High Performance Spatial and Spatio-temporal Data Processing

by

Suprio Ray

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

© Copyright 2015 by Suprio Ray

Abstract

High Performance Spatial and Spatio-temporal Data Processing

Suprio Ray
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2015

The rapid growth of spatial data volume and technological trends in storage capacity and processing power are fueling many emerging spatial and spatio-temporal applications from a wide range of domains. Spatial join is at the heart of many of the emerging spatial data analysis applications. However, spatial join processing on even a moderate sized dataset is very time consuming. At the same time, there is a rapid expansion in available processing cores, through multicore machines and Cloud computing. The confluence of these trends points to a need for effective parallelization of spatial query processing. Unfortunately, traditional parallel spatial databases are ill-equipped to deal with the performance heterogeneity that is common in the Cloud. We introduce a parallel spatial data analysis infrastructure that exploits all available cores in a heterogeneous cluster. We also present an approach to parallelize spatial join queries in a large main memory multicore machine.

Among the emerging spatio-temporal applications Location Based Services (LBS) are the most prominent. These applications are characterized by high rate of updates and many concurrent short running range queries. We present a parallel in-memory spatio-temporal indexing technique to support the demands of LBS workloads. Our system achieves significantly better performance than existing approaches to indexing spatio-temporal data. We also introduce a parallel in-memory spatio-temporal topological join approach.

Acknowledgements

The pursuit of a PhD degree is a significant endeavour in terms of time, energy and financial implications. In my journey in this pursuit, I have spent many a hours reading research papers, writing and testing code, running experiments and writing technical documents. But this effort may not have been successful without the support of a number of people.

First of all, I would like to thank my adviser Prof. Angela Demke Brown for all her support during these past years. Prof. Demke Brown has been a wonderful mentor, who is one of the smartest persons I have ever met. Her office door was always open for me whenever I needed to discuss something. No matter what she would have been doing, she always gave me undivided attention. She gave me the freedom to pursue topics that interest me. At the same time she provided valuable advice as to how to remain focused. Her tireless support during the writing of research papers is inspiring. I have learnt a lot from her as to how to address and solve a research problem and how to present it to an audience. I express my sincere gratitude to Prof. Demke Brown for her support and guidance that has made this thesis a reality.

I owe my deep gratitude to Prof. Ryan Johnson and Prof. Nick Koudas who are my thesis committee members. They have been my mentors and collaborators throughout this endeavour. During the various checkpoint meetings they provided me with valuable feedback. I could always count on them for insightful suggestions on how to improve my work.

I am grateful to Dr. Rolando Blanco and Dr. Anil Goel who were my mentors at SAP. They allowed me to pursue a research topic of my choice. I received valuable inputs from them during my internship at SAP.

I thank Prof. Ashvin Goel and Prof. Eyal de Lara for agreeing to be the internal examiners. They also generously offered support at different times by letting me use some of their servers. My special thanks to the external examiner Prof. Jörg Sander from University of Alberta for patiently reading my thesis, offering valuable suggestions and writing a kind appraisal report. I am grateful to Prof. Alexandra Fedorova from Simon Fraser University who believed in me and inspired me to pursue a PhD.

I would like to thank my lab mate Bogdan Simion. Bogdan has always been a wonderful friend of mine and I admire him for his gentle and cheerful character. He collaborated with me in the Jackpine project and helped me by exploring a few macro benchmarks, setting up databases and running experiments. He also assisted me in the Niharika project by helping to create the large dataset. We had many an interesting discussion about our research. I also thank all the members of SysNet lab. I have made many friends and I shall cherish the fond memories.

Finally, I thank my wife for her support and patience. I am indebted to my parents for their love and support and it is futile to try to thank them. I also thank my brother for his support.

Last but not least, I would like to thank all the people I met and worked with during these years, whose direct and indirect support has made a difference.

Contents

1 Introduction **1**

1.1 Benchmarking Spatial Queries . . . 2
1.2 Spatial Data Management . . . 2
1.3 Spatio-Temporal Data Management . . . 3
1.4 Organization Of The Dissertation . . . 4

2 Related Works **5**

2.1 Database Benchmarking . . . 5
2.1.1 Non-spatial database benchmarks . . . 5
2.1.2 Spatial Database Benchmarks . . . 6
2.2 Spatial Data Management . . . 7
2.2.1 Spatial Data Representation . . . 8
2.2.2 Spatial Index . . . 9
2.2.3 External Memory Spatial Join . . . 10
2.2.4 In-Memory Spatial Join . . . 13
2.2.5 Parallel Spatial Join . . . 13
2.2.6 Big Data Infrastructures For Spatial Query Processing . . . 16
2.3 Spatio-Temporal Data Management . . . 19
2.3.1 Spatio-Temporal Index . . . 19
2.3.2 Spatio-Temporal Join And Trajectory Index . . . 24

3 Benchmarking Spatial Queries **26**

3.1 Introduction . . . 26
3.2 The Benchmark . . . 27
3.2.1 Implementation Overview . . . 27
3.2.2 Data Model . . . 29
3.2.3 Micro Benchmark . . . 29
3.2.4 Macro Benchmark . . . 30
3.3 Experimental Setup . . . 34
3.4 Benchmark Results . . . 35
3.4.1 Data Loading . . . 35
3.4.2 Micro Benchmark . . . 36
3.4.3 Macro Benchmark . . . 40
3.4.4 Overall Score . . . 40

3.5 Discussion . . . 41

4 Performance Heterogeneity-Aware Parallel Spatial Join For The Cloud 43

4.1 Introduction . . . 43
4.1.1 A Case For Parallelizing Spatial Join Queries . . . 43
4.1.2 Heterogeneity In Computing Clusters . . . 44
4.1.3 Load Balancing . . . 46
4.2 Niharika . . . 47
4.2.1 Architecture . . . 47
4.2.2 Spatial Declustering . . . 48
4.2.3 Load Balancing . . . 48
4.2.4 Heterogeneity-aware single assignment of partitions . . . 52
4.2.5 Multi-Round Assignment Of Partitions . . . 52
4.3 Experimental Evaluation . . . 58
4.3.1 Experimental Setup . . . 58
4.3.2 Results With Dataset That Fits In Memory . . . 58
4.3.3 Results With Larger Dataset (USA) . . . 59
4.4 Scalability and Data Placement . . . 60
4.5 Chapter Summary . . . 63

5 Parallel In-Memory Spatial Join 64

5.1 Introduction . . . 64
5.1.1 Processing Skew . . . 64
5.2 SPINOJA . . . 65
5.2.1 System Organization . . . 66
5.2.2 MOD-Quadtree Declustering . . . 67
5.2.3 Work Metrics . . . 69
5.2.4 Processing Of Spatial Predicates . . . 72
5.2.5 Load Balancing . . . 73
5.2.6 Determining The Number Of Partitions (Tiles) . . . 76
5.2.7 Managing Processing Skew . . . 80
5.3 Experimental Evaluation . . . 80
5.3.1 Experimental Setup . . . 80
5.3.2 Multicore Scalability . . . 80
5.3.3 Comparison With Other In-Memory Spatial Join Approaches . . . 81
5.3.4 Comparison With PostgreSQL . . . 83
5.4 Chapter Summary . . . 84

6 High Performance Spatio-Temporal Index For Location-Based Services (LBS) 85

6.1 Introduction . . . 85
6.2 Design Considerations . . . 87
6.2.1 Main-Memory Storage For LBS . . . 87
6.2.2 A Novel Parallel In-Memory Index . . . 87
6.3 Overall System Organization . . . 88

6.4 Insert-Efficient Main-Memory Storage . . . 88
6.5 PASTIS . . . 90
6.5.1 Index Structure . . . 90
6.5.2 Update Processing . . . 92
6.5.3 Load-Balancing And Skew Handling . . . 93
6.5.4 Query Processing . . . 94
6.6 Discussion . . . 96
6.7 Performance Studies . . . 97
6.7.1 Experimental Setup . . . 97
6.7.2 Update Performance . . . 98
6.7.3 Query Performance . . . 101
6.7.4 Comparison With Other Indexes And Other LBS Systems . . . 103
6.8 Chapter Summary . . . 103

7 Trajectory-Based Spatio-Temporal Topological Join 105
7.1 Introduction . . . 105
7.2 Case Study And Motivation . . . 106
7.3 Spatio-Temporal Join Queries . . . 107
7.3.1 Trajectory-Based Spatio-Temporal Topological Join . . . 107
7.3.2 Spatio-Temporal Topological Predicates . . . 108
7.3.3 Predicates Evaluation . . . 108
7.4 Applicable Spatio-Temporal Join Algorithms . . . 109
7.4.1 Nested Loop Join (NLJ) . . . 109
7.4.2 Indexed Nested Loop Join With One Index (INLJ1I) . . . 109
7.4.3 Indexed Nested Loop Join With Two Indexes (INLJ2I) . . . 109
7.5 Our Approach . . . 111
7.5.1 Trajectory Filtering (INLJ2I-TF) . . . 111
7.5.2 Evaluation And Analysis . . . 112
7.5.3 Overview Of PISTON . . . 113
7.5.4 A Parallel In-Memory Join Algorithm . . . 114
7.5.5 In-Memory Spatial Index . . . 115
7.5.6 In-Memory Trajectory Index . . . 118
7.6 Experimental Evaluation . . . 120
7.6.1 Dataset With Static Polygons . . . 120
7.6.2 Dataset With Evolving Polygons . . . 125
7.7 Chapter Summary . . . 127

8 Conclusion 128
8.0.1 Future work . . . 129

Bibliography 130

List of Tables

3.1 Database tables used for micro and macro benchmark 28
3.2 Topological Relations in Dimensionally Extended 9-intersection Model included in benchmark . . 30
3.3 Micro benchmark - queries 31
3.4 Micro benchmark - inserts 32
3.5 Macro benchmark queries 33
3.6 Databases used in evaluation 35
3.7 Load time (importing shapefiles, index creation, update stats) 36
3.8 Overall scores 41

4.1 Some use cases of spatial join queries 44
4.2 Database tables 44
4.3 Jackpine queries and abbreviations with two datasets 45
4.4 Illustration of data placement 62

5.1 Jackpine queries and abbreviations 65
5.2 Point density per tile for Ed_cr_Ed 80
5.3 Database tables 81

6.1 Trace file details 101
6.2 Parameter settings 102

7.1 Queries with different predicates and abbreviations 108
7.2 Details of trajectory dataset 120
7.3 Trajectory dataset generation settings 121
7.4 Details of polygon dataset 121
7.5 Evolving polygons dataset generation settings 125

List of Figures

2.1 MBR representation and the two step spatial query evaluation 8
2.2 Spatial partitioning 9

3.1 Benchmark output report 27
3.2 Pairwise spatial joins involving polygons, lines and points 36
3.3 Pairwise spatial joins involving Polygons (Areas) only 37
3.4 Spatial join with a given object 37
3.5 Spatial analysis (N/S = scenario not supported due to unsupported operations) 38
3.6 Data insertion 39
3.7 Macro Benchmark (N/S = scenario not supported) 39

4.1 Min and max single-node PostgreSQL (with PostGIS) execution times for Jackpine queries, observed on 10 distinct EC2 m1.xlarge instances (warm runs) 46
4.2 Min and max observed execution times of 7 Jackpine queries in a cluster of local nodes (cold runs) 46
4.3 Architecture of Niharika 47
4.4 Spatial partitioning (a) and declustering using Hilbert SFC traversal and tile aggregation into partitions (b) 47
4.5 Spatial join with partitioned tables 50
4.6 Execution times for LiAus query with different partition strategies 51
4.7 Query execution times - original vs Sharding-StoredProc (California dataset) 52
4.8 Query execution times - original vs Sharding-StoredProc (USA dataset) 52
4.9 Algorithm1: Multi-round fixed-size batch assignment 53
4.10 Algorithm2: Multi-round fixed-size batch assignment using node processing capacity 55
4.11 Algorithm3: Multi-round batch assignment from a preferred partition set 56
4.12 Procedure StrategicAssignment for Algorithm3 57
4.13 Speedup of Algorithm1 compared with Algorithm2 (California dataset, 4 cores/node) 58
4.14 Algorithm3 vs Algorithm1 vs RR-ESP speedup (USA dataset, 4 cores/node) 59
4.15 Histogram of the processing times of the partitions for LtAus (USA dataset, Algorithm3) 59
4.16 Algorithm3 vs Algorithm1 vs RR-ESP speedup with modified dataset (USA dataset, 4 cores/node) 60
4.17 Algorithm4: Multi-round batch assignment to idle nodes with fetch 61
4.18 Algorithm4 speedup in a cluster (single-core) 62

5.1 % execution time spent in filter vs. refinement with Clone Join-IM and INLJ 65
5.2 Histogram of the execution time per spatial partition (tile) for query Ed_cr_Al 66
5.3 SPINOJA system organization 67

5.4 MOD-Quadtree object decomposition . . . 68
5.5 Procedure getWorkMetric4Tile . . . 68
5.6 Histogram of total area in hectares per tile of Arealm (landmass polygons) table . . . 69
5.7 Histogram of total length in km per tile of Edges (polylines) table . . . 70
5.8 The cost of evaluating polygon overlaps polygon . . . 71
5.9 Execution times of queries with different metrics (2 cores) . . . 72
5.10 Processing of predicate Polygon overlaps polygon . . . 72
5.11 Processing of predicate Polygon contains polygon . . . 73
5.12 Algorithm Load-balance TileNLJ . . . 75
5.13 Algorithm Load-balance TilePS . . . 76
5.14 Algorithm Load-balance TilePS_sepRefine . . . 77
5.15 Execution times of queries with different load-balancing strategies (2 cores) . . . 78
5.16 Candidate set size of queries with different load-balancing strategies (2 cores) . . . 78
5.17 Break-down of time with different load-balancing strategies (2 cores) . . . 78
5.18 Cache misses with different load-balancing strategies (2 cores) . . . 79
5.19 Execution times of queries with different number of tiles (2 cores) . . . 79
5.20 Candidate set size of queries with different number of tiles (2 cores) . . . 79
5.21 Execution times of different tiles for the query Ed_cr_Al . . . 79
5.22 Multicore speedup of SPINOJA over 1 core performance (2, 4, 6 and 8 cores) . . . 81
5.23 Memory usage for queries with SPINOJA vs. other approaches (8 cores) . . . 81
5.24 Execution times of queries with SPINOJA vs. other approaches (8 cores) . . . 81
5.25 Breakdown of execution times of queries with SPINOJA vs. other approaches (sequential execution) 82
5.26 Execution times of queries with SPINOJA vs. PostgreSQL . . . 83

6.1 System organization . . . 89
6.2 Completion time (1 billion record insert) with different storage organization . . . 89
6.3 L2 cache misses with different storage organization . . . 90
6.4 Structure of spatio-temporal index, PASTIS . . . 91
6.5 Algorithm Update . . . 92
6.6 Procedure TemporalIdx.insertLocnRecord . . . 93
6.7 Procedure CBmapHmRIDList.updateObject . . . 93
6.8 Algorithm Round-robin assignment . . . 94
6.9 Algorithm Adaptive . . . 94
6.10 Algorithm PastRangeQuery . . . 95
6.11 Procedure TemporalIdx.getIntervalMatchingObjs . . . 95
6.12 Procedure TemporalIdx.isObjInGrdCellAndIntrvl . . . 95
6.13 Procedure CBmapHmRIDList.isObjInGrdCellAndIntrvl . . . 96
6.14 Breakdown of time for location update (dataset 10 million) . . . 97
6.15 Update throughput with different load-balancing approaches . . . 97
6.16 Standard dev. of number of records processed with different load-balancing approaches . . . 98
6.17 Update throughput with different number of partitions . . . 98
6.18 Update throughput with different temporal index interval length . . . 99
6.19 Total size of all bitmaps with different temporal index interval length . . . 99
6.20 Query throughput . . . 100

6.21 Average query response time . . . 100
6.22 Query throughput - spatial extent . . . 101
6.23 Query throughput - effect of interval length . . . 102
6.24 PASTIS vs B^x-tree vs TPR . . . 103

7.1 Geofence crossing scenario . . . 106
7.2 Evaluation of the spatio-temporal predicates . . . 107
7.3 Algorithm1 - NLJ . . . 110
7.4 Algorithm2 - INLJ1I . . . 110
7.5 Algorithm3 - INLJ2I . . . 111
7.6 Algorithm4 - INLJ2I-TF . . . 112
7.7 Execution time of Crosses query with different spatio-temporal join algorithms . . . 113
7.8 Breakdown of time with INLJ2I-TF (TB-tree & STR) . . . 113
7.9 PISTON overview . . . 114
7.10 Algorithm5 - PISTON . . . 116
7.11 Splitting spatial objects along tile boundaries (a) and partial trajectory MBR filtering (b) . . . 117
7.12 Different Level1 grid resolutions . . . 117
7.13 Grid setup and testing . . . 117
7.14 Procedure evalPointInPoly . . . 119
7.15 Trajectory index update process . . . 120
7.16 Execution times of query Crosses with PISTON vs NLJ2I-TF (TB-tree & STR) with Dataset-10mi 121
7.17 Execution times of query Crosses with PISTON vs NLJ2I-TF (TB-tree & STR) with Dataset-100mi 122
7.18 Index creation time (including preprocessing) with PISTON vs TB-tree & STR . . . 122
7.19 Memory usage: PISTON vs TB-tree & STR . . . 123
7.20 Execution times of query Crosses with different number of worker threads (interval length 1000) . 123
7.21 Execution times of query Crosses with different number of Level0 grid cells (interval length 1000, 8 cores) . . . 124
7.22 Execution times of Crosses with different predicates (interval length 1000) . . . 125
7.23 Execution times of query Crosses with different distributions for evolving polygons (single threaded) 126
7.24 Execution times of query Crosses with different number of threads for Dataset-1bi (interval length 1000, Gaussian distribution) . . . 126
7.25 Distribution of polygon objects . . . 127
7.26 Trajectory dataset Dataset-10mi . . . 127

Chapter 1

Introduction

Spatial data is everywhere. Geospatial Web services such as Google Maps, in-vehicle GPS navigation systems, GPS-enabled mobile phones, and a host of accompanying Location Based Services (LBS) have become part of our daily experience. Enormous quantities of spatial data is constantly being generated from various sources such as satellites, sensors and mobile devices. NASA's Earth Observing System (EOS), for instance, generates 1 terabyte of data every day [68]. A decade ago, it was estimated that 80% of all business data stored in existing databases had spatial attributes [49]. The percentage today is probably even higher, as the ability to track customers and inventory has become cheaper and easier.

The deluge of spatial data and increasingly sophisticated end-user demand is changing the landscape for spatial data management. The term "spatial database" refers to relational database management systems that support spatial data types in the same way as any other data in the database. Spatial queries are used to extract useful information related to features of interest with any location criteria. Although spatial support was once a niche feature provided by high-end systems for a small set of customers, or found only in research systems, it is now widely used with new applications appearing regularly. Depending on the nature of the data and the queries that operate on this data, these applications can be generally classified into two categories. The first category of applications involve somewhat static spatial datasets (eg. map data) and complex analytical queries. These spatial analytics applications are characterized by long-running compute-intensive spatial join queries. Traditional Geographic Information Systems (GIS) applications and many emerging spatial analytics applications belong to this category. The second category of applications involve a high rate of updates. In contrast to the first category, these are driven by short-running range queries, Many emerging spatio-temporal applications, such as the Location-Based Services (LBS), are prime instances of this.

In this thesis we investigate topics pertinent to both categories of applications. Due to the importance of high performance systems in the landscape of rapidly growing data volumes [110], a key focus in this thesis is to address the performance and scalability issues of data management systems that deal with spatial and spatio-temporal data. To this end, we first developed a spatial database benchmark to identify such issues. Then we built two spatial query processing systems to deal with parallel spatial join in different settings. Next we developed a main-memory spatio-temporal data management system to handle high update rates and many concurrent range queries that are typical in LBS. Finally, we built a parallel main-memory system for trajectory-based topological join queries.

1.1 Benchmarking Spatial Queries

The emergence of novel spatial and spatio-temporal applications naturally gives rise to performance questions from multiple stakeholders and there exists a need for a spatial database benchmark that incorporate such applications. Although a number of database benchmarks exist, such as the TPC benchmarks [121], there are only a few spatial database benchmarks and these were developed decades ago. We introduce a spatial database benchmark, Jackpine, which we describe in Chapter 3. It includes a micro benchmark that attempts to provide a comprehensive coverage of spatial operations. The benchmark also has a macro benchmark suite that models a number of real-world applications. Some of the Jackpine micro benchmark queries are used to evaluate the performance of the data management systems that we built subsequently.

1.2 Spatial Data Management

Spatial join is a crucial operation in many spatial analysis applications in scientific and geographical information systems. With the growing data volume and popularity of online services new spatial applications are emerging in areas as diverse as medical imaging, building information management and environmental risk management. Due to the compute-intensive nature of spatial predicate evaluation, spatial join queries can be slow even with a moderate sized dataset. Efficient parallelization of spatial join is therefore essential to achieve acceptable performance for many spatial applications. The current surge in available processing cores, through multicore architectures and the Cloud computing model, presents an opportunity to dramatically reduce the latency of spatial join queries by creating a scalable parallel data processing infrastructure. The key challenge is how to balance the spatial processing load across a large number of worker nodes, given significant performance heterogeneity in the nodes and processing skew in the workload.

Modern data-centers have evolved as very large distributed systems built from hundreds of machines from multiple hardware generations. Performance heterogeneity among these machines occurs naturally, as disks or nodes fail and are replaced with newer components. The same effect occurs in smaller clusters within an organization. Cloud computing adds a new dimension, in that the infrastructure itself is no longer fixed. Cloud infrastructure providers, such as Amazon EC2, offer different instance types, each with different processing and storage capacity. Furthermore, due to virtualization overhead and load consolidation, different machines of a single instance type may vary widely in performance.

Although the issue of performance heterogeneity has received widespread attention in distributed data processing systems such as Map-Reduce, not much work has been done in parallel databases in this context. The challenges with parallel query processing in a heterogeneous cluster were recognized by Mayr et al. [75], but their evaluation was limited to User Defined Functions (UDFs) for traditional workloads in a 2 node system. Spatial database workloads are more compute-intensive than traditional database workloads [113] and the amount of computation required to evaluate a spatial join predicate can vary widely for different tuples. Thus, assigning the same number of tuples to each node, even when the nodes are identical, does not guarantee good load balancing. Due to this processing skew, the performance heterogeneity may get aggravated. Moreover, traditional database systems are unable to fully exploit multicore machines. [57]. Most databases, including PostgreSQL, do not yet support intra-query parallelism. Although a few recent projects [3, 7] explored support for intra-query parallelism, none of them looked at spatial join queries.

In Chapter 4 we introduce *Niharika*, a distributed spatial query processing system that provides a framework for spatial declustering and dynamic load balancing on top of a cluster of worker nodes, each of which runs a stan-

dard PostgreSQL/PostGIS relational database. The overall architecture of Niharika is inspired by HadoopDB [2], which aims to provide a parallel database implementation for Cloud computing. However, Niharika uses a multi-round query execution model that is better suited to exploit multiple processing cores and to address performance heterogeneity. Niharika's mechanisms for data partitioning and dynamic load balancing directly support intra-query parallelism, by concurrently executing multiple customized queries on multiple cores in each machine. A key advantage of our approach is that it requires no change in the underlying relational database engine. By taking advantage of the optimized relational database systems Niharika offers good query execution performance and scalability.

Due to the availability of machines with increasingly large main memory, many moderately large spatial datasets may fit in the RAM. With the rising core count per node, it is natural to consider an in-memory single node alternative to the Cloud model in improving the performance of spatial join. Furthermore, even with the multi-round query execution approach of Niharika, a key limiting factor to achieving parallel performance is the processing skew due to object properties. Previous parallel spatial join approaches tried to partition the dataset so that the number of spatial objects in each partition was as equal as possible. They also focused only on the filter step. However, when the more compute-intensive refinement step is included, significant processing skew may arise due to the uneven size of the objects. This processing skew significantly limits the achievable parallel performance of the spatial join queries, as the longest-running spatial partition determines the overall query execution time.

In Chapter 5 we present our solution, *SPINOJA*, a skew-resistant parallel in-memory spatial join infrastructure. SPINOJA introduces MOD-Quadtree declustering, which partitions the spatial dataset such that the amount of computation demanded by each partition is equalized and the processing skew is minimized. We compare three *work metrics* used to create the partitions and three load-balancing strategies to assign the partitions to multiple cores. SPINOJA uses an in-memory column-store to store the spatial tables. Our evaluation shows that SPINOJA outperforms in-memory implementations of previous spatial join approaches by a significant margin and a recently proposed in-memory spatial join algorithm by an order of magnitude.

1.3 Spatio-Temporal Data Management

The widespread adoption of GPS-enabled mobile devices, and rapid growth of spatio-temporal data have driven Location Based Services (LBS) to become a prominent technology. From its earliest application in mobile asset tracking and location navigation, LBS is breaking ground in applications as diverse as location aware search, personalized advertising and weather services, location-based games and social networks.

The important characteristics of many LBS applications are a very high rate of location data generation and a significant number of simultaneous location based queries. These queries can be classified into "historic" or "past" queries, "now" or "present" queries and "predictive" or "future" queries. Examples of "present" queries include nearest-neighbor queries, spatio-temporal range searches and fleet-status queries. Historic queries can be fleet activity, fleet performance, popular routes, driving behavior analysis, finding points of interest etc.

An important feature of the spatio-temporal applications, particularly the LBS, is the very high rate of updates. Depending on the size of the fleet and the update frequency, this could easily surpass over a million location updates per second. Furthermore, the users of these applications could issue thousands of queries at any given time. The majority of these query requests involve recent data. Traditional relational databases were not designed for these use-cases involving spatio-temporal data, particularly, the high update rates. To address these issues, we develop a spatio-temporal data processing system and in-memory indexing technique for LBS. We present our

work in Chapter 6.

The explosive growth of spatio-temporal and spatial data is fueling the emergence and growth of many spatio-temporal applications. Many of these applications are characterized by complex spatio-temporal queries. An important category of such queries is the trajectory-based spatio-temporal topological join queries, which combine a trajectory dataset and a spatial objects dataset based on spatio-temporal predicates. Although these queries have many important use cases, including in Location-Based Services (LBS), they have not received much attention from the research community. To support these queries commercially with a reasonable response time, a restrictive "a priori procedure" must be followed, which includes the uploading of polygons dataset into mobile devices and event tagging. This severely restricts the usage of these queries.

In Chapter 7 we describe several applicable in-memory spatio-temporal topological join algorithms that we systematically implemented. These algorithms *do not* require the "a priori procedure", using existing trajectory index (TB-tree [95]) and spatial index (STR [69]). We show that even the best among these algorithms is long running and not scalable. To address the performance problems of these algorithms we introduce PISTON, a parallel in-memory indexing system targeted for in-memory spatio-temporal topological join. With extensive evaluations, we demonstrate that even the single-threaded performance of PISTON is significantly better than the applicable approaches that use existing trajectory and spatial indexes. Moreover, the parallel performance of PISTON is several orders of magnitude better than these approaches.

1.4 Organization Of The Dissertation

The rest of this thesis is organized as follows. The works related to our research are discussed in Chapter 2. Then we introduce our spatial database benchmark in Chapter 3. The main motivation behind the benchmark is to identify performance issues with spatial and spatio-temporal query processing. We present two high performance spatial query execution systems, namely, Niharika in Chapter 4 and SPINOJA in Chapter 5 respectively. Niharika is a parallel spatial query processing system for the Cloud, and SPINOJA is designed to parallelize spatial join in an in-memory setup. Next we describe two parallel in-memory spatio-temporal data management systems in Chapters 6 and 7. The goal of the first of these two systems is to handle hundreds of thousands of location updates per second and many concurrent range queries. The second system, PISTON is a parallel in-memory approach for trajectory-based topological join. Finally, we draw conclusions in Chapter 8.

Chapter 2

Related Works

First we present works related to database benchmarking in Section 2.1. Then in Section 2.2 we present previous works on spatial data management. Finally, Section 2.3 outlines related research in spatio-temporal data management.

2.1 Database Benchmarking

The objective of benchmarking is to evaluate a system against a reference system in order to compare performance based on various criteria. Database benchmarks can be used to compare different hardware configurations, different database vendor software and different software releases. They play a key role in supporting business decisions on what database to use, as well as database research and development. In this section we review general database benchmarks as well as spatial database benchmarks.

2.1.1 Non-spatial database benchmarks

There are many commercial and free database benchmarks. They differ in their approaches to quantify system performance, workloads and features; we focus here on the most well-known ones. Although the goal of the database benchmarks is to simulate real-world workloads as closely as possible, they may not reflect an organization's actual workload. For this reason, many of the benchmarks choose a few specific business process scenarios and evaluate the databases against synthetic workloads. The TPC benchmarks [121] and DBT [32] are prime instances of this. Others use variations of generic database operations - select, project, join and update to generate the workloads. The Wisconsin Benchmark [14] and Bristlecone [20] are in this category.

The Wisconsin Benchmark was one of the first attempts to develop a scientific methodology for performance evaluation of databases. The benchmark consists of a standard set of queries which measure the cost of different relational operations. Bristlecone is a recent database performance testing utility that is similar in spirit to the Wisconsin Benchmark. Bristlecone provides the ability to create flexible test scenarios by varying the database operations, number of threads, number of tables, number of rows per table etc. Since it is implemented in Java, Bristlecone offers the possibility to support many different databases that have a JDBC driver implementation available. Although it is designed to evaluate database clusters, Bristlecone can also be used on single database instances. Because of its flexibility, we have used Bristlecone as the basis for Jackpine, which we present in Chapter 3.

The Transaction Processing Performance Council (TPC) is a non-profit organization devoted to defining database-related benchmarks. TPC has developed several benchmarks, each suited to a different workload domain, which are generally considered the industry standard by many enterprises. For example, TPC-C is an On-line Transaction Processing (OLTP) benchmark that simulates an order-entry environment. The transactions involved in this benchmark include entering and delivering orders, recording payments, checking the status of the orders, and monitoring the level of stocks at the warehouse. TPC-E is another benchmark that models the OLTP workload after a brokerage firm. TPC-H is a decision support benchmark that is characterized by the execution of queries with a high degree of complexity and concurrent data modifications. Notably, the TPC reports price/performance metrics, highlighting the cost of achieving a particular performance level. None of the existing TPC benchmarks address spatial database workloads.

The popularity of the TPC benchmarks has spawned a number of open source clones. The Open Source Development Labs Database Test Suite [32], known as DBT, is modeled after the TPC benchmarks, but differs in a few areas and is not certified. DBT is among the most comprehensive of all open source benchmark suites. jTPCC [65] and BenchmarkSQL [11] are open source Java implementations that resemble TPC-C. However, they are not fully compliant. TPCC-UVa [72] is an open source implementation that includes the functionalities specified by TPC-C. However, the focus of TPCC-UVa is to compare the performance of different Linux file systems. It also attempts to evaluate relative system performance in multicore CPU systems as opposed to a single core system.

In addition, there are several open source benchmarks that are not modeled after the TPC benchmarks. PolePosition [96] is a benchmark suite that was developed to compare database engines and object-relational mapping technology. OSDB [86] is an open source database test suite that is based on the ANSI SQL Scalable and Portable Benchmark (AS3AP), as documented by Grey [50].

SPECweb2009 [114] is a benchmark to evaluate the performance of web servers when serving both static and dynamic pages. The benchmark itself does not include or specify any of the web server software. However, a common web server architecture uses a multi-tier environment with a back-end database server. Thus, although SPECweb2009 is not a database benchmark, it provides indirect evidence of the underlying database performance.

There are several other benchmarking tools available but they are not as commonly used. None of the database benchmarks discussed so far assess the performance of spatial features.

2.1.2 Spatial Database Benchmarks

Objects in a geographic information system (GIS) are stored in a two-dimensional space and the queries asked of such a system are different from regular SQL queries. Typical geospatial queries [47] include partial matching queries, range queries, nearest-neighbor queries and where-am-I queries. However, as an emerging application domain, novel business scenarios and bleeding edge Web applications are being developed with GIS. Moreover, spatial data features arise in non-geographic applications as well, although GIS is still the largest application area. As a result, one of the key challenges in the development of a spatial benchmark is defining the workloads. Moreover, many spatial functions provided by different relational database vendors are still work-in-progress and the available features across different databases are not always standard offerings. Thanks to the work of the Open Geospatial Consortium (OGC) [84], however, standards for spatial functions have been defined and are being adopted by many databases. These developments make this a promising time to revisit the design of a spatial benchmark.

There have been only a handful of research projects that attempted to benchmark spatial database systems. The best known existing benchmark for spatial databases is SEQUOIA 2000 [117], which was specifically de-

signed to be an earth sciences benchmark and focused on raster data. In contrast, many of today's emerging GIS applications, and the spatial database extensions, operate on vector data. In addition, while the queries specified by SEQUOIA 2000 are all relevant to Earth Sciences, they are all isolated queries rather than a sample Earth Sciences workload. Also, there is no indication that they represent a complete set of queries that would be used in this domain. Finally, SEQUOIA 2000 specifies only the queries, and rules for reporting price/performance. It includes no implementation and no test or timing harness, making it difficult for others to adopt.

VESPA [92] is a vector-based spatial database benchmark that includes a range of query and update tasks that could be executed over synthetic data sets. It was used to compare PostgreSQL with the Rock & Roll deductive object oriented database [44]. VESPA includes a large number of "tasks" (essentially, individual queries) that test update, set union, containment, overlap, intersect, adjacent, search inside, measurement, and analysis operations. Although there are a large number of operations, it is difficult to assess how comprehensively these operations cover the available spatial functions. The use of synthetic data solves problems with scaling, but raises issues of how well it represents real-world data, which can have an impact on measured performance. Finally, VESPA does not include any attempt to model real spatial workloads and is not publicly available.

In recent years there have been very few reports of spatial benchmarking efforts. The few that exist were conducted on an ad hoc basis, simply by executing a few spatial queries against one or two databases and comparing the latency and throughput. Power [97] compared MySQL and PostgreSQL/PostGIS using a small dataset and contrived queries. The author suggested that the study used a dataset that is almost trivial and used rather simple spatial queries. Dynamark [77] and BASIS [51] are benchmarks that are geared towards evaluating spatial index structures, rather than full database systems.

2.2 Spatial Data Management

A spatial data management system is concerned with organizing and representing real-world spatial and geographical features such as landmarks, roads and rivers. A spatial query processing system is essentially a relational database system with enhanced support for spatial data types, spatial indexing and spatial join [52]. Geometric objects are modeled by spatial data types such as point, line and polygon. Topological relations are used to specify how the geometric objects are associated in a two dimensional space. Some of the common relationships, also called spatial predicates, are *Intersects*, *Overlaps*, *Touches*, *Equals*, *Contains* and *Disjoint*. Additionally, spatial analytic functions such as *Distance*, *Dimension*, *Convex Hull* and *Bounding Box* are commonly provided by a spatial database. Recently, the Open Geospatial Consortium (OGC) defined a core set of operations [83].

Among the spatial queries supported by a spatial data management system, spatial join is one of the most complex database operations. It combines two different datasets on spatial predicates. Spatial join is important in many traditional GIS applications, as well as, in many emerging spatial analysis applications. Given the importance of spatial join, a number of research papers explored various ways to improve its performance. Jacox and Samet [62] provides a comprehensive survey of the spatial join techniques. They classified various spatial join techniques into two main categories: external memory and internal memory methods. We adopt their classification for spatial joins. We also consider related works on parallel spatial join, followed by big data infrastructures for spatial query processing. Then, we present works related to performance heterogeneity aware query processing. But, to set the stage for subsequent discussions we start with a discussion on how spatial data is represented.

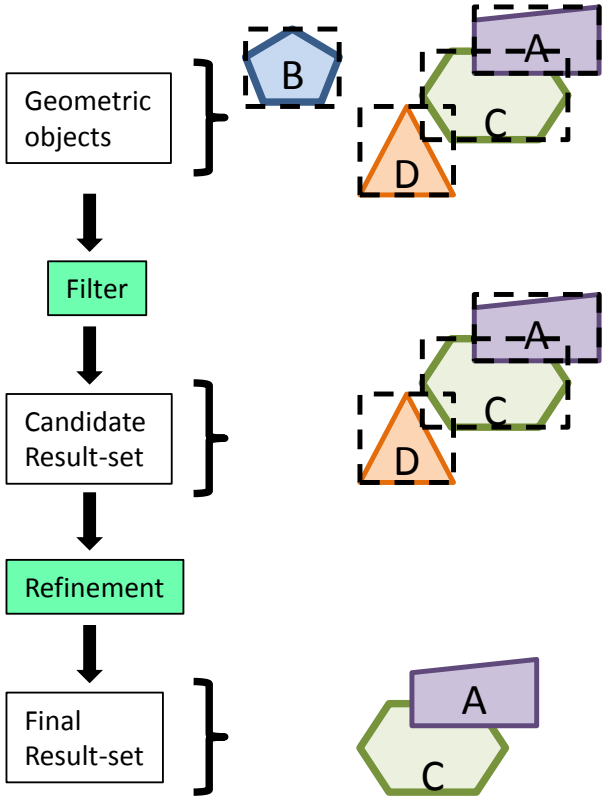


Figure 2.1: MBR representation and the two step spatial query evaluation

2.2.1 Spatial Data Representation

Most spatial applications today use vector data to represent complex map features such as parks and lakes (represented as polygons) or roads and rivers (represented by lines or polylines). Vector data is much more compact than its raster data counterpart, since only the coordinates delimiting the geometric shapes in the map are stored. Vector data also enables the use of sophisticated computational geometry algorithms for common spatial operations such as shape intersection, or distances between points. The results of the queries on vector data can then be overlayed on the high-resolution raster data, that has been cropped to a bounding box encompassing all the results that match the query.

Since spatial query processing is compute intensive, a key technique used in spatial indexing is the use of "approximations". Instead of indexing the exact geometry, an approximation of the feature is indexed. Common approximations are the Minimum Bounding Rectangle (MBR) and multiple bounding rectangles/boxes [108]. Spatial query processing involving the evaluation of topological predicates and distance join follows a two step evaluation mechanism [85]. As shown in Figure 2.1, the first step is *filter*, which returns a superset of the candidate objects satisfying a spatial predicate, by comparing an approximation of actual objects (called the minimum bounding rectangle, or MBR). The second step is *refinement*, in which the actual geometry of the candidate objects are inspected. The filter step tries to eliminate as many objects as possible, since the refinement step uses time-consuming compute-intensive computational geometry algorithms.

The axis-aligned MBR is widely used as the approximation of choice in spatial query processing, because of its concise representation, small storage overhead and the ability to perform fast calculations with MBR. However, this is not the only way to encode an object's approximation. For instance, Brinkhoff et al. [16, 17] proposed

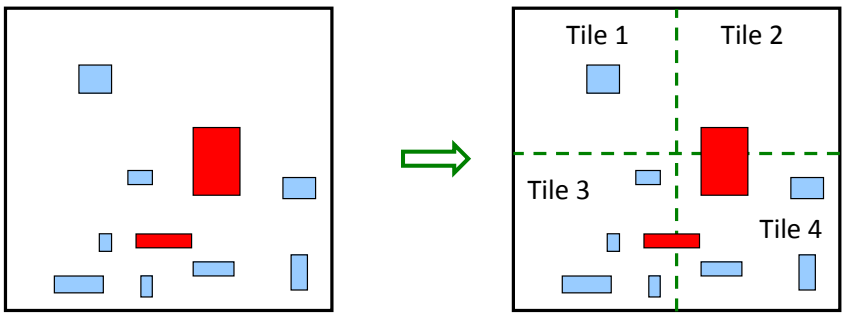


Figure 2.2: Spatial partitioning

several approximations, namely, Rotated Minimum Bounding Rectangle (RMBR), Convex Hull (CH), minimum bounding m-Corner (m-C), Minimum Bounding Circle (MBC) and Minimum Bounding Ellipse (MBE). In their evaluation, they introduced a "progressive filter" step based on these approximations, immediately after performing the filter step with MBR. They reported that the the best results were obtained with the minimum bounding 5-Corner (5C) in terms of high accuracy and low storage overhead.

In many approaches to spatial join (that are described in Sections 2.2.3.2, 2.2.3.3 and 2.2.5), the spatial domain is logically partitioned to create "buckets" or "tiles". This process is known as "spatial declustering". An object's extent may overlap more than one of these spatial partitions. In Figure 2.2, the spatial domain is partitioned into four tiles and the red objects overlap multiple tiles. To accurately compute the spatial join query result, each such object must be replicated to every partition that it overlaps. There exists a few design choices as to how to perform this. In a number of approaches, such as spatial hash join [73], PBSM [90] and Clone Join [91], the complete object is replicated to all the partitions whose MBR it overlaps. On the other hand, in the Shadow join approach [91] the actual object is assigned to a particular tile and the MBR of that object is split along the boundaries of those spatial partitions that is overlapped by it. Another possibility is to assign the actual object to a designated tile and replicate its MBR to all the tiles it overlaps. There are trade-offs in each approach, that we discuss at length later. In Chapter 5, we propose a novel spatial join approach in which the actual object is decomposed by clipping along the tile boundaries based on a few metrics.

2.2.2 Spatial Index

Before describing related works pertaining to spatial join, we start with a survey of the works related to indexing spatial objects. The most well-known spatial index is the R-tree. It is a hierarchical data structure that stores MBR in the internal nodes and is susceptible to degraded search performance due to node overlap and multi-path probing. Node overlap occurs due to the fact that the MBRs of two different internal nodes in an R-tree can overlap. As a result, during the search it might be necessary to walk down multiple subtrees until a match is found. This is the known as multi-path probing. A number of heuristics have been proposed to reduce the node overlap. The R*-tree [10] uses a custom split heuristics by removing and reinserting a portion of the entries when a node overflow occurs.

The ad hoc insertion of spatial objects in the R-tree may result in significant waste of space and inefficient tree organization. If the dataset is static it may be possible to sort the objects based on their spatial coordinates and pack them more tightly. A number of packing approaches have been proposed. The STR (Sort-Tile-Recursive) [69] sorts the records in each axis and splits the space along each dimension into strips having similar number of records. The Hilbert R-tree [66] uses the Hilbert space-filling curve and its good locality properties to sort pages

according to the Hilbert distance metric and group them.

2.2.3 External Memory Spatial Join

The external memory or disk based approaches can be further classified based on whether a pre-existing index exists on one, both or neither of the datasets.

2.2.3.1 Both Datasets Indexed

When both datasets are indexed by the same indexing technique such as R-tree (or variations) it is possible to utilize both indexes simultaneously. Brinkhoff et al. [18] presented a hierarchical synchronous traversal approach, for R*-tree based index, that recursively traverses both the trees in a top down manner. This is based on the idea that the MBR of any internal node is the minimum bounding box of all the MBRs of the children nodes in its subtree. So if the MBRs of two internal nodes N_A and N_B from two different R*-trees A and B do not intersect, then there will be no pair of MBRs in the subtrees of N_A and N_B that would intersect. Otherwise, there might be pair(s) of intersecting MBRs in their subtrees. The basic algorithm compares each pair of nodes from two R*-trees RT_A and RT_B at each level. If there is an intersecting node pair N_A and N_B at the same level, their subtrees are recursively traversed. If the intersecting node pairs are at the leaf level they are reported.

The CPU cost of the basic algorithm described above is rather high, because each entry of one node is compared against all entries of the other node. Moreover, pages corresponding to the R*-tree nodes are read from the disk without any consideration for the I/O cost. To deal with these issues, the paper suggests a number of improvements. The first optimization involves restriction of the search space. Specifically, if N_A and N_B satisfy MBR intersection condition, then only the entries of $N_A.subtree$ and $N_B.subtree$ that intersect the intersecting rectangle $MBR(N_A) \cap MBR(N_B)$ need to be compared. The second optimization involves sorting the entries using spatial locality, such that the pages required to compute the spatial join would be in the buffer with high probability. An additional optimization involves pinning some of the pages in the buffer that are more likely to be needed soon. The evaluation presented by the paper suggests that these optimizations reduce the number of node pair comparisons by a factor of 4 to 8 and the number of disk accesses up to 45%.

The techniques presented above were for two R*-trees having the same height. For two trees with different heights, the authors present the idea of performing window queries on the subtrees rooted in N_B using the same data MBRs of N_A as query rectangles.

2.2.3.2 One Dataset Indexed

When a spatial index such as an R-tree based index exists on one of the datasets A, then an indexed nested loop join based technique can be used. With this approach, the dataset B is iterated over and for each entry $b \epsilon B$ the index on A is traversed to perform a window search based on the MBR of b. However, if the cardinality of dataset B is much higher than that of A, this approach is not very efficient.

Lo and Ravishankar applied the hash-join paradigm into spatial join to introduce spatial hash join (SHJ) [73]. The authors showed the difficulty of directly applying the hash-join into a spatial dataset with the "coherent-assignment problem". Whereas relational hash-join is suitable for handling equi-joins because the hash function creates buckets that fully include the "equivalent classes", it is not the case with spatial join predicates. It is possible for a spatial object to overlap more than one bucket created by a spatial partitioning function. Presented with a few alternative design choices, the authors settled for a variation of multiple assignment in which a partition function may map an object to multiple buckets. This conceptually resembles the approach of replication of

objects that overlap multiple grid cells in PBSM [90]. However if both the inner and outer buckets' extents (spatial dimensions) are equal-sized, non-overlapping and immutable, there is a high possibility of load imbalance, because some buckets may be empty, while others may have many objects. To address this, the authors proposed that the inner bucket extents be single assignment and mutable, whereas the other bucket extents be multiple assignment and immutable.

In their implementation, the number and initial extents of the inner buckets are determined by a bootstrap-seeding process. As new objects are assigned to a bucket, its extent is enlarged such that its MBR contains all member objects. An object is assigned to a bucket whose extent would be enlarged the least. The outer buckets extents are set to the final extents of the corresponding inner buckets and are immutable. An object is replicated to all buckets whose extent overlaps it. If a particular object overlaps no bucket extent it is discarded. This reduces the eventual amount of processing required while joining. In the join phase pairs of inner and outer buckets with the same extents are joined using an indexed nested loop join. This involves constructing a quadratic-split-cost R-tree on the inner bucket and probes the objects from the outer bucket against it. They evaluated their approach against R-tree join (with and without pre-existing indices). With a TIGER/LINE dataset (rivers and railway tracks) the best performance demonstrated was 2X to 3X faster than the R-tree join.

2.2.3.3 Neither Dataset Indexed

When there is no pre-existing index on either of the datasets, partitioning-based join techniques can be used. The general principle is to decompose the spatial universe into partitions and then perform pairwise spatial join for each of these partitions. During the object to partition assignment an object may overlap multiple partitions. There are two approaches to address this: *multiple assignment* and *multiple matching*.

Multiple assignment: This approach replicates each object to all partitions it overlaps with. However, this could be expensive since it increases the amount of computation. Also the result-set must be processed to eliminate duplicates.

PBSM (Partition Based Spatial-Merge) [90] is a spatial join technique that uses a space partitioning. It decomposes the spatial domain into equal sized tiles and maps each tile to a partition using round-robin or hashing. The number of tiles is chosen to be much larger than the number of partitions. In the Filter step for each tuple in a relation the <MBR, OID> pair is appended into a disk file. Then these key-pointer elements of both the relations are read into the memory in such a manner that the set of MBRs from the two relations fit in memory. To do that PBSM determines the number of partitions based on available memory prior to applying the declustering. The key-pointer elements from the relations are loaded into memory for one single partition at a time and MBR-join is performed (Filter) using plane-sweep. The resultant candidate set consists of OID pairs from both the relations. The candidate set is sorted and the tuples (i.e. actual geometry objects) are read sequentially into memory to perform the Refinement step on the join attributes. This also uses a plane-sweep based approach to determine the intersection. After performing Filter and Refinement for all corresponding partition pairs, the matching pairs are sorted to eliminate duplicates from the result-set.

