] Jignesh M. Patel and David J. DeWitt, *Partition based spatial-merge join*, SIGMOD Rec. **25** (1996), no. 2, 259–270.
- [46] *R-tree. (2023, december 30). in wikipedia.*, <https://en.wikipedia.org/wiki/R-tree>.

- [47] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman, *Compiled query execution engine using jvm*, 22nd International Conference on Data Engineering (ICDE'06), IEEE, 2006, pp. 23–23.
- [48] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson, *A parallel spatial data analysis infrastructure for the cloud*, SIGSPATIAL, 2013, p. 284–293.
- [49] Suprio Ray, Bogdan Simion, and Johnson Ryan Brown, Angela Demke, *Skew-resistant parallel in-memory spatial join*, International Conference on Scientific and Statistical Database Management (SSDBM), 2014.
- [50] *Redis*, <https://redis.io/>, 2024, Accessed: 2024-10-25.
- [51] Hanan Samet, *The design and analysis of spatial data structures*, Addison-Wesley Longman Publishing Co., Inc., USA, 1990.
- [52] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database system concepts*, 6 ed., McGraw-Hill, New York, 2010.
- [53] *Simple feature access – part 1: Common architecture*, <https://www.ogc.org/standard/sfa/>, 2024, Accessed: 2024-10-25.
- [54] Ruby Y. Tahboub and Tiark Rompf, *Architecting a query compiler for spatial workloads*, International Conference on Management of Data (SIGMOD), 2020, p. 2103–2118.
- [55] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref, *LocationSpark: A distributed in-memory data management system for big spatial data*, VLDBJ **9** (2016), no. 13, 1565–1568.
- [56] *Tiger dataset*, <http://www.census.gov/geo/www/tiger>, 2024, Accessed: 2024-10-25.

- [57] Saumya Verma, *Efficient in-memory processing of sql queries with jit compilation*, Master's thesis, University of New Brunswick, Dec 2023.
- [58] David W Walker and Jack J Dongarra, *Mpi: a standard message passing interface*, Supercomputer **12** (1996), 56–68.
- [59] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo, *Simba: Efficient in-memory spatial analytics*, SIGMOD, 2016.
- [60] Simin You, Jianting Zhang, and Le Gruenwald, *Large-scale spatial join query processing in cloud*, ICDEW, 2015.
- [61] Jia Yu, Zongsi Zhang, and Mohamed Sarwat, *Spatial data management in apache spark: the geospark perspective and beyond*, Geoinformatica **23** (2019), no. 1, 37–78.
- [62] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster computing with working sets.*, HotCloud **10** (2010), no. 10-10, 95.
- [63] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick, *Upc++: a pgas extension for c++*, 2014 IEEE 28th IPDPS, IEEE, 2014, pp. 1105–1114.

Vita

Candidate's full name: Rahul Sahni

University attended:

- Bachelor of Technology in Computer Science, Guru Gobind Singh Indraprastha University, India, 2019
- Master of Computer Science, University of New Brunswick, Canada, Expected December 2024

Publications:

- Rahul Sahni, Xiaozheng Zhang, Sudip Chatterjee and Suprio Ray. 2024. Query Compilation based Distributed Morsel-driven Parallel Spatial Query Processing. In The 32nd ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '24), October 29-November 1, 2024, Atlanta, GA, USA, 4 Pages, <https://doi.org/10.1145/3678717.3691289>
- Rahul Sahni, Xiaozheng Zhang, Sudip Chatterjee and Suprio Ray. 2024. Scalable Big Spatial Data Processing with SQL Query Compilation and Distributed Morsel-driven Parallelism. In 2024 IEEE International Conference on Big Data (IEEE BigData 2024), Dec 15-18, 2024, Washington DC, USA, 10 Pages