PBSM was evaluated using two different datasets against Indexed Nested Loop Join and R-tree based join. The Indexed Nested Loop Join constructs an R-tree based index on one of the two relations, whereas the R-tree based approach creates an index on both or one of the relations. The key finding is that R-tree join performs the best when both relations have pre-existing index. However, if an index exists only on the smaller of the two relations, PBSM does a better job. When there is no pre-existing index on either of the relations, PBSM performs better than the two other approaches.

Multiple Matching: In order to avoid duplicates, this approach assigns one object to only one of the partitions

that it overlaps. However, this implies that while performing the join each object from one relation must be compared with objects from more than one partitions.

Size Separation Spatial Join (S^3J) [67] is a multiple matching technique that was developed with the primary goal of avoiding replication of objects. To achieve this, the spatial domain is hierarchically partitioned into uniform grids of increasing resolutions. An object is assigned to the level in which it is completely enclosed by the smallest single grid cell. This ensures that large objects are assigned to higher levels and smaller objects belong to lower levels. Specifically, at level l (where l = 0 to L), the D dimensional grid has $(2^l)^D$ cells. An object is represented by the tuple <OID, MBR, Ptr> and at each level the dataset is sorted so that objects are monotonically increasing by the midpoint Hilbert value. To assign an object, the algorithm starts with level L and moves up the levels (having increasing granularity), until it finds the level where the object overlaps only one grid cell.

Two level hierarchies, H_A and H_B are maintained to assign the objects from relations A and B respectively. Finally, the two relations are joined by performing a synchronized scan over the pages of all level files and reading one page at a time. Since each object is placed at a level that is determined by its size, the join processing algorithm can determine which pages are needed at each step. Specifically, a grid cell c_B of H_B is joined with its corresponding cell c_A of H_A and all cells at higher levels (in increasing granularity) of H_A that completely overlap c_A. This process of joining a cell with its counterpart and cells on higher level is repeated on all levels. Therefore, objects at the higher levels will be compared against all objects at the same and lower levels. So, the more objects at the higher levels, the more processing will be required.

The authors also proposed a bitmap structure, called Dynamic Spatial Bitmaps (DSB), to assist in filtering out objects. Conceptually, the bitmap is a compressed representation of the projection of all the levels of a particular hierarchy into a single level. If the bitmap is a projection at level l, its size is 4^l. After the level hierarchy H_A is constructed, a bitmap representing H_A is also created. Then while assigning an object b to level hierarchy H_B, the bitmap is consulted to see if a corresponding object from H_A exists at the same level. If no object exists, then object b can be filtered out.

S^3J was evaluated against PBSM and spatial hash join (SHJ) with several synthetic and real-world datasets. The authors note that the relative performance of these algorithms depends heavily on the statistical characteristics of the dataset. For instance when the replication overhead for SHJ is low its performance is comparable to S^3J. But when this overhead is high S^3J outperforms SHJ by a significant margin. The performance of PBSM depends on the number of tiles used, the amount of replication introduced and the number of pages required by the sorting stage. In general, S^3J with DSB is able to do better than both PBSM and SHJ in most cases.

The Scalable Sweeping-based Spatial Join (SSSJ) [8] is also a multiple matching technique that avoids replication. Since plane-sweep join requires the dataset to be in memory it is suitable to join two relations small enough to fit completely in memory. However, for a pair of relations P and Q that are much larger than memory, SSSJ partitions the spatial domain into n equal width vertical strips. If the MBR of an object is contained in a single strip it is termed as *small*. Otherwise it is *large* and each such object MBR is split into three pieces: two *end pieces* that the MBR intersects with, and a *center piece* in between. The main idea is to compute the intersections between each pair of *center piece* and *small* MBRs from the two relations and then in each strip join the *end piece* and *small* MBRs recursively.

Specifically, each *small* MBR $p \epsilon P$ is assigned to a set LP_n and each *small* MBR $q \epsilon Q$ is assigned to a set LQ_n. Then for all n the intersecting pairs are determined by joining LP_n and LQ_n. A *large* object MBR, that overlaps multiple strips, is assigned to set LP_{jk} and LQ_{jk} for P and Q respectively, where j is the strip where the object starts and k is strip where it ends. While joining LP_n and LQ_n all sets LP_{jk} and LQ_{jk} with $j \leq n \leq k$ is

also considered in the plane-sweep.

The authors evaluate SSSJ against PBSM with a real-world dataset from TIGER and two synthetic datasets that are deliberately built with significant skew. With the TIGER dataset SSSJ performs slightly worse than PBSM. However, with the skewed datasets SSSJ significantly outperforms PBSM. This is because PBSM uses a uniform grid to partition the dataset and hence is very sensitive to data distribution properties.

2.2.4 In-Memory Spatial Join

Keeping in pace with Moore's law, memory capacity has increased significantly. Machines with hundreds of GB, even a few TB, are not uncommon in many organizations that have to process large datasets. With the advent of large main-memory machines, it is now possible to accommodate many spatial datasets in memory. Two approaches that were designed for in-memory spatial join are the nested loop join (NLJ) and plane-sweep join [98]. Although disk-based approaches (R-tree, partitioning, hash based) can be utilized to perform in-memory spatial join, a technique specifically designed to perform in-memory spatial join may perform better than others. Because many disk restrictions, such as R-tree node size and fanout or the size of spatial partition no longer apply, there is more freedom in choosing various design parameters.

These insights led Nobari et al. to develop an in-memory spatial join called TOUCH [82], in the context of computational neuroscience simulation. TOUCH creates an R-tree index on one of the relations. The key idea is to directly assign the objects of the other relation to the index of the first to a non-leaf level where they are fully contained by an MBR. It helps avoid excessive object duplication in other approaches such as PBSM. Also those objects from the second relation that do not intersect any node MBR of the index, can be filtered out. Specifically, to join two relations A and B, a R-tree based index is created on A. The tree is built by starting at level 0 (leaf level) by grouping several objects into a node in the next higher level. Nodes on higher levels are recursively grouped until the process arrives at the root node. Next, the objects $b \epsilon B$ are assigned to the nodes of R-tree index of A, such that each b is assigned to the lowest level of the tree where it overlaps with a single node's MBR. If b overlaps with more than one of the nodes' MBR, then it is assigned to the parent of those nodes. Also, if b overlaps with no node MBR, it can be filtered out. In the next step the actual join is performed where the dataset A is referenced by the leaf nodes and the dataset B by the inner nodes. This involves dividing the spatial domain into uniform grid cells and iteratively joining only those leaf and inner nodes that overlap a particular grid cell. Those grid cells that are not overlapped by any nodes, do not need to be considered.

TOUCH was evaluated against nested loop join, plane-sweep and in-memory implementations of PBSM, S^3J and synchronous traversal. TOUCH showed superior performance over all approaches with synthetic datasets and a neuroscience dataset. The performance of PBSM came very close, however, PBSM required significantly more memory due to object replication overhead. However, a key limitation of TOUCH is that it only works for a particular pair of relations, because of the need to assign the objects of second relation to the R-tree index of the first relation. If there are more than two spatial relations in a dataset, a separate R-tree construction and object assignment must be performed for each pairwise combination of the relations.

2.2.5 Parallel Spatial Join

Since spatial join is a compute intensive operation it is natural to consider parallelizing it. However, only a few have studied parallelizing spatial join.

Brinkhoff et al. [19] presented an R*-tree index based spatial join in a multiprocessor machine with virtual shared memory architecture. Their work extends the hierarchical synchronous traversal approach [18] by par-

allelizing the processing of intersecting pairs of MBRs. Essentially, the filter step is parallelized by assigning matching subtrees of the indexes to each processor.

A spatial join processing task is executed in three phases: task creation, task assignment and task execution. The paper presented a few task assignment algorithms. In the first approach the m intersecting pairs of MBRs at the top level (where the number of nodes, $n \ll m$) are sorted according to local plane sweep-order and the adjacent subtrees are assigned to the nodes. However this causes two or more processors to operate on the same object at the same time and increases I/O due to independently accessing the object from local disks or local buffer. In the second algorithm the previous algorithm is modified to use a global buffer. But instead of assigning adjacent subtrees to each node, they were assigned in a round-robin manner to reduce simultaneous access of the same object from the global buffer. The third algorithm is a dynamic task assignment approach where the first n tasks are assigned to n nodes. When a node finishes its task, the next task (from a globally accessible task queue) is assigned to it. The paper considered two approaches to load balancing by reassigning tasks from a busy node to an idle node. The first is a "buddy node pair" approach and the second involves the node with the highest load. They also considered whether to assign task from all levels of the R*-tree or just the the top level.

The paper evaluates the various approaches to buffering, task assignment and load balancing with two line tables from TIGER dataset. They found that the global buffer with dynamic task assignment offered the best performance and the best speedup was 22.6X for 24 processors. However, this technique does not work for distributed spatial join in shared-nothing architecture, and they mentioned it as their future work.

Zhou et al. [130] presented a grid partitioning based parallel spatial join approach. The paper also addressed workload balancing to deal with processing skew in the parallel Filter and Refinement stages. Any spatial objects that overlap multiple grid cells are replicated to those cells. No index exists on either of the two relations.

Their parallel join processing framework consists of a number of steps. First, the spatial data tuples are evenly distributed among the n processors. Each processor declusters the data using the same grid partitioning scheme. A central processor gathers the information about the local partitions (such as sizes) from each processor and merges them using a bucket merge algorithm. The bucket merge algorithm attempts to balance the filter workload and reduce duplication of key-pointer data (i.e. <OID, MBR> pair) by sorting the data using Z order and then merging N small partitions into n filtering tasks using a greedy algorithm. This process generates grid cell to filtering task mapping information, which is conveyed by the central processor to all the processors. Next, this mapping information is used to redistribute the key-pointer data records among the processors. Duplicate record at each node is eliminated before performing a MBR join to produce the candidate set. The MBR join is performed by joining each pair of small partitions using a nested-loop method. Then the key-pointer candidates are sent back to the home processor (where the actual object tuples reside) in order to request conducting the Refinement. Each processor calculates the potential cost of Refinement (based on the number of vertices) using the received candidates and conveys the information to the central processor. To re-balance the workload for Refinement, the central processor sorts the candidates by Z order and groups them by that order into Refinement tasks such that each task has roughly the same cost. This process again generates cell to tasks mapping, which are sent to the processors. Next, according to this mapping information full objects (with geometry) are distributed among the processors. Finally, each node performs the Refinement by executing a Polygon Intersection Check algorithm.

The authors experimented with several strategies for merging the buckets in the parallel Filter stage, namely, round robin, Z-order greedy and improved Z-order greedy (which considers duplicates) algorithms. Their experiments suggest that the improved Z-order greedy algorithm performed the best in terms of minimizing the variation in bucket sizes. The authors also demonstrated that without proper selectivity information it is difficult to rebalance workload in the parallel Refinement stage. They claimed that redistribution of the actual object pairs based

on a cost metric is more effective to improve overall Refinement performance than doing no load balancing at all. They demonstrated this with an experiment where the response time of the parallel Refinement was significantly higher without any load balancing. We also take a similar position that it is important to perform load balancing to address skew issues in spatial join. In particular, we evaluate several cost metrics and propose a cost based declustering scheme that is discussed in Chapter 5.

The authors of PBSM presented some followup work [91], in which they dealt with parallel spatial join. Two partitioning-based parallel spatial join algorithms, clone join and shadow join, were proposed. In both the partitioning (declustering) approaches the entire spatial universe is divided in uniformly sized tiles. The number of tiles are chosen to be much larger than the number of nodes in the system so that the tuples are evenly distributed among the nodes. Each tile is assigned an identifier and a hash function maps each tile identifier to a node.

Clone join relies on a declustering technique similar to PBSM, called D-W (decluster with whole tuple replication). As the same suggests, entire tuples are replicated or cloned to nodes whose MBRs overlap with the tiles mapped to those nodes. With Clone join both relations are declustered using D-W and locally joined at each node. Finally, the duplicates are eliminated from the result using a `distinct` operator. The local join is very similar to PBSM.

Shadow join uses a different declustering algorithm called Partial Spatial Surrogates or PSS. PSS stores the complete tuple at its designated home node. The parts of the MBR of that tuple that overlap the tiles covered by any node are called *fragment box*. The fragment box and the OID of each tuple (together called *partial spatial surrogate*) are shipped to the destination node. While performing the join, both of the two relations are declustered using PSS and the candidate set is produced by joining partial spatial surrogates. Next, a redeclustering step sends the candidates to their home nodes where they are joined with the first relation. A sort is performed prior to join. Then another redeclustering step sends the intermediate result set to their home nodes to join with the second relation. The PSS declustering is similar to the partitioning approach of Zhou et al. [130]. The key difference is that PSS distributes a portion of an MBR, whereas Zhou et al. replicates the whole MBR to all nodes whose tile MBRs are overlapped by it. Clone join and Shadow join were evaluated first using an analytical model, then with a real dataset in a testbed. The key results are that the Clone join performs better than Shadow join when the join selectivity is high, and Shadow join performs better than Clone join when the Precision (ratio of the cardinalities of candidate set and the final result set) is high. Moreover, the impact of replication probability is insignificant.

Partitioning based spatial join such as PBSM and its parallel version, Clone join, duplicate each tuple to all spatial partitions that it overlaps. Consequently, the same tuple pair from two relations may exist multiple times in the join resultset. So a post processing step is needed to remove duplicates from the resultset, which makes these algorithms blocking. A non-blocking spatial join algorithm, NBPS was proposed by Luo et al. [74]. Their goal was to quickly produce result tuples continuously as they are generated, rather than completing the query execution.

NBPS uses a similar declustering approach as PBSM where the spatial domain is decomposed into tiles using regular grids and the tiles are mapped to the nodes using a hash function. A tuple is replicated to multiple nodes whenever it overlaps the MBRs of the tiles that are mapped to those nodes. The key idea of NBPS is to perform duplicate avoidance rather than post process duplicate elimination. This is done using a revised reference point method. Specifically, for a pair of intersecting MBRs M_A and M_B, four reference points are defined that are the corner points of the MBR intersected by M_A and M_B. To evenly distribute processing load a random number is generated using the reference points as seed and one of the points are chosen based on the random number.

To continuously generate result tuples as they are produced, each node in NBPS maintains two in-memory R-trees, RT_A and RT_B, for the two relations M_A and M_B. Also two disk-resident bucket tables BT_A and BT_B

are maintained each of which contains a subset of the partitions (generated by the spatial declustering function) that are mapped to the node. Each node also maintains two write buffers P_A and P_B of one page each in memory. As tuples are redistributed, each node enters an incoming tuple T_A into its R-tree RT_A and joins it with the R-tree RT_B for relation B. The tuple T_A is checked against the MBRs of all partitions of BT_A and for each matching partition in BT_A, T_A is written into P_A. if P_A becomes full it is written into disk so that it can process tuples from A again. The same previous steps happen for an incoming tuple T_B. After all tuples of A and B have arrived from redistribution, RT_A and RT_B are freed and the corresponding pair of partitions written to disk in the previous phase are read in a random order into memory to perform spatial join. The evaluation of NBPS against parallel PBSM with a Sequoia dataset demonstrated superior performance of NBPS. The good performance was attributed to its duplicate avoidance.

The previous approaches to parallel spatial join were developed before the advent of Cloud computing and hence they do not address the issue of performance heterogeneity. Most of these approaches, except for Zhou et al. [130], focussed on the Filter step. Also, the datasets they used for experimental evaluation were either based on polylines or based on a relatively small set of polygons. Consequently, the issue of processing skew were not featured in their evaluation results. We demonstrate (Chapter 4) that when large spatial datasets involving polygons are used, processing skew becomes a kew bottleneck to performance.

2.2.6 Big Data Infrastructures For Spatial Query Processing

2.2.6.1 Parallel Database Approaches

In the previous section we have mentioned works related to parallel spatial join. Next we describe related works that considered the complete infrastructure for spatial query processing, not just parallel spatial join in isolation. Paradise [89] is the most prominent parallel spatial database. The authors identified the key differences between regular relational databases and spatial databases and the need to tailor parallel database techniques for spatial needs. Paradise provides all the necessary support for parallel spatial query execution. This includes support for spatial data types, storage support to deal with large spatial objects and architectural and software support for spatial join and spatial aggregation query processing.

The architecture of Paradise follows the classic master-slave parallel model with a central query coordinator (QC) node and a number of query processing server (DS) nodes. The basic parallelism mechanism of Paradise is partitioned parallelism. During the first phase of a query execution the tuples of the two relations are partitioned on the join attribute using the same uniform grid in order to bring the candidate tuples together in a single DS node. In order to minimize skew, many more partitions than the number of DS nodes are created. Also tuples overlapping multiple partitions are replicated to the nodes in which those partitions are mapped. In the second phase each of the nodes joins the two partitions that it receives in the first phase. Any single processor spatial join algorithm could be used in this phase. Paradise uses the PBSM [90] spatial join technique described earlier. Finally, the QC node removes the duplicates from the query result before reporting.

Paradise was evaluated with the dataset from Sequoia 2000 benchmark [117]. Also raster images obtained from NASA's AVHRR satellite were used for some of the queries. The scale-up and speedup results in Paradise were mixed. The authors argue that the queries showing poor speedup and scale-up were just those that already ran efficiently on a single node, while queries that were slow on a single node achieved good speedup and scale-up.

Titan [23] is another implementation of a parallel shared-nothing database, dedicated to processing earth science-specific queries. Titan was designed for a specific type of raster-style data set and used for a quite narrow range of earth-science applications. A crucial drawback of Titan is that they achieve high parallelism of I/O

operations by sacrificing data locality. Experimental results show that Titan spends a lot of time in communication, shuffling data blocks between processing nodes. Also, Titan suffers from a high computational imbalance, with a high variation between the minimum and maximum number of data blocks processed on each node for each of the test queries.

In a later work [109], the authors of Titan attempted to address some of its limitations by completely redesigning their system to reduce the high communication overhead. They show better response times, given that queries are much more localized to the processing nodes. However, Titan remains a system dedicated to a particular type of earth science applications (with limited query semantics) and a specific raw raster dataset (the AVHRR dataset) and therefore has many hard-coded design decisions. As a result, Titan may not work as well with other applications and datasets.

T2 [22] is a subsequent parallel database system specifically designed for range queries. The main focus of this work is to provide users with a generic framework for parallel processing of a large-scale multi-dimensional dataset. Similar to Titan, the authors attempt to achieve as much parallelism as possible by pipelining critical operations and placing the data in an efficient manner for range queries. Unfortunately, performance results for the two applications they ported into the T2 framework (Titan and Virtual Microscope) are not encouraging, as they show only comparable or slightly worse performance. Furthermore, although this framework strives to be generic, integrating an application into T2 is left entirely up to the user, which makes it less appealing. Finally, T2 is also specialized for particular data and specific types of range query computations.

2.2.6.2 Map-Reduce Based Approaches

The Map-Reduce framework was first developed at Google to perform text processing in order to optimize text search. With the advent of big-data processing infrastructures based on Map-Reduce framework, such as Hadoop, there have been a few attempts to perform database query processing tasks with them. As such these systems lack many of the functionalities of a full-fledged database system. It has been demonstrated [93] that these Map-Reduce based systems are significantly slower than parallel databases.

SJMR [129] is a system that parallelizes spatial join using Map-Reduce on clusters of commodity machines. It is targeted for datasets without any spatial index. They also identify the importance of spatial partitioning for good load balancing. The spatial domain is decomposed into N regular tiles, where $N >> P$, P being the total number of partitions. Each tile is traversed using Z-order and mapped to a partition in a round-robin fashion. In the Map stage each spatial object's MBR is checked against all the tiles and is redistributed to the tiles with overlapping MBRs. The key of a generated record is set to be the partition id and value to be a tuple $\{$OID, MBR and spatial properties$\}$. In the Reduce stage the actual filter and refinement steps are performed. In the filter step the value tuples for the same partition are read for each of the two relations and the tuple "pairs" (T_R and T_S) from the relations are joined using in-memory plane-sweep. SJMR splits each partition into strips and uses a variation of the plane-sweep called "strip-based plane sweep". To avoid random seeks in the refinement step the OID pairs are sorted and then the actual geometries are read from disk sequentially and compared. To avoid duplicates SJMR uses a reference point based method. If a pair of objects (T_R and T_S) are replicated into several partitions and consequently at several Reduce tasks, they are only processed by the Reduce task where the Common Smallest Tile lies in.

SJMR was evaluated with two relations (Roads and Hydrography) created out of the TIGER/Line dataset. The experiments were conducted on a cluster of 8 machines with Hadoop 0.18.1. The results show a slight reduction in execution time over a parallel implementation of PBSM, when joining the two relations. The best speedup achieved by SJMR was about 6X on the 8 node cluster. One key issue is that SJMR requires that partition data fit

in main memory at each node. This may not happen even for many moderately sized datasets on machines in the Cloud.

Recently, Aji et al. produced a Map-Reduce based system for medical image processing, which also focuses on spatial join operations [5]. It consists of several components, namely, the SQL to Map-Reduce compiler YSMART-S, the spatial query engine RESQUE, and the Hadoop core. Like other space-partitioning based approaches, each medical image is decomposed into N fixed sized regular tiles. However, unlike normal spatial partitioning approaches that use replication for objects on the boundary of the tiles, they discard those objects. They argue that this is done for the sake of simplification and the missing objects do not significantly impact the final result.

RESQUE creates a R*-Tree based spatial index on-the-fly and performs a R-Tree based synchronous traversal [19] to join two datasets. Given two R*-Tree indexes, the algorithm recursively traverses each pair of nodes from the two indexes starting from the root nodes. When the MBRs of a node pair intersect, the traversal continues down to their children until the leaf nodes are reached. Then the actual geometries indexed by the two leaf nodes compared. They evaluated STAR and CLIQUE spatial join performance with a dataset of 18 images with diverse disease stages, where each nucleus is represented as a polygon in vector format. Their system consisted of 10 physical nodes with 192 cores in total. The number of Reducers ranged from 20 to 200. However, results are reported starting at 20 reducer nodes, so speedup and parallel efficiency against a sequential implementation cannot be calculated.

With the STAR join query, their system had better speedup against PBSM for low cardinality joins and worse performance for high cardinality joins. For the CLIQUE join, their system showed better speedup against PBSM. They also observed issues with skew in the dataset and to address that they proposed a simple approach where tiles are greedily assigned to partitions based on an estimated completion time.

2.2.6.3 Hybrid Approaches

HadoopDB [2] is an architectural hybrid between the Map-Reduce system Hadoop [55] and traditional relational database. It follows a master-slave model of parallel task processing and uses Hadoop infrastructure to manage the slave or service nodes. Hadoop also provides support for the recovery from failure for a query execution task, which is not typical of a traditional parallel database system. On each service node an instance of a relational database, such as PostgreSQL, is run. HadoopDB requires the data to be partitioned using hash or range partitioning. Direct support for spatial partitioning is not provided by HadoopDB. It also does not provide complete support for join. Although HadoopDB does not directly support spatial join, we mention it for completeness.

2.2.6.4 Performance Heterogeneity-Aware Query Processing

Modern data-centers have evolved as very large distributed systems built from hundreds of machines from multiple hardware generations. Performance heterogeneity among these machines occurs naturally, as disks or nodes fail and are replaced with newer components. The same effect occurs in smaller clusters within an organization. Cloud computing adds a new dimension, in that the infrastructure itself is no longer fixed. Cloud infrastructure providers, such as Amazon EC2, offer different instance types, each with different processing and storage capacity. Furthermore, due to virtualization overhead and load consolidation, different machines of a single instance type may vary widely in performance. Performance heterogeneity is a norm [106, 43], rather than an exception, in these platforms.

Although the issue of performance heterogeneity has received widespread attention [128, 4] in distributed data

processing systems such as Map-Reduce, not much work has been done in parallel databases in this context. The challenges with parallel query processing in a heterogeneous cluster were recognized by Mayr et al. [75]. They proposed a query execution engine framework that takes advantage of both partitioning parallelism and pipeline parallelism. To this end they utilize several load-balancing techniques. The first technique migrates operations, such as selection, projection and User Defined Functions (UDFs) from overloaded servers to less busy nodes. The second technique migrates part of the join processing, such as the sort phase in a sort-merge join, to a different node. The task of data partitioning could also be migrated. Other techniques include selective compression to trade off CPU bandwidth with network and rerouting data-streams from overutilized servers to a third server. A drawback of the paper is that it provided a very general description of these techniques, but no specific algorithm is given that determines when and how the load-balancing is performed. They evaluated the migration of a UDF operation and simulated the performance heterogeneity by varying the UDF processing cost. But their evaluation was limited to a 2 node system and the workload was a traditional workload (non-spatial).

Most traditional parallel database systems [34] do not deal with straggler nodes, as the underlying assumption is that the member nodes have identical performance. HadoopDB [2] deals with straggler nodes by launching speculative tasks, but its speculative task execution model (inherited from Hadoop) may lead to critical performance issues in heterogeneous environments. Although Zaharia et al. [128] introduced a new scheduling algorithm for speculative task execution, any speculative task could prove to be detrimental to performance of a parallel spatial query execution system.

Spatial database workloads are more compute-intensive than traditional database workloads [113] and the amount of computation required to evaluate a spatial join predicate can vary widely for different tuples. Thus, assigning the same number of tuples to each node, even when the nodes are identical, does not guarantee good load balancing in a Cloud setting. Due to this processing skew, the performance heterogeneity may get aggravated. The Big Data infrastructures discussed so far, do not address the performance heterogeneity in the context of spatial join, even though this is an important issue in the Cloud.

2.3 Spatio-Temporal Data Management

With the rapid growth of spatio-temporal data and increasing popularity of Location Based Services (LBS), it is becoming important to deal with the very high update rates typical in LBS. A key issue in this regard is spatio-temporal indexing. A significant body of research exists on the topic of spatio-temporal indexing. These indexing methods can be classified by whether they support indexing historical, present, predictive or combined past, present and future data. A popular benchmark [25] evaluates a number of prominent present or predictive indexing methods. A comprehensive survey of recent developments in spatio-temporal access techniques can be found in Nguyen-Dinh et al. [79]. We provide a brief survey of the combined past, present and future indexing methods, due to their relevance to our work.

The combined spatio-temporal access methods that have been proposed can be roughly classified into two categories: external access methods and main-memory access methods. We present related works in each category in the next two sections.

2.3.1 Spatio-Temporal Index

2.3.1.1 External Access Methods

Indexing moving objects has been an active area of research for the past decade. As the importance of LBS systems grew, a number of disk based techniques were developed to index location based dataset. B-tree and R-tree based indexing techniques are among the most notable external access based approaches. First, we present the previous works on B-tree based approaches and then the R-tree based approaches.

BB^x-index [70] is a B-tree technique that extends B^x-tree [63]. B^x-tree converts the object locations into 1D Space Filling Curve (SFC) codes and uses B-tree to index these locations. BB^x-index maintains a forest of B^x-trees, each tree for a separate timestamp interval. STCB$^+$-tree [71] is another B-tree access method. It maintains a compressed B$^+$-tree, called TCB$^+$-tree, to index the temporal dimension and one compressed B$^+$-tree, termed as SCB$^+$-tree, for each spatial dimension. To answer a query the TCB$^+$-tree and SCB$^+$-trees are searched separately and the final results are the union of all outputs.

A 3D R-tree could be used as a spatio-temporal index. The time dimension in such a tree can be viewed as another dimension and it can be integrated into the tree during the construction process. This allows modeling the movement of objects as distinct boxes in a three dimensional space. When an object moves into a different position a new box is created to represent its new position and temporal extent. If the objects indexed by the 3D R-tree move rapidly, as the vehicles on a highway do, there would be a lot of nodes created, depending on the spatial extent of the boxes. So 3D R-tree can be very inefficient with updates and so a direct implementation of this is not useful.

An interval query in a 3D R-tree is also modeled as a box representing the temporal interval and partial extent of the query. Since there is a single tree for the entire history, *short-interval* query performance may suffer, as it depends on the total records, rather than the number of entries active during the query interval.

Among the R-tree based techniques, Historical R-tree (HR-tree) [78] is one of the earliest. It generates a new logical R-tree for each update, which leads to significant space usage and poor performance. The index structure maintains an R-tree for every timestamp, but common branches of the consecutive trees are stored only once in order to save space. When the position of an object o changes at time t_1, its old version o_0 needs to be deleted from the tree for timestamp t_1 and the new version o_1 needs to be inserted into the tree. To achieve this, the changes are propagated to the root of the tree. This may causes some of the existing nodes to be duplicated. So even if a single object changes its position, the entire path may need to be duplicated.

A *timestamp* query is processed in HR-tree by directing the query to the corresponding R-tree, where an ordinary window query is performed. To process an *interval* query, the corresponding trees within the time interval are involved in the search process.

The Multi-version 3D R-tree (MV3R-tree) [118] attempts to address some of the issues with HR-tree and 3D R-tree. The basic idea is to maintain a combination of two independent structures, a Multi-version R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaf nodes of the MVR-tree. Each entry in an MV3R-tree node is a tuple $< S, t_{start}, t_{end}, ptr >$, where S is the spatial MBR, t_{start} and t_{end} denote the times when the entry was inserted into and deleted from the index respectively. An entry is alive at timestamp t if $t_{start} \leq t \leq t_{end}$. When a new entry is inserted into a full node, a *block overflow* condition occurs. This triggers a *version split* in which the live entries are copied to a new node, with their t_{start} modified to the insertion time. If the total number of live entries in the new node are above a threshold, a *strong overflow* occurs that results in a *key split* of that node. Whereas the version split is always performed for intermediate nodes on a block overflow, MV3R-tree attempts to avoid this split for leaf nodes. This helps in reducing the space usage, as the leaf nodes constitute a

large proportion of all nodes.

When an entry is deleted its t_{end} is changed from * to the current timestamp. Only when the t_{start} is equal to current timestamp (which may be caused by multiple updates at the same timestamp), the entry is deleted physically. On deletion of an entry, if an *underflow* condition occurs an attempt is made to acquire a live entry from its sibling node. The auxiliary 3D R-tree is built on the leaves of MVR-tree to facilitate answering interval queries. Any change made to a leaf node of the MVR-tree is propagated to the corresponding entry in the 3D R-tree.

Processing timestamp queries involves a two step process: determining the root node of the MVR-tree whose jurisdiction interval encompasses the query datestamp and then performing a local search in that tree. Since 3D R-tree is more efficient than MVR-tree with long interval queries, an interval threshold is used to determine whether to use the 3D R-tree while answering interval queries. A particular problem of duplicate visits to the same node via different parents may occur if the interval query processing involves multiple trees. Such redundant visits may significantly increase the cost of query processing. The paper suggests the idea of *postponed visit* of the branches of a shared node if it is detected that some branches would be visited in near future. Evaluation studies presented by the paper suggest that MV3R-tree performs better than 3D R-tree and HR-tree for various query answering scenarios.

The STR-tree [95] uses R-tree to index moving object trajectories that are represented as connected line segments. The stated goal of the index is to be able to deal with trajectory based queries (involving the topology of the trajectories), in addition to the regular coordinate-based queries (point, range and nearest neighborhood). To this end STR-tree organizes the line segments, traced by the object movements, not only by their spatial properties, but also according to the trajectories they belong too. They call this approach *trajectory preservation*. Consequently when inserting a new line segment STR-tree attempts to insert it as close as possible to its predecessor in the trajectory. An insertion in the tree involves finding the node that contains its predecessor. If the returned leaf node is full, the insert algorithm checks if the $p-1$ parents are full. In case one of them is not full, the leaf node is split. But if all of the $p-1$ parents are full, the insert algorithm expands the search path on the subtree including all nodes further to the right of the current insertion path. Finally, a regular R-tree insertion process is followed ($ChooseLeaf$ and $QuadraticSplit$). The split algorithm attempts to put more recent segments into new nodes.

For the coordinate-based query processing, STR-tree uses the classical range query approach of R-tree. A combined query (involving coordinate-based and trajectory-based) is processed in two steps: an initial set of segments are selected based on spatio-temporal query range and then for each of the found segments from the previous step attempts are made to find its connected segments. The performance evaluation of STR-tree suggests that it does better than R-tree only when the number of trajectories is small. STR-tree would perform poorly when the trajectories are long, which would naturally occur as time progresses.

The PPFI [42] indexes a road network using a 2D R*-tree. The time interval of past trajectories along a road segment is indexed by a 1D R*-tree. Specifically, the road networks are indexed by a R*-tree R_{road} and for each of its leaf node entry an R*-tree R_s is maintained. In addition, a hash structure H_{data} stores the most recent update time, position and velocity.

Processing a new location update by the PPFI index involves several steps. First, the status of the object is updated in H_{data} if the location changed. If the object is still in the same polyline (of a particular road segment) as a previously reported update, a new leaf node entry is created in the same R_s. However, if the new position is not in the same polyline as before, two entries E_1 and E_2 are created. Entry E_1 corresponds to the previous polyline and is inserted into R_s in the same way as before. Entry E_2 corresponds to new polyline and it records the trajectory from the start point of the new polyline up to the position that was just reported.

Answering a range query in PPFI involves first finding the polylines within R_{road} that intersect the spatial extent of the query. Then for each matching leaf node the corresponding R_s is examined to see if the temporal query interval matches. Answering predictive queries are done using the H_{data} by applying a linear predictive model.

2.3.1.2 Main-Memory Access Methods

Since B-tree and R-tree are both external access methods they are not scalable when it comes to supporting a very high rate of updates. R-tree based approaches, in particular, perform poorly with update-heavy workloads due to the need to maintain the target structure constantly. B-tree based access methods on the other hand perform worse than those of R-trees for query-rich workloads, since R-trees can prune the search space much more efficiently than B-trees. With spatio-temporal workloads involving millions of moving objects and thousands of simultaneous queries, neither approach can sustain high throughput.

In Šaltenis et al. [125] the authors reported that the use of grids could offer superior performance for update-intensive workloads. The primary purpose of the paper was to compare update and query performance of R-tree and grid based structures in the context of main-memory index evaluation. It has been demonstrated that R-tree based indexing is not update efficient, because a single update operation results in four tree traversals. To address the limitation of top-down traversal (such as in an R-tree), the authors presented the case for bottom-up updates by leveraging a secondary index. Such an index, that could be based on hashtable, provides a direct access to the object's data in the primary index. The bottom-up update approach takes advantage of the *update locality* property, in which the next update for an object is likely to be close to the previously reported position. It can avoid primary index scanning for a significant portion of the overall updates.

In the case of a grid, the grid cell determined by the current update location can be compared with the previously determined cell by inspecting the secondary index. If they are the same, the previous object data can be overwritten with the new location data without any primary index scanning. If the newly determined grid cell is different from the previous one, it is necessary to delete the object from the previous grid cell and move it into the new grid cell. The authors argue that an R-tree based indexing can also benefit from secondary-index-based bottom-up updates. With a number of experimental settings the paper demonstrates that when it comes to query processing, a grid based index augmented with a secondary structure is competitive with the query performance of R-tree.

Due to the availability of large main memory machines, a few recent research projects focussed on in-memory indexing techniques. The MOVIES approach [37] enables concurrent queries and updates by letting the queries run on a read-only copy of the index structures, while a newer version of the index is being built from an update buffer.

Drawing an analogy with a cinematographer, MOVIES creates a snapshot index called an *index frame* such that each snapshot is active during a short interval called *frame time*. During each frame time T_i, all updates are collected in update buffer U_i and all queries are answered with the read-only index I_{i-1} built in the previous frame time. Additionally, a new read-only index I_i is built based on the updates collected in the update buffer U_{i-1} during T_{i-1}. As soon as the index I_i is completely built, a new frame is started in which all incoming queries are handled by I_i.

In order to handle high update rates the update buffers in MOVIES must be organized such that they are fast and their total size does not exceed the available memory in the system. Since the update buffers may contain several different updates from the same object, for each object the buffer only stores the most recent position. The aggregation buffers are implemented as arrays where slot position i stores object aggregate $i = oid$. The

read-only index is organized as virtual kd-trie where each indexed object's Z-order location code is computed and the codes are stored in a sorted index.

MOVIES supports predictive range queries that ask for the predicted positions of moving objects at a future time t_q. The paper presents two approaches to compute the predicted position from the actual position: during the indexing time (PI) or querying time (NPI). In the PI approach, for each incoming update the predicted position of a moving object is computed and indexed. In the NPI approach, this computation is postponed and the index is built based on the update datestamp. However, the query window needs to be enlarged accordingly to answer a predictive query in this approach.

The paper presents a number of evaluation studies, demonstrating that MOVIES achieve high update throughput. However, the key drawback of this remains that it requires very frequent rebuilding of short-lived index structures, which results is a significant wastage of CPU resources.

The PGrid approach [126] avoids the "stop the world problem" of MOVIES by using fine-grained concurrency control mechanisms and two separate index structures. PGrid maintains a grid-based directory structure and an additional secondary hashtable based structure. The main idea is to service the queries using the grid index, and to use the secondary index for the updates in a bottom-up fashion. The spatial domain is statically partitioned into uniform grid cells and the grid is represented by a two dimensional array. No object is stored in the grid directory, but rather object data is stored in linked lists of buckets. A pointer to the first bucket of such a list is in a grid cell. The object data consists of a four tuple $< oid, longitude, latitude, datestamp >$. The hashtable (secondary index) indexes objects by their oid. An entry in this hashtable is a tuple $< oid, cell, bckt, idx, ldCell, ldBckt, ldIdx >$. The pointers $cell$, $bckt$ and idx point to the current cell and bucket an object belongs to and its offset in the bucket respectively. The pointers with ld prefix represent the previous object location in the grid index.

An incoming location update could be local or non-local. If the object's new position remains the same as its currently stored position the update is local, otherwise it is non-local. For local updates, there is no explicit object deletion and the updater can just overwrite the previous object data with the new data. However, for non-local updates it is necessary to maintain the previous object positions. This is due to PGrid's "freshness semantics", that guarantees to include in the result any object that moves during update processing from a grid cell yet to be scanned by a query to a cell already scanned. To deal with this issue, a non-local update does not actually remove an object's previous position, but rather it is marked for deletion (called *logical deletion*). A logically deleted position is deleted physically during a subsequent update.

To process a new update the grid cell is computed using its actual position. The secondary index lookup is performed to retrieve the primary index related information. If the update type is local, the outdated object data is overwritten with the new data. If it is non-local, the new object data is inserted into the new cell and the old data is logically deleted. Moreover, if that object was previously logically deleted, that information is physically deleted.

Processing a range query involves determining the grid cells that fully and partially overlap the query window. Only the objects in the partially overlapped cells are inspected to see if they are within the query range. To distinguish multiple copies of the same object, PGrid uses the update timestamp. Specifically, a positive timestamp signifies the most up-to-date object location, whereas a negative timestamp indicates the previous location. The object data with previous location can be included in the query result if the object was updated after the query started and query has not yet seen the new updated position.

To support multiple updaters and concurrent update and query processing PGrid uses synchronization techniques. The paper conducts experiments with several synchronization mechanisms, including mutex lock, read-write mutex lock, spin lock, atomic latch and optimistic lock-free traversal. Among them the spin lock performed

the best in a multi-threaded scenario. In an architecture with SIMD technology support, SIMDized reads and writes were the most efficient. Experimental evaluation of PGrid demonstrates that it outperforms the snapshot based approaches in various scenarios.

MOIST from Google [64] is a full-fledged LBS system. It employs traffic shedding to reduce updates and save storage. MOIST is not strictly an in-memory approach. However, it maintains m records in memory for each object and it archives aged data to disk. This helps memory caching of recent history for all objects.

MOIST performs traffic shedding by a technique called "object schooling", in which nearby objects with similar moving patterns are clustered into one school. For each object school (OS) it only keeps track of the leader object, and records the distance offsets of the followers from the leader. The main table in MOIST is the *Location* table that stores the location and velocity information of each object. To keep track of the object schools MOIST maintains the *Affiliation* table that supports two basic operations: to check if an object is a leader or follower and to calculate a follower's displacement from its leader. There is also a *Spatial Index* table to index the objects. The spatial index is called S2Cell index, which linearizes the 2D domain into uniform grid cells using Hilbert SFC encoding and indexes them using a B-tree. The underlying implementation of the tables is based on Google's distributed key-value storage system called BigTable [24].

The updates are handled by determining whether the object is a leader or a follower. If it is a leader, the corresponding entries in the Location table and the Spatial Index table are updated. If the object is a follower and its estimated location is within a threshold distance from the leader, the update is shed. Otherwise the object is considered to have left the school and it is transformed into a leader by updating the tables.

To archive aged data double buffering could be used, in which updates take place in one memory buffer while the other buffer is flushed to disk. However since the disk latency is much higher than memory, it may not be possible to ensure that the time it takes to flush the buffer to disk is less than the time it takes to fill the other buffer in memory. Therefore, MOIST uses a parallel ping-pong system that uses locality-preserving buffer partitioning and parallel disks.

The existing approaches to external access methods fall short in terms of update throughput and concurrent query performance. Although the main-memory access method PGrid [126] addresses some of the issues, it only supports present queries, and not historical queries or predictive queries. Our indexing system (described in Chapter 6) supports all three types of queries and at the same time offers very high update throughput.

2.3.2 Spatio-Temporal Join And Trajectory Index

The spatio-temporal queries were classified into two main categories by Pfoser et al. [95] : coordinate-based and trajectory-based. The former type of queries deal with the selection of all objects with respect to a given range, and examples of these queries are range queries and nearest neighbor queries. The trajectory-based queries on the other hand deal with properties of the trajectory of each individual object, such as topology and direction. The trajectory-based queries were further classified into topological queries and navigational queries. The trajectory-based topological join queries deal with part or all of the trajectory of each object from a trajectory dataset and combine this dataset with a spatial object dataset based on a spatio-temporal predicate such as enters, leaves and crosses. The spatial objects are typically polygons.

Most of the past research projects in spatio-temporal query and indexing addressed coordinate-based queries. A survey of recent developments in such indexing approaches is presented by Nguyen-Dinh et al. [79]. The few projects that looked into spatio-temporal join dealt with problems that are orthogonal to trajectory-based topological join. For instance, Chen and Patel [26] proposed a framework for trajectory distance join and trajectory *k* Nearest Neighbor join between two trajectory datasets. Iwerks et al. [61] introduced an algorithm to maintain a

dynamic view of the "spatial semijoin" results as time progresses. However, their focus was on the present time. Bakalov et al. [9] addressed the problem of identifying all pairs of similar trajectories between two datasets using symbolic representation.

The work that most closely relates to ours is that of Pfoser et al. [95]. They introduced two spatio-temporal indexes TB-tree and STR-tree, both based on R-tree. They showed that TB-tree performed better than STR-tree with trajectory-based queries. The main idea behind TB-tree is to bundle segments from the same trajectory into the leaf nodes of the R-tree. In their paper they focussed on range and time-slice queries, but provided no evaluation or result with the topological queries. The Scalable and Efficient Trajectory Index (SETI) [21] is another trajectory index that uses a two level organization to separate the spatial from temporal indexing. SETI partitions the 2D space into disjoint hexagon cells that remain static during the structure's lifetime. Like Pfoser et al. [95], the SETI paper only evaluated range and time-slice queries. Recent works, including [54, 80] addressed the problem of how to optimally split trajectories for the purpose of improving range query performance. However, none of the indexes addressed trajectory-based topological join. In our work (Chapter 7) we present a high performance indexing system that is specifically designed to deal with this.

Chapter 3

Benchmarking Spatial Queries

3.1 Introduction

With the rapid growth of spatial and spatio-temporal data and the emergence of novel applications many questions arise that are to relevant performance and scalability of spatial databases. Developers or organizations deploying spatially enabled applications want to evaluate which spatial database will support their needs in the most cost-effective manner. Database researchers and developers want to know which features or operations limit performance in real use, to focus the design and implementation of new algorithms. Finally, systems researchers and architects want to understand the stresses imposed on processor, memory, and storage resources by spatial database workloads. Understanding these questions is the key to the design and implementation of new algorithms. Some of these questions involve performance and systems design aspects. For instance, which features or operations limit performance in real use, or how different spatial database workloads stress the processor, memory, and storage resources. These types of questions are the purview of benchmarking. Unfortunately, there has been no widely used, industry standard spatial benchmark. The best known existing benchmark for spatial databases is SEQUOIA 2000 [117], which was specifically designed to be an earth sciences benchmark and focused on raster data. In contrast, many of today's emerging GIS applications, and the spatial database extensions, operate on vector data. In addition, while the queries specified by SEQUOIA 2000 are all relevant to Earth Sciences, they are all isolated queries rather than a sample Earth Sciences workload.

We developed a spatial database benchmark, Jackpine, [103] to fill this gap. Sim et al., in a study on the value of benchmarking, noted that the adoption of a benchmark fosters both community building and technical advances in a research area [112]. Our aim is to provide the basis of such a benchmark, providing both micro benchmark coverage of basic spatial operations as well as modeling a number of real-world applications for spatial databases. One of our goals in this regard is to see how low-level systems software is exercised by spatial database workloads. We believe this affords some advantages in the design of a benchmark, as we have no bias toward (or against) any particular implementation or application. Our hope is that Jackpine will provide the basis for ongoing discussion and refinement within the community, leading to a benchmark that will be widely used.

Due to the rising data volume and the growing popularity of spatial applications, most traditional relational database systems (RDBMS) offer spatial features. However, the degree to which spatial functions are supported in RDBMS varies widely. For example, true geodetic support (i.e., support for true measurement along a spherical coordinate) is offered in Oracle, DB2 and SQL Server, but not by MySQL, or Ingres. PostgreSQL has true geodetic support only for point-to-point non-indexed distance functions. Informix provides both a no-cost Spatial

Avg Duration	Avg Ops Sec	Count Of Resultset	Duration	Iterations	Other Exceptions	SQL Exceptions	Warmup Count	Warmup Duration	DBType	DBUser
0.236	4.237288	310.0	0.708	3.0	0.0	0.0	1.0	0.972	MySQL 5.0.91	mysql
0.945333	1.057827	80.0	2.836	3.0	0.0	0.0	1.0	2.415	PostgreSQL 8.4.2	postgres
1.954666	0.511596	80.0	5.864	3.0	0.0	0.0	1.0	2.759	Informix	informix

Figure 3.1: Benchmark output report

module (which lacks geodetic support) and a Geodetic module (which is not free). In terms of spatial functions, SQL Server, Oracle, DB2, Informix and PostgreSQL support spatial predicate functions defined by the Open Geospatial Consortium (OGC) [84], as well as custom geodetic functions. MySQL only supports OGC functions using minimum bounding rectangles (MBRs). Support for spatial indexing also varies, with most of the spatially enabled databases offering the R-tree index [53], except for SQL Server which provides a Grid. PostgreSQL supports a generalized R-tree index called GiST [58]. Finally, the cost of these different options also varies widely.

3.2 The Benchmark

One of our goals in developing Jackpine is to be able to support as many different databases as possible, with minimal effort. We believe this is a key feature if the benchmark is to achieve widespread adoption by other users. Another objective of Jackpine is to include a comprehensive set of workloads, that would include basic spatial operations on one hand, and representative real-world applications on the other. To this end, Jackpine consists of two different parts: a micro benchmark and a macro benchmark. The micro benchmark is comprised of a number of spatial join queries with topological relationships, several queries with spatial analysis functions, and data loading queries. The macro benchmark includes queries that are modeled on 6 different real-world spatial applications.

3.2.1 Implementation Overview

To leverage the development effort of existing database benchmarks, we evaluated a number of open source options. Our motivation was to reuse the test harness or part of the implementation of the chosen benchmark. In our evaluation, Bristlecone [20] fared better than the others, in terms of support for open-source databases, available features and extensibility. Bristlecone allows running test cases with systematically varying parameters. Also, Bristlecone is written in Java, allowing it to be used with any database that provides a JDBC driver implementation. We thus chose to use the test harness of Bristlecone as the basis of our own implementation.[1]

A key concept in our benchmark is the spatial scenario, which is an extension of a scenario in Bristlecone. A spatial scenario is essentially a test case that includes SQL queries and the specification of various parameters. A Java class implements the functionalities relevant to a particular spatial scenario. The run-time parameters such as the number of warmup runs, number of iterations, number of threads, width and length of data selected, databases to run against, etc. are specified in a properties file. Database-specific support for a particular scenario is provided by specifying the SQL query in the corresponding SQL dialect of that database. A Java class implements the SQL

[1] This is reflected in the name Jackpine.

Table 3.1: Database tables used for micro and macro benchmark

Database table	Geometry	Cardinality	Data file size (KB)	Index file size (KB)	Scenario
edges_merge	line	5895060	1651629	416972	Micro benchmark, Reverse Geocoding, Map Search and Browsing and Toxic Spill
pointlm_merge	point	13441	920	985	Micro benchmark, Map Search and Browsing
arealm_merge	polygon	5963	28490	1644	Micro benchmark, Map Search and Browsing
areawater_merge	polygon	374053	200734	26144	Micro benchmark, Map Search and Browsing
gnis_names09	point	103413	10225	10562	Map Search and Browsing
s_fld_haz_ar	polygon	3403	70339	267	Flood Risk Analysis
s_gen_struct	line	938	93	71	Flood Risk Analysis
land_use_2006	polygon	404599	162400	36327	Flood Risk Analysis and Land Information Management
parcels2008	polygon	321578	131808	25179	Land Information Management
hospitals	point	48	7	5	Land Information Management
s_wtr_ar	polygon	3	2813	4	Land Information Management
frontyard_parking_restrictions	polygon	76	321	7	Land Information Management
landfills	point	79	30	9	Land Information Management
geocoder_address		2587672	214835	8653	Geocoding
cityinfo	point	41660	5836	2974	Reverse Geocoding

dialect for each supported database. The benchmark is packaged as a set of libraries, configuration properties files and scenario properties files. A Unix shell script invokes the scenarios and executes them. The script can take additional parameters (such as the databases to run against), which is useful if a user wishes to override the parameters specified in the scenario properties files.

After the end of the execution of each scenario, an output report is generated. For each database, the output report includes information about the scenario execution, such as, the number of iterations, average duration of the iterations, average operations per second, total duration of the iterations, number of records returned by the scenario queries, number of warmup runs and total warmup duration. A screenshot of the output report from a micro benchmark scenario run is shown in Figure 3.1.

Like Bristlecone, Jackpine can support any database with a JDBC driver implementation. For the purpose of conducting the evaluation study, we have implemented support for three databases: PostgreSQL, MySQL and Informix. Although Jackpine is a complete spatial benchmark, a user of Jackpine may decide to include additional spatial scenarios. Due to its extensible nature, such a new scenario can be incorporated by implementing a new

spatial scenario class, creating a properties file and including the SQL queries in the corresponding SQL dialect class. Support for a new database requires only implementing the SQL dialect class for that database.

3.2.2 Data Model

Although there is no lack of large spatial data sets, the challenge is to find a suitable data set that can be used to build compelling applications representative of real-world use cases. To this end, we looked for data sets that are large enough to be interesting while also containing the labeled features necessary to construct the micro and macro scenarios.

The TIGER® (Topologically Integrated Geographic Encoding and Referencing) system [120] is produced by the US Census Bureau to support its mapping needs. It is a public domain data source available for each US state. We obtained the TIGER dataset for the state of Texas as a collection of shapefiles and imported them into the databases. We chose this dataset because Texas is the largest state in the contiguous United States, and it contains many interesting and diverse spatial features. Also, we were able to find sophisticated GIS data sets developed by the City of Austin and Travis County that we used to develop a number of macro scenarios. These are both located in the state of Texas, and several of our scenarios combine tables from the TIGER data set with the Austin and Travis County data.

Table 3.1 shows the database tables with their geometry, cardinality, data file size (KB) and index file size (KB). The largest table in terms of cardinality and size is the edges_merge (polylines) table. The information regarding data file size and index file size corresponds to the MySQL database. The micro benchmark queries (described in the next subsection) use 4 tables from the TIGER data set. The macro benchmark scenarios (described in Section 3.2.4, also use the database tables from the TIGER data set, as well as the remaining tables which are based on shapefiles obtained from the GIS data sets developed by the City of Austin and Travis County. Depending on the scenario, different combinations of these tables are used.

3.2.3 Micro Benchmark

The goal of the micro benchmark is to test the basic topological relationships and spatial analysis functions. The topological relationships describe how two spatial objects relate to each other in terms of topological constraints. Spatial analysis functions are analytic operations to determine the spatial properties of interest.

Ideally, the queries included in the micro benchmark should provide complete, yet minimal, coverage of topological relations. To this end, we explored several formal models. The four-intersection model, proposed by Egenhofer [38], is based on the comparison of the intersections of the boundaries and interiors of simple regions. By also taking into account the intersections with the exteriors, Egenhofer extended this model to a nine-intersection model [39], which is based on the comparison of the nine intersections between the interiors, boundaries, and exteriors of two geometric objects. The geometric objects could be point, line or polygon. The model specifies a total of 60 basic topological relations between two objects. However, many of these relations are not easy for humans to conceptualize, and are difficult to embed in a DBMS query language. In response to these issues, Clementini and Di Felice proposed a method that derives a subset of the relationships in the nine-intersection model by considering the maximum dimension of the intersection of two geometric objects [28]. This model, known as the Dimensionally Extended Nine-Intersection Model (DE-9IM) proposes the relationships: Equals, Disjoint, Intersects, Touches, Crosses, Overlaps, Within and Contains. The DE-9IM has been adopted by the Open Geospatial Consortium. Hence, we also use it as the basis for defining our topological relation queries.

The possible pairwise topological relationships among polygon, line and points according to the DE-9IM are

Table 3.2: Topological Relations in Dimensionally Extended 9-intersection Model included in benchmark

	Polygon and Polygon	Line and Line	Line and Polygon	Point and Polygon	Point and Line	Point and Point
Equals	Y		NA	NA	NA	Y
Disjoint	Y					
Intersects			Y	Y	Y	
Touches	Y		Y			NA
Crosses	NA	Y	Y	NA	NA	NA
Overlaps	Y		NA	NA	NA	NA
Within	Y		Y	Y		NA
Contains	Y					NA

outlined in Table 3.2. The relationships that are not applicable are marked with "NA" (for example, a line object cannot be equal to a polygon object). The relationships that are included in the benchmark are marked with "Y" and each row and each column is covered at least once. Looking across a row, different pairs of geometric objects will have different costs for computing the specified relationship. Looking down a column, the same database tables will be involved, but the computational cost of the different relationships can differ greatly. The micro benchmark queries pertaining to the selected topological relationships involve all-pair spatial join between the geometry columns of two tables. By using all entries in the tables, we avoid issues that may arise from selecting a particular object that may not be representative.

In addition to the all-pair spatial join queries, the micro benchmark includes four additional spatial join queries, each entails performing a spatial join with a given spatial object.

Besides the queries based on topological relations, we have also identified a number of queries that are related to spatial analysis. The focus of spatial analysis is to study spatial attributes of interest using formal techniques [33]. Finally, in addition to the read queries, the benchmark includes a workload that inserts a number of records with spatial attribute into the database tables. Depending on the table in which the record is inserted, the geometry of the spatial attribute could be point, line or polygon. Table 3.3 summarizes all the queries that are included in the micro benchmark.

3.2.4 Macro Benchmark

With the emergence of popular geospatial Web services such as Google Maps, the application of the spatial features of the relational databases have become widespread. The macro benchmark scenarios attempt to model some of these common use cases. Each scenario consists of a series of queries that are executed in sequence. Most of these queries involve spatial join or spatial analysis operations. The total time to execute all the queries in a scenario is considered to be its execution time. Table 3.5 summarizes all the queries that make up each of the scenarios. Here, we briefly describe each scenario.

3.2.4.1 Geocoding

Geocoding is the process of determining the geographic coordinates (expressed as latitude and longitude) of a terrestrial surface location based on other geographic data, such as street addresses, or postal codes. The geographic coordinate of an entity can be used to accurately place it in the map and to display in a mapping application. A common use case of geocoding is to locate addresses of people, organizations and businesses.

Table 3.3: Micro benchmark - queries

Operation	Description	Query
Topological relations, all pair joins		
Equals	Polygon equals polygon	Find the polygons that are spatially equal to other polygons in arealm_merge table
Equals	Point equals point	Find the points that are spatially equal to other points in pointlm_merge table
Disjoint	Polygon disjoint polygon	Find the polygons that are spatially disjoint from other polygons in arealm_merge table
Intersects	Line intersects polygon	Find the lines in edges_merge table that intersect polygons in arealm_merge table
Intersects	Point intersects polygon	Find the points in point_merge table that intersect polygons in arealm__merge table
Intersects	Point intersects line	Find the points in point_merge table that intersect lines in edges_merge table
Touches	Polygon touches polygon	Find the polygons that touch polygons in arealm_merge table
Touches	Line touches polygon	Find the lines in edges_merge table that touch polygons in arealm_merge table
Crosses	Line crosses line	Find first 5 lines that crosses other lines in edges_merge table
Crosses	Line crosses polygon	Find the lines in edges_merge table that cross polygons in arealm_merge table
Overlaps	Polygon overlaps polygon	Find the polygons that overlap other polygons in arealm_merge table
Within	Polygon within polygon	Find the polygons that are within other polygons in arealm_merge table
Within	Line within polygon	Find the lines in edges_merge table that are inside the polygons in arealm_merge table
Within	Point within polygon	Find the points in pointlm_merge table that are inside the polygons in arealm_merge table
Contains	Polygon contains polygon	Find the polygons that contain other polygons in arealm_merge table
Topological relations, given object		
Intersects	Longest line intersects	Given the longest line in edges_merge table, find all polygons in areawater_merge table intersected by it
Intersects	Largest polygon intersects	Given the largest polygon in arealm_merge table, find all lines in edges_merge table that intersect it
Overlaps	Largest polygon overlaps	Given the largest polygon in arealm_merge table, find all polygons in areawater_merge table that overlap it
Contains	Largest polygon contains	Given the largest polygon in the areawater_merge table, find all points in pointlm_merge table contained by it
Spatial analysis		
Distance	Distance search	Find all polygons in arealm_merge table that are within 1000 distance units from a given point.
Within	Bounding box search	Find all lines in edges_merge table that are inside the bounding box of a given specification.
Dimension	Dimension of polygons	Find the dimension of all polygons in arealm_merge table
Envelope	Envelope of lines	Find the envelopes of the first 1000 lines in edges_merge table
Length	Longest line	Find the longest line in edges_merge table
Area	Largest area	Find the largest polygon in areawater_merge table
Length	Total line length	Determine the total length of all lines in edges_merge table
Area	Total area	Determine the total area of all polygons in areawater_merge table
Buffer	Buffer of polygons	Construct the buffer regions around one mile radius of all polygons in arealm_merge table
ConvexHull	Convex hull of polygons	Construct the convex hulls of all polygons in arealm_merge table

Table 3.4: Micro benchmark - inserts

Operation	Description	Query
Insert record	Load points	Insert 1000 records in the pointlm_merge table
Insert record	Load lines	Insert 1000 records in the edges_merge table
Insert record	Load polygons	Insert 1000 records in the arealm_merge table

The scenario query involves finding the matching street segment given street number, name and postal code. The latitude and longitude of the location can be calculated from the latitudes and longitudes of the street segment. In the macro scenario, we geocode a total of 50 addresses in sequence, to emulate an application receiving a series of requests.

3.2.4.2 Reverse Geocoding

Reverse geocoding is the opposite process of geocoding - obtaining an associated textual address such as a street address or postal code from geographic coordinates. Location Based-Services (LBS) use reverse geocoding to convert coordinates obtained by mobile GPS devices into addresses that are more easily understandable by the end users. Such readable addresses can be displayed in Web-based GIS applications such as Activity reporting. An Activity Report shows various information about a GPS-enabled mobile user such as the timestamp, textual address, speed, direction etc. of that device at various moments in time during an interval.

There are two queries in this scenario to determine the complete textual address from the given latitude and longitude. The first query returns the closest city name and the second query finds the closest street name.

3.2.4.3 Map Search And Browsing

Searching for a point of interest and displaying it on a map is a common use case in Web-based mapping applications. Usually, during such a search process a user performs several successive searches that are relevant to her interest. A tourist visiting a new city may look for nearby airport, hotel, bar, and popular sites. A student visiting a University campus during a Graduate Visit Day may be interested in the nearby airport, hotel, library and student hostel facility.

In the scenario, we model these two different visit cases: student visit and tourist visit. During a benchmark run one of them is randomly picked. Then a search for a place of interest is performed. Once the most suitable match is returned, a series of 5 queries are executed that fetch the spatial objects inside a bounding box centered around the found place of interest. This is synonymous with the queries that retrieve spatial objects to be displayed in a map. The sequence of search for a nearby place of interest and 5 map display queries are repeated a number of times depending on the visit case. This reflects a typical pattern of searching for several nearby points of interest.

3.2.4.4 Flood Risk Analysis

Flooding is the most common natural disaster in many countries including the United States. Identifying the flood-prone areas, known as floodplains, is crucial to mitigate flood damages. The Federal Emergency Management Agency (FEMA) in the United States publishes the Flood Insurance Rate Map (FIRM) that depicts Special Flood Hazard Areas (SFHAs) and the risk premium zones. FIRM is used by emergency managers to administer floodplain management regulations, by builders and potential buyers to determine flood risks associated with buildings and properties and by the insurance agencies to assess whether flood insurance is required when offering loans. FEMA developed the DFIRM Database, which is the digital version of the Flood Insurance Rate Map, and

Table 3.5: Macro benchmark queries

Description	Query	Macro Scenario
Geocode address	Find the street segment that best matches with the given street number, name and postal code	Geocoding
Closest city	Find the city that is closest to the given latitude and longitude	Reverse Geocoding
Closest street	Find the street name that is closest to the given latitude and longitude	Reverse Geocoding
Best place of interest	Find the location of the place of interest that best matches the search criteria (by name or type)	Map Search and Browsing
Retrieve landmarks	Find all the points from gnis_names09 table that intersect a bounding box	Map Search and Browsing
Retrieve points	Find all the points from pointlm_merge table that intersect a bounding box	Map Search and Browsing
Retrieve lines	Find all the lines from edges_merge table that intersect a bounding box	Map Search and Browsing
Retrieve land polygons	Find all the polygons from arealm_merge table that intersect a bounding box	Map Search and Browsing
Retrieve water polygons	Find all the polygons from areawater_merge table that intersect a bounding box	Map Search and Browsing
Protected by dams	Find the flood risk areas that are protected by one or more dams	Flood Risk Analysis
Risk area categories	Determine the total flood risk area in acres grouped by risk area category	Flood Risk Analysis
Flood insurance required	Which residential property owners are required to carry flood insurance	Flood Risk Analysis
High risk industrial	Which industrial complexes are in high risk flood areas	Flood Risk Analysis
Residential properties	Determine the average property value per sq foot for Single-Family Residential properties	Land Information Management
Number of nearby hospitals	How many residential properties have a hospital within one mile	Land Information Management
Properties near hospitals	Determine the average property values within a one mile radius of all hospitals	Land Information Management
Lake properties	Find any 10 properties within 100 feet of the three major lakes	Land Information Management
Parking restrictions	Which office buildings have front yard parking restrictions	Land Information Management
Un-permitted properties	Find all the commercial properties that are built on unpermitted landfills	Land Information Management
Segment on spill point	Determine if the toxic spill point is on any segment of any waterway	Toxic Spill
Downstream segments	Find all segments of any waterway that are within 20 mile downstream of the spill point	Toxic Spill

is intended to be used with digital mapping and analysis software. The primary risk classifications used by the DFIRM Database are the High-risk areas (areas with 1% annual chance of flooding), the Moderate-to-low risk areas (areas with 0.2% annual chance flooding), and Undetermined-risk areas.

The DFIRM data set used in the benchmark is based on the Travis County Digital Flood Insurance Ratemap Database [35]. The scenario consists of a sequence of four queries that are relevant to flood risk analysis.

3.2.4.5 Land Information Management

The objective of land information management is to secure property rights for the land owners and to enable land administrators to make fundamental policy decisions about the nature and extent of investments in the land. Key to this process is the registration and maintenance of land ownership information, documenting the boundaries and precise location of parcels of land. Parcel-based digital land use information management systems are useful in land appraisal tasks such as property conveyancing, property tax assessment, property valuation and mortgage. It also deals with various issues in the management of utilities, the maintenance of land resources such as forestry, the enforcement of land use regulations and environmental impact assessments [122].

The land information data set used in the benchmark was obtained from the City of Austin GIS data sets [29]. Several queries, each related to land information management, are executed in sequence during the benchmark run.

3.2.4.6 Toxic Spill

Accidents involving toxic chemicals can cause significant hazard to human health and wreck havoc to the ecosystem. If the toxic substance is spilled into waterways, its detrimental impact may spread to a wider region, even to places many miles away. For example, in 2010 Hungary declared a state of emergency in three counties after a torrent of toxic red sludge from an aluminum plant tore through nearby villages, killing several people.

The toxic spill scenario in Jackpine is modeled after the Dunsmuir spill query from SEQUOIA 2000 benchmark. The scenario first detects the waterway segment on which the spill point is located. Then all segments of any waterway that are within 20 miles downstream of the initial spill point are recursively determined. The data set is based on the edges_merge table obtained from the TIGER data set. The initial spill point was chosen within the Trinity river in Texas.

3.3 Experimental Setup

In Section 3.4 we use Jackpine to assess the spatial features in three databases. In this section, we first describe the experimental setup. Table 3.6 summarizes the characteristics of each database instance.

The machine used to run the benchmark was an Intel Pentium 4 CPU (2.4 GHz) with 512 MB memory and 240 GB disk, running Ubuntu 10.04 Lucid 32-bit with kernel version 2.6.32. The purpose of using a machine with relatively little (512 MB) memory was to prevent the entire dataset from being memory resident. The database systems evaluated were PostgreSQL, MySQL and Informix without the Geodetic module. The databases were installed “as is” and “out of the box”. No tuning was performed on any of the databases. Query result caching in MySQL was disabled. Note that the buffer pool size with Informix is calculated by multiplying the number of buffers (1000) and the native page size (2KB). The buffer pool size reported for MySQL is the key buffer size.

All the tables, except the geocoder_address, have a spatial column (called “shape” in MySQL and Informix, and “the_geom” in PostgreSQL). For these tables, 2 indexes were created on each of them: a) a spatial index on

Table 3.6: Databases used in evaluation

Database	Version	Total data set size (GB)	Buffer pool size (MB)	Disk block size (KB)
PostgreSQL	8.4.2	6.1	12	8
MySQL	5.0.91	4.6	16	1
Informix	11.50	6.1	2	2

the spatial column and b) a regular (B-Tree) index on the spatial id column (the column is called "ogr_fid" in MySQL, "gid" in PostgreSQL, and "se_row_id" in Informix). For the geocoder_address table, a composite index (B-Tree) was created. Additional indexes (B-Tree) were created on the "roadflg" and "hydroflg" columns of the edges_merge table.

The benchmark uses JDBC connection pool and creates the connections before the actual execution of the queries. This ensures that the overhead of establishing a connection to the database is not accounted for in the metrics (elapsed time). Since the benchmark uses JDBC to connect to databases and execute queries, the test harness could be run in a different machine than the actual databases. However, to avoid taking network delays into consideration, the test harness was run in the same machine as the database server. Only one database was run at any time. During each benchmark scenario execution, a warmup run was first performed, followed by three successive iterations. The average of the total elapsed times of the three runs are plotted in the graphs.

3.4 Benchmark Results

The results of the benchmark runs comparing PostgreSQL, MySQL and Informix are presented in this section. We measured the average elapsed time to execute the benchmark queries. All time measurements are reported on log scales, since the results differ widely across queries and databases.

While analyzing the results, some of differences between the 3 databases in terms of spatial indexing and spatial query processing must be taken into account. The indexing method used in MySQL is the R-tree with quadratic splitting. PostgreSQL supports GiST indexing, whereas R-tree is the preferred indexing approach in Informix.

Of the two steps of filter and refinement, the MySQL query execution mechanism only conducts the filter step and does not execute the refinement step. The result is faster performance for many queries, but the inaccuracy due to omitting the refinement may make it unsuitable for many applications.

3.4.1 Data Loading

The first step in the evaluation was to measure the ramp-up time for our experimental testbed. We measured the time it takes to load all the datasets in the three databases (MySQL, PostgreSQL and Informix), create the necessary indexes (both spatial and non-spatial) and update the statistics used by the query optimizer (e.g. "vacuum full analyze" in PostgreSQL, or "update statistics" for Informix). We report these loading times in Table 3.7.

We used both generic and dedicated loading tools for importing shapefiles into our database tables. Since MySQL lacks a dedicated loading tool for shapefiles, we used GDAL's *ogr2ogr* tool. For PostgreSQL we used its dedicated *shp2pgsql* import tool, while for Informix we used its dedicated *loadshp* tool. Each of the loading tools performs similar operations: retrieve the records from a given shapefile, insert them in a specified database table and create two indexes, a non-spatial primary index which uniquely identifies each record and a spatial index on the geometry column.

Table 3.7: Load time (importing shapefiles, index creation, update stats)

Dataset	MySQL	PostgreSQL	Informix
pointlm_merge	5.786s	1.648s	31.9s
arealm_merge	9.099s	2.355s	19.1s
areawater_merge	4min 56.061s	1min 48.983s	20min 2.1s
edges_merge	74min 25.482s	34min 42.839s	436min 53s
Total	**80 min**	**37 min**	**7.5 hrs**

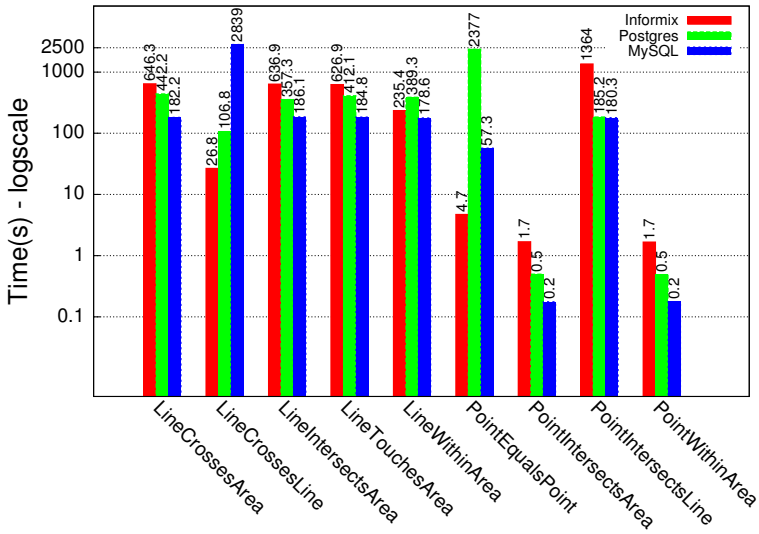


Figure 3.2: Pairwise spatial joins involving polygons, lines and points

We noticed that in general, dedicated bulk loading tools achieve much faster performance than generic ones. For example, we attempted to load the data into PostgreSQL using *ogr2ogr* before using its dedicated tool *shp2pgsql*. Although *ogr2ogr* is able to operate with several DBMSes and convert several types of spatial formats, we found that it performs several orders of magnitude worse with PostgreSQL compared to the bulk loading tool *shp2pgsql*.

With Informix unfortunately, the *loadshp* tool has significantly higher loading time because the server by default performs checkpoints every 1000 records, which introduces a significant overhead. While this chunksize can be changed to a larger number (thus resulting in less checkpointing), we decided to stick to the out-of-the-box setting. Furthermore, the speed of *loadshp* can be improved by using an insert cursor, which basically buffers rows before writing them to database tables. However, this limits the ability to handle errors, because if an error is encountered during loading, all buffered rows since the last successful write will be lost.

3.4.2 Micro Benchmark

The micro benchmark consists of several different sets of workload scenarios. In the scenario names, “area” and “polygon” have been used synonymously. The first category of scenarios is comprised of 15 pairwise spatial join queries among polygon, line and point objects. Due to the number of queries, the results are split into Figures 3.2 and 3.3.

Figure 3.2 shows the elapsed times of the pairwise spatial join queries between polygon and line, line and line, point and line, point and polygon, and point and point objects. Note that the result set to be returned by

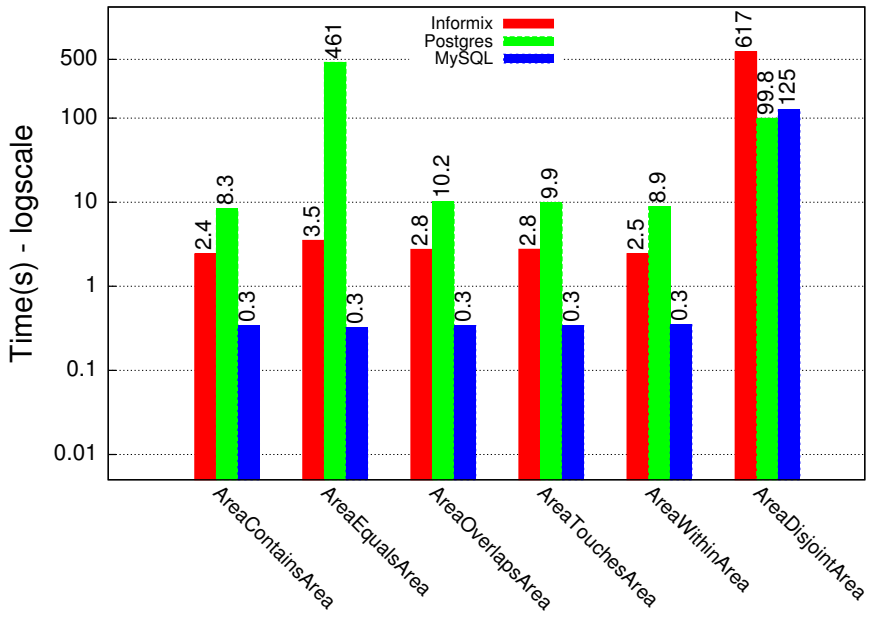


Figure 3.3: Pairwise spatial joins involving Polygons (Areas) only

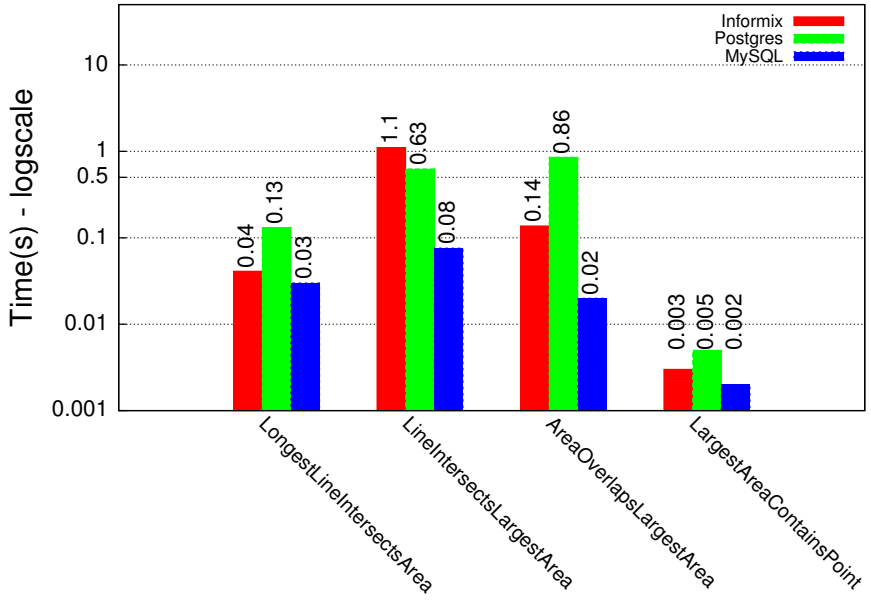


Figure 3.4: Spatial join with a given object

the LineCrossesLine scenario query was limited to 5 records. Without this limit, PostgreSQL ran this query for a number of days. The two other databases also took many hours to complete. This is due to the data size of the line table (edges_merge), as can be seen in Table 3.1. Interestingly, when the query returns a limited result set, as shown in LineCrossesLine scenario results, PostgreSQL performs better than MySQL. Among the remaining queries, MySQL does better than both PostgreSQL and Informix in all except the PointEqualsPoint scenario.

Figure 3.3 shows the elapsed times of the pairwise spatial join queries between polygon and polygon. Except

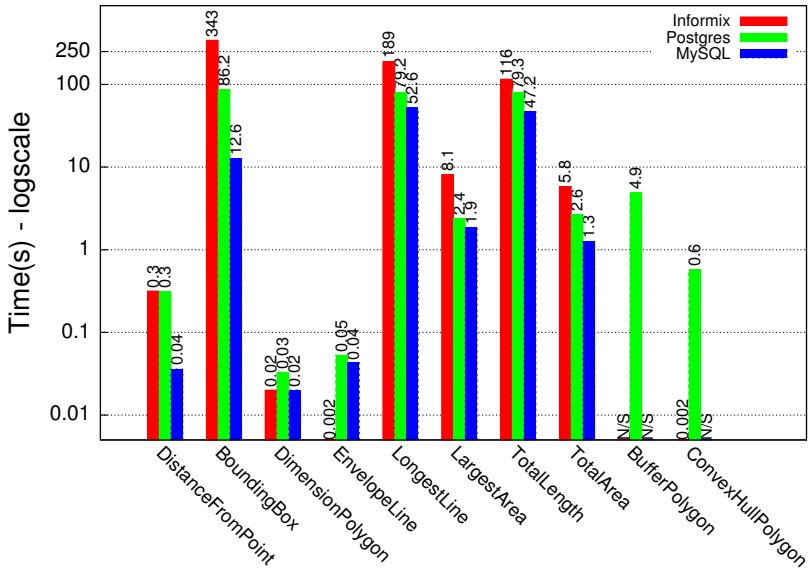


Figure 3.5: Spatial analysis (N/S = scenario not supported due to unsupported operations)

for the AreaDisjointArea scenario, MySQL performs better over PostgreSQL and Informix. In fact, the performance of MySQL is an order of magnitude better than the others, because it does not perform the refinement phase. Surprisingly, in the AreaDisjointArea scenario Informix is about 6 times slower than PostgreSQL and over 5 times slower than MySQL. This demonstrates the merit of comprehensively covering all the topological relationships as specified in Table 3.3, rather than an ad hoc selection. An application with many Disjoint operations involving polygons may not benefit from Informix's superior performance with other operations.

For the queries involving spatial join with a given object, the identifier of the given object (for instance, the largest polygon) was first determined offline. Then, the queries involving spatial joins of the polygon, line, and point tables with this given object were executed. Figure 3.4 shows the elapsed times of these queries. Informix performs best in 3 out of 4 scenarios, however it has the worst performance for the LineIntersectsLargestArea scenario.

The spatial analysis scenarios are comprised of queries with analytic functions and aggregation operations. Figure 3.5 shows the 10 spatial analysis scenario results. Note that MySQL does not support a few of these spatial functions such as Distance, Buffer and Convex hull. Consequently, no results were reported with MySQL for Buffer and ConvexHull. However, finding all the spatial objects of interest, such as restaurants, within a certain distance from a point is a common application. Hence, we decided to simulate the Distance function for MySQL by determining if the lengths of the lines constructed between the point origin and other neighboring points are less than or equal to the distance offset. Informix reported a runtime issue with the Buffer function, so no result is reported for that scenario.

Overall, MySQL is faster than PostgreSQL and Informix in 7 out of the 8 scenarios supported by MySQL. Informix performs the worst in 6 of these scenarios. However, Informix has surprisingly good performance for the Envelope and Convex Hull operations.

In many spatio-temporal applications such as Location Based Services, GPS-enabled mobile devices continuously generate location records. These records need to be loaded into the database in a real-time manner in order to serve up-to-the-minute status and activity reports. In other applications large shape files with different spatial

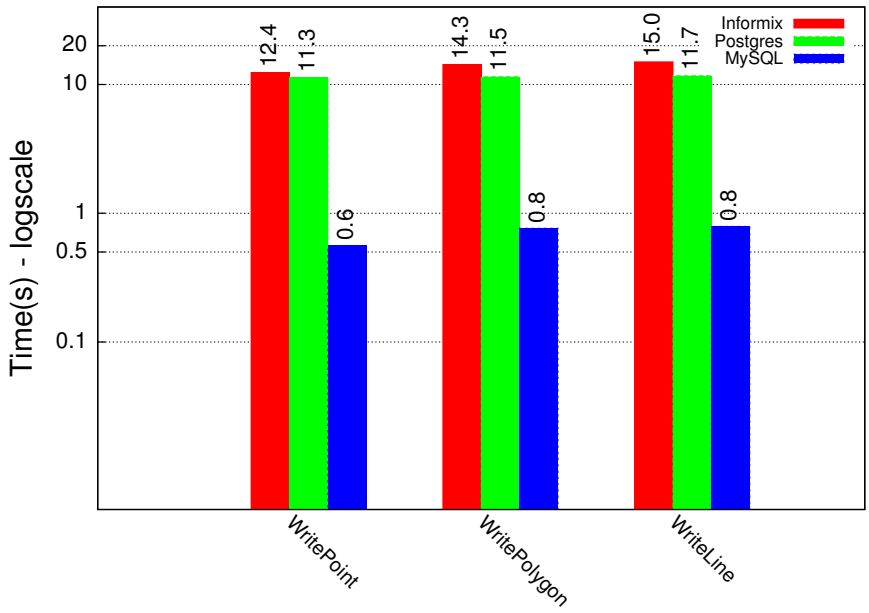


Figure 3.6: Data insertion

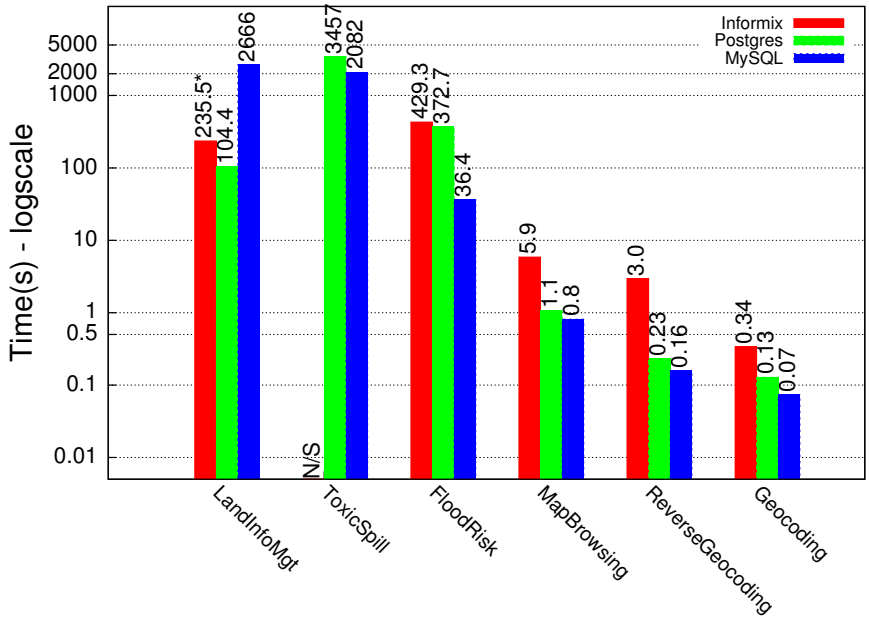


Figure 3.7: Macro Benchmark (N/S = scenario not supported)

attributes may be uploaded into the database. But they are not time critical.

The data insertion scenarios consist of inserting 1000 records consecutively into the database. Figure 3.6 shows the elapsed times to load database tables with point, line and polygon geometries. In all of the scenarios MySQL performed the best and Informix performed the worst.

3.4.3 Macro Benchmark

The performance of the 3 databases in the macro scenarios are shown in Figure 3.7. The query in the Geocoding scenario does not involve any spatial attributes and the performance of all three databases are quite comparable. The Reverse Geocoding scenario queries find the closest city and street from a Geographic location. The queries make use of the Distance function (simulated in MySQL as described earlier). Informix performs the worst in this scenario and MySQL performs the best. Map Search and Browsing scenario queries involve the Distance function and the Intersects relation. Once again, the performance of MySQL is better than PostgreSQL and Informix and PostgreSQL was faster than Informix.

The Flood Risk Analysis scenario executes a set of 4 queries involving spatial and non-spatial joins and aggregation operations. The topological relations Overlaps and Intersects and spatial analysis function Area feature in these queries. The performance of MySQL is several times better than that of Informix and PostgreSQL. PostgreSQL is slightly worse than Informix.

The Land Information Management scenario is a mix of 6 different queries and entails aggregation operations and spatial and non-spatial joins. Overlaps and Intersects are the featured topological relations. The spatial analysis function Dwithin (the distance within a certain offset) features prominently in these queries. Dwithin is supported only in PostgreSQL. Informix does not directly support this, but the Distance function can be utilized to express similar operations. MySQL, again, does not support Dwithin function and was simulated in the same manner described earlier. Another spatial function used in this scenario is the Area. MySQL performs quite poorly in this scenario, while Informix was the overall winner.

The Toxic Spill scenario consists of two queries, involving topological relations Intersects and Equals, and spatial analysis functions StartPoint, EndPoint and Distance. The Distance function was simulated in MySQL in a manner described earlier. Due to issues with the StartPoint and EndPoint functions, we were not able to implement this scenario for Informix. As a result, we only report MySQL and PostgreSQL results, and MySQL had the upper hand.

As noted earlier, even though MySQL performs better than Informix and PostgreSQL in many scenarios, MySQL may return many "false positive" records as result sets because it does not perform the refinement phase of the spatial query execution process. As can be been in Figure 3.1 in the column "Count Of Resultset", the number of records returned by a micro benchmark scenario run is the same for both PostgreSQL and Informix, whereas MySQL returns more records. By making this information explicit, Jackpine lets the users be cognizant of this issue. For some spatial applications, MBR-based matching of the spatial features offered by MySQL may be sufficient and no refinement step is needed at the application level. The users of the database must decide whether a particular database is suitable for their application domain based on the supported functions.

The results seen in these experiments are intended to illustrate the use of Jackpine, and not to draw any conclusions about the suitability of these databases in any particular application. To truly assess the relative merits of the databases, effort should be made to tune each database configuration appropriately. Nonetheless, the results shown here highlight the benefits of benchmarking with both a wide range of micro queries, and testing representative application workloads.

3.4.4 Overall Score

The previous evaluation sections provide a detailed view of the performance under different scenarios. Since different applications may stress different operations, this information can be valuable to an end user. Nonetheless, providing an overall performance score for each database is also desirable. Such a unified metric for evaluating

Table 3.8: Overall scores

Query Set	PostgreSQL	MySQL	Informix
Microbenchmark	1.00	5.17	1.48
Macrobenchmark	1.00	1.06	0.29

the performance of spatial database systems is hard to define, however. Given the various database parameters and categories of spatial queries, proposing a metric that incorporates all of them is a challenge in itself.

There has been much previous debate regarding the appropriate metric for summarizing overall benchmark results [31, 27]. Proposed metrics include arithmetic mean, weighted geometric mean or harmonic mean, each with advantages and disadvantages. Since our query run times differ by orders of magnitude within a single database, we prefer a measure that is insensitive to the influence of a long-running query. Hence, we decided to use a geometric mean over all queries, where each time result is normalized to a reference DBMS. Since only PostgreSQL supports all the queries in the benchmark, we decided to use it as our baseline DBMS. Ideally, the score would be computed over all queries in the benchmark, however a small number of queries are not supported by MySQL and Informix. We have chosen to keep these queries in the benchmark, however the overall score is computed for only those queries that are supported by all 3 databases.

We calculate the overall database score as:

$$Score_{DBMS_X} = \sqrt[N]{\prod_{q=1}^{N} \frac{Time_q^{DBMS_{ref}}}{Time_q^{DBMS_X}}}$$

where N is the total number of queries, and the reference DBMS is PostgreSQL. We calculate separate scores for the micro and macro benchmarks, reported in Table 3.8 for each database (higher is better). In the micro benchmark MySQL significantly outperforms the other two. However, this is because it did not perform the refinement step while processing the queries, and hence the query execution times were significantly faster than the others. Informix gets a slightly better score than PostgreSQL for the the micro benchmark queries.

In the macro benchmark Informix does worse than PostgreSQL. The particular mix of queries used in the macro scenarios lead to a lower score. This is consistent with the results for the Spatial Analysis microbenchmark queries in Figure 3.5. We emphasize again that the databases are untuned.

3.5 Discussion

With the growing popularity of Web Mapping and Location based services, the spatial support in the major relational databases has become increasingly important. The spatial functionalities across the commercial and open-source databases differ widely and there is no standard spatial database benchmark to compare such diverse offerings. We have introduced Jackpine, a database benchmark to evaluate spatial database performance. Our benchmark is flexible, as it can support any database with a JDBC driver implementation. It includes a number of carefully chosen spatial workloads and is extensible so that new test scenarios can be added. Jackpine subsumes all the vector query types of the SEQUOIA 2000 benchmark and includes many more.

We have presented a performance evaluation of Informix, MySQL and PostgreSQL with Jackpine as an example of using the benchmark. We have attempted to model real application workloads, however efforts to trace queries seen in actual usage would help to refine these scenarios.

Jackpine has been reasonably successful as a spatial database benchmark. It has been used by a few projects

so far and has been cited by a number of papers. For instance, it was extended to support concurrent workload mixes of queries from the benchmark [113]. As a future work, I plan to incorporate support for spatio-temporal workloads, including high rate of location data updates and many concurrent range queries. It is our hope that Jackpine will continue to be useful on its own, and will serve as a basis for ongoing discussion and development of a standard spatial benchmark for the community.

Chapter 4

Performance Heterogeneity-Aware Parallel Spatial Join For The Cloud

4.1 Introduction

Spatial analysis applications are rapidly gaining in importance, fueled by the explosive growth in vector spatial data and the availability of spatial features in commercial databases. Alongside traditional uses such as land surveys, city planning and environmental risk assessment, new classes of spatial applications are emerging in domains as diverse as building information management and medical image analysis.

These applications perform complex analyses of spatial datasets to provide vital information to governments and industries ranging from insurance to natural resource development to medical diagnostics. Typically, the analytical processing involves spatial join queries, which are long running and compute intensive. For example, a spatial join of a polyline table (73 million records representing the contiguous USA) with itself takes roughly *20 hours* to complete on an Amazon EC2 m1.xlarge instance.

4.1.1 A Case For Parallelizing Spatial Join Queries

Spatial join queries are used to combine two different datasets based on a spatial predicate. For instance, a polygon dataset representing landuse can be joined with another polygon dataset of flood-plains to determine flood-risk areas [103]. To find river bridges, a polyline dataset of roads can be joined with another polyline dataset representing hydrography. Table 4.1 shows some use cases for spatial join queries. These queries involve computational geometry algorithms to evaluate the relationships between spatial data types. These geometric computations on datasets with many records impose a high computational load, even with spatial indexes, leading to very long query latencies.

To illustrate the performance of the spatial join queries, we selected seven representative medium and long running queries from the Jackpine spatial benchmark's micro-benchmark suite [103]. These queries perform spatial join operations using different topological relations on the edges (polylines) and area (landmass and water polygons) tables from our datasets. We use PostgreSQL with the PostGIS spatial extension as the relational database. We use two real-world spatial datasets that contain diverse geographical features, drawn from the TIGER® data [120], produced by the United States (US) Census Bureau. This is a public domain data source

[1] We use Line to refer to polylines and Area to refer to polygons.

Table 4.1: Some use cases of spatial join queries

Use case	Queries
Flood Risk Analysis	Line/Area Intersects Line/Area[1]
	Area Overlaps Area
Land Info Management	Area Overlaps Area
	Area Intersects Area
Medical Imaging	Area Overlaps Area
	Area Intersects Area
Building Info Management	Line/Area Intersects Line/Area
	Line/Area Touches Line/Area
Water/Gas Utilities	Line Crosses Line
	Line Touches Line

Table 4.2: Database tables

Dataset	Database table	Geometry	Cardinality
California (4 GB, sharded)	edges_ca	polyline	4173498
	arealm_ca	polygon	7953
	areawater_ca	polygon	39334
Contig. USA (54 GB, sharded)	edges_us	polyline	73233790
	arealm_us	polygon	112492
	areawater_us	polygon	2290815

available for each US state. The first dataset consists of the polyline, polygon and point shapefiles for all the counties of California. The second is a much larger dataset covering the contiguous US. Full details of the dataset are in Table 4.2. Table 4.3 summarizes these queries on the two datasets, and the abbreviations that we use to refer to them in the text. Note that in the original Jackpine benchmark, the "Line and Line" queries were limited to return 5 result records due to very long execution times; we remove this limit in our experiments.

To establish a baseline for our work, we evaluated the single-node PostgreSQL (with PostGIS) performance for the queries in Table 4.3 on 16 m1.xlarge instances on Amazon EC2. We ran Ubuntu 10.04 Lucid 64-bit with kernel version 2.6.32-33-generic as the OS on each machine. The best (minimum) and worst (maximum) observed execution times are shown in Figure 4.1. As can be seen, some of the queries are very long running. For instance, the AoAus query takes 2 hours (7341 seconds), whereas the LcLus takes over 20 hours (71159 seconds) on average. So it is natural to try to parallelize the spatial join queries, in light of the abundance of processing cores per machine.

Figure 4.1 also shows that there is a significant difference between the best and worst observed query execution times. This is indicative of performance heterogeneity, which we discuss next.

4.1.2 Heterogeneity In Computing Clusters

Both modern data centers and smaller local clusters are typically built from commodity machines. Over time, new machines are added to deal with increasing loads or to replace failed nodes. These new nodes will usually come from newer hardware generations and will have many differences including different CPU models, numbers of cores, cache sizes, and disk models. Even without adding new nodes to the cluster, failed disks may be replaced with newer models, delivering higher performance. Such disk upgrades or replacements are commonplace in real-world clusters.

Table 4.3: Jackpine queries and abbreviations with two datasets

Description	**California**	**US**
Line Intersects Area (edges and arealm)	LiAca	LiAus
Line Touches Area (edges and arealm)	LtAca	LtAus
Line Crosses Area (edges and arealm)	LcAca	LcAus
Line Intersects Line (edges and edges)	LiLca	LiLus
Line Crosses Line (edges and edges)	LcLca	LcLus
Area Overlaps Area (areawater and areawater)	AoAca	AoAus
Area Touches Area (areawater and areawater)	AtAca	AtAus

Performance heterogeneity in a homogeneous cluster can also arise because system performance may degrade over time. Disk performance degradation due to partial media failure (i.e., bad sectors) is a common phenomenon. Therefore, to maintain performance homogeneity over time, a full replacement of the system would be required, which is prohibitively expensive.

Borthakur [15] observed that, in Facebook's datacenters, heterogeneity is the norm and anomalous behaviour is far more common than complete failure; at any given time, perhaps 10% of machines in the cluster run 50% slower than the others. This observation suggests that performance heterogeneity in large clusters of machines is more severe than academic researchers usually assume.

To make matters worse, modern servers may employ a range of techniques to improve reliability at the expense of performance at high operating temperatures, affecting disk throughput, CPU speeds, and memory bus speeds [40]. Since temperature also varies with the position of a node in a rack, or a rack in a machine room, some nodes in a cluster may be operating with these performance-limiting reliability mechanisms activated, while others are still operating at full speed, giving rise to dynamically varying performance heterogeneity.

In Cloud environments, additional factors such as virtualization overheads, contention for shared physical resources from multiple virtual machine (VM) instances, and the effects of VM migration, can also contribute to performance heterogeneity. Figure 4.1 shows significant differences between the fastest and slowest execution times, even though we used warm runs and the instances had enough RAM to hold the California dataset (the US dataset also benefited from OS caching). For instance, with the "Line and Area" queries on the US dataset, the slowest node takes roughly 50% longer than the fastest node (1844s extra for LcAus). We also found that the slowest instance varied across different queries, which indicates that the degree of performance heterogeneity may vary dynamically in Cloud environments, as others have observed [106, 43].

Perhaps more surprisingly, we also observed performance heterogeneity in an apparently homogeneous local cluster. We show the min and max execution times of 7 Jackpine queries from Table 4.3 on 4 distinct local nodes. Each local machine includes 8 Intel Xeon CPU cores (model E8400), 6 MB cache, 2 GB memory and an 880 GB 7200-RPM SATA disk. Every query was run in isolation on an instance of PostgreSQL running on each of these machines. For these experiments, we use cold runs to include disk effects.

As can be seen, there is a significant difference between the best and worst times for the queries. For instance, the LiAca (Line Intersects Area, California dataset) query took 111 seconds longer on the slowest node than on the fastest node. For LtAus (Line Touches Area, USA), the difference was 2633 seconds. On investigation, we found that the cluster was not truly homogeneous: the disk models differ due to replacements, which are common in real-world clusters. In this case, the node with the highest disk read bandwidth was always the first to complete the query.

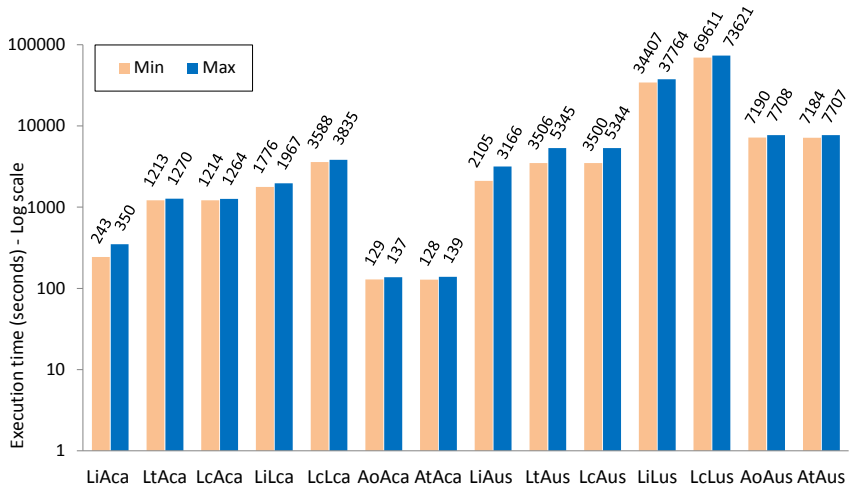


Figure 4.1: Min and max single-node PostgreSQL (with PostGIS) execution times for Jackpine queries, observed on 10 distinct EC2 m1.xlarge instances (warm runs)

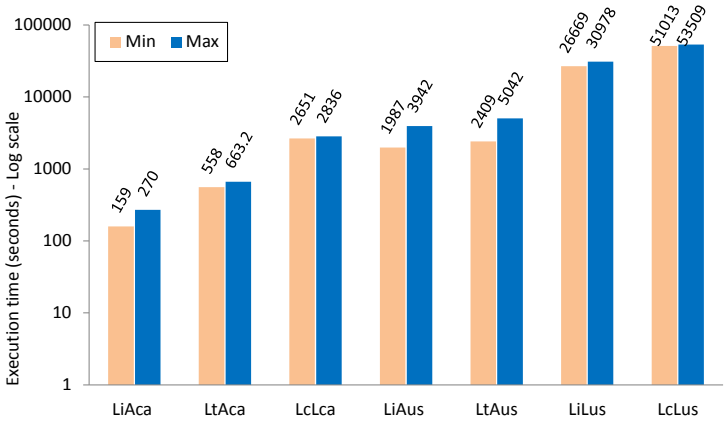


Figure 4.2: Min and max observed execution times of 7 Jackpine queries in a cluster of local nodes (cold runs)

4.1.3 Load Balancing

In a parallel query execution system operating on traditional workloads, the dataset is statically distributed to nodes using either range partitioning or hash partitioning, so that each node can work on a disjoint portion of the dataset. However, these partitioning schemes are not suitable for spatial data because, to execute a spatial join involving two tables, the tables need to be partitioned on the same spatial boundaries. The process of partitioning spatial datasets, known as spatial declustering, must take into account the spatial locality of the objects. This process involves dividing the spatial domain into two-dimensional disjoint subspaces and assigning them to individual shared-nothing database nodes so that the load (in terms of number of objects) is evenly distributed. However, due to the nature of spatial datasets, spatial declustering approaches suffer from a few issues caused by skew. The first issue is the variation in the number of records among different partitions, known as **tuple distribution skew**. The second issue is the **processing skew** caused by variation in the size of objects. In this chapter we primarily address the tuple distribution skew.

In a real-world cluster, the combined effects of data skew and machine performance heterogeneity make load balancing even more challenging. For instance, the machine with the weakest CPU could be assigned to

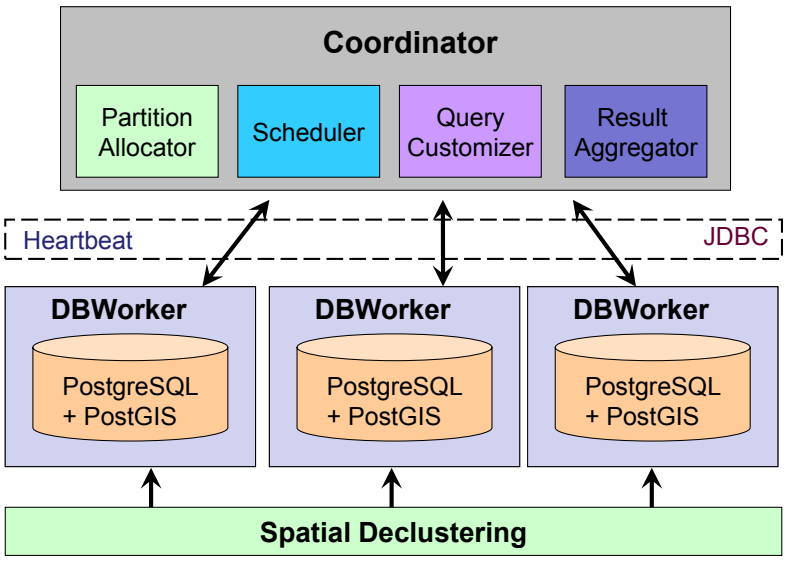
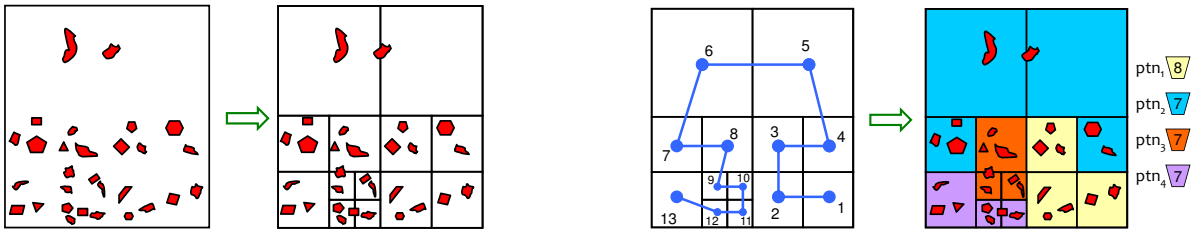


Figure 4.3: Architecture of Niharika



(a) Recursive tiling (b) Declustering

Figure 4.4: Spatial partitioning (a) and declustering using Hilbert SFC traversal and tile aggregation into partitions (b)

process the spatial partitions with the heaviest workload. Therefore, significant opportunity exists to improve load balancing, which would in turn reduce query execution time and achieve better speedup.

4.2 Niharika

We propose a parallel spatial query execution infrastructure that aims to accelerate the execution of long running spatial analysis queries by exploiting all available processing cores. It addresses the challenges of performance heterogeneity and processing skew by combining a data partitioning scheme and task scheduling with dynamic load balancing. This mechanism can naturally take advantage of the multiple processing cores in each machine.

4.2.1 Architecture

Our system is based on a master-slave architecture, inspired by HadoopDB, with a task scheduler called the Coordinator, and worker nodes called DBWorkers, as shown in Figure 4.3. The Coordinator handles the scheduling of query jobs, while DBWorkers run tasks locally. Each DBWorker node hosts a PostgreSQL/PostGIS instance, which is used to locally process its portion of the dataset. The role of the Coordinator resembles that of the JobTracker in HadoopDB. It pushes the query execution into the database instances on each DBWorker, thus

leveraging the highly-optimized RDBMS processing. The Coordinator performs aggregation and group-by operations on the resultset returned by the DBWorkers.

4.2.2 Spatial Declustering

We implemented a spatial declustering scheme (see Figure 4.4), which aims to assign neighboring tiles to the same node as much as possible, while still reducing tuple distribution skew and offering good load balancing. The *Scheduler* component of the Coordinator is responsible for making these assignments, with the help of a *Query Customizer* that re-writes the submitted query for execution on each node so that nodes only process records from their assigned partitions.

We illustrate the tile spatial declustering algorithm with an example. Given a set of 29 spatial objects and a threshold of at most 4 objects per tile, Phase 1 generates 13 tiles (see Figure 4.4(a)). The tiles are then numbered in the Hilbert space filling curve (SFC) traversal order (see Figure 4.4(b)(left)). Any object that overlaps multiple spatial partitions must be replicated to those partitions. We use Hilbert SFC to improve spatial locality so that object duplication is minimized. It is assumed that there are 4 nodes, N_1 to N_4, in the cluster. If each node gets an equal share of tiles (last node gets extra tiles) the tile-to-node allocation is:

$N_1 : T_1, T_2, T_3[8 \text{ objects}]$
$N_2 : T_4, T_5, T_6[4 \text{ objects}]$
$N_3 : T_7, T_8, T_9[8 \text{ objects}]$
$N_4 : T_{10}, T_{11}, T_{12}, T_{13}[9 \text{ objects}]$

Here, N_2 gets only 4 objects while N_4 gets 9 objects, giving a significant tuple distribution skew with this approach.

Next, we apply our tile aggregation algorithm with a partition size of 8. Traversing the tiles in SFC order, ptn_1 is assigned tiles T_1, T_2 and T_3. The next tile T_4 will not fit in ptn_1, so ptn_2 is created and filled similarly. The end result is:

$ptn_1 : T_1, T_2, T_3[8 \text{ objects}]$
$ptn_2 : T_4, T_5, T_6, T_7[7 \text{ objects}]$
$ptn_3 : T_8, T_9, T_{10}[7\text{objects}]$
$ptn_4 : T_{11}, T_{12}, T_{13}[7\text{objects}]$

Since the number of partitions is the same as the number of nodes, each node is assigned one partition. As can be seen, the data distribution skew is significantly reduced.

To enable dynamic load balancing, we actually generate many more partitions than the number of nodes.

4.2.3 Load Balancing

Load-balancing is essential to parallelize any computing task, including a long running analytics query. An imbalanced workload assignment may result in a significant delay between the completion times of the fastest and the slowest (aka *straggler*) machines. Even in a perfectly homogeneous cluster of machines, the inherent skew in the spatial dataset may cause workload imbalance. In the real world, machine performance can vary widely, worsening the straggler effect.

In a traditional parallel database, each member node is statically assigned a disjoint subset of the dataset. In the context of spatial query execution, this implies the static assignment of spatial partitions that are generated by the spatial declustering phase. When a query job is submitted, each node can be issued an identical query task, which it executes on its own dataset. The resultset returned by all member nodes can be aggregated to produce

the final query result. For instance the query "find all the line segments that intersect polygon objects", can be expressed as follows in PostgreSQL:

```
select distinct a.gid from
  arealm_us a, edges_us e where
  ST_Intersects(e.the_geom, a.the_geom)   (1)
```

Each node executes this query on its data partition.

However, static partition assignment does not take into account node performance heterogeneity, dynamic load conditions, or the inherent processing skew due to variation in spatial object sizes and join selectivity. These factors can easily result in straggler nodes. Instead, we propose *dynamic partitioned parallelism* in which the data partitions that each node processes are determined "just in time" based on the performance of presently available nodes. This scheme requires that nodes either host the entire dataset locally, or obtain the dataset for their assigned partitions at run-time, since it is not known ahead of time which partitions will be needed. The compact nature of vector spatial data makes it tractable to host the entire dataset on each node, which is the approach we take.

Once the Scheduler has determined a partition assignment, we need to ensure that each node processes only the partitions that are mapped to it. We cannot send the original query to each node because this would result in each node processing the original query over the entire dataset. Instead, a form of virtual dynamic partitioning can be achieved by customizing queries on-the-fly to direct each node to process only its assigned partitions. This query customization scheme requires the addition of a partition_id column to the database tables. Then, the query can be tailored for each node by the Coordinator to include the appropriate partition identifiers.

Dynamic partitioning implies that the workload should be discretely divisible into chunks of almost equal size. Spatial partitions are the natural granularity for dividing spatial query workloads. The physical division of a table can be achieved by a widely supported database technique called *sharding*, which allows a single instance of a database table to be split into smaller physical pieces. PostgreSQL supports sharding by letting the sharded tables "inherit" from a master table and adding a constraint on the partitioning key [88]. In Niharika, the partitioning key is the spatial partition id and an index is created on this for each partition. We illustrate the sharding of the edges_us table.

Master-table:

```
CREATE TABLE edges_us (
  gid integer NOT NULL,
statefp varchar(2) default NULL,
  ...
  the_geom geometry NOT NULL,
  partition_id integer,
  PRIMARY KEY (gid) )
```

Here, gid is the primary key attribute in the table, the_geom is the spatial attribute and partition_id is the partitioning key.

Child-tables:

```
CREATE TABLE edges_us_1 (
  CHECK ( partition_id = 1 ) )
  INHERITS (edges_us);


CREATE TABLE edges_us_2 (
```

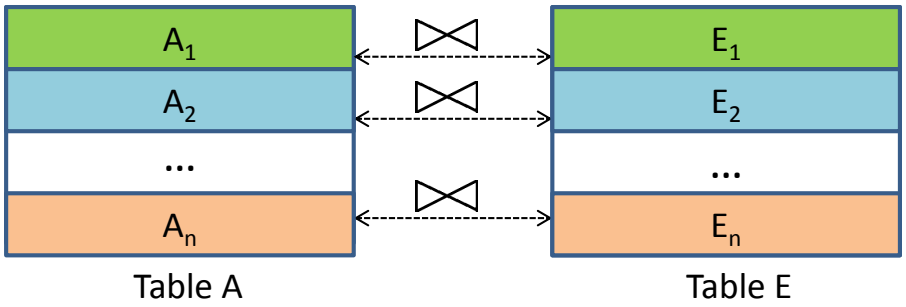


Figure 4.5: Spatial join with partitioned tables

```
    CHECK ( partition_id = 2 ) )
    INHERITS (edges_us);
    ...  ...
```

Query customization can be done in several ways. First, the **where** clause of the original query can be extended with a set of partition ids specified either as a range (using **between**) or as a list (using **in**). For example, we can specify the partitions for node N_2 from Section 4.2.2 by rewriting Query (1) using *Between*:

```
select distinct a.gid from
 arealm_us a, edges_us e where
 ST_Intersects(e.the_geom, a.the_geom)
 ...  and a.partition_id between T4 and T7
     and e.partition_id between T4 and T7
```

We can also specify the partitions as a list using *In*:

```
select distinct a.gid from
 arealm_us a, edges_us e where
 ST_Intersects(e.the_geom, a.the_geom)
 ...  and a.partition_id in (T4, T5, T6, T7)
     and e.partition_id in (T4, T5, T6, T7)
```

We can also use a **union** of multiple queries, in which each query specifies a single partition.

```
select distinct a.gid from
 arealm_us a, edges_us e where
 ST_Intersects(e.the_geom, a.the_geom)
 ...  and a.partition_id = T4
     and e.partition_id = T4
 union select distinct a.gid from
    arealm_us a, edges_us e where
    ST_Intersects(e.the_geom, a.the_geom)
     and a.partition_id = T5
     and e.partition_id = T5
 union ...
```

The choice of customization strategy depends on the underlying database engine. For instance, we want the query optimizer to select an ideal spatial join plan. Figure 4.5 shows a spatial join between partitioned tables A and E. An ideal plan would involve join between matching partitions only (e.g. partition 1 of table A would be joined only with partition 1 of table E). Joining partition 1 of table A with any other partition of table E is

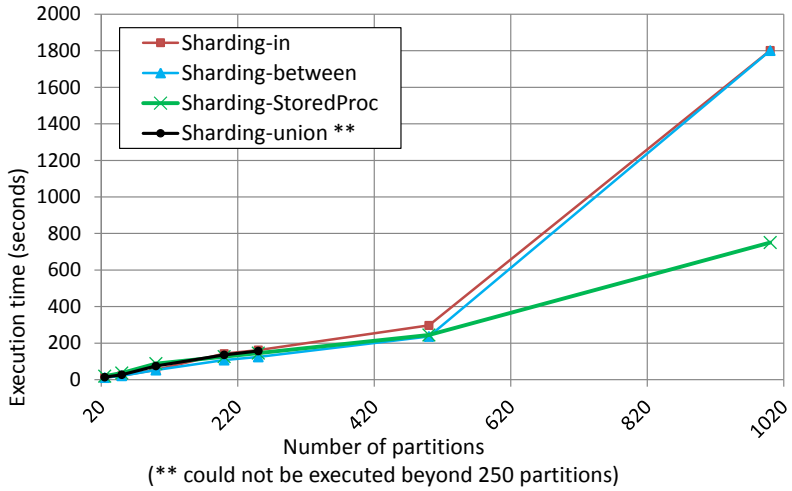


Figure 4.6: Execution times for LiAus query with different partition strategies

unnecessary, since it would not contribute any tuple to the query resultset. Other features of the database engine may affect the choice between query customization strategies that result in ideal query plans.

Inefficient query plan: $A \bowtie E = A_1 \bowtie E_1 U A_1 \bowtie E_2 ... U A_1 \bowtie E_n \cup A_2 \bowtie E_1 U A_2 \bowtie E_2 ... U A_2 \bowtie E_n ... \cup ... A_n \bowtie E_n$

Ideal query plan: $A \bowtie E = A_1 \bowtie E_1 \cup A_2 \bowtie E_2 ... \cup A_n \bowtie E_n$

We evaluated each of these customized queries for dynamic partitioning on PostgreSQL, and found that an ideal query plan was generated for only the **union** approach with sharded tables (*Sharding-union*). Unfortunately, we also found that memory usage ballooned with Sharding-union when a large number of partitions is specified (i.e. $>$250), leading to dramatic increases in execution time. As a result, it could not be run with any query beyond 250 partitions. To work around this limitation in PostgreSQL, we created a *Stored procedure* in which the Sharding-union query with multiple partitions is expressed as an iteration over each partition:

```
FOR p IN partition_list LOOP
    ...  and a.partition_id = $p
    and e.partition_id = $p ...
END LOOP
```

Figure 4.6 shows the effect of the different dynamic partitioning strategies on query execution time for LiAus on PostgreSQL as the number of partitions increases. The shared procedure approach (*Sharding-SharedProc*), gives us an ideal query plan that has stable performance; we use this strategy in our implementation.

Next we execute all the queries with Sharding-StoredProc approach on single PostgreSQL instances (results are averaged over the set of m1.xlarge nodes). The execution times are compared against those of the *original* PostgreSQL dataset (non-sharded). Figure 4.7 shows the comparison for California and Figure 4.8 for US dataset. In some cases the *original* PostgreSQL does better than *sharded* PostgreSQL, and in other case the reverse occurs. In all subsequent experiments we use the execution times of the single-node *sharded* PostgreSQL as the baseline to compute speedup for our scheduling and dynamic load-balancing strategies.

We present several alternatives for partition assignment to achieve load balancing.

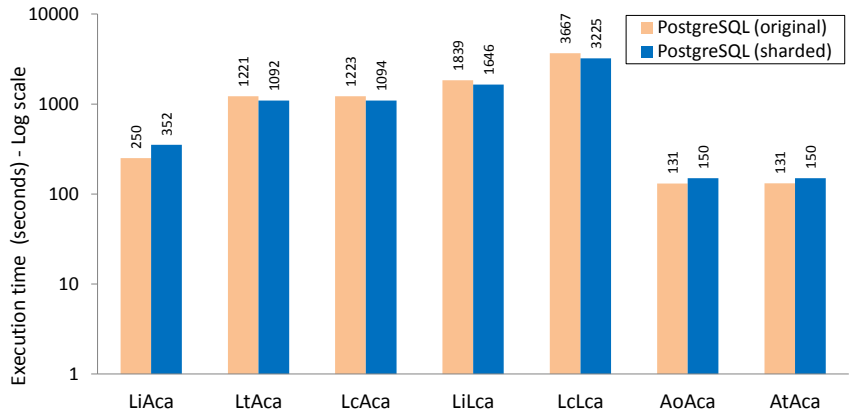


Figure 4.7: Query execution times - original vs Sharding-StoredProc (California dataset)

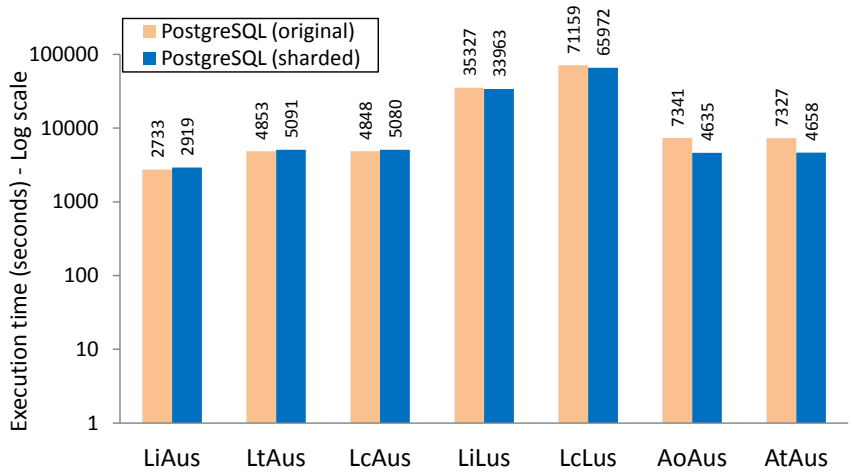


Figure 4.8: Query execution times - original vs Sharding-StoredProc (USA dataset)

4.2.4 Heterogeneity-aware single assignment of partitions

An uneven assignment of workloads to nodes may cause a long delay between the completion times of the fastest and the slowest nodes, which increases the overall elapsed time of the job. However, an *a priori* balanced assignment of partitions to nodes in a single round is surprisingly difficult. Intuitively, such an assignment can be made by approximating the processing cost of each partition, and the processing capacity of each node, and allocating partitions to nodes proportionally. Node processing capacity can be approximated by running the same query against the same dataset on each, and comparing their relative performance.

We experimented with various strategies for estimating the node processing capacity and the work required per partition, for different queries, and then assigning partitions accordingly. The speedups achieved were inconsistent. Heterogeneity-aware single assignment is limited because (a) it does not adapt to dynamic load conditions and (b) the work required per partition depends on the query being executed.

4.2.5 Multi-Round Assignment Of Partitions

Due to the limitations of single round assignment, the desired outcome of the smallest possible difference between the completion times of the fastest and slowest node may not be achieved. To overcome the limitations of single-

Require: W is the number of DBWorkers; P is total number of partitions; $BatchSize$ is the size of assignment

```
1:  TotAssigned ← 0
2:  MaxIdx ← −1
3:  while true do
4:    for i = 0 to W − 1 do
5:      Status ← status of DBWorker i
6:      if Status = IDLE then
7:        if TotAssigned < P then
8:          TotAssigned ← TotAssigned + BatchSize
9:          if TotAssigned > P then
10:           TotAssigned ← P
11:         end if
12:         MinIdx ← MaxIdx + 1
13:         MaxIdx ← MaxIdx + BatchSize
14:         if MaxIdx > P − 1 then
15:           MaxIdx ← P − 1
16:         end if
17:         RwQuery ← Rewrite(MinIdx, MaxIdx)
18:         Assign RwQuery to DBWorker i
19:         Status ← ASSIGNED
20:         AsgndCnt ← AsgndCnt + 1
21:       end if
22:     else if Status = COMPLETE then
23:       CmpltCount ← CmpltCount + 1
24:     end if
25:   end for
26:   if CmpltCnt = AsgndCnt and TotAssigned = P then
27:     break
28:   end if
29: end while
```

Figure 4.9: Algorithm1: Multi-round fixed-size batch assignment

round assignment, we need to adjust the size of the assigned workload as the query executes. This goal can be achieved by assigning work in multiple rounds, where a node processes a smaller portion of its overall assigned workload in each round. In effect, nodes that complete their work faster can be assigned a larger share of the total workload.

A useful abstraction for scheduling is Divisible Load Theory (DLT) [13]. It attempts to devise a task assignment plan such that all processors finish their computation simultaneously and any deviation from the optimal plan can be improved by transferring load from a busy processor to an idle one. We present three multi-round scheduling algorithms inspired by DLT.

4.2.5.1 Fixed-Size Batch Assignment

The first algorithm is the multi-round fixed-size batch assignment to idle nodes (Algorithm1, see Figure 4.9). Initially, each node is assigned a batch of spatial partitions (generated in the declustering phase). Thereafter, any node that finishes processing its batch gets the next batch to work on. The intermediate results generated after processing each batch are combined and aggregated by the Coordinator. At each step the min and max partition indexes of the batch are updated (lines 11 through 14) and a query rewriting step (line 15) customizes the query before issuing it to the DBWorker. This process is repeated until all the partitions are completed and the final

result is produced. Note that the list of partitions is traversed sequentially and any DBWorker can be assigned the next batch of partitions if it has finished processing its previous batch.

4.2.5.2 Batch Assignment Using Processing Capacity

Intuitively, the choice of batch size requires a tradeoff between the overhead of executing the query in multiple rounds, and the opportunity to do load balancing. A smaller batch size requires more rounds, which creates more opportunities to adjust the workload assignment, but also incurs more overhead. It may be useful to employ different batch sizes for different nodes, however, we have found that a fixed-size batch for all nodes works well. The main advantage of Algorithm1 is its simplicity and that there is no setup step required, except for selecting the batch size.

Algorithm1 (Figure 4.9) is implemented as a Java program running in the Coordinator node. The Coordinator launches a worker thread for each DBWorker, which acts as a state machine. When the status is *IDLE* (line 6), the corresponding DBWorker is available for more work. When the status is *ASSIGNED* (line 17), the DBWorker executes the query assigned to it. When a DBWorker finishes its query and the result is available, the status is changed to *COMPLETE*. When all P partitions are processed, the final result is printed.

In Algorithm1, node processing capacity is implicit, in the sense that more work is assigned to faster nodes because they become idle sooner. Our second algorithm (Algorithm2, described in Figure 4.10) explicitly considers node processing capacity. The idea is to assign batches of workload partitions to each node such that the total number of partitions assigned to each node is proportional to its processing capacity in the cluster. The capacity ratio of a node's performance can be based on the different types of nodes that are in the cluster (for instance, m1.xlarge, m2.xlarge, m2.2xlarge, etc). Another way to do this is by executing a representative query on each node and comparing their relative performance. Explictly accounting for the node capacity enables a more even load assigment, than an approach that is completely agnostic. To address skew in the dataset, the spatial partitions are grouped together in histogram buckets such that the partitions in the same bucket have similar execution times. The execution histogram is constructed during a preprocessing step, in which a representative query is executed on a single node for each partition and the partition id is placed in the histogram based on its execution time.

While executing the query, the partitions are assigned from each bucket to nodes in proportion to their processing capacity (lines 7 through 13). The buckets with the longest running partitions are processed first. If a faster node completes processing all its share of partitions from all the buckets, it looks for yet unprocessed partitions from the buckets (lines 15 through 19). The query is customized with the assigned list of partitions (line 21) before the DBWorker executes the query.

4.2.5.3 Batch Assignment With Node Affinity

The previous algorithms are agnostic to which node is assigned to process which partition. For relatively small datasets that can fit comfortably in the node memory, this should not be a concern, as all partitions can be cached in RAM. However, for larger datasets, an essentially random assignment of partitions to nodes can lead to very poor caching behavior. In this case, partitions will frequently need to be fetched from disk, since they are unlikely to be found in the memory of the node to which they are assigned.

Our third algorithm, Algorithm3, assigns partitions to each node from a preferred set of partitions, thereby maximizing the likelihood that the partitions are available in that node's memory. Of course, some spatial partitions may still need to be fetched from disk. However, Algorithm3 is expected to reduce disk I/O substantially compared to a random assignment such as Algorithm1.

Require: W, P, $BatchSize$ as before; $HistBucket[B][]$ is execution profile histogram having B buckets; $InitAssigned[B][W]$ is a boolean array initialized false

```
1: while true do
2:   for i = 0 to W − 1 do
3:     CapacityRatio ← capacity ratio of DBWorker i
4:     Status ← status of DBWorker i
5:     if Status = IDLE then
6:       Assigned ← false
7:       for j = 0 to B − 1 do
8:         if InitAssigned[j][i] = false then
9:           AsgnList ← assign from HistBucket[j][]
10:            in proportion to CapacityRatio
11:          InitAssigned[j][i] ← true
12:          Assigned ← true
13:          break
14:        end if
15:      end for
16:      if Assigned = false then
17:        for j = 0 to B − 1 do
18:          if any unprocessed left in HistBucket[j][] then
19:            AsgnList ← assign from HistBucket[j][]
20:            Assigned ← true
21:            break
22:          end if
23:        end for
24:      end if
25:      if Assigned = true then
26:        RwQuery ← Rewrite(AsgnList)
27:        Assign RwQuery to DBWorker i
28:        Status ← ASSIGNED
29:        AsgndCnt ← AsgndCnt + 1
30:      end if
31:    else if Status = COMPLETE then
32:      CmpltCount ← CmpltCount + 1
33:    end if
34:  end for
35:  if CmpltCnt = AsgndCnt and TotAssigned = P then
36:    break
37:  end if
38: end while
```

Figure 4.10: Algorithm2: Multi-round fixed-size batch assignment using node processing capacity

Require: W, P, $BatchSize$ as before; $NodeAsgnList[W]$ is the list of partitions assigned to DBWorkers by Procedure StrategicAssignment; $FetchList[W]$ is the list of partitions fetched by DBWorkers; $PartAssigned[P]$ is a boolean array initialized to false

```
1: while true do
2:   for i = 0 to W − 1 do
3:     Status ← status of DBWorker i
4:     if Status = IDLE then
5:       BatchList ← NULL
6:       if TotAssigned < P then
7:         CurrAsgnList ← NodeAsgnList[i]
8:         while CurrAsgnList has more items do
9:           if size(BatchList) = BatchSize then
10:            NodeAsgnList[i] ← CurrAsgnList
11:            break
12:          end if
13:          PartId ← next(CurrAsgnList)
14:          if PartAssigned[PartId] = false then
15:            PartAssigned[PartId] = true
16:            append(BatchList, PartId)
17:            remove(CurrAsgnList, PartId)
18:            TotAssigned ← TotAssigned + 1
19:          end if
20:        end while
21:        if size(BatchList) = 0 then
22:          for j = 0 to P − 1 do
23:            if PartAssigned[j] = false then
24:              PartAssigned[j] = true
25:              append(BatchList, j)
26:              TotAssigned ← TotAssigned + 1
27:              if size(BatchList) = BatchSize then
28:                break
29:              end if
30:            end if
31:          end for
32:        end if
33:        if size(BatchList) > 0 then
34:          RwQuery ← Rewrite(BatchList)
35:          Assign RwQuery to DBWorker i
36:          Status ← ASSIGNED
37:        end if
38:      end if
39:    else if Status = COMPLETE then
40:      CmpltCount ← CmpltCount + 1
41:    end if
42:  end for
43:  if CmpltCnt = AsgndCnt and TotAssigned = P then
44:    break
45:  end if
46: end while
```

Figure 4.11: Algorithm3: Multi-round batch assignment from a preferred partition set

Require: W, P as before; $LargePartsList$ is a small list of partitions with the most skew
1: $i \leftarrow 0$
2: **while** $LargePartsList$ has more items **do**
3: $append(NodeAsgnList[i], next(LargePartsList))$
4: $i \leftarrow i + 1$
5: **if** $i = W$ **then**
6: $i \leftarrow 0$
7: **end if**
8: **end while**
9: $Q \leftarrow P - size(LargePartsList)$
10: $PartIndex \leftarrow 0$
11: **for** $i = 0$ to $W - 1$ **do**
12: $CapacityRatio \leftarrow$ capacity ratio of DBWorker i
13: $CurrAsgn \leftarrow Q * CapacityRatio/TotalCapacity$
14: **while** $PartIndex < Q$ **and** $CurrAsgn > 0$ **do**
15: **if** $PartIndex$ not in $LargePartsList$ **then**
16: $append(NodeAsgnList[i], PartIndex)$
17: $CurrAsgn \leftarrow CurrAsgn - 1$
18: **end if**
19: $PartIndex \leftarrow PartIndex + 1$
20: **end while**
21: **end for**
22: $returnNodeAsgnList$

Figure 4.12: Procedure StrategicAssignment for Algorithm3

The assignment approach of Algorithm3 (Procedure StrategicAssignment, Figure 4.12) first assigns to DBWorkers from a small list (typically less than 5% of total) of partitions that have the most skew. This ensures that the long running partitions will be the first to get processed. To assign from this list we do not consider node processing capacity, since this is a rather small set. Then from the remaining list of partitions each DBWorker is assigned a set of preferred partitions. This takes into account the static capacity ratio of a node's performance. The assignment algorithm makes sure that a node is always assigned the same set of partitions to maximize caching benefits. It also ensures that a node will have a sufficient number of partitions to process. We assume that a number of spatial queries are processed by the system. As outlined in lines 7 through 17 of Algorithm3 (Figure 4.11), each DBWorker processes from the list of partitions assigned to the node. However, if a DBWorker has processed all its assigned partitions, it may be assigned to work on any unprocessed partitions originally assigned to another node (lines 20 through 25). This allows Algorithm3 to deal with node performance heterogeneity and skew, while still providing good affinity between nodes and the partitions they usually process.

4.2.5.4 Round Robin With Equal Share Of Partitions

To evaluate the benefits of dynamic load balancing, we also implemented a single-assignment approach in which each node is given an equal number of partitions. We call this approach Round Robin with Equal Share of Partitions (RR-ESP). As the name suggests, it assigns the partitions to cores in a round robin fashion such that each node gets about the same number of partitions.

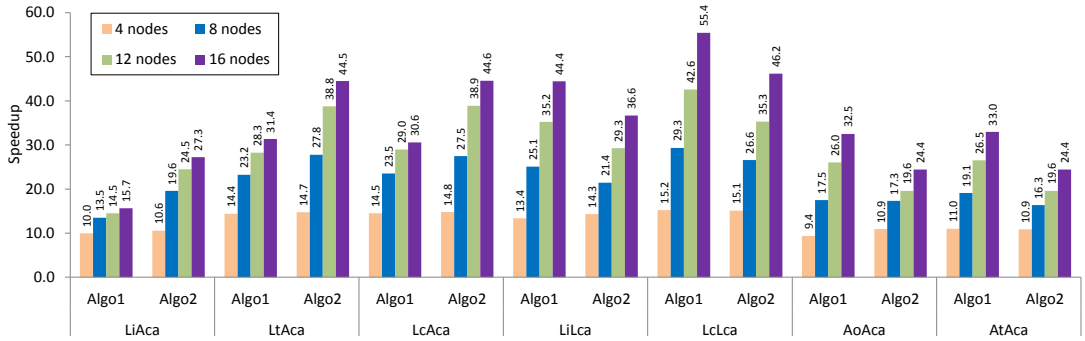


Figure 4.13: Speedup of Algorithm1 compared with Algorithm2 (California dataset, 4 cores/node)

4.3 Experimental Evaluation

In this section we evaluate our load balancing algorithms in various settings on the California and USA datasets. We first describe the datasets and the settings that we use in our experiments.

4.3.1 Experimental Setup

We obtained the polyline (edges) and polygon (area landmass and area water) shapefiles for all the counties of California and created single shapefiles by merging them. Similarly, we created single US shapefiles for all the 48 states in the contiguous US. These shapefiles were then uploaded to a PostgreSQL database using the shp2pgsql tool. Table 4.2 outlines the database tables. To represent a real-world database system that processes many queries, we use the best of 3 warm runs to calculate the speedup results. We observe that the overhead of duplicate elimination from resultset is minimal.

4.3.2 Results With Dataset That Fits In Memory

Niharika's query execution model is naturally able to exploit multiple processing cores in each database node. PostgreSQL executes each query as a separate OS process, hence if multiple concurrent queries are issued to a multi-core node, the OS will naturally schedule them on different processors. Niharika's scheduler simply assigns multiple concurrent customized queries, each representing a separate batch of work, to each multi-core node. The number of concurrent batches is set to the number of cores in a node.

Algorithm1 and Algorithm2 were evaluated using Amazon EC2 with 4, 8, 12 and 16 **m1.xlarge** nodes, each of which had 4 cores and 15GiB memory. The speedup of each query is computed relative to the sharded single-node PostgreSQL execution time across all nodes. The speedup achieved by Algorithm1 and Algorithm2 for the seven queries is compared in Figure 4.13.

We observe that queries involving the same database tables have similar speedup profiles. For instance, the "Line and Area" queries generally have the lowest speedups for all numbers of cores, for both algorithms, while the "Line and Line" queries have the best speedups. The maximum achievable speedup is limited by the longest processing time of any single partition, and our analysis showed that large polygons in the area table led to some partitions with very long processing times in joins with the line table. We also observe that Algorithm2 has better speedup with these "Line and Area" queries than Algorithm1. However, with the other queries Algorithm2 performs worse than Algorithm1. Essentially, Algorithm2 tries to assign larger batches of work to the more powerful nodes in the cluster, thus reducing the number of rounds that are needed. When the estimates of node

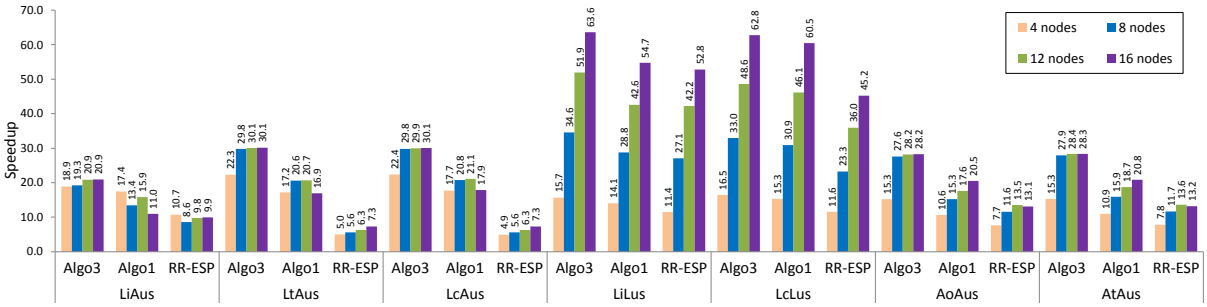


Figure 4.14: Algorithm3 vs Algorithm1 vs RR-ESP speedup (USA dataset, 4 cores/node)

processing capacity and work per partition are reasonably accurate, Algorithm2 performs well. However, the larger batch size also means fewer opportunities for load balancing when conditions change dynamically, or when the static estimates are less accurate. Since Algorithm2 is more complex than Algorithm1, we do not use Algorithm2 for subsequent experiments.

4.3.3 Results With Larger Dataset (USA)

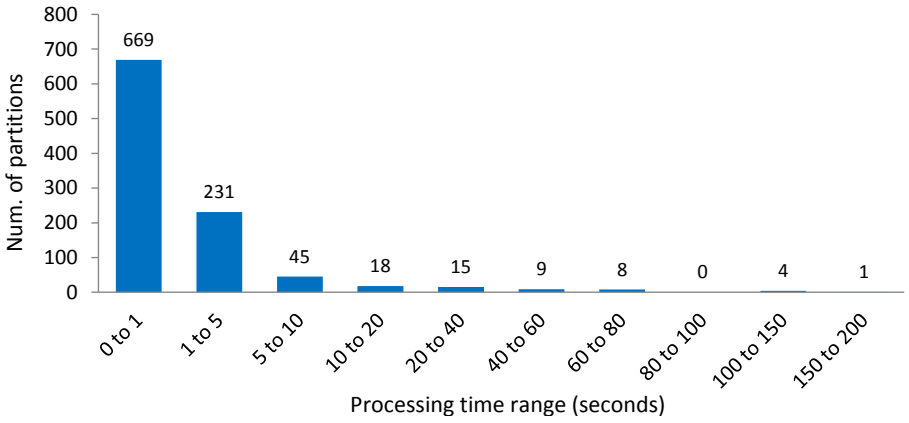


Figure 4.15: Histogram of the processing times of the partitions for LtAus (USA dataset, Algorithm3)

When the dataset is too large to remain in node memory, as is the case for the US dataset on m1.xlarge EC2 instances, we expect that Algorithm3 should have a significant advantage over the randomized assignment of Algorithm1. For comparison, we also include the single-assignment RR-ESP results in this section. Figure 4.14 shows the speedup attained by Algorithm1, Algorithm3 and RR-ESP against the best sharded single-node PostgreSQL performance. With Algorithm3, Niharika achieves excellent speedup in the number of cores for all queries, especially the "Line and Line" queries (LiLus and LcLus). For instance, the LcLus query achieves linear speedup in the number of cores, reaching 16.5X with 4 nodes (16 cores) and 62.8X with 16 nodes (64 cores). With "Line and Area" and "Area and Area" queries, Algorithm3 reaches the maximum achievable speedup, which is limited by the processing time of the longest running partition. For instance, with LtAus, Niharika attains speedup close to 30X with 8 nodes; beyond 8 nodes the speedup does not improve.

To explain this effect, we measured the processing time of each partition and found significant skew, as shown in Figure 4.15. The histogram (count) shows that only 1 partition took more than 150s (168.8s, allowing a maximum speedup of 30.1X), with 4 others taking between 100 and 150s. The remaining partitions took less than

80s.

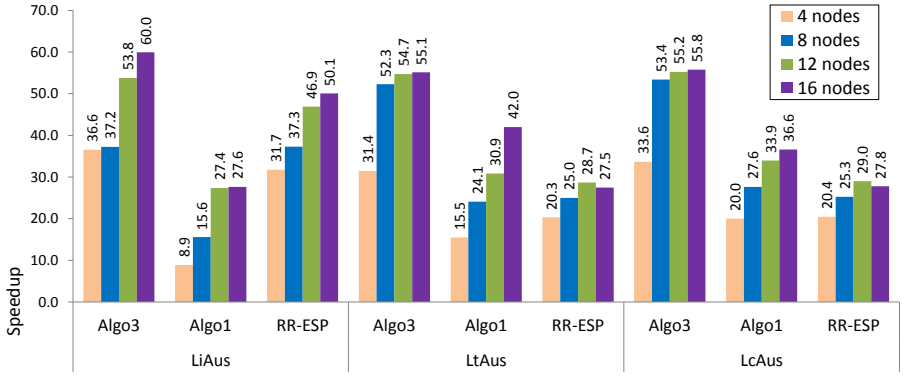


Figure 4.16: Algorithm3 vs Algorithm1 vs RR-ESP speedup with modified dataset (USA dataset, 4 cores/node)

This result suggests that the spatial dataset properties can be a key constraining factor with Area queries. Although the TIGER dataset refactors long line objects, it does not do as well with large Area objects, as it does with the Line objects. To illustrate, we removed the largest 50 objects (0.4% of the dataset) from the arealm_us dataset and re-ran the Line and Area queries with Algorithm1, Algorithm3, RR-ESP and the sharded single-node PostgreSQL. As can be seen in Figure 4.16, Algorithm3 achieves near linear speedup in the number of cores. In a few cases Algorithm3 achieves super linear speedup. This can be attributed to improved memory caching behaviour because of the affinity scheduling of Algorithm3 and the significant reduction of processing skew due to the deletion of the largest 50 objects. In Chapter 5 we explore a more fine-grained declustering approach that takes object size as well as object count into consideration.

In comparison to Algorithm3, the speedup of Algorithm1 is significantly worse on all queries. Although Algorithm1 does a good job with in-memory datasets, that is no longer the case with datasets that do not fit in memory. Therefore, Algorithm3 can be the considered the load balancing algorithm of choice among the ones we present. We also see that Niharika with Algorithm3 is superior to an approach that is agnostic to performance heterogeneity, that is, RR-ESP. In both Figures 4.14 and 4.16 RR-ESP showed much poorer speedups compared to those of Algorithm3. Also, RR-ESP shows reduced speedups when going from 48 to 64 cores for all but one case (LiAus, where the speedup is unchanged) when the queries involve an area table. This suggests that RR-ESP is more susceptible to processing skew.

4.4 Scalability and Data Placement

So far, we have evaluated Niharika's scheduling algorithms with the assumption that the dataset is hosted in every member node in the parallel database system. Although this may be tractable for many spatial datasets, especially in light of growing disk capacities, some datasets are too large for full replication at every node. They must be partitioned such that each node hosts only a portion of it. In traditional shared-nothing parallel databases, each node is allocated an equal portion of the original dataset because node performance heterogeneity is not considered. As noted earlier, this may degrade overall system performance and result in long idle times for the faster nodes.

We designed a two step heterogeneity-aware data placement algorithm that assigns spatial partitions to nodes in proportion to their processing capacity in Step 1. In Step 2, each node is allocated an extra k-times the initial

Require: W, P, $BatchSize$ as before; $NodeAsgnList[W]$ is the list of partitions assigned to DBWorkers; $FetchList[W]$ is the list of partitions fetched by DBWorkers; $PartAssigned[P]$ is a boolean array initialized to false

```
1: while true do
2:   for i = 0 to W do
3:     Status ← status of DBWorker i
4:     if Status = IDLE then
5:       BatchList ← NULL
6:       if TotAssigned < P then
7:         CurrAsgnList ← NodeAsgnList[i]
8:         while CurrAsgnList has more items do
9:           if size(BatchList) = BatchSize then
10:            NodeAsgnList[i] ← CurrAsgnList
11:            break
12:          end if
13:          PartId ← next(CurrAsgnList)
14:          if PartAssigned[PartId] = false then
15:            PartAssigned[PartId] = true
16:            append(BatchList, PartId)
17:            remove(CurrAsgnList, PartId)
18:            TotAssigned ← TotAssigned + 1
19:          end if
20:        end while
21:        if size(BatchList) > 0 then
22:          RwQuery ← Rewrite(BatchList)
23:          Assign RwQuery to DBWorker i
24:          Status ← ASSIGNED
25:        else
26:          FetchList[i] ← NULL
27:          for j = 0 to P do
28:            if PartAssigned[j] = false then
29:              if j not in FetchList then
30:                append(FetchList[i], j)
31:                if size(FetchList[i]) = BatchSize then
32:                  break
33:                end if
34:              end if
35:            end if
36:          end for
37:          if size(FetchList[i]) > 0 then
38:            Status ← FETCHING
39:          end if
40:        end if
41:      end if
42:    else if Status = FETCH_COMPLETE then
43:      append(NodeAsgnList[i], FetchList[i])
44:      Status ← IDLE
45:    else if Status = COMPLETE then
46:      CmpltCount ← CmpltCount + 1
47:    end if
48:  end for
49:  if CmpltCnt = AsgndCnt and TotAssigned = P then
50:    break
51:  end if
52: end while
```

Figure 4.17: Algorithm4: Multi-round batch assignment to idle nodes with fetch

Table 4.4: Illustration of data placement

Node	Processing capacity	Step1	Step2	Total
N_1	6	300	300	600
N_2	6	300	300	600
N_3	5	250	250	500
N_4	3	150	150	300

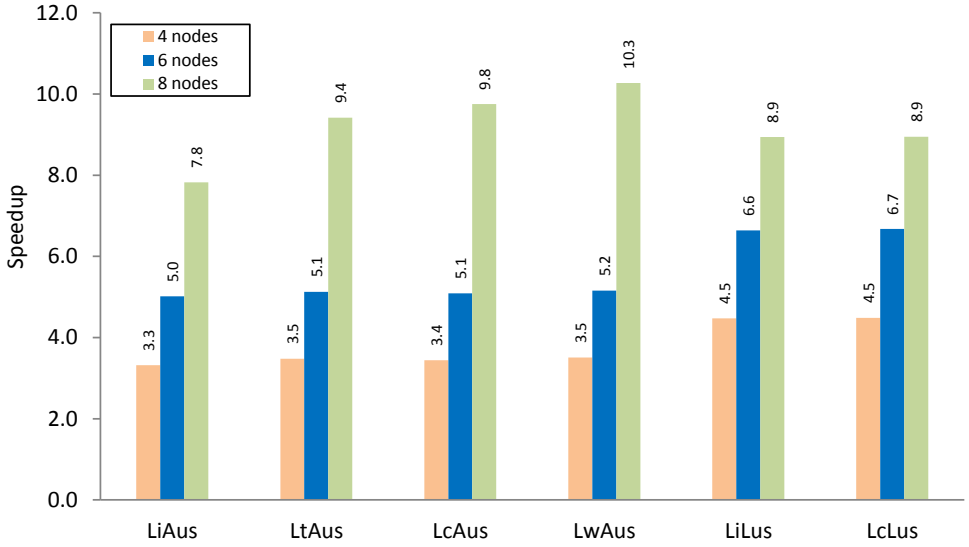


Figure 4.18: Algorithm4 speedup in a cluster (single-core)

number of partitions, to account for dynamic load variation and minimize the copying of partitions from other nodes. With k=1, this is similar to a data placement strategy with a replication factor of 2.

To illustrate, assume that 1000 partitions are generated by the declustering algorithm (Section 4.2.2) and we have 4 nodes (N_1 to N_4) with different processing capacities, as shown in Table 4.4. In Step 1, each node is assigned partitions according to its processing capacity. In Step 2, each node is assigned a different set of partitions, also in proportion to its capacity.

The placement algorithm attempts to ensure that a node hosts a sufficient number of partitions to process, in the event of dynamic load changes or skew in the dataset. If a node has processed all its assigned workload partitions, it may (depending on its disk capacity) initiate a task to fetch additional partitions from other nodes. When the fetch task is complete, it can process the newly acquired partitions. If another node, hosting those partitions, becomes free before the fetch task is complete, the fetch is aborted and the other node is allowed to process those partitions. We present a scheduling algorithm (Algorithm4, Figure 4.17) that incorporates the fetch mechanism and takes advantage of our data placement strategy.

Algorithm4 was evaluated with the USA dataset in a local cluster of machines. Each machine has 4 3.0 GHz Intel Xeon CPU core with 6 MB cache, 4 GB memory and an 880 GB 7200-RPM SATA disk.

Each node was statically allocated a part of the dataset, using our data placement strategy. Any additional spatial partitions that were fetched during a query execution were deleted before starting the next query execution, to ensure we capture the cost of fetching missing partitions. The speedup against the best single-core performance (by enabling only 1 core) is shown in Figure 4.18. Using more machines allows more of the dataset to be cached in memory, leading to super-linear speedup in some cases.

4.5 Chapter Summary

Spatial join is at the heart of many emerging spatial data analysis applications. We have introduced Niharika, a parallel spatial query execution infrastructure, designed to exploit multiple cores in modern processors to accelerate spatial join performance.

Performance heterogeneity in a cluster is natural in Cloud computing settings. We have also shown that even an apparently homogeneous cluster can have significant performance heterogeneity, which hurts parallel database query execution times. With its load-balancing techniques Niharika is able to dynamically perform "performance proportional" assignment of workloads to nodes according to their processing capacity. Niharika significantly reduces idle times for faster nodes in relation to a straggler.

Niharika does not require changes to the database engine. Its query execution model can naturally use all available processing cores for intra-query parallelism, which is not currently supported by many databases including PostgreSQL. We presented three multi-round load-balancing algorithms and showed the importance of dynamically adapting to performance heterogeneity. Niharika's Algorithm3 scales well with an in-memory dataset and a dataset that does not fit in memory on a cluster of multicore nodes in the Cloud. We also introduced Algorithm4, that utilizes a heterogeneity-aware data placement strategy and exhibits good speedup.

Chapter 5

Parallel In-Memory Spatial Join

5.1 Introduction

The rapid growth in core counts and main memory sizes presents significant opportunities for improving the performance of spatial join queries. Today's high-capacity servers may have hundreds of cores and up to a terabyte of main memory. Due to the compactness of the vector data representation, many spatial datasets can fit completely in the main memory of a more modest machine. For instance the TIGER [120] dataset, comprised of the polyline and polygon features of all the contiguous states of the USA, is roughly 54 GB in size [104]. To exploit the potential of large memory multi-core machines, we need to revisit previous research works related to parallel spatial join. Many of these approaches [90, 91, 129, 6] rely on spatial declustering to partition the dataset, enabling parallelization. These techniques are based on the idea of minimizing data skew (tuple distribution skew) by creating spatial partitions or tiles such that each tile has roughly the same number of objects. However, when both the filter and refinement steps are considered, significant processing skew may be observed due to the variation in the time it takes to process different tiles. This variation results from differences in object properties, such as size or point density, which are typical in many spatial datasets such as geospatial and VLSI. As we have demonstrated in Section 4.3.3, in addition to data skew, the issue of processing skew must be addressed to improve parallel performance of spatial join.

5.1.1 Processing Skew

As described in Section 2.2, spatial join queries use a two-step evaluation process, of which the refinement step is very compute-intensive. The refinement step typically involves computational geometry algorithms to evaluate the relationships between spatial objects, which impose a high computational load, even with spatial indexes. Consequently, a spatial join query involving even a moderately sized dataset may lead to long query latencies. Previous spatial join research efforts focused on the filter step and did not consider refinement (e.g., [90, 18, 67]). Moreover, they only considered one spatial predicate based on MBR comparison (MBR overlaps or intersects). Since the refinement step dominates the overall query latency, we argue that it is important to consider both the filter and refinement steps. To illustrate our point, we selected 8 spatial join queries from the Jackpine spatial database benchmark [103]. These queries are shown in Table 5.1. To identify the specific tables involved in the join we use abbreviations with the 2 initial letters of the tables. Note that the query abbreviations are different from those in Table 4.3. We also selected five different spatial predicates defined by OGC. In Figure 5.1 we show the percentage of execution time spent in filter and refinement steps for these queries. We implemented

Table 5.1: Jackpine queries and abbreviations

Description (tables involved)	Abbreviations
Polygon overlaps Polygon (Areawater and Areawater)	Aw_ov_Aw
Polygon overlaps Polygon (Areawater and Arealm)	Aw_ov_Al
Polygon within Polygon (Arealm and Areawater)	Al_wi_Aw
Polyline Crosses Polygon (Edges and Arealm)	Ed_cr_Al
Polyline Intersects Polygon (Edges and Areawater)	Ed_in_Aw
Polyline Crosses Polygon (Edges and Areawater)	Ed_cr_Aw
Polyline Touches Polygon (Edges and Areawater)	Ed_to_Aw
Polyline Crosses Polyline (Edges and Edges)	Ed_cr_Ed

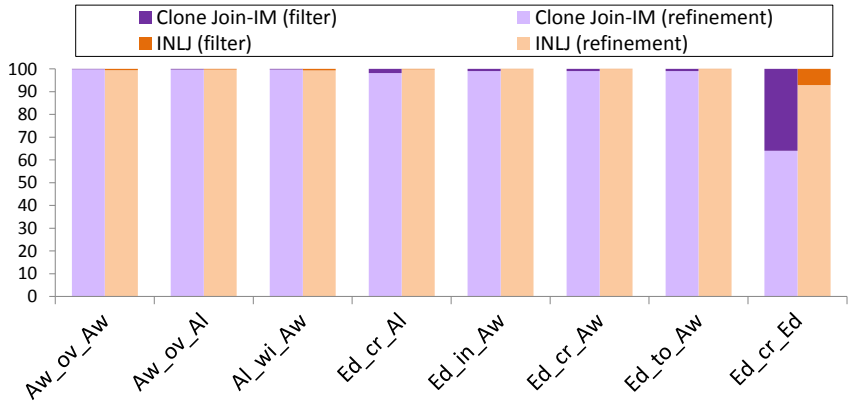


Figure 5.1: % execution time spent in filter vs. refinement with Clone Join-IM and INLJ

in-memory versions of two disk based spatial join approaches: Clone Join and Indexed Nested Loop join (INLJ) based on in-memory R-tree filtering. We use the label **Clone Join-IM** to distinguish the in-memory version of Clone Join. We use the TIGER California dataset, details of which are in Table 5.3 in Section 5.3.1. As Figure 5.1 shows, for all queries, except for Ed_cr_Ed, the refinement step takes over 98% of the overall filter+refinement time. Therefore, it is imperative that a spatial join approach be optimized for refinement to address the long query latencies.

The recent TOUCH [82] in-memory spatial join approach, like previous disk-based approaches, also only considers the filter step for the evaluation. With an in-memory spatial join, disk latency is no longer an issue, and the refinement step dominates the overall query latency. When the actual object geometries are used during refinement, the processing skew can become a critical issue.

To illustrate the problem, we use the Clone Join-IM algorithm. We execute the Ed_cr_Al query for each tile generated by Clone Join-IM declustering and report the execution times in a histogram in Figure 5.2. As can be seen, the vast majority of the tiles took less than 0.5 seconds. Only three tiles took more than 5 seconds, and only one took over 6 seconds, which indicates significant processing skew. In a parallel spatial join query processing system it is important to distribute equal amounts of work to each processor. If a processor takes significantly longer than others due to processing skew, it becomes a "straggler" and the overall query latency suffers.

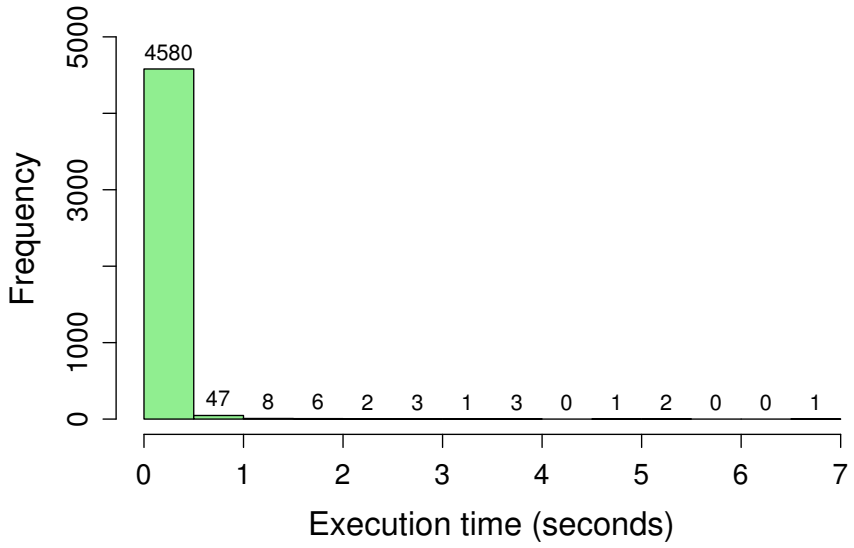


Figure 5.2: Histogram of the execution time per spatial partition (tile) for query Ed_cr_Al

5.2 SPINOJA

We introduce SPINOJA (Skew-resistant Parallel IN-memOry spatial Join Architecture), a main memory query execution infrastructure designed specifically to address processing skew for parallel in-memory spatial join. SPINOJA takes into account the overall query execution time, including the filter and refinement steps. It introduces a new spatial declustering scheme called MOD-Quadtree declustering, which alleviates processing skew. Unlike previous declustering approaches that try to distribute the number of objects evenly, our approach attempts to equalize the amount of required computation per partition. The main idea is to decompose objects along tile (spatial partition) boundaries such that the amount of work involved in processing the tiles is roughly equal. We explore three work metrics to determine the best way to approximate the amount of work per tile. We also present three load-balancing strategies to assign the tiles to worker threads.

To evaluate SPINOJA fairly against prior work, we implement parallel in-memory versions of previously proposed disk-based spatial join approaches, in addition to a parallel version of the TOUCH in-memory spatial join approach. We evaluate eight spatial join queries involving five spatial predicates that require complex refinement processing. Our experimental results show that SPINOJA does a very good job of minimizing the processing skew and achieves superior performance over the other approaches. In addition, the memory requirements of SPINOJA are much lower than those of Clone Join and comparable to those of TOUCH.

5.2.1 System Organization

SPINOJA's system organization is modeled after the master-slave architecture. As shown in Figure 5.3, the task scheduler is called the *TaskManager*, and worker threads are called *Workers*. The TaskManager is responsible for scheduling a query job and assigning parts of each job or tasks to the available Workers. The TaskManager also performs result aggregation, which involves spatial predicate evaluation and group-by operations, and eliminates duplicates from the resultset using a distinct operator.

SPINOJA uses a spatial declustering scheme (described next in Section 5.2.2) that attempts to reduce processing skew. Key to this declustering process is the work metric and we describe three such metrics. The *Table loader and Index builder* component loads the tiles created by the spatial declustering into in-memory tables.

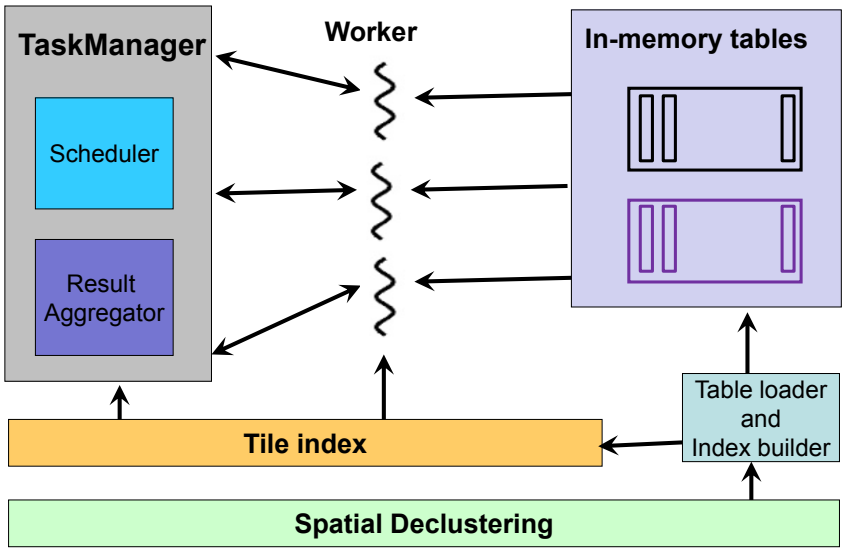


Figure 5.3: SPINOJA system organization

It also constructs the *Tile index* to be used during the query execution. To implement the Tile index we use a compressed bitmap, which requires less memory than other data structures. More specifically, we use the CONCISE [30] compressed bitmap. The Scheduler component of TaskManager assigns the tiles to the workers using load-balancing algorithms such that each Worker only processes objects from its assigned tiles. We evaluate three algorithms for dynamic load balancing within this framework.

5.2.2 MOD-Quadtree Declustering

A join query can be parallelized by partitioning the data domain into disjoint chunks and assigning the corresponding pair of chunks from the two tables to a separate processor. This spatial declustering process involves subdividing the spatial domain into 2-dimensional *tiles*. Each tile contains those objects whose MBRs it overlaps, which implies that any object whose MBR is overlapped by multiple tiles may need to be replicated to each of those tiles. As noted in Section 5.1, previous approaches to spatial declustering aimed to equalize the number of objects in each tile to try to ensure that each tile would incur roughly the same amount of computation. This may work if only the filter step is considered, as it makes only four numeric comparisons to check if the MBRs of two objects overlap. However, when the refinement step is also taken into account, this approach does not work because the amount of computation is dependent on the properties of the objects and the underlying computational geometric algorithm. Moreover, the replication of large objects to multiple tiles increases the amount of computation for each such tile.

To address the issues with existing spatial declustering approaches, SPINOJA uses a novel technique we call MOD-Quadtree (Metric-based Object Decomposition Quadtree) declustering. The main idea behind our technique is to create the tiles such that the amount of computation required by each tile is roughly the same. As the name suggests, MOD-Quadtree is a region quadtree variant that recursively decomposes a tile into four equal-sized tiles. The decomposition criteria is not based on the number of objects, but rather on the amount of computation estimated by a suitable *work metric*. When the amount of computation among the tiles is equalized, it is expected that it would take roughly the same amount of time to process each tile and that there would be no "straggler effect". A suitable work metric is therefore important to evenly distribute the work among the tiles.

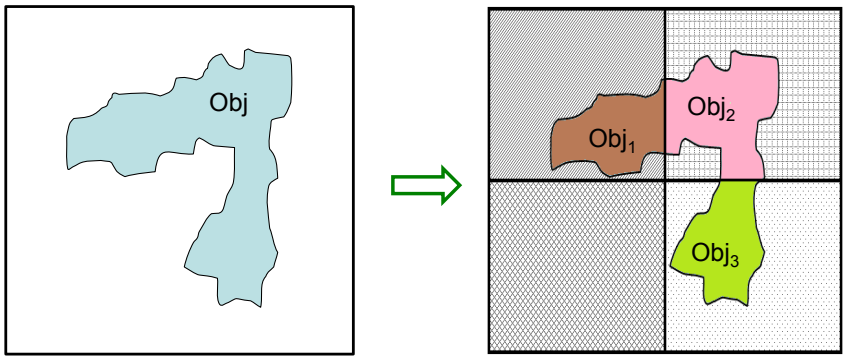


Figure 5.4: MOD-Quadtree object decomposition

Require: *tileMBR* is the MBR of the tile for which the metric is calculated; *tableList* is a list of tables containing the **original** spatial objects

```
1:  workMetric ← 0
2:  for tab in tableList do
3:      origObjList ← retrieveOverlappingObjects(tileMBR, tab)
4:      for obj in origObjList do
5:          objFrag ← getIntersection(obj,tileMBR)
6:          workMetric ← workMetric + calculateMetric(objFrag)
7:          . . .
8:      end for
9:  end for
10: return workMetric
```

Figure 5.5: Procedure getWorkMetric4Tile

Another key aspect of MOD-Quadtree is that when a tile is subdivided into four, any object that overlaps with the newly created (smaller) tiles is also decomposed along the boundaries of them. This is illustrated in Figure 5.4, in which an object *Obj* is decomposed into three fragments Obj_1, Obj_2 and Obj_3 by clipping against the four tile boundaries. The termination criteria of quadtree approaches is typically related to a threshold on the number of objects, for instance, with point region quadtree the decomposition of a tile halts when a tile contains one point. In MOD-Quadtree, the recursive decomposition stops when the total number of tiles created so far exceeds a threshold.

In each round of the MOD-Quadtree declustering algorithm the tile with the largest work metric value is selected for decomposition. The center point of its MBR is calculated before removing the tile from a global tile list. A new tile *tile0* is created and its properties are set, in particular, the work metric is calculated for the newly created tile by invoking the procedure *getWorkMetric4Tile*. Then *tile0* is inserted into the global tile list. Similarly, tiles *tile1*, *tile2* and *tile3* are created and inserted into the global tile list. The MOD-Quadtree declustering algorithm is parallelized to attain multi-threaded performance and reduce overall execution time. Note that the tiles are ordered with Hilbert SFC (space-filling curve) orientation to enable traversing them using Hilbert order when assigning to Workers. This allows nearby tiles with similar object densities to be assigned to different Workers during query execution. Hilbert ordering also prevents a drawback with tile assignment approaches such as round robin or range partitioning. With round robin assignment, the tiles assigned to a Worker may form long columns when the number of tiles are integral multiples of the number of Workers. This may degrade overall performance, as was observed by Patel et al. [90]. With range assignment large blocks of tiles may be formed and also degrade performance.

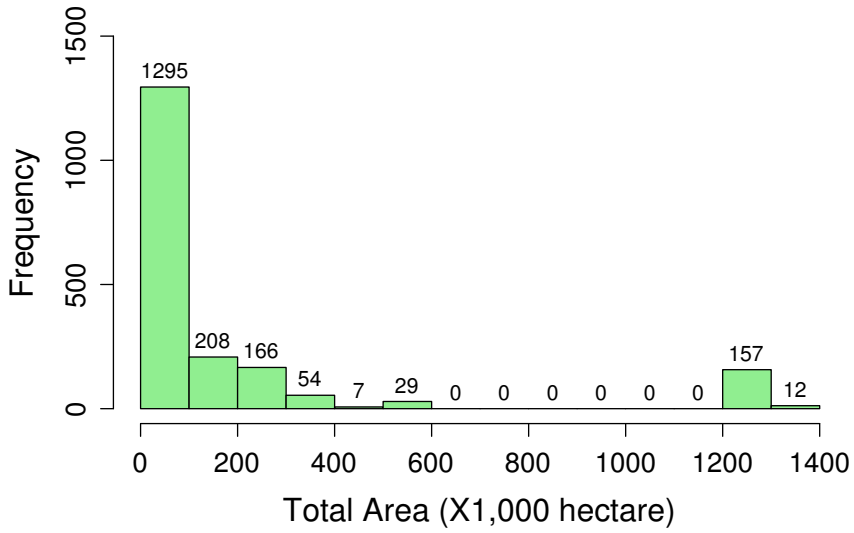


Figure 5.6: Histogram of total area in hectares per tile of Arealm (landmass polygons) table

Figure 5.5 shows the *getWorkMetric4Tile* procedure. It iterates through each spatial object that overlaps the MBR of a newly created tile *tileMBR* and determines the object fragment (line 5). Then the work metric is calculated and a local variable is updated (line 6). These steps are repeated for each of the tables involved in the spatial join queries. For convenience, we upload the original dataset into PostgreSQL tables and issue SQL queries as part of the procedure call *retrieveOverlappingObjects()*. The actual work metric calculation (procedure *calculateMetric*) depends on which work metric to use. We evaluate three metrics in Section 5.2.3.

Once the declustering process completes, the generated tile list is iterated with Hilbert SFC order and for each object fragment a record is generated. The fields of the record include the tile id, original object id, the MBR of the object, the fragment id, the fragment geometry, the fragment MBR and other fields. These records are persisted in disk files to be later retrieved when the in-memory tables are populated. For parallel retrieval multiple disk files are used for a particular table.

When the query execution engine of SPINOJA starts, first each in-memory table is populated by reading the corresponding disk files. The schema of a given in-memory table *tab* is as follows:

$\{$*ObjectId*, *ObjectMBR*, *FragmentId*, *FragmentMBR*,*FragmentGeom*, ... $\langle$other fields of *tab*$\rangle\}$

As part of the table loading process a tile bitmap index is also created. The index maps each fragment id to the tile id to which it belongs. The index is essentially a dictionary as shown below:

$\{$**key**: *TileId*, **value**: compressed bitmap of *FragmentId*s$\}$

The use of a compressed bitmap is meant to keep the memory footprint of the index low.

5.2.3 Work Metrics

The declustering phase in SPINOJA attempts to create tiles with about the same amount of computation demands. Key to this process is choosing a good work metric – one that is able to estimate the actual amount of computation needed while processing a tile. We describe and compare two different work metrics: object size based and point density based.

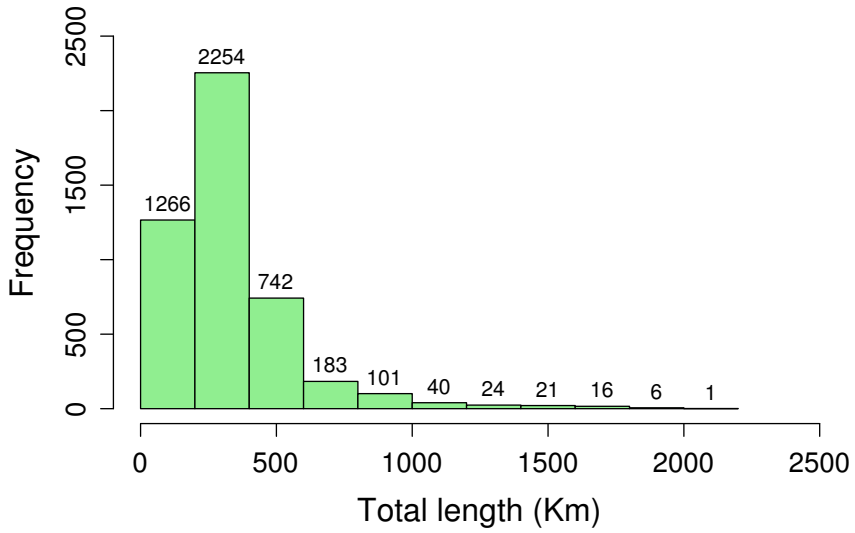


Figure 5.7: Histogram of total length in km per tile of Edges (polylines) table

5.2.3.1 Object Size Based Metrics

A potential work metric candidate is the object size. Intuitively, the larger the object the more work is needed in the refinement step. Many spatial datasets, such as geographical and VLSI datasets, are characterized by significant variation in the sizes of the objects. To demonstrate, we calculated a frequency histogram of the total area of all the polygons in the tiles created by Clone Join declustering from the Arealm table. Figure 5.6 shows the histogram for tiles with non-zero area. As can be seen, in 169 out of 1928 tiles (8.7% of the tiles), the total area of all polygons is over 1200K hectares. Usually these tiles contain very large objects such as the Death Valley National Park. Moreover, several large polygons usually cluster within the same spatial partition. These large objects require expensive refinement processing more frequently, since their minimum bounding rectangle (MBR) interacts with a larger number of other objects. In the rest (91.3%) of the tiles the total area of all polygons is less than 600K hectares. Note that the TIGER dataset already does a good job of refactoring very long polyline features (such as rivers, or roads) into smaller polyline segments. This is illustrated in Figure 5.7, which shows the frequency histogram of the total length for all polylines in each tile (with non-zero length) from the Edges table. To contrast the variability in object sizes between Arealm and Edges tables, we computed the standard deviation of the total area of all polygons and the total length of all polylines among the tiles in Arealm and Edges respectively. The standard deviation is 354335.2 for Arealm, whereas it is 235.5 for Edges. Therefore, it is expected that the processing skew would be less pronounced in "polyline and polyline" queries.

Since the MOD-Quadtree declustering decomposes the spatial objects, a metric based on object size would lead to creating smaller objects and hence reduce the overall computation. For a polygon the size of an object is its area, but for polyline objects the size is related to its length. To use a metric with uniform dimension we use the area of the MBR of an object or *AreaOfMBR* as the metric. Formally, the work incurred by tile t with this metric is given by,

$$W_t = \sum_{i=1}^{R} AreaOfMBR_i$$

where the number of objects in t is R.

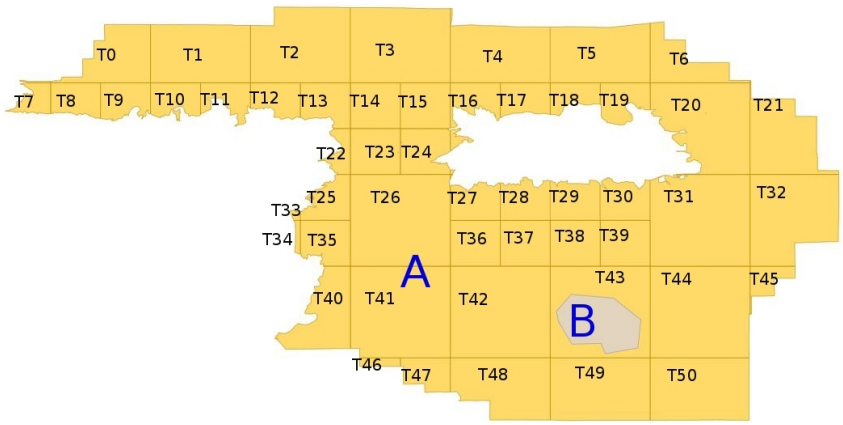


Figure 5.8: The cost of evaluating polygon overlaps polygon

5.2.3.2 Point Density Based Metrics

The processing of a spatial predicate is dominated by the refinement step, which itself consists of two steps: an intersection determination step, based on the Bentley-Ottmann plane sweep algorithm [12], and a matrix creation step, based on the dimensionally extended 9-intersection model [28]. The dominant cost component of these two is the plane sweep algorithm, which has a running time of $O((n+k)\ \log n)$. Here n is the total number of points in all objects involved and k is the number of intersections in the output. Since k is not known in advance, this cost can be approximated by $O(n\ \log n)$ for relatively small k. In the existing approaches to spatial join, the actual geometry of each object from the first dataset must be compared with that from the second dataset. For objects with many points this may incur significant processing cost in the plane sweep algorithm. For instance, in Figure 5.8 there are two polygons: A (representing San Bernardino National Forest with 11,931 points) from the first dataset and B (a small lake B with 9 points) from the second dataset. The cost of the plane sweep, to determine if B overlaps A, can be approximated as $(11931+9)*\log(11931+9) \approx 112078.5$.

In the SPINOJA approach, the objects are decomposed by clipping against the tile boundaries. Therefore, only the object fragments within each tile need to be compared. In Figure 5.8 there were 51 tiles generated by SPINOJA, T0 to T50. For demonstration purposes, we label the tiles sequentially from left to right and then top to bottom, rather than using Hilbert SFC order. As can be seen, object A is split into multiple polygons at the tile boundaries and object B is entirely contained by tile T43. Thus, only the object fragment of A contained within T43 needs to be evaluated. The cost of the plane sweep can be approximated as, $(4+9)*\log(4+9) \approx 33.3$, which is significantly lower than the original. As a result, we choose the plane sweep cost based on the number of points as another work metric. We call this *PSC_NumPts*. The work incurred by tile t with this metric is:

$$W_t = P_t * \log(P_t)$$

where P_t is the total number of points in all objects in tile t.

The factor k (the number of intersections found) in the true cost of Bentley-Ottmann's plane sweep algorithm can be approximated by the selectivity of the spatial join predicate for a particular tile being evaluated. We use this as the third work metric and call it *PSC_SelectNumPts*. The work involved in tile t with this metric is:

$$W_t = Sel_t * R * P_t * \log(P_t)$$

where P_t is the total number of points in all objects in tile t, number of objects R and Sel_t the selectivity of t. For selectivity estimation any technique such as [1] can be used. Since Intersects is one of the least restrictive OGC predicate, we use the aggregate of the selectivity estimates for this predicate between each pair of tables.

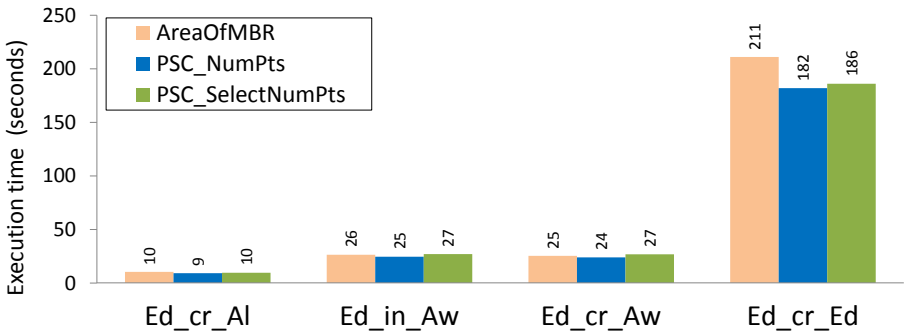


Figure 5.9: Execution times of queries with different metrics (2 cores)

5.2.3.3 Evaluation Of The Metrics

To evaluate the three metrics we executed the queries (Table 5.1) with 10K tiles generated by MOD-Quadtree declustering. We used the setup described in Section 5.3.1 with 2 cores. Figure 5.9 shows the execution times of 4 queries with the three partitioning metrics. We see that the PSC_NumPts metric performs best in all cases. Note that the difference in query execution times between the two point density metrics (PSC_NumPts and PSC_SelectNumPts) is not very significant. However, PSC_SelectNumPts requires the additional step of selectivity estimation and hence more computation.

Intuitively, a point density based metric may be superior since the cost of the plane sweep algorithm in the refinement step is dependent on the number of points. PSC_NumPts is a clear winner over AreaOfMBR, especially with the longest running query Ed_cr_Ed. Therefore, we use PSC_NumPts as the default metric in all subsequent experiments where applicable.

5.2.4 Processing Of Spatial Predicates

In the existing spatial join approaches that rely on spatial declustering, a *result aggregation* step is needed to remove duplicates from the resultset. This may be necessary because in a grid partitioning approach such as Clone Join, the same object can be replicated to multiple grid tiles. In SPINOJA, the objects are not replicated, but rather decomposed into fragments along tile boundaries. Therefore, the result aggregation step is different from other approaches and depends on the type of spatial predicate.

The spatial predicates are used to describe how two spatial objects relate to each other in terms of topological constraints. For some of the Dimensionally Extended Nine-Intersection Model (DE-9IM) [28] predicates, such as Intersects, Touches, Crosses and Overlaps, the satisfaction of the predicate by any one pair of the object fragments in one of the tiles is sufficient. This is illustrated in Figure 5.10. The polygon A has been decomposed into fragments A0 through A8. Another polygon B has been split into fragments B0, B1 and B2. Since the fragment pairs from each tile are evaluated individually, if the predicate is satisfied by *any one* fragment pair of A and B, the predicate is satisfied for the entire object. In this example, if the predicate is "A Overlaps B", it is satisfied for fragment pairs A3 and B0, A6 and B1, and A7 and B2.

For several of the DE-9IM predicates, namely, Equals, Within and Contains, *all* the fragment pairs from every applicable tile for object A and B must satisfy the predicate. In order to avoid costly refinement involving the fragments, after the filter step and prior to refinement we check to see if the MBRs of the *entire* objects satisfies the predicate, i.e., whether the predicate is satisfied by MBR_A and MBR_B. Whereas in the filter step the fragment MBRs are used, in this evaluation step the MBRs of the entire objects are used. This is illustrated in Figure 5.11.

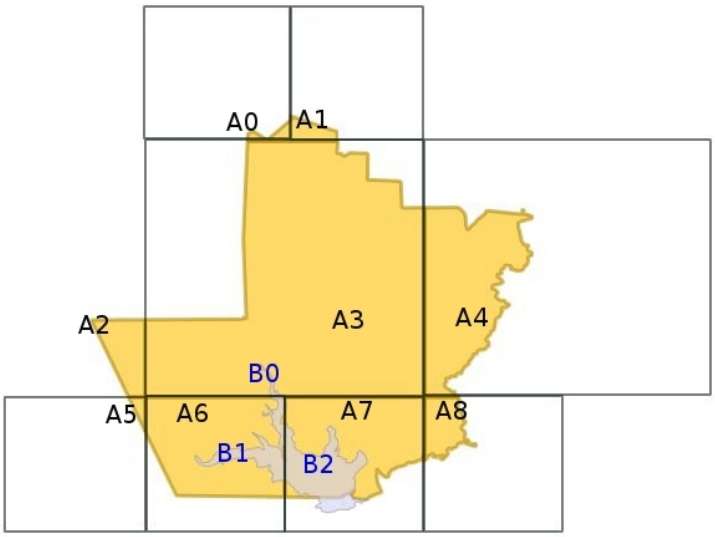


Figure 5.10: Processing of predicate Polygon overlaps polygon

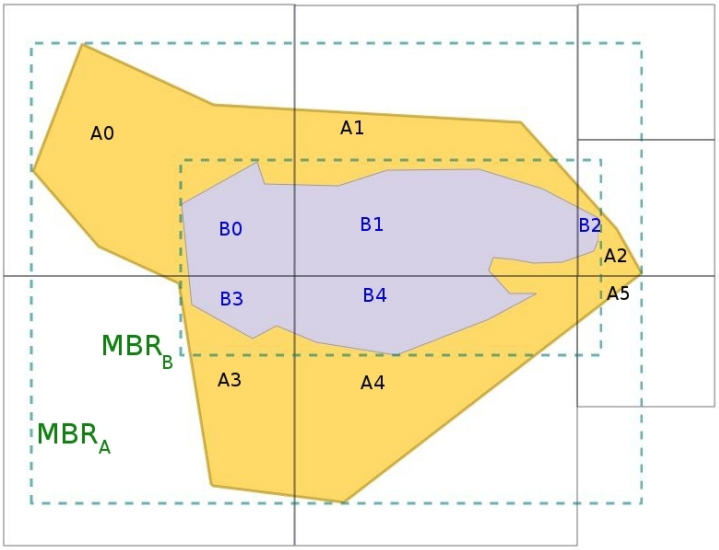


Figure 5.11: Processing of predicate Polygon contains polygon

If the predicate is “A Contains B”, all the fragment pairs, namely, A0 and B0, A1 and B1, A2 and B2, A3 and B3, and A4 and B4 must satisfy the predicate Contains.

5.2.5 Load Balancing

In a parallel query execution system load balancing is necessary to evenly distribute workload to worker threads to ensure that the overall query execution time is reduced. The aim of load balancing is to minimize the idle time by attempting to assign an equal share of work to the workers so that they are always kept busy. In practice, however it is not easy to allocate an equal amount of work to each worker. If a certain task takes much longer than others it will increase the idle times for the remaining worker threads and the slowest task will determine the overall query latency.

Load balancing could be based on static or dynamic work assignment. In the static approach, the workload is partitioned into equal sized tasks before assigning to the workers (e.g., Clone Join statically assigns the spatial partitions to workers in a round-robin fashion). This method cannot guarantee uniform idle times among the worker threads even when number of objects are the same per spatial partition. As noted in Section 5.1.1, some partitions may take much longer to process due to the properties of the objects. Dynamic work assignment addresses the load imbalance using a scheduler and a task queue. The scheduler enqueues the next task from a task pool until exhausted. Each worker dequeues an entry from the task queue and once the execution is complete it looks for the next task.

SPINOJA uses the dynamic task assignment approach. For each query job the scheduler creates many tasks, each with the id of a tile generated during the declustering step. These tasks are inserted into a synchronized task queue. Each worker thread picks the next task and performs the spatial join on the objects from both tables that belong to the corresponding tile. Since there are two distinct phases of filter and refinement, it is possible to use different strategies as to when to schedule the filter and refinement. We describe and evaluate three strategies.

In the first approach, the tuples from the left table belonging to the current tile are iterated over. For each such tuple, the tuples from the right table in the current tile are checked to see if their MBR satisfies the spatial predicate; if so, the actual geometries are compared immediately in the refinement step. This is essentially a per tile nested loop join and we call this approach *TileNLJ* (tile-wide nested loop join). The sketch of the algorithm is shown in Figure 5.12. The TaskManager creates and enqueues a task *SPJTask* for each tile (lines 1-3). A Worker retrieves the next task from the queue and obtains the tile bitmaps for the tileid from the left and right tile indexes (lines 10-12). Then for each entry in the left tile bitmap the fragment geometry is retrieved from *leftTable*. Similarly, in the inner loop the fragment geometries are retrieved from the *rightTable* (lines 13-16). In the filter step the MBRs of the two geometries are checked for intersection (line 18). If true, the refinement step is performed with the actual geometries and for the spatial predicate *spPredicate* (line 20). If the predicate is satisfied, the corresponding object ids are added to the local resultset (lines 21-23). When all the tasks are completed, the TaskManager retrieves the local resultsets and performs the result aggregation and duplicate elimination (line 5-7).

In the second approach, a tile-wide plane sweep is performed to do the filter and then the refinement step is immediately conducted on the candidate tuple pairs generated by the filter. The plane sweep technique, used in [98, 18, 90], involves sorting the object fragment MBRs on their lower x-coordinate. Then the MBR with the smallest x-value is chosen and all the MBRs from the other table that overlap it along the x-axis are chosen. Then those MBRs are checked for overlap along the y-axis, and candidate pairs are generated with the fragment ids of the matching MBRs. This process is repeated until all tuples from both the tables are merged. All the candidate pairs from a tile that pass the filter step then go through the refinement. We call this scheduling approach *TilePS* (tile-wide plane sweep) and the algorithm sketch is shown in Figure 5.13. The TaskManager steps for this are the same as in Figure 5.12. For a particular tile, the Worker retrieves and adds the MBRs of all fragment geometries from the *leftTable* and *rightTable* into two lists (lines 7-12). These two lists are used as inputs to the tile-wide plane sweep in the filter step (line 14). The fragment id pairs from two tables that satisfy this plane sweep step constitute the *candidate set*. The actual geometries for each member of this set are tested to see if they satisfy the predicate in the refinement step (lines 17-19).

The filter step of the third scheduling strategy is similar to that of the second approach. However, unlike an immediate refinement that follows the filter step, in this approach the refinement step is performed separately for a fixed size batch of candidate set. The idea behind this strategy is to evenly distribute the refinement load among the worker threads. Some tiles may produce more candidate set pairs in the filter step than other tiles. So if the refinement step is also performed by the same thread as the filter, some threads may end up doing more

refinement work. For this reason, the candidate pairs generated in the filter step are grouped together in batches of size *BATCH* (we use *BATCH*=100) to create a new task. The Worker enqueues these tasks in the queue after the tile-wide plane sweep filter step. Since the refinement step is executed as part of a separate task, we call this scheduling approach *TilePS_sepRefine* (tile-wide plane sweep with separate refinement; Figure 5.14). Again, the TaskManager steps for this are the same as before. After the tile-wide plane sweep is performed (line 15), the Worker iterates over the candidate set and produces batches of size *BATCH*. For each batch it enqueues a task *SPJTask* of type *REFINE* (lines 17-23). When a Worker retrieves such a task from the queue, the actual geometries for each entry of the batch are obtained and the refinement step is performed (lines 26-30).

```
Require: leftTable and rightTable are the 2 tables, and leftTileIndex and rightTileIndex are the corresponding
    indexes; spPredicate is the spatial join predicate
TaskManager:
 1: while tileId in leftTileIndex do
 2:    Create a new SPJTask with tileId
 3:    TaskQueue.push(SPJTask)
 4: end while
 5: {//Wait for all tasks to complete}
 6: for worker in listOfWorkers do
 7:    localResultset ← worker.getLocalResultset()
 8:    update query resultset with localResultset
 9: end for
Worker:
10: while true do
11:    SPJTask ← TaskQueue.pop()
12:    tileId ← SPJTask.tileId()
13:    leftObjBitmap ← leftTileIndex.getBitmap()
14:    rightObjBitmap ← rightTileIndex.getBitmap()
15:    for leftBitIdx in leftObjBitmap do
16:       leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
17:       for rightBitIdx in rightObjBitmap do
18:          rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
19:          {//Filter step: pairwise MBR compare}
20:          if MBRIntersects(leftGeom.MBR,rightGeom.MBR) then
21:             {//Refinement step}
22:             if satisfies(spPredicate,leftGeom,rightGeom) then
23:                leftObjId ← leftTable.getObjId(leftBitIdx)
24:                rightObjId ← rightTable.getObjId(rightBitIdx)
25:                UpdateLocalResultset(leftObjId,rightObjId)
26:             end if
27:          end if
28:       end for
29:    end for
30: end while
```

Figure 5.12: Algorithm Load-balance TileNLJ

We implemented the three load balancing strategies and evaluated them with 10K tiles generated by MOD-Quadtree declustering. We assign each Worker to a different core using Linux setaffinity. We used the setup in Section 5.3.1 with 2 cores. Figure 5.15 shows the execution times of four queries with the load balancing approaches. TileNLJ does significantly worse than the other scheduling strategies with query Ed_cr_Ed, but does slightly better with the other three queries. The candidate set size or the number of candidate pairs generated from

```
TaskManager:
 1: {//same as before}
Worker:
 2: while true do
 3:    SPJTask ← TaskQueue.pop()
 4:    tileId ← SPJTask.tileId()
 5:    leftObjBitmap ← leftTileIndex.getBitmap()
 6:    rightObjBitmap ← rightTileIndex.getBitmap()
 7:    for leftBitIdx in leftObjBitmap do
 8:       leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
 9:       leftMBRList.add(leftGeom.MBR,leftBitIdx)
10:    end for
11:    for rightBitIdx in rightObjBitmap do
12:       rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
13:       rightMBRList.add(rightGeom.MBR,rightBitIdx)
14:    end for
15:    {//Filter step: tile-wide plane sweep}
16:    rsFragmentIdPairList ← PlaneSweep(leftMBRList,rightMBRList)
17:    {//Refinement step}
18:    for fragIdPair in rsFragmentIdPairList do
19:       leftGeom ← leftTable.getGeomAtFragId(fragIdPair.leftId)
20:       rightGeom ← rightTable.getGeomAtFragId(fragIdPair.rightId)
21:       if satisfies(spPredicate,leftGeom,rightGeom) then
22:          leftObjId ← leftTable.getObjId(fragIdPair.leftId)
23:          rightObjId ← rightTable.getObjId(fragIdPair.rightId)
24:          UpdateLocalResultset(leftObjId,rightObjId)
25:       end if
26:    end for
27: end while
```

Figure 5.13: Algorithm Load-balance TilePS

the filter steps of these approaches with different queries are shown in Figure 5.16. Clearly, for Ed_cr_Ed query the candidate set size is significantly larger with TileNLJ than the other two strategies and this is reflected in the query execution times. Figure 5.17 shows the break-down of execution times of the scheduling strategies with two queries Ed_cr_Aw and Ed_cr_Ed. For scheduling approaches TilePS and TilePS_sepRefine there is a *Tile-wide plane sweep* step, whereas for TileNLJ there is a *Pairwise filter* step. The tile-wide plane sweep, as explained earlier, is a filter step performed for all the objects in a tile, whereas the pairwise filter is done for each pair of objects as part of the nested loop in TileNLJ. The tile-wide plane sweep does a better job of reducing the candidate set size than the pairwise filter (as is evident from Figure 5.16), but takes longer. The overhead of this step causes the queries to take longer with TilePS and TilePS_sepRefine than TileNLJ for the polyline and polygon queries. However, for Ed_cr_Ed query this overhead is more than compensated by the reduction of time in refinement due to the much smaller candidate set, as can be seen in Figure 5.17, This suggests that for queries with relatively smaller filter selectivity, TileNLJ is the best approach, but for queries with a large candidate set TilePS is a better option. It is possible to design a *hybrid* strategy that chooses either TileNLJ or TilePS depending on the filter selectivity.

Interestingly, TilePS_sepRefine does slightly worse than TilePS. To see why, we present the last level cache misses with TilePS and TilePS_sepRefine in Figure 5.18. As can be seen, TilePS exhibits fewer cache misses for all queries. With this strategy, the same worker thread performs both the filter and refinement step for the

```
TaskManager:
 1: {//same as before; Create tasks SPJTask with type FILTER}
Worker:
 2: while true do
 3:   SPJTask ← TaskQueue.pop()
 4:   if SPJTask.getType = FILTER then
 5:     tileId ← SPJTask.tileId()
 6:     leftObjBitmap ← leftTileIndex.getBitmap()
 7:     rightObjBitmap ← rightTileIndex.getBitmap()
 8:     for leftBitIdx in leftObjBitmap do
 9:       leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
10:       leftMBRList.add(leftGeom.MBR,leftBitIdx)
11:     end for
12:     for rightBitIdx in rightObjBitmap do
13:       rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
14:       rightMBRList.add(rightGeom.MBR,rightBitIdx)
15:     end for
16:     {//Filter step: tile-wide plane sweep}
17:     rsFragIdPairList ← PlaneSweep(leftMBRList,rightMBRList)
18:     {//Enqueue Refinement task}
19:     Create a new SPJTask with type REFINE
20:     for fragIdPair in rsFragIdPairList do
21:       SPJTask.add(fragIdPair)
22:       if SPJTask.batchSize() = BATCH then
23:         TaskQueue.push(SPJTask)
24:         Create a new SPJTask with type REFINE
25:       end if
26:     end for
27:     TaskQueue.push(SPJTask)
28:   else
29:     rsFragmentIdPairList ← SPJTask.getFragmentIdPairList()
30:     for frIdPair in rsFragmentIdPairList do
31:       leftGeom ← leftTable.getGeomAtFragId(frIdPair.leftId)
32:       rightGeom ← rightTable.getGeomAtFragId(frIdPair.rightId)
33:       {//Refinement step}
34:       if satisfies(spPredicate,leftGeom,rightGeom) then
35:         leftObjId ← leftTable.getObjId(frIdPair.leftId)
36:         rightObjId ← rightTable.getObjId(fragIdPair.rightId)
37:         UpdateLocalRefineResultset(leftObjId,rightObjId)
38:       end if
39:     end for
40:   end if
41: end while
```

Figure 5.14: Algorithm Load-balance TilePS_sepRefine

objects of a particular tile and so the queries with TilePS have better data locality. We use TilePS as the default load-balancing strategy unless otherwise specified.

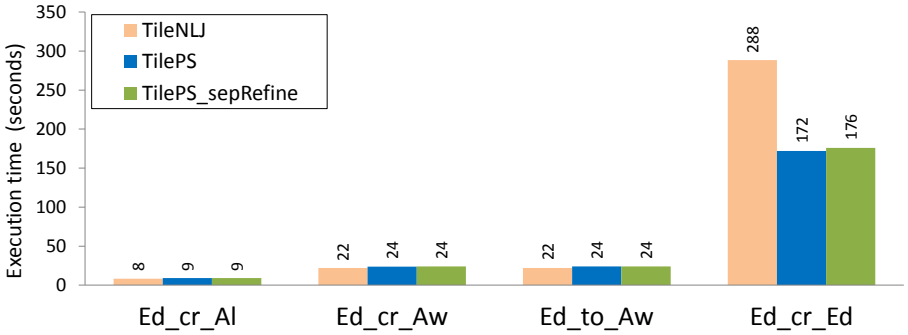


Figure 5.15: Execution times of queries with different load-balancing strategies (2 cores)

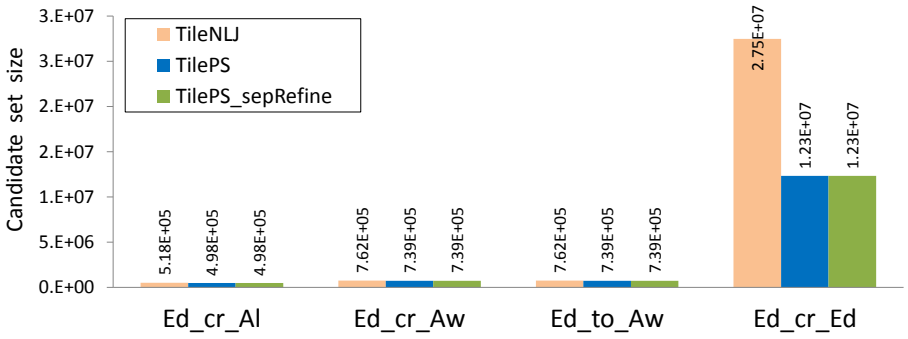


Figure 5.16: Candidate set size of queries with different load-balancing strategies (2 cores)

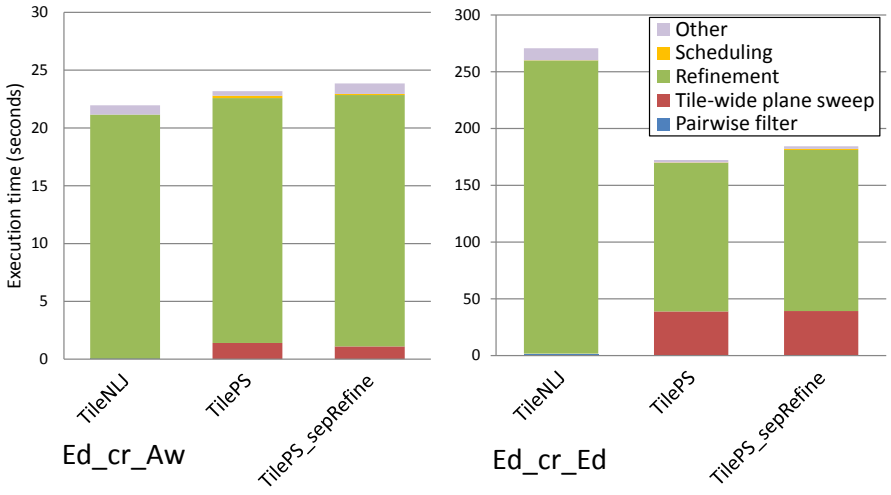


Figure 5.17: Break-down of time with different load-balancing strategies (2 cores)

5.2.6 Determining The Number Of Partitions (Tiles)

An important question in spatial declustering is how many tiles to create in order to achieve the best query execution time. In existing declustering approaches that replicate objects to all tiles that they overlap, there is a trade-off. The more tiles that are created, the more object replication is needed because of the increased probability of overlapping with the tile boundaries, leading to more memory consumption. However, more tiles imply smaller tiles with fewer objects and faster processing time per tile.

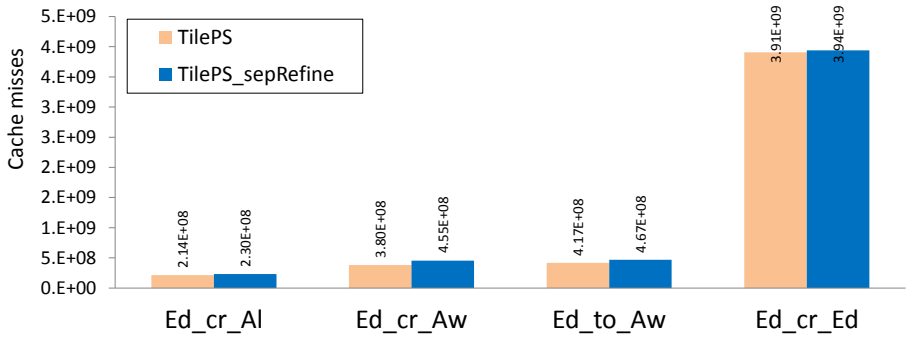


Figure 5.18: Cache misses with different load-balancing strategies (2 cores)

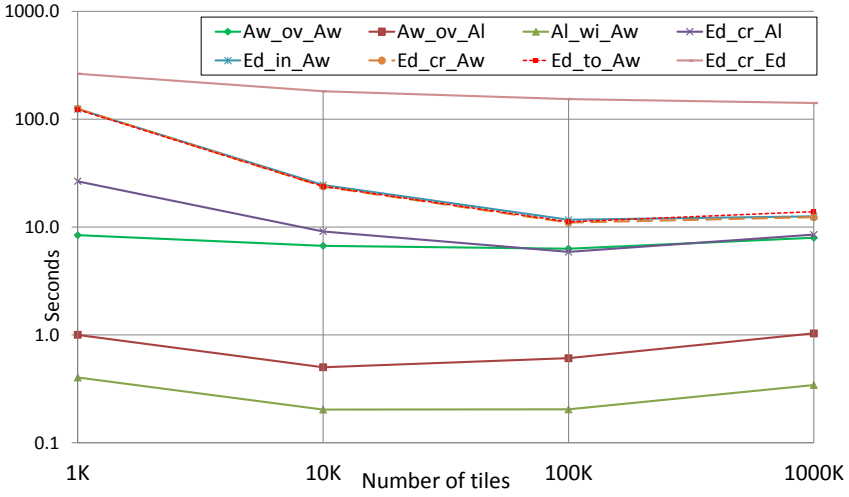


Figure 5.19: Execution times of queries with different number of tiles (2 cores)

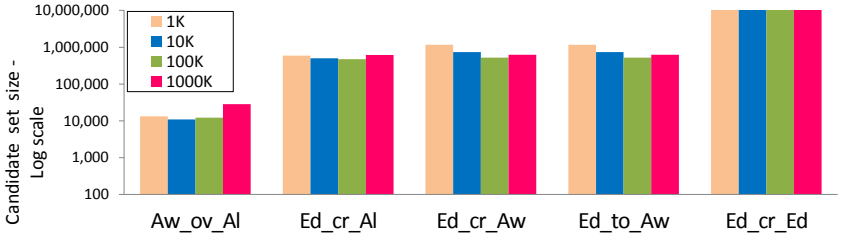


Figure 5.20: Candidate set size of queries with different number of tiles (2 cores)

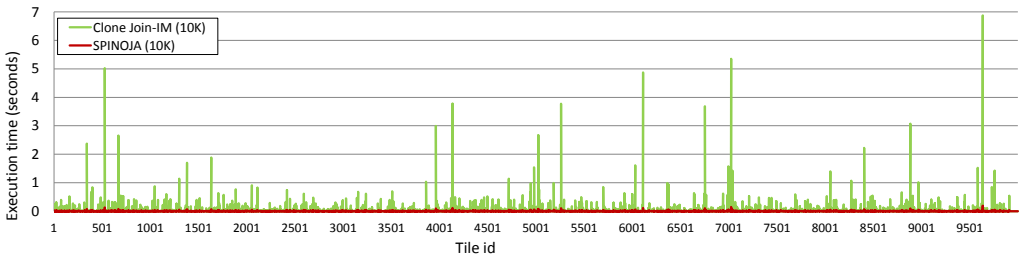


Figure 5.21: Execution times of different tiles for the query Ed_cr_Al

Table 5.2: Point density per tile for Ed_cr_Ed

1K	10K	100K	1000K
43494.7	4329.4	444.2	50.75

Since SPINOJA's MOD-Quadtree declustering does not replicate objects, but rather decomposes objects along tile boundaries, the increase in memory usage is less significant. Although smaller tiles incur less processing costs, if there are too many tiles the scheduling overhead may outweigh the benefits of partitioning. To explore these issues, we evaluate SPINOJA with 1K, 10K, 100K and 1000K tiles using the setup in Section 5.3.1 with 2 cores. With MOD-Quadtree declustering it is possible to specify how many tiles to generate as part of the termination condition. Figure 5.19 shows the query execution times with the four tile number configurations. As expected, 1K tiles results in the worst execution times for all queries. For short-running queries (i.e., overall latency of less than a second) the best performance is achieved with 10K tiles. From 1K to 10K tiles, the execution times of all queries improve. For relatively longer-running queries (i.e., overall latency greater than 10 seconds) the performance further improves when the number of tiles is increased from 10K to 100K. Beyond 100K tiles, the performance of all queries worsens, except for the longest-running Ed_cr_Ed. To explain the result, in Figure 5.20 we plot the candidate set size generated by the filter step of some of the queries for different numbers of tiles. As can be seen, for the short-running query Aw_ov_Al, the candidate set size is the smallest for 10K tiles. For the longer running queries, Ed_cr_Al, Ed_cr_Aw and Ed_to_Aw, it is the smallest for 100K tiles. The candidate set size remains relatively unchanged for the query Ed_cr_Ed. However, since the point density per tile decreases (Table 5.2) as the number of tiles increases, the amount of work per tile is reduced. Therefore, the execution time for Ed_cr_Ed continues to decrease as the tile count increases.

5.2.7 Managing Processing Skew

Parallel spatial join query execution is susceptible to processing skew due to the dataset properties. Object replication based declustering may aggravate the situation. As shown in Figure 5.2, although the vast majority of tiles take less than 0.5 seconds to process the Ed_cr_Al query with in-memory Clone Join (which we call Clone Join-IM), a few take significantly longer. SPINOJA was designed to address this processing skew. To show how well it achieves this goal, we execute the Ed_cr_Al query with 10K tiles and plot the execution times of each individual tile in Figure 5.21. As can be seen, none of the tiles takes more than 0.2 seconds. To contrast, we also plot the execution times of the tiles with Clone Join-IM. The standard deviation of the tile execution times is 0.167 for Clone Join-IM and 0.006 for SPINOJA. Thus, it is apparent that SPINOJA does a much better job in reducing the processing skew and hence improving on the overall query execution time.

5.3 Experimental Evaluation

In this section we evaluate SPINOJA in various settings. We first describe the datasets and the settings used in our experiments.

5.3.1 Experimental Setup

We use a real-world spatial dataset that contains diverse geographical features, drawn from the TIGER® data [120], produced by the United States (US) Census Bureau. This is a public domain data source available for each US

Table 5.3: Database tables

Dataset	Database table	Geometry	Cardinality
California	Edges	polyline	4173498
	Arealm	polygon	7953
	Areawater	polygon	39334

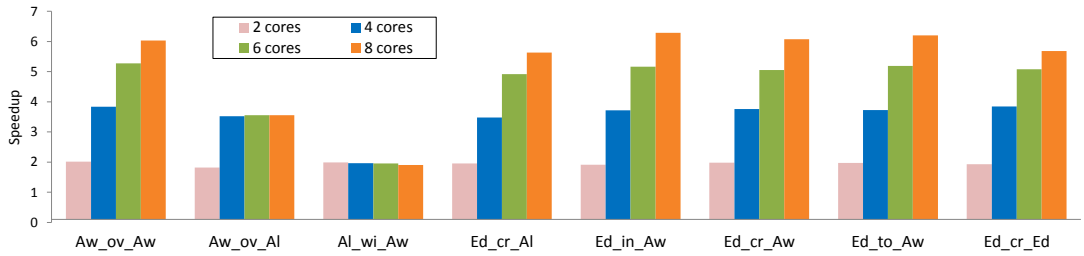


Figure 5.22: Multicore speedup of SPINOJA over 1 core performance (2, 4, 6 and 8 cores)

state. The dataset that we use consists of the polyline and polygon shapefiles for all the counties of California. We merged the shapefiles to create single shapefiles. Table 5.3 shows the details of each, including the geometry and cardinality. We use the best of 3 warm runs to report the query execution time results for the queries in Table 5.1.

The experiments were conducted on a machine having 16 GB memory, eight 3.0 GHz Intel Xeon CPU cores with 6 MB cache, and an 880 GB 7200-RPM SATA disk. We run Ubuntu 10.04 Lucid 64-bit with kernel version 2.6.32-33-generic as the OS.

5.3.2 Multicore Scalability

SPINOJA's declustering mechanism already does a good job of reducing the overall query execution time by minimizing the processing skew. Since it is a parallel spatial join approach, we are also interested in its multicore scalability. To evaluate the multicore performance we executed the queries with 2, 4, 6 and 8 cores. We use TilePS as the load-balancing strategy and 100K tiles.

Figure 5.22 shows the speedup achieved with 2, 4, 6 and 8 cores over the execution times of SPINOJA with 1 core. All queries with latency more than 1 second exhibit significant decrease in execution time, obtaining benefit from the addition of cores. These queries show a "staircase" pattern in the reduction in latency. With SPINOJA the queries achieve near linear speedup in the number of cores.

5.3.3 Comparison With Other In-Memory Spatial Join Approaches

To evaluate the performance of SPINOJA against in-memory spatial join approaches we implemented TOUCH [82]. We also implemented **in-memory** versions of Clone Join [91], INLJ (indexed nested loop join) and NLJ (nested loop join). Both TOUCH and INLJ use a bulk-loading in-memory R-tree variation called STR-tree that offers the best query performance. Since SPINOJA is a parallel spatial join, in order to do a fair comparison we **parallelized** each of these approaches. The Join Phase of TOUCH was parallelized by assigning cells to different threads in a round-robin manner. We call this parallel version **TOUCH-P**. The other phases of TOUCH (Tree Building and Assignment) were processed using pre-processing steps and are not part of the comparison.

The INLJ approach was parallelized by range partitioning the larger table among the worker threads and creating an STR-tree index on the other table. Similarly NLJ was parallelized by range partitioning the larger table

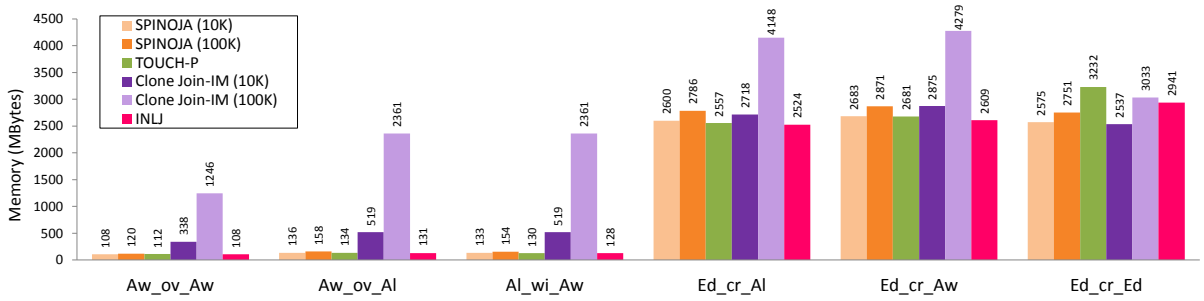


Figure 5.23: Memory usage for queries with SPINOJA vs. other approaches (8 cores)

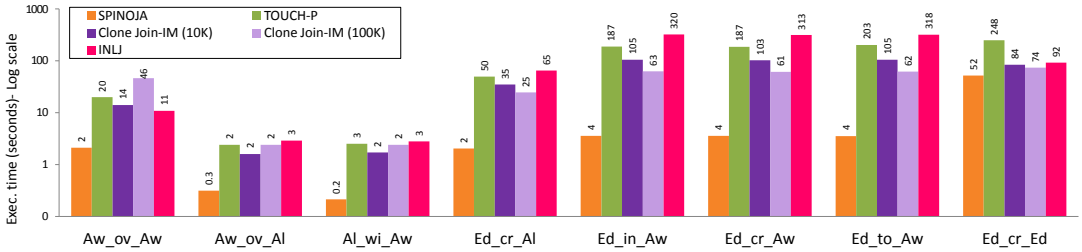


Figure 5.24: Execution times of queries with SPINOJA vs. other approaches (8 cores)

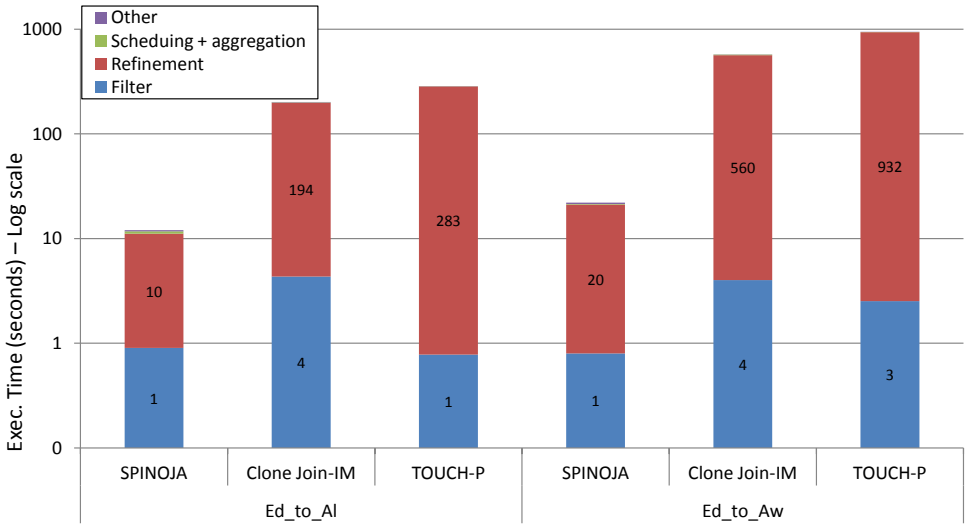


Figure 5.25: Breakdown of execution times of queries with SPINOJA vs. other approaches (sequential execution)

among different threads. We found that the NLJ approach takes too long for queries with Edges and Areawater tables and so we omit NLJ from the results.

First, we report the memory usage of the different approaches. With Clone Join the number of tiles is a crucial factor in the overall memory consumption. The authors of Clone Join used 10K tiles in the spatial declustering function for their experiments and did not report the effect of the number of tiles. We implemented in-memory versions of Clone Join with both 10K and 100K tiles and call them **Clone Join-IM (10K)** and **Clone Join-IM (100K)** respectively. To compare, we also present the memory usage of SPINOJA with the same tile configurations. As shown in Figure 5.23, Clone Join-IM with 100K tiles requires significantly more memory than all other approaches. This is because with more tiles the probability of an object overlapping multiple tiles increases and

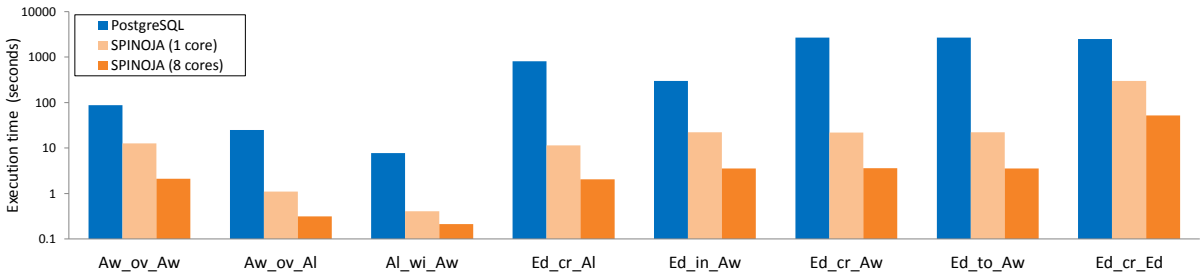


Figure 5.26: Execution times of queries with SPINOJA vs. PostgreSQL

hence the likelihood of it getting replicated to each such tile also increases. SPINOJA uses slightly more memory with 100K tiles than with 10K, but is still comparable to the memory usage of TOUCH-P.

For SPINOJA we report the execution times with 100K tiles, as this gives the best performance for the longer running queries. The execution times of the queries with SPINOJA and the other approaches are shown in Figure 5.24. All the query execution times are reported for the 8-core execution. It is evident that SPINOJA outperforms all other approaches by wide margins. SPINOJA is 90X faster than INLJ in the best case. Compared to TOUCH-P, SPINOJA is significantly faster, particularly, with the queries involving Edges and Areawater. For instance, SPINOJA is about 50X faster than TOUCH-P with Ed_cr_Aw query. Note that TOUCH-P does better than INLJ in most queries but TOUCH-P performs worse with self-join, namely Aw_ov_Aw and Ed_cr_Ed. This is because its Assignment step is unable to filter objects for self-join.

Between the two Clone Join variations, there is no clear winner in terms of execution time: Clone Join-IM (10K) does better than Clone Join-IM (100K) with the short running queries and Clone Join-IM (100K) performs better than the other with the longer running queries. In fact, Clone Join-IM (100K) is faster than both TOUCH-P and INLJ with the longer running queries. However, Clone Join-IM (100K) requires significantly more memory than Clone Join-IM (10K). Due to this increased memory usage, Clone Join-IM (100K) may not be practical with larger datasets. SPINOJA does better than Clone Join-IM (100K) and Clone Join-IM (10K) in all cases. For instance, with Ed_to_Aw query SPINOJA is about 30X faster than Clone Join-IM (10K) and 17X faster over Clone Join-IM (100K). With the purely polyline based query Ed_cr_Ed, SPINOJA achieves the least (but still considerable) speedup over the other approaches: 1.4X over Clone Join-IM (100K) to 4.8X over TOUCH-P. Because, this query exhibits less processing skew than the others, as explained in Section 5.2.3.1.

To explain why SPINOJA performs better than Clone Join-IM and TOUCH-P, we plot a graph (Figure 5.25) with breakdown of execution times spent in filter, refinement, scheduling and aggregation and other. As can be seen, TOUCH-P and Clone Join-IM spends 98% and 96% of the times in Refinement. For instance, with Ed_to_Aw query, Clone Join-IM and TOUCH-P take 560 seconds and 932 seconds respectively in refinement, whereas for SPINOJA this was only 20 seconds. This difference contributed toward the significant performance gap between SPINOJA and others.

5.3.4 Comparison With PostgreSQL

SPINOJA is essentially a main-memory column store database optimized for spatial join. A direct comparison with a general purpose relational database, such as PostgreSQL, cannot be made because SPINOJA lacks several features such as transactions and a query optimizer. However, by comparing SPINOJA with PostgreSQL we make the case that a custom-built database might be better suited for spatial join than a general purpose relational database. In Figure 5.26 we compare the execution times of the queries with PostgreSQL 8.4.4. The bufferpool

of PostgreSQL was set to 8 GB so that the database completely fits in memory. Since PostgreSQL does not yet support intra-query parallelism, we present the query execution times with SPINOJA for one core, alongside those with 8 cores. The single-core performance of SPINOJA over PostgreSQL is quite good for both short and long-running queries. For instance, SPINOJA attains a speedup of 19X with Al_wi_Aw and a speedup of 123X with Ed_cr_Aw query. Consequently, the 8-core speedup of SPINOJA over PostgreSQL is significant. For example, this speedup is 749X with Ed_cr_Aw. Note that with the Intersects predicate (i.e. Ed_in_Aw query), SPINOJA achieves relatively less speedup than other "polyline and polygon" queries with different predicates. This is because PostgreSQL uses an optimization with this predicate, which SPINOJA currently does not implement.

5.4 Chapter Summary

Spatial join is important in many traditional and emerging spatial data analysis applications. We have introduced SPINOJA, a parallel in-memory spatial query execution infrastructure. It is designed to accelerate spatial join performance by exploiting large main-memory available in modern machines and rising core count.

Previous approaches to disk based parallel spatial join used spatial declustering that attempted to distribute the objects to tiles based on object count. Moreover, they focused on the filter step. However, when the more compute-intensive refinement step is taken into account, the object properties, such as size or point density, become important. As a result, even when the number of objects per tile is roughly equal, processing skew can occur. In an in-memory spatial join query, in the absence of disk latency, the processing skew becomes the key bottleneck to parallel performance.

SPINOJA addresses this processing skew by using an object decomposition based declustering. The declustering uses a work metric to equalize the amount of computation demanded by each tile. We also present three load balancing strategies. With extensive evaluation we demonstrate that SPINOJA does significantly better than in-memory implementations of previous parallel spatial join approaches and the parallel performance of a recently proposed in-memory spatial join.

Chapter 6

High Performance Spatio-Temporal Index For Location-Based Services (LBS)

6.1 Introduction

Started as an enabling technology for mobile asset tracking, LBS is the driving force behind a range of emerging applications such as location-based games and social networks, location-aware search and personalized advertising and weather services. The combination of factors, including the proliferation of mobile devices and the rise of novel applications, has led to the rapid growth of spatio-temporal data. Besides the data volume, LBS workloads also exhibit "velocity" of data. LBS applications are characterized by a very high rate of location updates and many concurrent location-oriented queries. These queries can be classified into "historical" or "past" queries, "now" or "present" queries and "predictive" or "future" queries.

Commercial LBS offerings need to satisfy a diverse and often conflicting set of features. The customers would like to generate reports with different degrees of complexity. Some of these reports are based on present queries, or predictive range queries including spatio-temporal range searches and fleet-status queries. Others are long running historical reports based on past queries, such as fleet activity during the last week or driving behavior analysis etc. With a large customer base the query workload in a given LBS system can be significant. At the same time the service providers must store and efficiently index location updates received at very high frequency. Šidlauskas et al. [126] noted that this rate can easily surpass 1 million or more location record updates per second. To address this complex set of requirements commercial LBS providers rely on relational databases to manage their data. To determine how traditional databases perform with update-intensive spatio-temporal workloads, we benchmarked [101] a database popular with the LBS industry. The database engine supports an R-tree based spatial index. We disabled the autocommit property and configured the buffer pool size to 64 GB so that the database completely fits in memory. The throughput achieved by the database was about 45K inserts/second, which is arguably low in this context. Our goal is to build database support for a full-fledged commercial LBS system. To this end, we intend to support millions of updates per second for one million moving objects, along with thousands of concurrent historical, present and predictive queries. This presents a few research challenges, including insert-optimized database storage and efficient spatio-temporal indexes.

The insert performance of traditional relational databases falls short, even when the database fits completely in memory. We argue that to meet the performance demands of Location-Based Services, it is essential to exploit recent advances in hardware and database technologies. Modern server-grade machines have large memory capac-

ity, and some have hundreds of GB, even a few TB of memory. Each machine comes equipped with several tens or even hundreds of cores. Database technologies have evolved to take advantage of the abundant memory and processor capacity and in-memory databases are an active area of research. Several database vendors also have in-memory offerings, such as SAP HANA [111] and VoltDB [123]. When it comes to main-memory database storage design, there are a few options available. However, we argue that it is possible to achieve better insert throughput than those from the in-memory row or columnar storage techniques. The location record updates are insert-only operations. There is no need to delete records from random locations in the table. A new record can always be inserted at the end of a table. To insert a new record in the table a new record id (RID) must be acquired, which is a unique identifier for each record. This can be performed without acquiring a lock, by incrementing the RID counter using an atomic CPU instruction. Also, location records are stored without any data compression, because typical compression techniques such as dictionary encoding are not that beneficial for position data. So, the associated cost of compressing records can be avoided. Furthermore, unlike regular transactions, the location updates do not need the strongest consistency guarantee. It may be considered okay even if a few location updates are missed.

Due to its importance in LBS applications, indexing moving objects has been an active area of research. Although a number of spatio-temporal indexes have been proposed [79], most of them are based on B-tree or R-tree and hence they suffer from the underlying limitations. These indexes are unable to sustain a very high rate of location data inserts/updates and the majority of these proposals deal with answering only present or predictive queries. Only a few of them can answer historical, present and predictive queries. A recently reported LBS system that supports historical and present queries is MOIST [64]. MOIST achieves 8K+ and 60K updates per second with one and 10 servers respectively for one million objects. Realizing the potential of memory based approaches, recently a parallel main-memory moving object indexing technique was proposed [126]. They achieved good update performance by exploiting efficient in-memory data structures. However, their system was designed to support present queries only, as at most 2 location records per object are maintained in memory. Hence it is not suitable as a real-world LBS system. To offer commercial LBS it is essential to maintain the history of each object and have support for historical queries.

We propose a few key ideas that take advantage of the modern hardware resources [102]. They are outlined below:

1. By taking advantage of large memory capacity we maintain the location table data and indexes for the past N (configurable) days in memory. Older data and indexes are maintained on disk. We also present an insert-efficient main-memory storage that is particularly suitable for LBS. Our storage model, *column fragment*, exploits insert parallelism by creating "multiple insertion points" that enable lockless insertion in each fragment.

2. We introduce a novel parallel in-memory spatio-temporal index called PASTIS (PArallel Spatio-Temporal Indexing System). PASTIS decomposes the spatial domain into grid cells and for each grid cell partial temporal indexes are maintained for moving objects that visited the cell. This makes it possible for separate threads to process the updates concurrently for different grid cells. The partial indexes are constructed as compressed bitmaps. Bitmaps offer memory efficiency and fast bitwise operations.

With extensive evaluation studies we demonstrate that our system achieves excellent update throughput (millions of updates per second) and at the same time achieves very good query execution throughput (thousands of queries per second). For instance, our system achieves 3.2 million+ updates per second with 1 million moving

objects. The update throughput of our system is significantly better than the reported throughput of MOIST and the relational database that we benchmarked.

6.2 Design Considerations

We outline several key ideas to achieve the goal of handling high update rates and concurrently supporting past, present and predictive queries for LBS. We expand on the ideas introduced in Section 6.1.

6.2.1 Main-Memory Storage For LBS

Many applications exhibit skew in the service requests for objects. A news website, for instance, receives significantly more page hits for breaking news than older news from the previous week. Such skew patterns can also be observed with the LBS workloads. More recent data from a vehicle fleet, for instance, is accessed much more often than older data. Wu and Madden [127] noted the significant request skew in LBS and reported that more than 50% of all queries accessed data from just the last day in their LBS system. Consequently, companies providing LBS typically maintain the past 30 to 45 days of data online in relational databases and older data is archived using database archiving techniques. Although the volume of location data even for 45 days is significant, it is feasible to keep the entire online data and the corresponding index in main memory. This is made possible by the availability of large main-memory machines, having hundreds of GB and even a few TBs of RAM. These observations led us to design our system as an in-memory database. We maintain in memory the past N days of data, and data older than that are kept on disk.

Besides the data volume, LBS workloads also exhibit high "velocity" due to frequent location updates from each moving object. Therefore, it is not enough to just provide in-memory storage capacity, but the storage must support record insertion at a high rate. To maintain history for each moving object the location records need to be stored in a database table. We assume that this *Location* table has the following schema:
$\{ObjectId, Latitude, Longitude, Direction, Speed, Datestamp\}$

We propose an insert-efficient in-memory storage for the *Location* table. It is based on the idea of exploiting parallelism while inserting new location records in the table. For this, an in-memory column is partitioned into a number of segments called *column fragments*. Each column fragment is implemented as a dynamic array of primitive data types. The different column fragments from a table belonging in the same range of record identifiers are grouped as a *table fragment*. In Section 6.4, we provide further details of our storage structure.

6.2.2 A Novel Parallel In-Memory Index

To answer past, present and predictive queries a high performance spatio-temporal index is necessary. The index must support high update rates. Traditional relational databases use R-tree based spatial index. These indexes inherit the underlying limitations. For instance, to add a new record in the index page level or R-tree node level locks must be acquired. This limits the multi-threaded scalability of these approaches.

We propose a novel in-memory spatio-temporal index we call PASTIS (PArallel Spatio-Temporal Indexing System). The main idea behind PASTIS is to decompose the spatial domain into grid cells and for each grid cell maintain partial temporal indexes for moving objects that visited the cell. The partial temporal indexes are constructed as compressed bitmaps. The compression of bitmaps provides memory storage efficiency. Bitmaps support fast random update operations. Moreover, the fast bitwise logical operations enabled by bitmaps allow us

to attain good query performance. Our spatio-temporal index exploits multi-core parallelism by letting separate threads process the inserts/updates concurrently for different cells.

For an incoming location update u, a new record id *RID* is obtained and then the fields of the update record are appended into the corresponding column fragments. Updating the spatio-temporal index involves first computing the current grid cell id, then the corresponding temporal index data structures are updated. Further details of PASTIS can be found in Section 6.5.

6.3 Overall System Organization

Figure 6.1 shows the overall system organization. Our system is highly multi-threaded and takes advantage of the available processing cores. It uses three different thread-pools: to perform inserts into the Location table, to conduct updates into the index, and to execute queries. To reduce concurrency conflicts we use multiple record and index inserter queues and multiple query queues, such that each thread in a thread-pool manages a separate queue. Since the density of moving objects per grid cell changes continuously and is skewed, load-balancing is essential. This is discussed in Section 6.5.3

6.4 Insert-Efficient Main-Memory Storage

With the advent of large main-memory machines, there have been a number of projects addressing storage organization for main-memory databases. Row-store databases are typically utilized for update intensive workloads, whereas column-oriented databases are exploited for analytical workloads. SQL Server's OLTP engine Hekaton [36] is an example of row-oriented main memory storage. IBM's Blink [99] is also a row-store main-memory database. MonetDB [116] is among the first column-store databases and it materializes intermediate results in main memory. To support updates with column-store usually a separate write-efficient store is maintained, besides a read storage. The main-memory column-store database C-store [116] uses a read-optimized store (RS) and writable store (WS), where all the inserts and updates are handled by WS. Data is moved from WS to RS as batch inserts by a tuple-mover. C-store maintains its data without any compression. SAP HANA is a commercial in-memory column oriented database that has a delta store for inserts and updates and a main store to support analytical queries [111]. Records in delta store are periodically merged into the main store. It uses dictionary encoding to compress the data. However, the movement of data or the merge process is a relatively time consuming operation. As we have argued in Section 6.1, due to the requirements of LBS it is possible to improve on the insert performance over the existing approaches. Next we describe our insert-efficient main-memory storage.

To support high location update rates in LBS it is imperative that the underlying storage for the Location table be optimized for record inserts. To achieve parallel performance with concurrent insertions in the Location table, we use the concept of "multiple insertion points". In this organization, each in-memory column is partitioned into a number of segments called column fragments. The entire record id (RID) space is subdivided such that each column fragment operates in a different RID space. The column fragments from different columns that belong in the same range of record identifiers constitute a table fragment. Each object is assigned to a table fragment so that all RIDs corresponding to the location updates for an object are monotonically incremental in its RID space.

Formally, let the entire RID space be $\{x|x = 1, 2, 3, ...N\}$, where N is the highest RID. Let there be F table fragments, identified $y = 1, 2, 3, ..F$. Then the RID space R_y of table fragment y is the interval $[R * (y - 1) + 1, R * y]$, where $R = N/F$. Let there be M moving objects in the system. They are grouped into sets of objects $\{V_i|i = 1, 2, 3, ..F\}$, where each set V_i has $V = M/F$ objects. A function f is used to map each set V_i to a

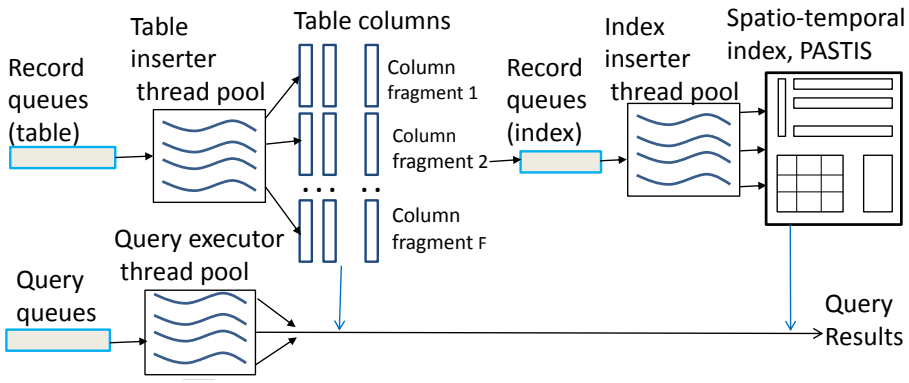


Figure 6.1: System organization

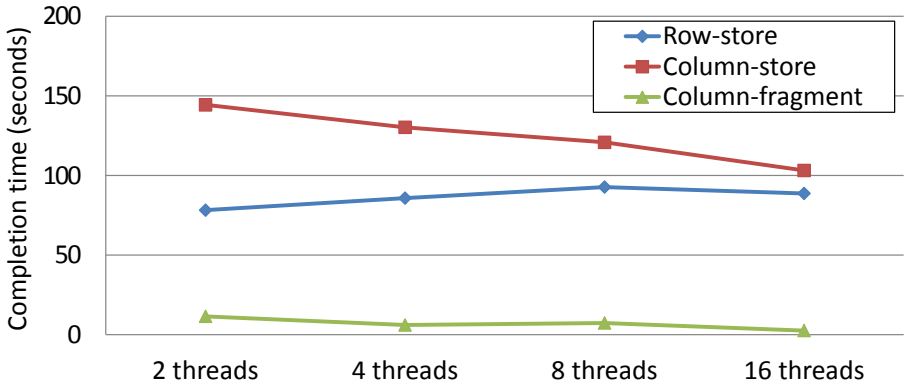


Figure 6.2: Completion time (1 billion record insert) with different storage organization

RID space R_y, where $f: V_i \rightarrow R_y$. When a new location update is received from a moving object $v_i \epsilon V_i$, it is always inserted at the end of its assigned table fragment. For instance, if v_i is mapped to R_y and r_y is the last assigned RID within R_y, a new RID is obtained by incrementing r_y by 1 before the new location update can be processed. A separate thread is dedicated to handle the inserts for each table fragment, allowing concurrent updates into the same table without lock conflicts. To increment RID in a lockless manner, atomic instructions such as Compare-and-Swap or Fetch-and-Add can be used. When determining how many table fragments to be created, one option is to consider how many threads that could be dedicated for processing inserts into the table. If there are F threads available for this task, then F table fragments could be created. We take this approach, but other criteria such as, the number of object groups, could be used.

As mentioned earlier, main-memory columnar storage models, such as C-store or SAP HANA, maintain a separate writable storage, which must be periodically merged to the read-optimized store. This merge process can be expensive. In our model we have a single storage for the Location table that is used for both reads and writes. Since location data does not benefit from compression, we store the data "as is" and avoid the corresponding cost of compression using techniques such as dictionary encoding. We also do not perform any sorting on the columns of the Location table, unlike other column-stores. The new records from a particular object are always inserted at the end of the same table fragment, thus they are implicitly sorted by the timestamp column.

To evaluate the record insert performance of column fragment against row-store and column-store, we implemented memory resident column-store and row-store storages. Our column-store implementation resembles the L2-delta store of SAP HANA. Records are inserted into the Location table as is, and no compression or sort-

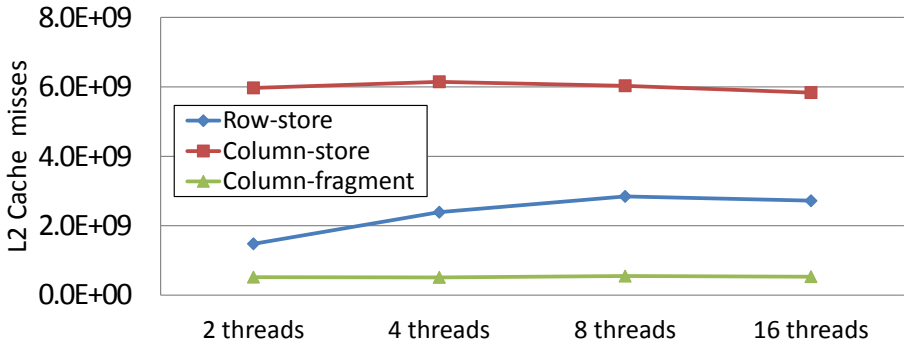


Figure 6.3: L2 cache misses with different storage organization

ing is performed. Prior to inserting a record, a new insert position or RID is acquired using the Fetch-and-Add atomic operation. We also experimented with atomic Compare-and-Swap (CAS), but found Fetch-and-Add to be faster on our hardware. Our experiment involves inserting 1 billion location records with different numbers of inserter threads (2, 4, 8 and 16). The details of the dataset are in Section 6.7.1. To avoid any disk I/O, we preallocated enough memory to completely accommodate the entire dataset. The completion times of inserting 1 billion records with different numbers of threads are shown in Figure 6.2. As can be seen, the column fragment is 7X and 12X faster with two threads, and 34X and 39X faster with 16 threads than row and column stores respectively. This is due to the fact that with column fragments there is no contention over the atomic operation of acquiring a new RID. Also as each column fragment is managed by a dedicated thread there are fewer cache misses compared to the other two. We report the L2 cache misses observed during the insert operations in Figure 6.3. Note that the completion time with row store is better than that of the column store, because of fewer cache misses. The experiment demonstrates that column fragment is efficient for inserts. Row-store could also benefit from such fragmentation.

We adopt column fragment for the Location table because column-based storage is more efficient for analytical workloads. Currently, persistence of the Location table is provided by memory-mapped files. Location updates do not require the strongest consistency guarantee. Unlike transactional workloads, missing a few location records may be tolerated.

6.5 PASTIS

In this Section we describe our spatio-temporal index, shown in Figure 6.4. First we describe the internal data structures. Then, we describe the update and query algorithms.

6.5.1 Index Structure

The index is organized as a versioned grid in which the spatial domain is subdivided into regular sized grid cells. For a highly skewed location dataset it is also possible to structure the grid cells as quad-tree blocks, in which each block is recursively split into four blocks until they meet a criteria. PASTIS orders the grid cells using Z-order (also known as Morton-order) space filling curve. Although it is possible to use a different space filling curve order such as Hilbert curve, the calculation of Z-order code is much faster than that of Hilbert order. Therefore any location record can be positioned into a corresponding grid cell with its Z-order code. Each grid cell maintains partial temporal indexes for the objects that visited that cell. Whenever a location update u is received,

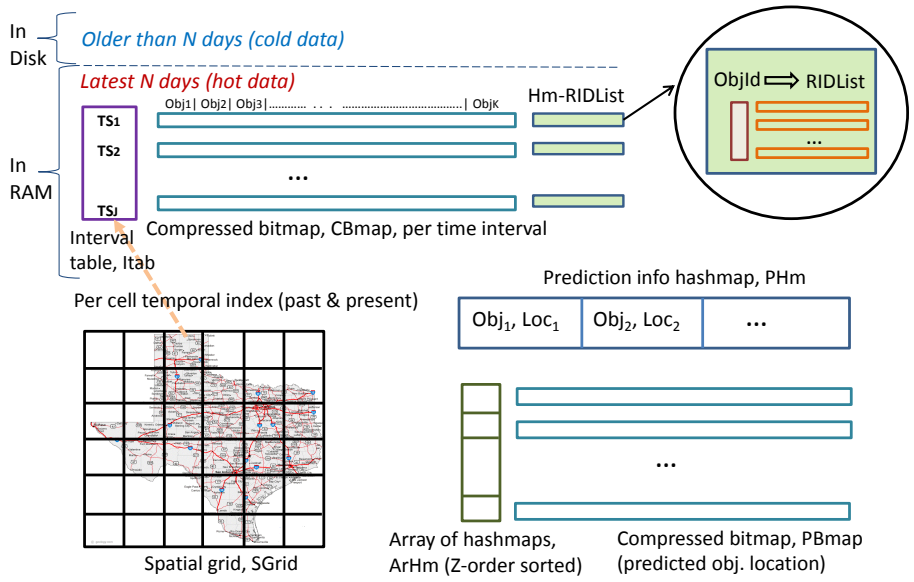


Figure 6.4: Structure of spatio-temporal index, PASTIS

the corresponding grid cell $SGrid_i$ (where i=1 to I, with I the total number of cells) is located and the partial temporal index structures are updated for that record.

A partial temporal index consists of an interval lookup table *Itab* with an entry for each time interval for the past *N* days. In each entry there is a compressed bitmap *CBmap* identifying the moving objects that were in the grid cell at the given time interval, and a hashmap *Hm-RIDList*. The hashmap associates each moving object with a list of record identifiers *RIDList*. Each identifier is used to locate in the Location table the actual records storing datestamp, latitude and longitude information for the object while at the grid cell during the time interval. The *RIDList* is implemented as a dynamic integer array. Time intervals are of *S* seconds (configurable value) for the past configurable number of hours *H*. For data records older than *H* hours, the compressed bitmap *CBmap* and the hashmap *Hm-RIDList* are maintained for configurable intervals of time *T* (where $T = m * S$, for a configurable factor m). The bitmap is constructed by bitwise ORing the *S* seconds interval bitmaps, while the hashmap is constructed by appending RID lists.

For a new update *u* with location *(x,y)*, datestamp ds_t, object id Obj_k and record id RID_l, let $SGrid_i$ be the determined grid cell and $Itab_i$ be the corresponding interval table. Here, ds_t is the datestamp at time *t*; k=1 to K, K being total objects and l=1 to L, L being the max record id. If ds_t maps to an existing interval TS_j in $Itab_i$, then the corresponding bitmap $CBmap_j$ and hashmap $Hm\text{-}RIDList_j$ are updated. Otherwise a new interval TS_j and related data structures are instantiated, where j=1 to J, J being total intervals. For object id Obj_k, the k-th bit of the bitmap $CBmap_j$ is set. Furthermore, to make an entry for the new record id RID_l, the object id Obj_k is used to locate the RID list $RIDList_k$ within $Hm\text{-}RIDList_j$. Finally, RID_l is appended in the RID list of $RIDList_k$.

While processing a query, the checking of the bitmap *CBmap* to identify if an object was present during an interval is a fast operation. On the other hand, inspecting the *Hm-RIDList* is a relatively expensive operation; it involves a lookup in the hashmap *Hm-RIDList*, then the retrieval of the corresponding RID list *RIDList* and finally, obtaining the actual datestamps etc, from the table. Moreover, if the temporal index interval is small there will be fewer entries in the corresponding *Hm-RIDList* and the lists of record identifiers. Hence accessing a *Hm-RIDList* and corresponding *RIDList* pertaining to a smaller interval is faster than that for a longer interval. If a query interval completely overlaps a temporal index interval, there is no need to inspect the *Hm-RIDList*. For a query

Ensure: The record was inserted into table and RId is record id
Require: *LocTable* is database table, *TemporalIdx* is temporal index
1: **while** *RecordConcurrentQueue* has more items **do**
2: {// First update predicted data structures}
3: *LocnRecord* ← *RecordConcurrentQueue.pop()*
4: *OId* ← *LocnRecord.getObjId()*
5: *PredctdGCellId* ← *computePredctdGridCellId(LocnRecord)*
6: *PrevGCellId* ← *PredictivLookupDS.getPrevGCellId(OId)*
7: **if** *PrevGCellId* != *PredctdGCellId* **then**
8: *PredictivLookupDS.removeBitmapEntry(PrevGCellId,OId)*
9: *PredictivLookupDS.addBitmapEntry(PredctdGCellId,OId)*
10: **end if**
11: *PredictivLookupDS.insertIntoPHm(OId,PredictedLatLon)*
12: {// Next update past/present data structures}
13: *CurrGCellId* ← *computeCurrGCellId(LocnRecord)*
14: *TemporalIdx* ← *SGridHashMap.find(CurrGCellId)*
15: ***TemporalIdx.insertLocnRecord****(LocnRecord)*
16: **end while**

Figure 6.5: Algorithm Update

interval that partially overlaps an index interval, the *Hm-RIDList* is checked. Smaller temporal index intervals are more likely to be completely overlapped by a query interval. These are the considerations behind maintaining partial temporal indexes with finer granular interval *S* for more recent "hot" data and maintaining partial indexes with coarser granular interval *T* for "colder" data. Therefore, queries accessing the "hot" data need to inspect smaller intervals and hence can be faster. Partial index structures for data older than *N* days are kept in disk.

The partial index structures described above are used for answering historical and present queries. To efficiently support predictive queries (anticipated future locations) we keep a hashmap *PHm* keyed by moving object id (Obj_k), the value being a tuple (Loc_k) with the predicted latitude and longitude. An array of hashmaps *ArHm* associates a compressed bitmap $PBmap_i$ with each grid cell $SGrid_i$. The bitmap represents the predicted object status within the grid for a future configurable time interval F, where $T_F = T_{now} + \Delta t$. The anticipated future positions are interpolated using the model of Dittrich et al. [37]. For a location update with current location *(x,y)*, the predicted position is calculated as $(x, y) + \vec{pv}.T_F$, (where $\vec{pv}$ is the projected velocity at time T_F) and entered in the hashmap *PHm*. Another prediction function, such as in Panda [59], could be used. If the anticipated grid cell id is different from current grid cell id, the array of hashmaps is used to locate and update the previously and currently predicted bitmaps.

6.5.2 Update Processing

In Figure 6.5 we describe the update processing algorithm. For an incoming location update *u*, a new record id *RID* is obtained and then the fields of the update record *LocnRecord* are appended into the corresponding column fragments. Updating the index involves first updating the prediction data structures. In line 5 the predicted grid cell id is computed. If the previous grid cell entry for that object is different from the predicted, then the object id is removed from the bitmap corresponding to the previously predicted grid cell and an entry is made into the bitmap for the predicted grid cell (lines 7 to 9). Also, the Loc_k object is updated in the predictive hashmap *PHm* (line 10). The data structure *PredictivLookupDS* encapsulates both *PHm* and the array *ArHm* of hashmaps. Next, the data structures for the past and present are updated. The current cell id is computed and the corresponding

```
Require: LocnRecord is the location record to insert
 1: OId ← LocnRecord.getObjId()
 2: Datestamp ← LocnRecord.getDatestamp()
 3: RId ← LocnRecord.getRecordId()
 4: RecInterval ← computeInterval(Datestamp)
 5: CBmapHmRIDList ← this.Itab.find(RecInterval)
 6: if CBmapHmRIDList = NULL then
 7:    this.RWLock.lock()
 8:    Create a new CBmapHmRIDList instance
 9:    CBmapHmRIDList.updateObject(OId,RId)
10:    this.Itab.update(RecInterval,CBmapHmRIDList)
11:    this.RWLock.unlock()
12: else
13:    CBmapHmRIDList.updateObject(OId,RId)
14: end if
```

Figure 6.6: Procedure TemporalIdx.insertLocnRecord

```
Require: OId is the moving object id, RId is the record id
1: RIDList ← this.HmRIDList.find(OId)
2: this.RWLock.lock()
3: if RIDList = NULL then
4:    Create a new RIDList instance
5: end if
6: this.CBmap.add(OId)
7: RIDList.add(RId)
8: this.HmRIDList.insert(OId,RIDList)
9: this.RWLock.unlock()
```

Figure 6.7: Procedure CBmapHmRIDList.updateObject

temporal index structures are updated (line 14).

Figure 6.6 details the steps for updating the temporal index structures with the info from *LocnRecord*. First, the temporal interval is computed (line 4) and the corresponding bitmap *CBmap* and *Hm-RIDList* structures (together encapsulated by *CBmapHmRIDList*) are updated (line 9 and 13). A lock is acquired only when a new instance of *CBmapHmRIDList* (lines 6 to 11) needs to be created. Updating the *CBmapHmRIDList* involves locating the RID list entry *RIDList* in the hashmap *HmRIDList* (Figure 6.7, line 1). Then a read-write lock is acquired, *CBmap* is updated and the RID for the location record is inserted into *RIDList* before releasing the lock (lines 2 to 8).

6.5.3 Load-Balancing And Skew Handling

PASTIS is based on spatial partitioning of the spatial domain into regular grid cells. The distribution of the moving objects (the number of objects per cell) can be highly skewed and this distribution changes over time. Therefore, load-balancing is important for location update performance, especially with very high update rates. The goal of the load-balancing approach is to minimize the variation in the number of location records processed by the update processing threads. This can be done in two ways: either assign the partitions (cells) to threads using *Range assignment* or *Round-robin assignment*. Figure 6.8 shows the Round-robin assignment algorithm. Each index inserter thread manages a concurrent queue. In the Assignment step the grid cells are assigned to index inserter threads. A table inserter thread inserts a location record in the table fragment and then inserts it into the index inserter queue based on the cell the record belongs to and the cell to thread mapping. An index inserter

thread simply processes the next record from its queue. A third load-balancing approach (we call *Adaptive*; Figure 6.9) does not do any cell assignment. Instead, a table inserter thread inserts a record into a randomly chosen queue. Each index inserter thread iterates over the queues and processes the next available record.

```
Assignment:
 1: Assign grid cells to index inserter threads in a round-robin manner to produce the hashtable Cell-
    ToThreadMappingHT
Table inserter thread:
 2: ...
 3: RecGridCellId ← computeCurrGridCellId(LocnRecord)
 4: threadId ← CellToThreadMappingHT.find(RecGridCellId)
 5: RecQueueArray[threadId].push(LocnRecord)
Index inserter thread:
 6: Initialize LocalRecQueue ← RecQueueArray[localThreadId]
 7: while true do
 8:     LocnRecord ← LocalRecQueue.pop()
 9:     Process LocnRecord and insert into index
10: end while
```

Figure 6.8: Algorithm Round-robin assignment

```
Table inserter thread:
 1: ...
 2: threadId ← RandomGenerator.nextRand()%NUM_THREADS
 3: RecQueueArray[threadId].push(LocnRecord)
Index inserter thread:
 4: for qIdx ← 0 to RecQueueArray.size() - 1 do
 5:     LocnRecord ← RecQueueArray[qIdx].pop()
 6:     Process LocnRecord and insert into index
 7: end for
```

Figure 6.9: Algorithm Adaptive

6.5.4 Query Processing

In this Section we describe the past, present and predictive (future) query processing algorithms. The algorithm for historical range query is shown in Figure 6.10. The grid cells that are fully or partially covered by the query window *QRect* are computed in lines 2 and 3. We proceed to describe the steps for the partially covered cells. For each partially covered grid cell the corresponding temporal index is used to obtain the objects that were inside the cell during the interval (*EndDS* - *StartDS*) using bitwise OR (lines 8 - 10 of Figure 6.10, and Figure 6.11). This list of such objects is encoded in a partial result bitmap *tmpResltPartlyCovrd*. For each such moving object (i.e. for each bit that is set in *tmpResltPartlyCovrd*) inside every partially covered grid cell, the corresponding temporal index is utilized to inspect if that object was inside the query window *QRect* (line 18 of Figure 6.10). The details of this inspection are shown in Figures 6.12 and 6.13. This involves retrieving the *RIDList* for that object (line 1 of Figure 6.13) and for each RID entry, obtaining the corresponding location record fields of latitude, longitude and datestamp from the location table (lines 3 - 6). If the object location was inside the query rectangle and the datestamp with the query interval (line 8), the object is part of the past range query resultset.

The present range query algorithm utilizes the past range query algorithm (Figure 6.10), by specifying the end

```
Require: Reslt is result-set bitmap, LocTable is database table, STIdx is spatio-temporal index, QRect is query
    window, StartDS and EndDS are query interval datestamps
 1: {//Local variables: tmpResltFullyCovrd, tmpResltPartlyCovrd }
 2: FullyCovrdCells ← getFullyCoveredCells(QRect)
 3: PartlyCovrdCells ← getPartiallyCoveredCells(QRect)
 4: for gridCell in FullyCovrdCells do
 5:    TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
 6:    TemporalIdx.getIntervalMatchingObjs(tmpResltFullyCovrd, StartDS,EndDS)
 7:    Reslt ← BitwiseOr(Reslt, tmpResltFullyCovrd)
 8: end for
 9: for gridCell in PartlyCovrdCells do
10:    TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
11:    TemporalIdx.getIntervalMatchingObjs(tmpResltPartlyCovrd, StartDS,EndDS)
12: end for
13: tmpResltPartlyCovrd.BitwiseMinus(tmpResltFullyCovrd)
14: NumBitsSet ← tmpResltPartlyCovrd.NumBitsSet()
15: BitmapSetIterator ← tmpResltPartlyCovrd.getIterator()
16: for gridCell in PartlyCovrdCells do
17:    TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
18:    BitmapSetIterator.ReSet()
19:    for (nxtBit ← BitmapSetIterator.GetnxtBitSet()) != NULL do
20:       if Reslt.IsSet(nxtBit) != TRUE then
21:          if (TemporalIdx.isObjInGrdCellAndIntrvl(nxtBit, QRect,StartDS,EndDS,LocTable)) then
22:             Reslt.SetBit(nxtBit)
23:          end if
24:       end if
25:    end for
26: end for
```

Figure 6.10: Algorithm PastRangeQuery

```
Require: tempReslt is temporary result-set bitmap, StartDS and EndDS are query interval datestamps, LocTable
    is database table
1: IntervalsCovrd ← getCoveredIntervals(StartDS,EndDS)
2: for Interval in IntervalsCovrd do
3:    CBmapHmRIDList ← this.Itab.find(Interval)
4:    PartialRsltBitmap ← CBmapHmRIDList.getObjsBitmap()
5:    this.ReadWriteLock.ReadLock()
6:    tempReslt ← BitwiseOr(tempReslt, PartialRsltBitmap)
7:    this.ReadWriteLock.UnLock()
8: end for
```

Figure 6.11: Procedure TemporalIdx.getIntervalMatchingObjs

```
Require: QRect, StartDS, EndDS, LocTable same as before
1: IntervalsCovrd ← getCoveredIntervals(StartDS,EndDS)
2: for Interval in IntervalsCovrd do
3:    CBmapHmRIDList ← this.Itab.find(Interval)
4:    IsInside ← CBmapHmRIDList.isObjInGrdCellAndIntrvl( ObjId, QRect,StartDS,EndDS,LocTable)
5: end for
```

Figure 6.12: Procedure TemporalIdx.isObjInGrdCellAndIntrvl

```
Require: ObjId, QRect, StartDS, EndDS, LocTable same as before
 1: RIDList ← this.HmRIDList.find(OId)
 2: for r=0 to ObjId.size()-1 do
 3:    Rid ← RIDList[r]
 4:    Lat ← LocTable.getLatitude(Rid)
 5:    Lon ← LocTable.getLongitude(Rid)
 6:    Datestamp ← LocTable.getDatestamp(Rid)
 7:    ReturnVal ← FALSE
 8:    if Datestamp ≥ StartDS AND Datestamp ≤ EndDS AND QRect.Contains(Lat, Lon) then
 9:       ReturnVal ← TRUE
10:       return
11:    end if
12: end for
```

Figure 6.13: Procedure CBmapHmRIDList.isObjInGrdCellAndIntrvl

datestamp parameter to be "now" and start datestamp parameter to a few seconds earlier (a configurable value *offset*).

The predictive range query algorithm uses two main data structures: the prediction hashmap *PHm* and the array of hashmaps *ArHm* (associating the each grid cell to a prediction bitmap). The algorithm performs the following steps:

1. Compute the grid cells that are fully or partially covered by the query window *QRect*
2. For each fully or partially covered cell the intermediate result bitmap is obtained by performing a bitwise OR with the prediction bitmap
3. Initialize final result bitmap with fully covered result bitmap
4. For each bit that is set in partially covered result bitmap do
5. Retrieve actual predicted latitude and longitude from *PHm*
6. If the predicted location is inside *QRect*
7. Set the bit representing the object in the final result bitmap

6.6 Discussion

PASTIS is essentially a versioned grid organization that maintains for each time interval the partial trajectory of each moving object. PASTIS utilizes insert-efficient compressed bitmap and dynamic integer array (RID list) to maintain the partial trajectory. It could be argued that this can also be implemented with an in-memory R-tree based approach. To evaluate the performance of such an approach we replaced the *CBmap* and *Hm-RIDList* structures with an efficient in-memory R*-tree. An in-memory R*-tree is used to track the partial trajectory of all the moving objects during a time interval for each grid cell. A location update is processed by the R*-tree as a combination of deletion, updating the trajectory MBR and re-insertion operations.

A location update involves inserting a new record in the table and updating the index. Figure 6.14 shows the breakdown of the completion time to perform 10 million location updates and contrasts PASTIS implemented with compressed bitmap and RID list vs. in-memory R*-tree. The data was obtained by the profiler tool Callgrind.

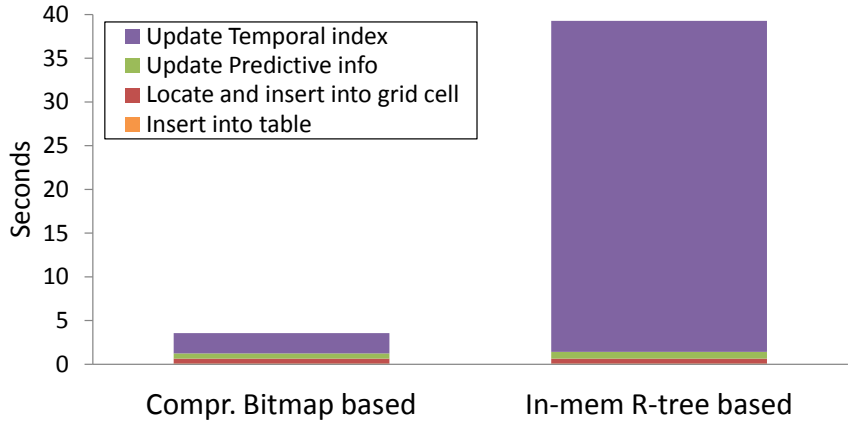


Figure 6.14: Breakdown of time for location update (dataset 10 million)

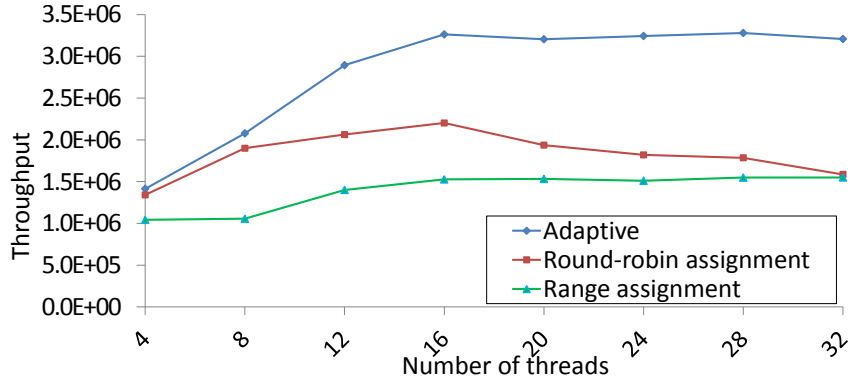


Figure 6.15: Update throughput with different load-balancing approaches

The R*-tree approach takes 10X more time. Also, whereas the *Update Temporal index* step takes 60.4% of the overall time with the bitmap approach, it takes 96% of the overall time with the R*-tree approach. Note that *Insert into table* step takes only a small fraction of overall time due to column fragment storage.

Since concurrency is a key performance bottleneck, our system exploits several high performance concurrency control features. The shared counter variables were implemented with atomic Fetch-and-Add. Efforts were made to avoid the usage of locks. However, in a few cases where it was unavoidable we used high performance spin locks. It was shown that spin locks provided the best performance compared to other options [126], especially when the locked objects are held for relatively short period of time. We used two concurrent collection classes: concurrent hashmap and concurrent blocking queue from the Intel TBB library.

6.7 Performance Studies

6.7.1 Experimental Setup

The performance studies were conducted on a HP Z820 machine with 256 GB of RAM and 2 Intel Xeon E5-2670 processors having a total of 16 cores (32 h/w threads), each running at 1200 MHz. The OS was Suse Linux SLES 11.1.

The polyline shapefiles of Texas from the TIGER dataset [120] were fed into the mobility trace generator

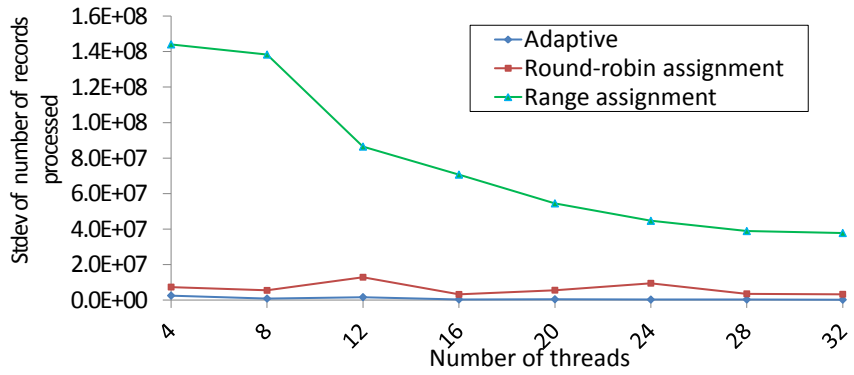


Figure 6.16: Standard dev. of number of records processed with different load-balancing approaches

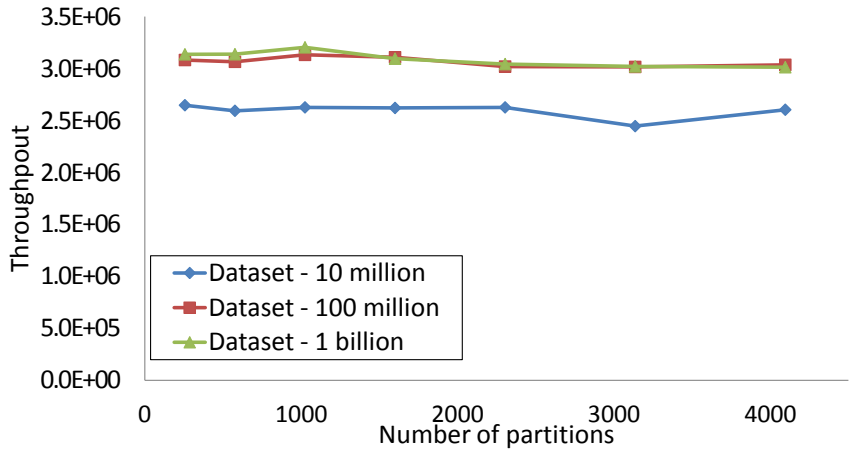


Figure 6.17: Update throughput with different number of partitions

MOTO [76] to generate the traces. We modified MOTO to generate multiple trace files, each file for a different table fragment. Mobility traces were generated for each object for 1000 timestamps (equivalent to 10,000 seconds). We generated the trace files for 3 different data sizes, as shown in Table 6.1. Table 6.2 shows various configuration parameter settings.

6.7.2 Update Performance

We first evaluate the location update performance (insertion into the Location table and updating the index). In our experimental setup we utilized 10 dedicated threads to handle inserts into the Location table for 10 different column fragments.

6.7.2.1 Load-Balancing Algorithms

To determine the impact of the load-balancing approaches on location update performance we implemented the algorithms mentioned in Section 6.5.3, namely, Range assignment, Round-robin assignment and Adaptive. The throughput achieved with 1 billion location updates is shown in Figure 6.15. The Adaptive algorithm consistently achieved better throughput than the other two approaches as the number of *index inserter threads* are varied. To determine the reason for this, we plot the standard deviation of the number of records processed by the threads in Figure 6.16. The standard deviation was the least for the Adaptive algorithm. This suggests that the Adaptive

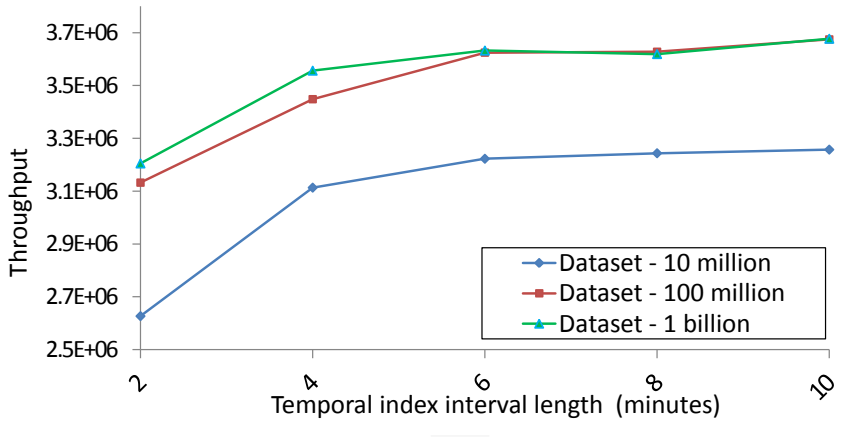


Figure 6.18: Update throughput with different temporal index interval length

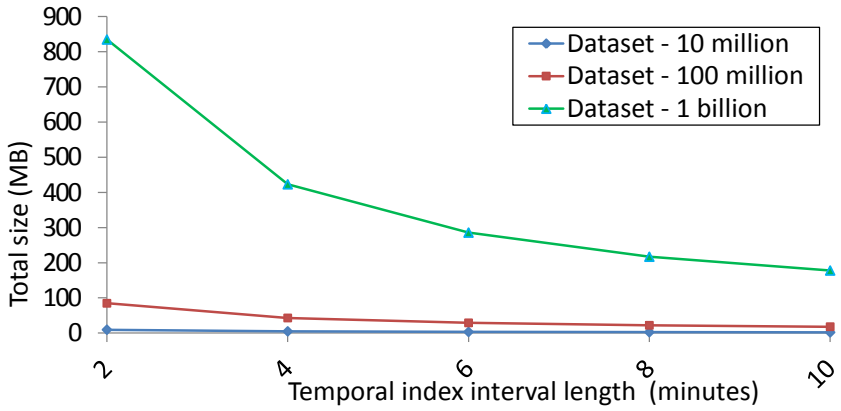


Figure 6.19: Total size of all bitmaps with different temporal index interval length

algorithm is the least sensitive to skew in the distribution of moving objects. It is expected that as the number of threads are increased the variation among them is reduced. This can be clearly observed for Range assignment. With Round-robin, this variation was lower than that of Range assignment, but still higher than Adaptive. Note that Adaptive achieved over 3.2 million updates per second and this throughput became steady after 16 to 20 threads. Part of the reason is that, although there are 32 hardware threads in the machine not all of them are available for index update processing. Because, 10 threads are dedicated as *table inserters* (regardless of the load-balancing approach). We use Adaptive as the load-balancing algorithm in all subsequent experiments.

6.7.2.2 Number Of Spatial Partitions (Grid Resolution)

To observe the effect of the number of partitions on location update performance, we vary the number of partitions (grid cells). Figure 6.17 shows the throughput achieved with the 3 datasets as the number of partitions are increased from 256 to 4096. The number of index inserter threads were fixed at 20. As can be seen, the throughput remains steady regardless of the number of partitions for all three datasets. This suggests that the Adaptive algorithm does a good job of load-balancing and grid resolution has no significant impact on update throughput. For the remaining experiments we use 1024 partitions.

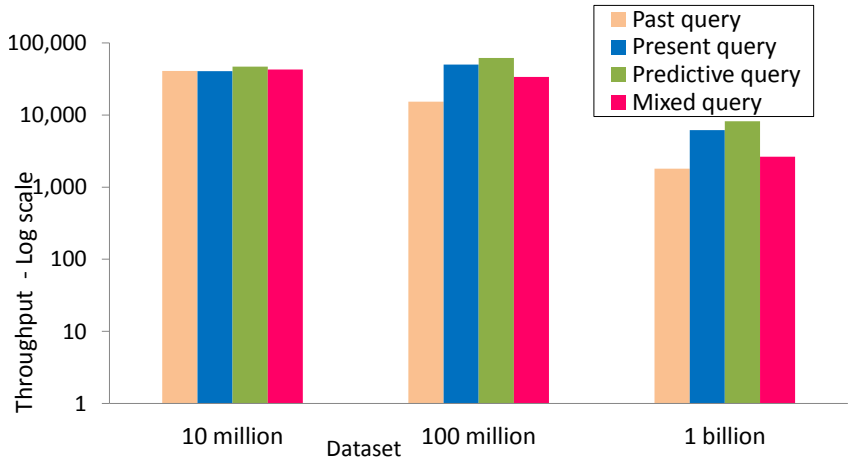


Figure 6.20: Query throughput

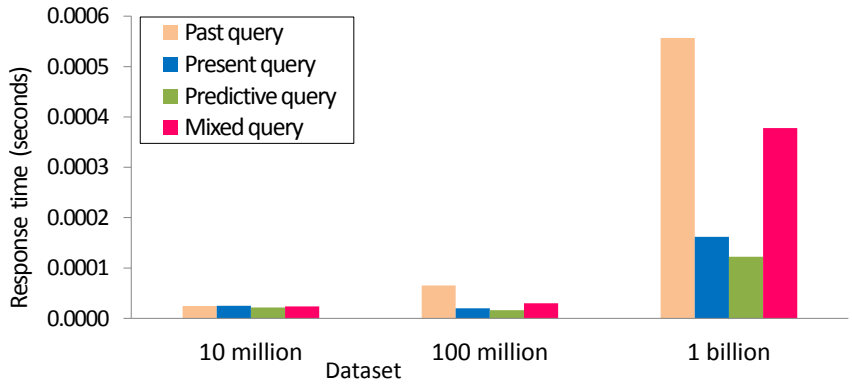


Figure 6.21: Average query response time

6.7.2.3 Temporal Index Interval Length

An index interval length is the duration of time interval for which all location records in a particular grid cell are recorded in the corresponding temporal index. The smaller the index interval length, the more temporal index instances are created. Intuitively, it would require more computation to process more temporal index instances. To evaluate the throughput at different index interval lengths, we vary the length to 2, 4, 6, 8 and 10 minutes. Figure 6.18 shows the corresponding throughputs. The throughput is increased as the interval length is increased, for instance, it is 3.5 million updates per second for Dataset - 1 billion with interval length 4. This is a good trend, because in a real-world LBS application this interval length S for the records received within the last H (where, H = 24 hours) would be higher than 2 minutes, such as 10 to 30 minutes. For records older than 24 hours, this length T could be much higher, such as a few hours. Unless otherwise specified, we use 2 minute as the default, because this is the most pessimistic scenario.

6.7.2.4 Storage Requirements

The storage requirement is a key consideration when its comes to adopting an index solution. To this end, we report the total size of all bitmaps created by our index by varying the temporal index interval lengths, to 2, 4, 6, 8 and 10 minutes. As shown in Figure 6.19, the total size of all the bitmaps is quite small, and is dependent

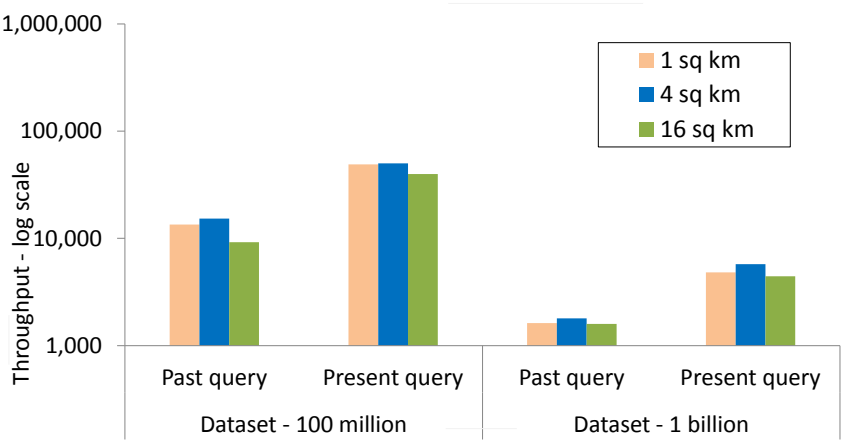


Figure 6.22: Query throughput - spatial extent

Table 6.1: Trace file details

Dataset name	**Num. of mobile objects**	**Num. of location records**	**Size on disk**
10 million	10,000	10,000,000	445 MB
100 million	100,000	100,000,000	4.5 GB
1 billion	1,000,000	1,000,000,000	46 GB

on the temporal index interval length. As noted previously, in a real-world LBS applications the temporal index interval length would be much larger, and hence the storage requirements would be lower. Note that the memory requirement for storing the RID Lists for a particular dataset is not reported in the Figure, because it a known and fixed quantity. That is calculated by multiplying the the total number of records and the number of bytes needed to represent one RID.

6.7.3 Query Performance

In this Section we evaluate concurrent range query performance. All queries were executed simultaneously with record updates. The query execution is started after at least 5% of the dataset is populated. Four types of range queries were evaluated: past (historical), present, predictive (future), and mixed. The mixed queries were generated by issuing past, present and predictive queries at equal ratio 1:1:1. Since LBS applications are dominated by updates, rather than queries, we assign the available threads in 2:1 ratio to update processing as opposed to query processing. We used update to query ratio 1000:1, so that one query was issued for every 1000 updates. No significant decrease in update throughput was observed.

Figure 6.20 shows the throughputs of past, present, predictive and mixed queries. For all three datasets, the system achieved throughputs of thousands of queries per second. The average query response times are shown in Figure 6.21. As expected, past query response times are the longest. However, even with Dataset - 1 billion, the past query response time was less than a millisecond. For present, predictive and mixed queries the response times were lower than those of past queries. This shows that in-memory bitmaps can help accelerate query performance.

Table 6.2: Parameter settings

Parameter	**Settings**
Space domain	Texas, 1251 km x 1183 km
Num. of road segments	56832846
Time duration, *timestamps*	1000
Update frequency, *seconds*	10
Updates (num. of records)	10 million, 100 million, **1 billion**
Range query area, km^2	1 , **4**, 16
Range query interval, %	1.5, 3, **6**, 12

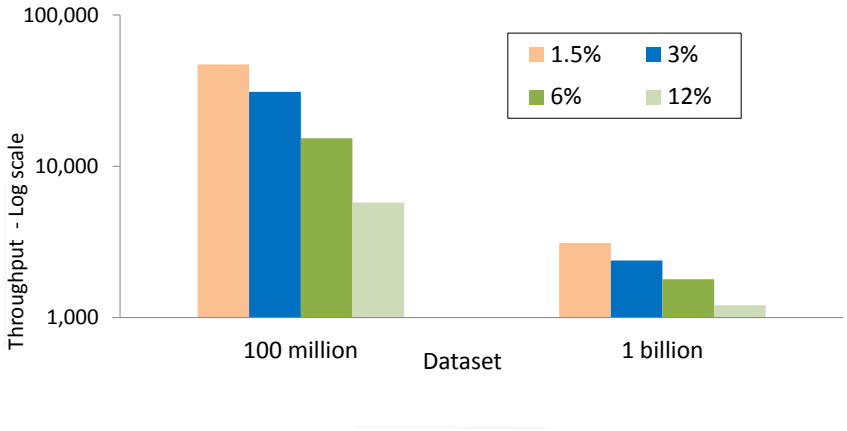


Figure 6.23: Query throughput - effect of interval length

6.7.3.1 Spatial Extent

A key parameter, when generating queries, is the spatial extent or the spatial dimension of the query window. We used three spatial extent values: 1 sq. km, 4 sq. km and 16 sq. km [126]. Figure 6.22 compares the throughput of past and present queries for the 3 different spatial dimension sizes. For both the queries the query throughput is the lowest when the spatial extent is 16 sq. km. This is expected because when the dimension is larger, there are more moving objects and correspondingly, more records to be processed. Interestingly, the throughput is slightly lower at 1 sq. km than at 4 sq. km. To explain this, note that if a grid cell is partially overlapped by the query window it is more expensive to process than if it is fully covered. A cell is more likely to be partially covered at 1 sq. km than at 4 sq. km.

6.7.3.2 Interval Length

The temporal length of the queries, expressed as a fraction of the recorded history [118], is an important parameter for the past queries. We used 4 query interval length values to evaluate the performance of the past queries: 1.5%, 3%, 6% and 12%. Since the dataset was generated for 1000 timestamps, an interval length of 1.5% represents a query interval with a duration of 15 timestamps. Figure 6.23 compares the throughput of past range queries for the 4 different interval lengths and datasets 100 million and 1 billion. In all cases, the throughput is the best at interval length 1.5%. The throughput is decreased as the interval length is increased.

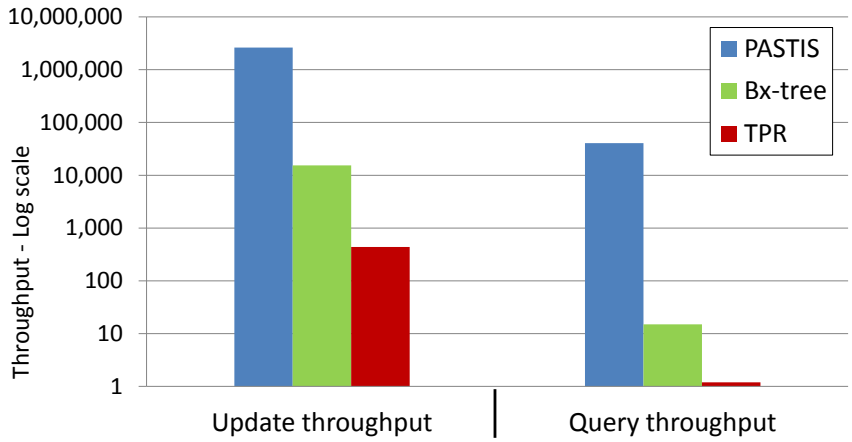


Figure 6.24: PASTIS vs B^x-tree vs TPR

6.7.4 Comparison With Other Indexes And Other LBS Systems

Spade [25] is a popular benchmark that compares a number of spatio-temporal indexes. We evaluate two representative spatio-temporal indexes, B^x-tree and TPR-tree [124], from Spade using a setup described in Section 6.7.1. Like other in-memory databases, such as SAP HANA, in our system the Location table is persisted, and the index is built when the table is loaded into memory. To do a fair comparison, we modified the Spade benchmark code to make the B^x-tree and TPR-tree indexes memory resident and not persist into disk. Figure 6.24 compares their update and query throughputs of B^x-tree and TPR-tree against PASTIS for the 10 million record dataset. For both update and query, our approach shows orders of magnitude better throughput than B^x-tree and TPR-tree.

A paper on LBS system MOIST reported [64] that it attained 8000+ updates per second and 60K updates per second with 1 server and 10 servers respectively for one million moving objects. Although a direct comparison is not possible, we demonstrated in Figure 6.17 that with a single server we achieve over 3.2 million updates per second for one million objects. This is a speedup of 400X over MOIST for a single server. MD-HBase [81] reported a peak throughput of 220K location updates per second (in a 16-node cluster), which is one order of magnitude lower than ours. They noted their best present range query average response time to be about 500 milliseconds. Our past and present range query response times are less than a millisecond for all 3 datasets. Most commercial LBS providers use traditional relational databases for data management. As reported in [101], a popular database achieves only about 45K updates per second even when it completely fits in memory.

6.8 Chapter Summary

Location-Based Services have come to play important roles in various facets of our lives. With the exponential growth of spatio-temporal data, many commercial LBS systems are unable to meet the growing customer demands. It is necessary that the databases, on which LBS providers depend, offer sufficient performance to handle very high update rates, while supporting many concurrent past, present and future queries.

To address these requirements, we propose a combination of in-memory based techniques. We present an insert-efficient main-memory storage for LBS. We introduce PASTIS, a parallel in-memory spatio-temporal index that supports historical (past), present and predictive (future) queries. PASTIS is a versioned grid organization that utilizes compressed bitmaps to maintain partial temporal indexes for objects that visited each grid cell in a 2D spatial domain. By completely maintaining the data and index for the past active N days in memory,

our system avoids any disk access latency for updates and queries. Thread-level parallelism and fine-grained concurrency control, supported by fast bitmap operations help our system to achieve high update throughput and query performance. With extensive evaluations we demonstrate the superior performance of our system over existing approaches.

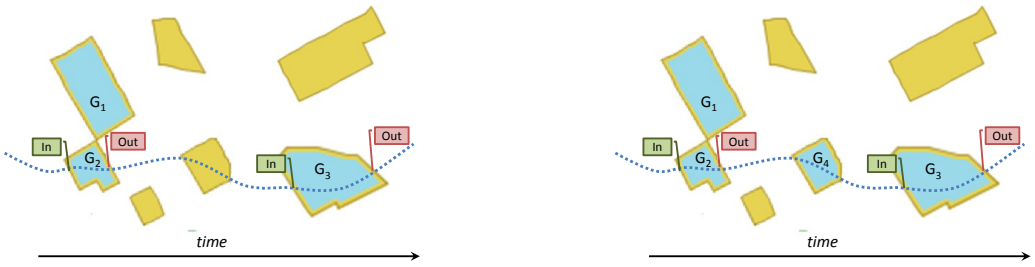
Chapter 7

Trajectory-Based Spatio-Temporal Topological Join

7.1 Introduction

The trajectory-based queries are important classes of spatio-temporal queries [95]. Trajectory-based topological queries deal with the whole or part of the trajectory of a moving object. Given a set of trajectories R and another set of spatial objects (polylines or polygons) S, a trajectory-based topological join query combines these two data sets on spatio-temporal predicates P, such as *enters*, *crosses* or *leaves*. An example of such a query is to select all trajectories that cross a set of polygons during a given time interval in the past. These queries are widely used in many Location-Based Services (LBS) applications. A food and beverage products company, such as Pepsi or Coca-Cola that delivers drinks from manufacturing plants to stores, may like to create polygons around the plants and stores. In LBS these polygons are often known as *geofences*. The company may want to know the details about the trajectories of their delivery trucks whose trajectories enter and leave the geofences during rush hours. Such information could then be used to optimize truck routes and in turn save on fuel costs and reduce stop times. Similar use-cases can be found with various other industries including utility services and transportation. Some of these fleets can have tens of thousands of vehicles [45]. New uses-cases are also emerging. For instance, RFID tags with GPS support can be used to track perishable produce, such as sea-food, and the determination and optimization of routes can reduce wastage. Similarly, GPS-equipped mobile phones can be used track the trajectories of people in urban areas and offer personalized services.

We have shown 4 that spatial join queries are compute intensive, leading to long query latencies. Although the study of spatial joins have received a lot of attention [90, 91, 6, 105], not much work has has yet been done with the spatio-temporal topological join queries. Also, unlike spatio-temporal range queries, that we have dealt with in Chapter 6, spatio-temporal topological join queries are more complex and we address them in this Chapter. To evaluate the performance of spatio-temporal topological join queries, given a dataset of moving object trajectories and polygon objects, we systematically describe different **applicable** in-memory join algorithms. In this process we gradually add in-memory versions of trajectory index, TB-tree (Trajectory-Bundle tree) [95] and spatial index, STR [69] on the datasets. We evaluate these algorithms with a moderately large sized trajectory dataset (having 100 million location records) and a polygon dataset (with 5651 polygons). Even the best among these algorithms, called *Indexed Nested Loop Join with 2 Index* or *INLJ2I*, takes **over 8 hours** when the query interval is 1000. Our study of these algorithms suggests that not unlike the spatial counterparts, spatio-temporal



(a) Event tagging

(b) Ad hoc inclusion of a new geofence G_4

Figure 7.1: Geofence crossing scenario

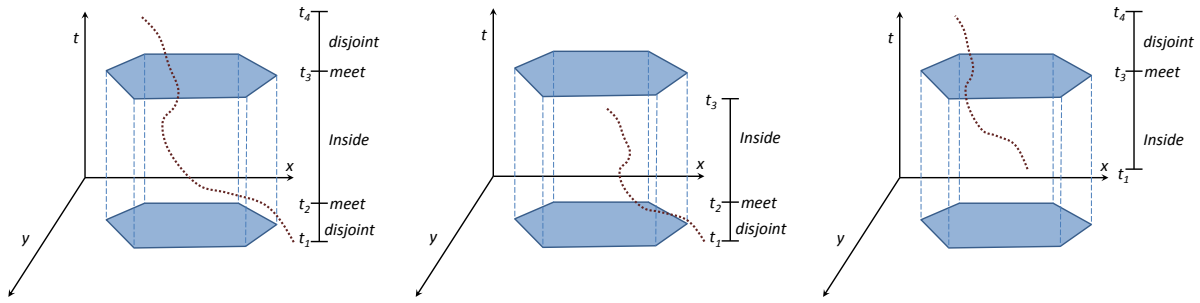
topological join queries are also compute intensive and long running. Obviously, the query latencies may not be acceptable for real-time use-cases. even with a moderately large data set. Our goal is to support ad hoc historical spatio-temporal topological join queries with real-time response times. We also support joining trajectories with evolving polygons, i.e., polygons that change their size and positions over time.

Due to the time-consuming nature of the spatio-temporal join queries, commercial LBS providers utilize an elaborate "a priori procedure" to support these queries. In particular, the polygon (geofence) definitions must be uploaded into each mobile device. Moreover, sophisticated embedded software within a mobile device must tag each crossing event when it happens, such as, moving inside a polygon (In event) or moving outside a polygon (Out event). Obviously, this approach can not answer ad hoc historical spatio-temporal join queries for newly created geofences or with polygons whose definitions were not available during the query interval or whose definitions changed since. We support ad hoc historical spatio-temporal topological join queries without the need for uploading polygon definitions into the mobile devices or tagging the polygon crossing events. We also support joining trajectories with evolving polygons, i.e., polygons that change their size and positions over time.

7.2 Case Study And Motivation

Trajectory-based spatio-temporal topological join queries are used to combine a trajectory dataset with a spatial objects dataset based on a given criteria. The criteria is expressed as a spatio-temporal predicate. Such queries are becoming increasingly important in many emerging spatio-temporal applications, such as Location-Based Services (LBS), location-aware advertising, national security and battle planning and analysis. A typical LBS scenario is the product dispatch case, where a delivery truck is loaded with products at a loading depot and then it delivers the products to a number of customer sites. A vehicle from a garbage disposal company or a utility services company visits different customer sites in a municipality. Such customer sites can be specified by spatial objects such as polygons. In LBS terminology they are called *geofences*. A dispatch manager may be interested in asking historical spatio-temporal topological join queries, such as "find the trucks that crossed a set of geofences during the rush hours last month" or "find the trucks that crossed a set of geofences last week where the duration inside the geofences was more than 15 minutes" or "find the trucks that always enter geofence A and leave C prior to entering B during the afternoon shift".

Due to the compute intensive nature of spatio-temporal topological join (also demonstrated in Section 7.5.2), commercial service providers follow a "formal procedure" to support these queries. As part of this process the geofence definitions must be uploaded into the on-board location tracking device of each truck. This definition



(a) Crosses (b) Enters (c) Leaves

Figure 7.2: Evaluation of the spatio-temporal predicates

includes the name and coordinates of each geofence polygon that a truck is expected to traverse. As a truck enters into or leaves from a particular geofence, the embedded software of its on-board device must *tag* such a event. This is illustrated in Figure 7.1(a). Here the geofence definition includes the set G = {G_1,G_2,G_3}. As a truck V enters and leaves a geofence, each location record must be tagged with the event label as either "In" or "Out". Without loss of generality, the sequence of location records could be $\langle t_1, G_2, In\rangle$, $\langle t_2, G_2, Out\rangle$, $\langle t_3, G_3, In\rangle$ and $\langle t_4, G_3, Out\rangle$, where each tuple is of the form $\langle timestamp, polygon, event\rangle$. It is feasible to answer spatio-temporal topological join queries for the trajectory of V and G, but based on the tagged events only. However, it is not practical to answer any ad hoc queries or queries based on "what if" scenarios. It is also impossible to answer queries for newly added geofences retroactively. If a dispatch manager thinks that V may have traversed a polygon in between G_2 and G_3, and label it as G_4 (Figure 7.1(b)), it is not possible to answer any historical spatio-temporal topological join query based on G_4.

Due to the limitation of the current approaches in answering spatio-temporal topological join queries, we believe that it is an important problem to address. A suitable solution must do away with the need to upload polygon definitions to the on-board devices upfront and having to tag the crossing events as they happen. Such an approach should also answer the queries reasonably fast.

7.3 Spatio-Temporal Join Queries

Spatial join queries are used to coalesce two different datasets based on a spatial predicate. In the case of trajectory-based spatio-temporal topological join, one of the two datasets is a trajectory dataset and the other is a dataset of spatial objects such as polylines or polygons. The objects in the spatial dataset could be stationary or evolve over time. Moreover, the predicate must be a spatio-temporal predicate. We formally define the spatio-temporal join operations.

7.3.1 Trajectory-Based Spatio-Temporal Topological Join

Definition: Given a trajectory dataset *R*, and spatial objects dataset *S*, a time interval *T* and spatiotemporal predicate *P*, the query operation finds for each $r \in R$, all $s \in S$ such that each *r* and *s* satisfies the predicate *P* within interval *T*. Formally,

$$R \bowtie_{P,T} S \equiv \{(r, s, \{t\}) \mid r \epsilon R \ \wedge \ s \epsilon S \ \wedge \ \exists_{t \subseteq T} P(r(t), s(t))\}$$

Table 7.1: Queries with different predicates and abbreviations

Description (tables involved)	Abbreviations
Trajectory crosses Polygon	Crosses
Trajectory enters Polygon	Enters
Trajectory leaves Polygon	Leaves

Here, *t* is a sub-interval of *T*. Each trajectory *r* is a sequence of triples:

$$r = \{(x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n)\},$$

where (x_i, y_i) location coordinates, t_i timestamps and $t_1 < t_2 < ... < t_n$. *S* can be polygon or polyline spatial objects. However, for the rest of the paper we mean polygons by spatial objects.

7.3.2 Spatio-Temporal Topological Predicates

Although the Open Geospatial Consortium (OGC) adopted the Dimensionally Extended Nine-Intersection Model (DE-9IM) [28] for spatial predicates, when it comes to spatio-temporal predicates, OGC has still not adopted a standard model. In fact the definition of data models and predicates in this context is a fertile ground of research. Erwig and Schneider [41] proposed a number of spatio-temporal predicates. A number of data models have also been proposed [94]. For our discussion we use three spatio-temporal predicates, which seems to be commonly present in all proposed models. Table 7.1 shows the predicates and the queries based on these predicates.

7.3.3 Predicates Evaluation

The efficient evaluation of the predicates is necessary to be able to use them in spatio-temporal join queries in a real database system. There have been a few research projects that dealt with the representation of moving objects, such as the work by Forlizzi et al. [46]. The main idea behind these works is to describe the moving objects using *sliced representation* or *unit representation*. In this approach, the temporal evolution of an object is decomposed into a sequence of *slices* or *time units* such that within each unit the evolution is specified by a "simple" function. Then the evaluation of a spatio-temporal topological predicate can be described by a sequence of topological relationships that may hold at different time units. Schneider [107] proposed a generic mechanism to evaluate spatio-temporal predicates based on this idea. According to this, for each pair of matching time unit a basic topological relation, called *unit development* is evaluated. For instance, for a trajectory of moving points and a polygon shape object the *unit development* is the outcome of the predicate *point inside polygon*. The concatenated sequence of all the unit developments collected over the entire query interval can then be inspected for certain patterns that evaluates the spatio-temporal topological predicate to be true. This process is illustrated in Figure 7.2(a), which checks if the spatio-temporal predicate Crosses is valid between a trajectory and a polygon object. The basic topological relations are evaluated at each time unit. Prior to t_2 the trajectory and the polygon are disjoint. The trajectory is inside the polygon between t_2 and t_3, after which they are disjoint again. The predicate evaluation process is shown in Figure 7.2(b) and Figure 7.2(c) for Enters and Leaves respectively. For the purpose of evaluating the spatio-temporal topological predicates Crosses, Enters and Leaves between a trajectory and a polygon, it is sufficient to evaluate basic topological relation *point inside polygon* (point-in-polygon test) at each time unit.

7.4 Applicable Spatio-Temporal Join Algorithms

We discuss several applicable spatio-temporal topological join algorithms in this section. They are all based on the same principle of spatio-temporal predicate evaluation approach used by previous works (as outlined in Section 7.3.3). In presenting different join algorithms, we take a methodical approach, by starting with an algorithm that uses no index and incrementally adding an index to the two tables corresponding to the two datasets.

7.4.1 Nested Loop Join (NLJ)

For this discussion we assume that in-memory table *R* is used to store the location updates corresponding to each moving object's trajectory and table *S* is used to maintain the spatial objects. The spatio-temporal predicate is *P*, where the predicates can be Crosses, Enters and Leaves. The query time interval is *T*. We assume that the records in table *R* are similar to "location" records, each with attributes such as timestamp, latitude, longitude, velocity, direction and the moving object id. The records in table *S* have the attributes: the spatial object geometry, name, start-timestamp, end-timestamp and object id. The start and end timestamps of the spatial objects are used to specify their active lifespan.

The first algorithm, *Nested Loop Join* or *NLJ*, uses no index on either table and thus this is a naive approach. The algorithmic sketch is presented in Figure 7.3. The algorithm performs a table scan on both tables *R* and *S* and materializes the records that are inside query time interval *T*. It iterates over each location record in table *R*, and checks if its coordinate (denoted by latitude, longitude) is inside any of the spatial objects. This means there is an inner loop that iterates over the spatial objects in *S*. Then for each coordinate and spatial object, a point-in-polygon test is performed and the outcome is recorded in the *unit development* sequence data structure, as described in Section 7.3.3.

Once all the location records are processed, the actual spatio-temporal predicate evaluation happens. This is done by iterating over each moving object's *unit development* sequence and evaluating the predicate *R* against each.

7.4.2 Indexed Nested Loop Join With One Index (INLJ1I)

The next algorithm uses a trajectory index, I_R, to index the location records in table *R*. A search on this index for all trajectories within the query time interval *T* returns the matching trajectories for all moving objects, $RO_{i,t}$. This data structure allows iterating over each moving object that is in the search result and retrieve its trajectory.

There is no index on the records in table *S* and table scan is done on *S* to materialize the records that are inside the query time interval. Since this algorithm is an indexed nested loop using one index, we call it *Indexed Nested Loop Join (1 Index)* or *INLJ1I*. The algorithm is presented in Figure 7.4. Each trajectory that is returned by index I_R is essentially a sequence of coordinates. The algorithm loops through each such coordinate to test against each spatial object to evaluate the point-in-polygon predicate (lines 7-11). As before, the evaluation result is recorded in the *unit development* sequence corresponding to the moving object (line 13). The spatio-temporal predicate evaluation steps (lines 15-18) are the same as in *NLJ*.

7.4.3 Indexed Nested Loop Join With Two Indexes (INLJ2I)

The next algorithm introduces an in-memory spatial index, I_S, on the geometry attribute of table *S*. It also uses an in-memory trajectory index, I_R, as before. Since this algorithm uses two indexes we call it *Indexed Nested Loop Join (2 Index)* or *INLJ2I*. The sketch of this algorithm is in Figure 7.5. In this case, the two outer loops are the

Require: *R* and *S* are the 2 tables. The spatio-temporal join predicate is *P* and query time interval is *T*. And *mobjDevelMap* is a map containing the unit development sequence of each moving object.

```
1:  R_t ← TableScan(R,T)
2:  S_t ← TableScan(S,T)
3:  Materialize R_t and S_t
4:  for r_t in R_t do
5:      l_t ← getLocation(r_t)
6:      movObjId ← getObjId(r_t)
7:      mObjDevel ← mobjDevelMap.get(movObjId)
8:      for s_t in S_t do
9:          {//Evaluate basic relation}
10:         e_{l,s,t} ← evalPointInPolygon(l_t,s_t)
11:         {//Add to unit development sequence}
12:         mObjDevel.addToUnitDevelSequence(e_{l,s,t},l_t,s_t,t)
13:     end for
14: end for
15: mObjList ← mobjDevelMap.getKeys()
16: for mObj_i in mObjList do
17:     mObjDevel ← mobjDevelMap.get(mObj_i)
18:     {//Evaluate spatio-temporal predicate: crosses, enters, leaves etc.}
19:     result_i ← evalSTPredicate(P,mObjDevel)
20: end for
```

Figure 7.3: Algorithm1 - NLJ

Require: *R* and *S* are the 2 tables. I_R is an in-memory trajectory index on table *R*. And parameters *P*, *T* and *mobjDevelMap* are as before.

```
1:  RO_{i,t} ← I_R.IndexLookup(T)
2:  for O_i in RO_{i,t} do
3:      mObjDevel ← mobjDevelMap.get(O_i)
4:      R_t ← RO_{i,t}.getTraj(O_i)
5:      S_t ← TableScan(S,T)
6:      Materialize S_t
7:      for r_t in R_t do
8:          l_t ← getLocation(r_t)
9:          for s_t in S_t do
10:             {//Evaluate basic relation}
11:             e_{l,s,t} ← evalPointInPolygon(l_t,s_t)
12:             {//Add to development sequence}
13:             mObjDevel.addToUnitDevelSequence(e_{l,s,t},l_t,s_t,t)
14:         end for
15:     end for
16: end for
17: mObjList ← mobjDevelMap.getKeys()
18: for mObj_i in mObjList do
19:     mObjDevel ← mobjDevelMap.get(mObj_i)
20:     {//Evaluate spatio-temporal predicate: crosses, enters, leaves etc.}
21:     result_i ← evalSTPredicate(P,mObjDevel)
22: end for
```

Figure 7.4: Algorithm2 - INLJ1I

same as in INLJ1I. The algorithm iterates over each record from each trajectory that is in returned by index I_R

(lines 1-5). Next the index I_S is used to return those spatial objects whose MBRs intersect with a given record's coordinate (line 7). The coordinate then must be checked against the actual geometries of all spatial objects that matched the search criteria (line 8-10). The rest of the algorithm is the same as in INLJ1I.

Require: *R* and *S* are the 2 tables. I_S is an in-memory R-tree index on table *S*. I_R is an in-memory trajectory index on table *R*. And parameters *P*, *T* and *mobjDevelMap* are as before.

```
1:  RO_{i,t} ← I_R.IndexLookup(T)
2:  for O_i in RO_{i,t} do
3:     mObjDevel ← mobjDevelMap.get(O_i)
4:     R_t ← getTraj(RO_{i,t},O_i)
5:     for r_t in R_t do
6:        l_t ← getLocation(r_t)
7:        SI_t ← I_S.IndexLookup(l_t)
8:        for s_t in SI_t do
9:           {//Evaluate basic relation}
10:          e_{l,s,t} ← evalPointInPolygon(l_t,s_t)
11:          {//Add to unit development sequence}
12:          mObjDevel.addToUnitDevelSequence(e_{l,s,t},l_t,s_t,t)
13:       end for
14:    end for
15: end for
16: mObjList ← mobjDevelMap.getKeys()
17: for mObj_i in mObjList do
18:    mObjDevel ← mobjDevelMap.get(mObj_i)
19:    {//Evaluate spatio-temporal predicate: crosses, enters, leaves etc.}
20:    result_i ← evalSTPredicate(P,mObjDevel)
21: end for
```

Figure 7.5: Algorithm3 - INLJ2I

7.5 Our Approach

First we introduce an optimization into the algorithm INLJ2I. We also examine the performance of applicable spatial join approaches against our optimization.

Then we describe our system that we call PISTON (Parallel In-memory trajectory-based Spatio-temporal TOpological joiN).

7.5.1 Trajectory Filtering (INLJ2I-TF)

In any real-world dataset many moving object trajectories would not intersect any spatial object during the specified query interval *T*. If those trajectories could be eliminated prior to the point-in-polygon test step, there is a potential for significant improvement in the runtime of the algorithm, because the point-in-polygon test is performed on each coordinate of a trajectory with each spatial object returned by I_S. Inspired by the Filter step of the two-step processing of spatial predicates, we introduce a "trajectory filter" step in the INLJ2I algorithm. This involves checking the minimum bounding rectangle (MBR) of a trajectory for intersection against the spatial index I_S. If this trajectory MBR does not intersect the MBR of any spatial object, then no further action is needed for this particular trajectory. Otherwise the point-in-polygon test needs to performed. We call this algorithm *Indexed Nested Loop Join (2 Index) with Trajectory Filtering* or *INLJ2I-TF*. The algorithm is presented in Figure 7.6. As

can be seen, for each trajectory returned by I_R the trajectory MBR is built (lines 5-7). Then MBR intersection test is performed to filter out trajectories from further processing (line 8). Otherwise, the same processing steps are performed as in algorithm INLJ2I.

Require: *R* and *S* are the 2 tables. I_S is an in-memory R-tree index on table *S*. I_R is an in-memory trajectory index on table *R*. And parameters *P*, *T* and *mobjDevelMap* are as before.

```
1:  RO_{i,t} ← I_R.IndexLookup(T)
2:  for O_i in RO_{i,t} do
3:     mObjDevel ← mobjDevelMap.get(O_i)
4:     R_t ← getTraj(RO_{i,t},O_i)
5:     for r_t in R_t do
6:        l_t ← getLocation(r_t)
7:        trajMBR_t.expandToInclude(l_t)
8:     end for
9:     if I_S.MBRIntersects(trajMBR_t) then
10:       for r_t in R_t do
11:          l_t ← getLocation(r_t)
12:          SI_t ← I_S.IndexLookup(l_t)
13:          for s_t in SI_t do
14:             {//Evaluate basic relation}
15:             e_{l,s,t} ← evalPointInPolygon(l_t,s_t)
16:             {//Add to unit development sequence}
17:             mObjDevel.addToUnitDevelSequence(e_{l,s,t},l_t,s_t,t)
18:          end for
19:       end for
20:    end if
21: end for
22: mObjList ← mobjDevelMap.getKeys()
23: for mObj_i in mObjList do
24:    mObjDevel ← mobjDevelMap.get(mObj_i)
25:    {//Evaluate spatio-temporal predicate: crosses, enters, leaves etc.}
26:    result_i ← evalSTPredicate(P,mObjDevel)
27: end for
```

Figure 7.6: Algorithm4 - INLJ2I-TF

7.5.2 Evaluation And Analysis

To evaluate the spatio-temporal join algorithms that we have described so far, we select two trajectory datasets: Dataset-10mi consists of the trajectories of 10 thousand moving objects (having 10 million location records) and Dataset-100mi has 100 million records for 100 thousand moving objects. Furthers details about them can be found in Table 7.2. We select a spatial dataset: the Arealm polygons from TIGER Texas dataset. The details of this dataset are shown in Table 7.4. To index the trajectories we chose TB-tree. The polygons were indexed using an efficient spatial index, STR. In both cases we use **in-memory** implementations of the indexes. We ran the applicable algorithms presented in Section 7.4 with the Crosses query by varying the query intervals between 200, 400, 600, 800 and 1000. Algorithms NLJ and INLJ1I took too long to be practically used in any application. So we only present the execution times (in seconds) for INLJ2I. We compare its performance against INLJ2I-TF in Figure 7.7 for both trajectory datasets (label prefix "Dataset" is shortened as "D"). As can be seen, INLJ2I-TF took significantly less time than INLJ2I in all cases. However, even with the filtering algorithm INLJ2I-TF took hundreds of seconds. Naturally, the question arises if it is possible to improve the performance of spatio-temporal

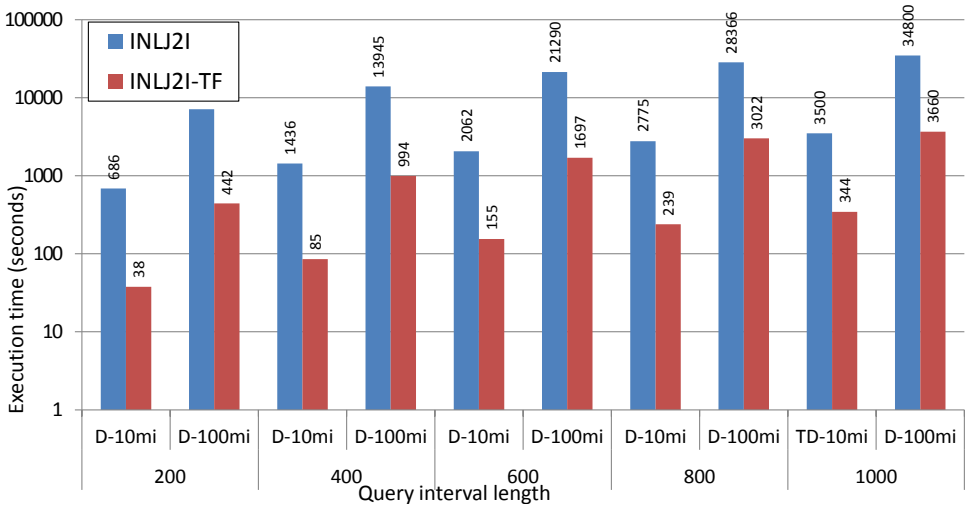


Figure 7.7: Execution time of Crosses query with different spatio-temporal join algorithms

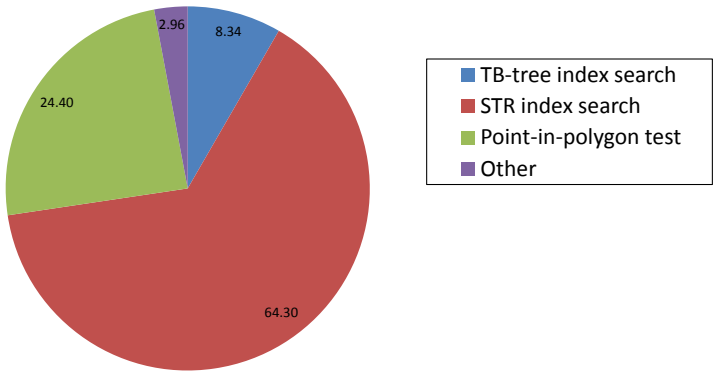


Figure 7.8: Breakdown of time with INLJ2I-TF (TB-tree & STR)

join further. We investigate this question in the next section.

To analyze in which part of the code the most time is spent, we profiled the execution of INLJ2I-TF with the profiling tool valgrind ("–tool callgrind" option). Since it takes a long time to run the profiling, we used the smaller trajectory dataset Dataset-10mi. The polygon dataset used is the same as before. We ran the Crosses query with time interval 100. In Figure 7.8 we show the breakdown of time reported by valgrind. As shown, the most time (over 64%) is spent in the spatial index (STR) search. About 24% of time is spent in point-in-polygon test and about 8% in trajectory index search. This suggests that reducing the costs of spatial index and point-in-polygon test are important to improve the overall performance.

7.5.3 Overview Of PISTON

As we have demonstrated, existing approaches to indexing trajectories and spatial objects do not scale. These approaches, such as TB-tree and STR, are usually tree-based and were designed for external memory (disk) join algorithms. Moreover, the spatial indexes do not optimize the basic topological relation such as point-in-polygon test. To improve the performance of trajectory-based spatiotemporal join algorithms, we propose a new approach that we call PISTON. This is a parallel in-memory spatio-temporal join approach. The architecture of PISTON

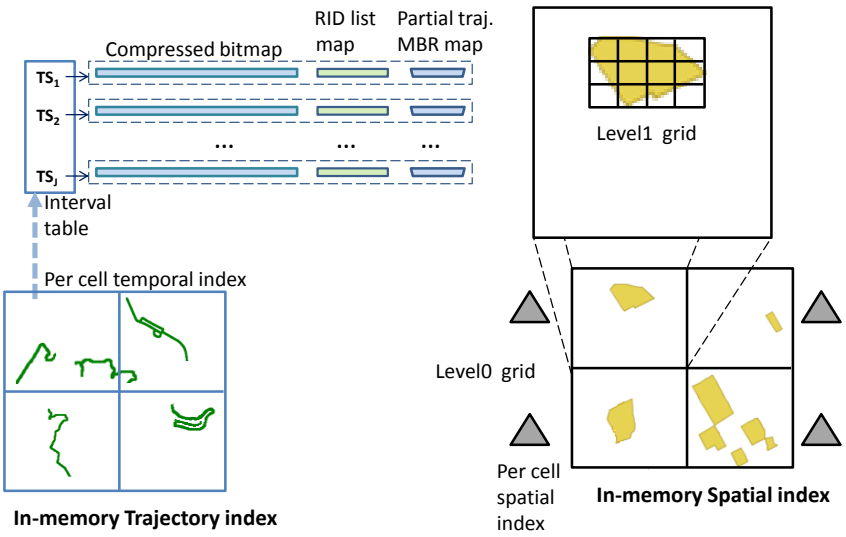


Figure 7.9: PISTON overview

is illustrated in Figure 7.9. PISTON is a grid-based approach that organizes the spatial domain into regular cells. Other schemes such as quad-tree or K-d-tree based splitting of the spatial domain is also possible. There are two in-memory indexes: the first to index the moving object trajectories and the second to index the spatial objects, such as, polygons. PISTON uses these indexes to implement a parallel in-memory spatio-temporal topological join algorithm; it is described in Section 7.5.4.

The trajectory index is a high performance index designed to handle high rate of location data updates. The index can handle both a coordinate-based range query and a trajectory-based query subject to a retrieval criteria. For each grid cell in the spatial domain, partial trajectory information is maintained for the objects that visited that cell. More details about this index is in Section 7.5.6.

The spatial index has a two level grid organization: *Level0* and *Level1*. The *Level0* grid of the spatial index uses the same grid organization and is aligned exactly with that of the trajectory index. The purpose of *Level1* grid is to impose grids inside the bounding box of each spatial object. Further details of the spatial index is provided in Section 7.5.5.

7.5.4 A Parallel In-Memory Join Algorithm

PISTON introduces a parallel in-memory spatio-temporal topological join algorithm. Like any other parallel task execution, load-balancing is needed to evenly distribute the query workload to the worker threads. This could be based on static or dynamic work assignment. In a static approach, the workload is partitioned and assigned to the workers before the execution starts. A dynamic approach, on the other hand, assigns the workload to the threads during runtime from a pool of subtasks until exhausted. PISTON follows the dynamic task assignment approach with a master/slave model, where the master is called *Manager* and the slaves *Worker*. For each query job the Manager creates many tasks, one for each grid cell, and enqueues them into a synchronized queue. Each Worker thread then picks the next task from the queue and performs the spatio-temporal join on the objects from both tables belonging to the corresponding grid cell. This is essentially a per cell indexed nested loop join and we simply call this approach *PISTON*.

The sketch of the algorithm is shown in Figure 7.10. The Manager creates and enqueues a task *STJTask* for each cell (lines 1-3). Then it waits for the Workers to complete. A Worker retrieves the next task from the

queue and with the cell id it retrieves the partial trajectory information for that grid cell from the trajectory index PiI_R (line 19-21). The trajectory index maintains information for all valid active time intervals in memory. For every time interval it is necessary to inspect the "partial trajectory" of each moving object. For each such moving object indexed by the trajectory index, the corresponding "partial trajectory MBR" is retrieved. Then this MBR is checked against the spatial index PiI_S to see if the *MBRIntersects* spatial predicate is satisfied (lines 23-27). This process is similar to the Filter step of the two step Filter and Refinement in regular spatial predicate evaluation. If the object's trajectory MBR does not satisfy the MBRIntersects predicate, no further action is needed for that partial trajectory. Otherwise, every coordinate of that partial trajectory is tested against the spatial index PiI_S to do a point-in-polygon test (line 32). The details of Procedure *evalPointInPoly* is in Figure 7.14. The index returns **all** spatial objects (polygons), $\{s_t\}$, that contain the coordinate. Each such spatial object becomes part of the moving object's *unit development* sequence (lines 32-34). This process is illustrated in Figure 7.11(b). As can be seen, the "partial trajectory MBRs" PM_1, PM_2 and PM_3 do not intersect the MBR of object *Obj1* and hence can be ignored. Since PM_4, PM_5 and PM_6 satisfies *MBRIntersects* with *Obj1*, only those points in their corresponding "partial trajectories" need to be further processed.

Once all the Workers have processed all the grid cells, the Manager iterates over each task that were scheduled and retrieves partial *unit development* sequences. For each moving object the partial sequences are merged to create the complete *unit development* sequences (line 7-12). Finally, for each such sequence the spatio-temporal predicate *P*, such as Crosses, Enters or Leaves, is evaluated (lines 14-17 in Figure 7.10).

7.5.5 In-Memory Spatial Index

We introduce a novel in-memory index for spatial objects that is particularly suitable for point-in-polygon queries. With a tree-based spatial index such as STR, evaluating point-in-polygon query for a given point involves first using the index to retrieve all candidate spatial objects whose MBRs are intersected by the point. Then each candidate spatial object's actual geometry is tested for point containment. However, with our index no separate filtering step is necessary. There is a single processing step that returns all the spatial objects that contain a given point.

Our index is a hierarchical grid based approach. The base level (*Level0*) grid is aligned exactly with the grid of the trajectory index. A higher level (*Level1*) grid is imposed on each of the spatial objects indexed. The resolution of the grid cells at *Level1* is different at different *Level0* cells. This depends on the dimension of the smallest object MBR in a cell. This is illustrated in Figure 7.12. In *Cell2* there are two objects: *Obj2.1* and *Obj2.2*. The smaller of two is *Obj2.2* and its MBR dimension determines grid cell resolution of *Cell2*. All *Level1* grids imposed on a *Level0* cell have the same resolution. So *Obj2.1* and *Obj2.2* have the same resolution.

A polygon that overlaps multiple *Level0* grid cells requires special treatment. In many of the existing approaches to parallel spatial join [90, 91], such an object is replicated to each cell that it overlaps with. We use a technique in which an object is split along the cell boundaries. This is shown in Figure 7.11(a). In this example the object *Obj* is split into 3 objects: Obj_1, Obj_2 and Obj_3. When a spatial object is split along the cell boundaries of multiple *Level0* cells, different fragments of that object would have *Level1* grid imposed on them with different resolutions. As explained previously, this depends on the dimension of the smallest MBR of that cell. During the processing of the spatio-temporal join query, an aggregation step is used to combine the partial results for the different fragments of an object.

There is a setup process when the spatial objects are indexed first time. There are 3 steps in this process. First, each spatial object in table *S* is iterated over and the *Level0* grid cell to which it lies is determined. Objects overlapping multiple cells are also split in this step. In Step 2, each *Level0* grid cell is checked to determine the

Require: *R* and *S* are the 2 tables. PiI_R is PISTON's trajectory index on table *R*; PiI_S is PISTON's spatial index on table *S*. The join predicate is *P*, query time interval is *T* and *mobjDevelopmentMap* is a map containing the unit development sequence of each object.

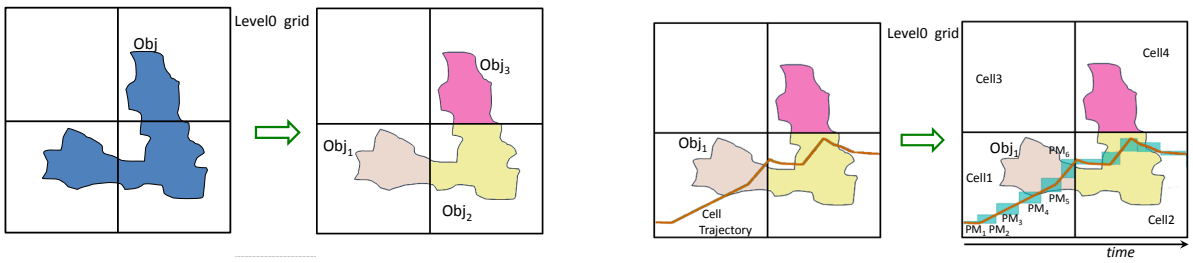
TaskManager:

for *cellId* in *cellList* **do**
  Create a new *STJTask* with *cellId*
  TaskQueue.push(STJTask)
  TaskList.add(STJTask)
end for
{//Wait for all tasks to complete}
...
for *STJTask* in *TaskList* **do**
  partialUnitDevelopmentSequence ← *STJTask.getPartialUnitDevelopmentSequence()*
  for $\{O_i$,*partialDevelopmentMap*$_i\}$ in *partialUnitDevelopmentSequence* **do**
    mObjDevelopment ← *mObjDevelopmentMap.get*(O_i)
    {//Merge to the unit development sequence}
    mObjDevelopment.merge(*partialDevelopmentMap*$_i$)
  end for
end for
mObjList ← *mObjDevelopmentMap.getKeys()*
for *mObj*$_i$ in *mObjList* **do**
  mObjDevelopment ← *mObjDevelopmentMap.get*(*mObj*$_i$)
  {//Evaluate spatiotemporal predicate: crosses, enters, leaves etc.}
  result$_i$ ← *evalSTPredicate(P,mObjDevelopment)*
end for

Worker:

while true **do**
  STJTask ← *TaskQueue.pop()*
  cellId ← *STJTask.cellId()*
  partialSTIndex ← PiI_R*.getPartialSTIndex*(*cellId*)
  {//Do for each interval and each moving object that was in the cell during the interval}
  for $\{interval_j$,*partialTrajectoryInfo*$_j\}$ in *partialSTIndex* **do**
    for $\{O_i$,*objTrajectoryInfo*$_i\}$ in *partialTrajectoryInfo*$_j$ **do**
      partialTrajectoryMBR$_i$ ← *objTrajectoryInfo*$_i$*.getPartialTrajectoryMBR()*
      partialTrajectory$_i$ ← *objTrajectoryInfo*$_i$*.getPartialTrajectory()*
      if PiI_S***.MBRIntersects***(*partialTrajectoryMBR*$_i$) **then**
        for r_t in *partialTrajectory*$_i$ **do**
          l_t ← *getLocation*(r_t)
          ts_t ← *getTimestamp*(r_t)
          if ts_t is in *T* **then**
            $\{e_{l,s,t}$,l_t,$\{\boldsymbol{s}_t\}$,*t*$)\}$ ← PiI_S***.evalPointInPoly***(l_t,T)
            {//Add to partial unit development sequence}
            mObjDevelopment.addToPartialUnitDevelpmntSequence($e_{l,s,t}$,l_t,$\{\boldsymbol{s}_t\}$,*t*)
          end if
        end for
      end if
    end for
  end for
end while

Figure 7.10: Algorithm5 - PISTON



(a) Shape object split

(b) Partial trajectory MBR filtering

Figure 7.11: Splitting spatial objects along tile boundaries (a) and partial trajectory MBR filtering (b)

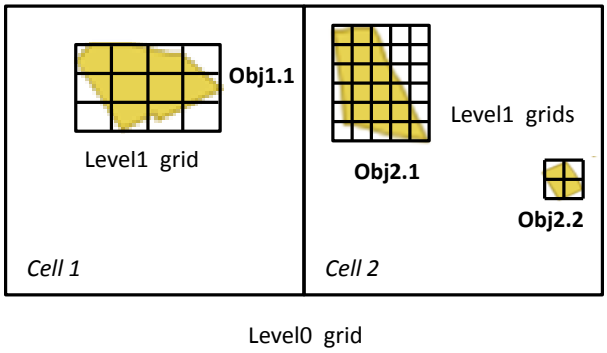


Figure 7.12: Different Level1 grid resolutions

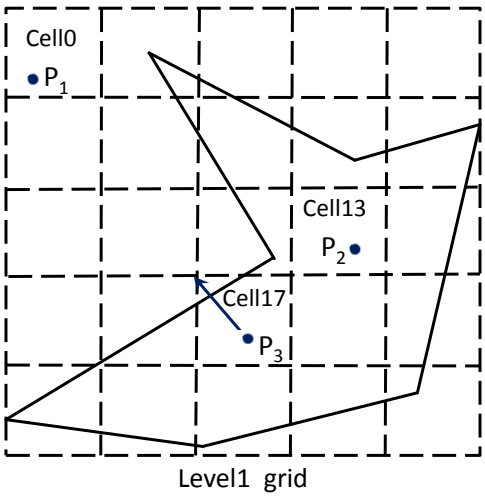


Figure 7.13: Grid setup and testing

spatial object with the smallest MBR dimension. Then based on that *Level1* grid is imposed on the spatial objects. The Step 3 involves recording the status of the *Level1* cells relative to the spatial objects. With respect to a given polygon, a *Level1* grid cell status could be fully inside, fully outside, or indeterminate. As shown in Figure 7.13, *Cell0* is fully outside of the polygon and *Cell13* is fully inside. However, *Cell17* is indeterminate with respect to the polygon because one of the edges of the polygon overlaps *Cell17*. The indeterminate cells could also have several polygon edges which overlap the cell. The setup process collects information for each *Level1* grid cell regarding which corner(s) of the cell is inside or outside of the polygon and whether any polygon edge crosses the

cell. To deal with overlapping polygons, status information is collected for all polygons and maintained separately within each *Level1* cell. To reduce memory consumption due to the setup process, the cell status is recorded for only those *Level1* cells that are overlapped by the MBR of any spatial object. If no record is found for a *Level1* cell, it means that it is outside of any polygon.

Once the setup process for all the polygons is complete, the spatial index can be used to answer point-in-polygon queries. The griding method [56] suggested a similar point-in-polygon test procedure, that works for a single polygon only. In our approach, this test is performed for the points of the trajectories against all polygons. To check if a point is inside any of the polygons indexed by the spatial index, first the *Level0* cell to which the point lies is determined and then its corresponding *Level1* cell is determined. For a given polygon many of the *Level1* grid cells are either completely inside or outside. Hence a simple look-up is all that is needed to determine if a point is inside the polygon. As a result, this operation could be extremely fast in most cases. For instance, in Figure 7.13, point P_1 is outside of the polygon because the status of *Cell0* is fully outside. Similarly, point P_2 is inside of the polygon because cell *Cell13* is fully inside.

If the *Level1* cell, inside which the point lies, contains any edges, then a line segment is formed from the test point to one of the corners of the cell. This is illustrated by an arrow pointing from point P_3 to the top-left corner of cell *Cell17* in Figure 7.13. This line segment is tested for intersection against all recorded polygon edges for the cell. Since the state of the cell corner (whether inside or outside) is known, the number of times this line crosses the edges of the polygon determines if the point lies inside or outside of the polygon. The steps of this procedure to perform a point-in-polygon test is shown in Figure 7.14. To support evolving polygons, such that they can change their locations and sizes over a period of time, the spatial index supports multiple versions of the same spatial object. Each version is associated with an active lifespan, with start-timestamp and end-timestamp. The lifespan must be active during query interval *T* for it to be included in the return result.

7.5.6 In-Memory Trajectory Index

In our previous work we introduced a parallel in-memory index for spatio-temporal range queries 6. Our in-memory trajectory index is designed to support both coordinate based (including range queries) and trajectory based queries. At the same time it supports fast index creation and update operations. The in-memory trajectory index discretizes both the spatial and temporal dimensions. It discretizes the temporal dimension by maintaining location update information for a fixed length interval *i*. Our trajectory index discretizes the spatial dimension by imposing regular grid $SGrid_c$ on the spatial domain. Here c=1 to C and C the total number of cells. A "cell trajectory", T_c, is part of a moving object's trajectory that is completely within a given grid cell *c*. Essentially, all the location coordinates corresponding to a moving object's trajectory that are inside a cell boundary belong to T_c. A "partial trajectory", T_i, is comprised of those location coordinates from T_c, whose timestamps are within an interval *i*. The minimum bounding rectangle (MBR), PM_i, corresponding to the a partial trajectory T_i is its "partial trajectory MBR". As described in Section 7.5.4 and illustrated in Figure 7.11(b), these MBRs are used to filter out those segments of the trajectory that do not intersect any polygons.

We describe the different components of the in-memory trajectory index, as shown in Figure 7.9. The main idea behind this is to maintain per cell temporal index structure for all the moving objects that visited a grid cell $SGrid_c$. For each temporal interval $i \epsilon I$ during the past *N* (configurable) days, an entry is maintained in an interval lookup table $Itab_c$ within $SGrid_c$. Each entry in $Itab_c$ corresponds to three data structures: a compressed bitmap $CBmap_{c,i}$, a hashmap $RIDList_{c,i}$ and a second hashmap $PtMBR_{c,i}$. $CBmap_{c,i}$ identifies the moving objects that were in the grid cell at the given time interval. Therefore, if object *m* was present during *i* inside $SGrid_c$, bit position *m* of $CBmap_{c,i}$ is set. Here m=1 to M, M being total number of objects. We use an insert-optimized

```
Require: pt is a given point and T is the query time interval. The procedure returns a list resultListOfPoly of all
    polygons inside which pt lies and the polygons are active during T.
 1: level0Cell ← getLevel0CellForGivenPoint(pt)
 2: level1Cell ← getLevel1CellForGivenPoint(pt,level0Cell)
 3: listPolyRecorded ← getPolygonsRecordedAtCell(level1Cell)
 4: if listPolyRecorded.count = 0 then
 5:    resturn NULL
 6: else
 7:    for poly in listPolyRecorded do
 8:       if level1Cell is fully inside of poly then
 9:          if isActiveDuring(poly,T) then
10:             resultListOfPoly.add(poly.id)
11:          end if
12:       else if level1Cell is fully outside of poly then
13:          continue
14:       else
15:          corner ← determine the best cell corner to test
16:          insideFlag ← initFlag(corner)
17:          send a line fragment ray from pt towards corner
18:          edges ← getPolyEdgesRecordedAtCell(poly)
19:          for edge in edges do
20:             if ray intersects edge then
21:                insideFlag ← !insideFlag
22:             end if
23:          end for
24:          if insideFlag and isActiveDuring(poly,T) then
25:             resultListOfPoly.add(poly.id)
26:          end if
27:       end if
28:    end for
29:    return resultListOfPoly
30: end if
```

Figure 7.14: Procedure evalPointInPoly

compressed bitmap, the details of which is beyond the scope of the paper. It is assumed that when a new location update is received, (prior to updating the trajectory index) a record is inserted into table, *R*, to obtain a unique record id RID. The table schema: {*ObjectId,Latitude,Longitude,Direction,Speed,Datestamp*}. Therefore, given an RID the actual record storing datestamp, latitude and longitude can be retrieved from table *R*. The hashmap $RIDList_{c,i}$ is used to maintain for each moving object a list of RIDs corresponding to the location updates sent while at the grid cell *c* during the time interval *i*. Essentially the RID list constitutes the partial trajectory for a moving object. To keep the minimum bounding rectangle (MBR) of each partial trajectory, the hashmap *PtMBR* is used, which is keyed by the object id.

Next we describe the trajectory index update workflow. As a new location update is received from a moving object the trajectory index is updated to modify its trajectory information. To support high update rates with thousands of mobile objects, the system is organized to be highly parallel. As shown in Figure 7.15, an incoming location update is first placed in one of a set of queues *RQ-t* from which it is picked up by a thread in the thread-pool *TP-t*. The location table *R* is partitioned by dividing the RID domain into a number of sub-ranges and a separate *TP-t* thread handles insert into a different partition. Upon inserting a record into the corresponding table partition, it is then enqueued randomly in one among a set of queues *RQ-x*. Since the distribution of the objects

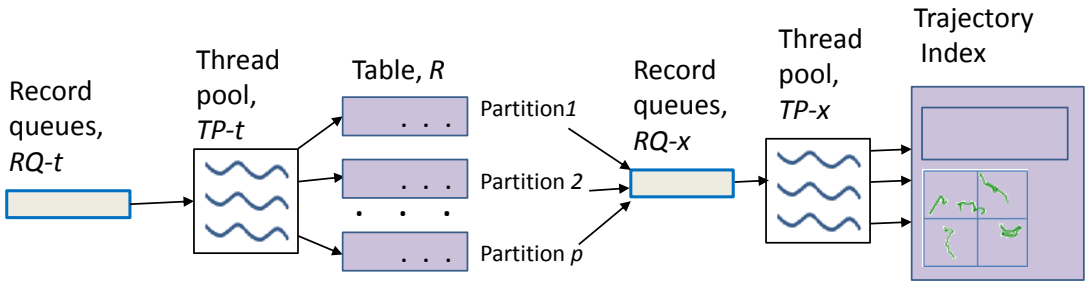


Figure 7.15: Trajectory index update process

Table 7.2: Details of trajectory dataset

Dataset name	Num. of mobile objects	Num. of location records	Size on disk
Dataset-10mi	10,000	10 million	445 MB
Dataset-100mi	100,000	100 million	4.5 GB
Dataset-1bi	1,000,000	1 billion	46 GB

and their trajectories could be highly skewed, it is necessary to load-balance the trajectory index update process. The load-balancing scheme involves the threads in the thread-pool *TP-x* iterate over the queues *RQ-x*. Each *TP-x* thread inspects the currently chosen queue and processes the next available record. Let the record fields be coordinate *(x,y)*, datestamp ds_t, object id *m*, and record id *RID*, among others. The target grid cell $SGrid_c$ is determined from its coordinate *(x,y)*. The datestamp ds_t determines the corresponding interval *i* in the interval table $Itab_c$, where ds_t is the datestamp at time *t*; i=1 to I, I being total intervals. If ds_t does not map to an existing interval *i* in $Itab_c$, a new interval *i* and related data structures are instantiated. The data structures $CBmap_{c,i}$ and $RIDList_{c,i}$ are updated from the record fields as described earlier. To update $PtMBR_{c,i}$, the old trajectory MBR of object *m*, $MBR_{c,i,m}$ is retrieved and is expanded to include new location *(x,y)*.

7.6 Experimental Evaluation

In this section we evaluate PISTON in various settings. We first describe the experiments involving polygon dataset with shapes that are static and next we conduct experiments with evolving polygons dataset. The trajectory dataset remains the same in both cases.

7.6.1 Dataset With Static Polygons

7.6.1.1 Experimental Setup

We use a real-world spatial objects dataset that contains diverse geographical features, drawn from the TIGER® data [120], produced by the United States (US) Census Bureau. This is a public domain data source available for each US state. The dataset that we use consists of the area landmass polygon shapefiles for all the counties of Texas. We merged the shapefiles to create single shapefiles. Table 7.4 shows the details of the static polygons dataset, including the geometry and cardinality.

To generate the trajectory dataset, the polyline shapefiles of Texas from the TIGER® dataset were fed into the mobility trace generator MOTO [76] to generate the traces. We modified MOTO to generate multiple trace files, each file for a different partition of table *R*. Mobility traces were generated for each object for 1000 timestamps,

Table 7.3: Trajectory dataset generation settings

Parameter	**Settings**
Spatial domain	Texas, 1251 km x 1183 km
Num. of polyline segments	56832846
Time duration, *timestamps*	1000
Update frequency, *seconds*	10
Updates (num. of records)	10 million, 100 million, **1 billion**

Table 7.4: Details of polygon dataset

Dataset	**Database table**	**Geometry**	**Cardinality**
Texas	Arealm	polygon	5651

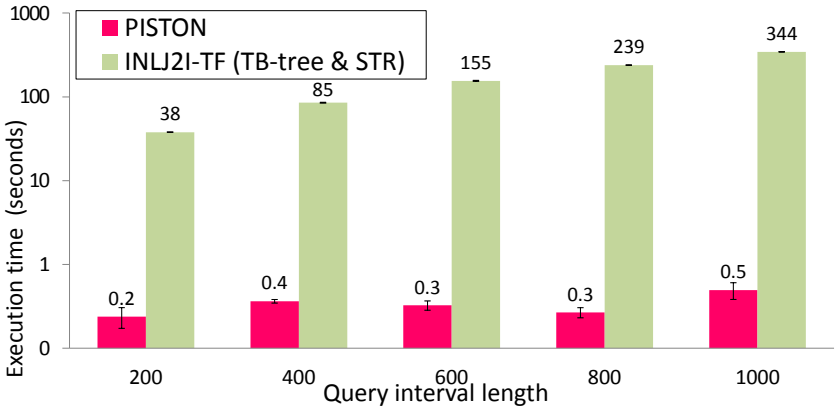


Figure 7.16: Execution times of query Crosses with PISTON vs NLJ2I-TF (TB-tree & STR) with Dataset-10mi

which is equivalent to 10,000 seconds. As shown in Table 7.2, we generated the trace files for 3 different sizes. The largest trajectory dataset, Dataset-1bi, contains one billion location records for 1 million moving objects. The different configuration parameter settings used to generate the trajectory dataset is shown Table 7.3.

The experiments were conducted on a machine having 256 GB memory, 8 Intel Xeon processors with 64 cores, each running at 1064 MHz. We run Ubuntu 10.04 Lucid 64-bit with kernel version 3.13.0-43-generic as the OS.

7.6.1.2 Comparison With Existing Trajectory And Spatial Index

In Section 7.4 we introduced a few applicable algorithms to perform trajectory-based spatio-temporal topological join. Except for *NLJ*, the other join algorithms uses in-memory index. The best performing algorithm among them was INLJ2I, that uses in-memory implementations of the trajectory index TB-tree and the spatial index STR. We presented a trajectory MBR filter optimization to INLJ2I and called it INLJ2I-TF. In Section 7.5.2, we compared the performance of INLJ2I-TF against INLJ2I and demonstrated its superior performance. Note that these algorithms use GEOS [48], a C++ geometry library, to perform the point-in-polygon test. INLJ2I-TF also uses both indexes (in-memory TB-tree and STR) and hence it can be considered as the best performing algorithm that uses existing trajectory and spatial indexing approaches.

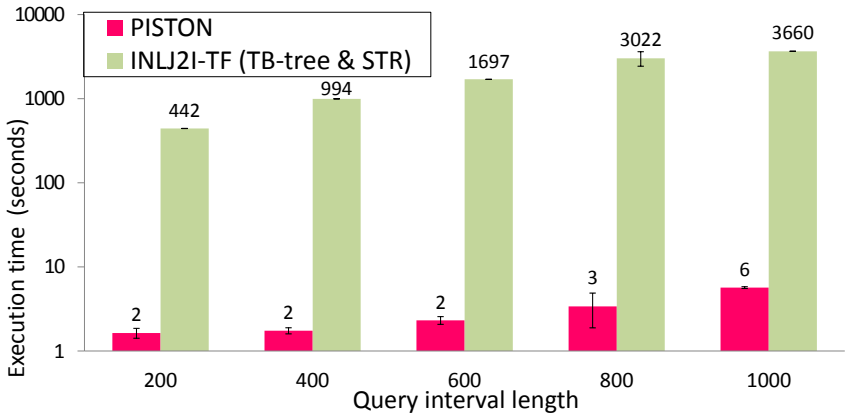


Figure 7.17: Execution times of query Crosses with PISTON vs NLJ2I-TF (TB-tree & STR) with Dataset-100mi

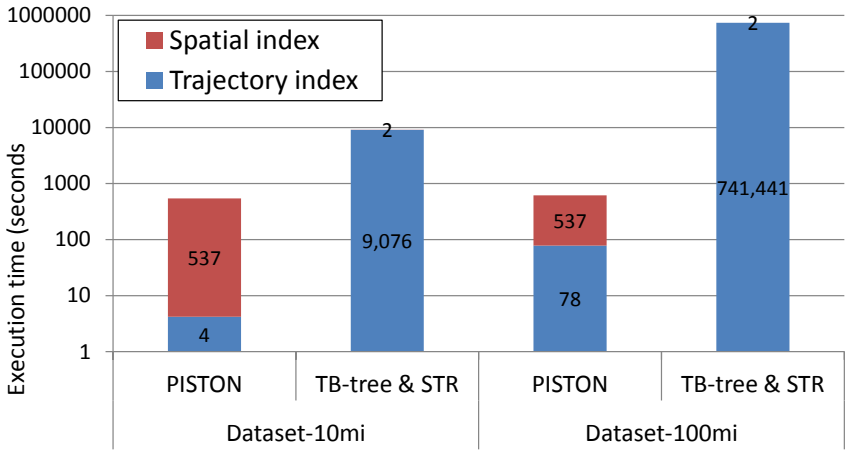


Figure 7.18: Index creation time (including preprocessing) with PISTON vs TB-tree & STR

Execution time: Since INLJ2I-TF still takes quite a long time to execute spatio-temporal topological join queries with a moderately large trajectory dataset, we introduced PISTON. In order to compare PISTON with existing index based approaches, we evaluate its performance against the best performing algorithm, INLJ2I-TF. Since INLJ2I-TF does not support intra-query parallelism we use **single-threaded** execution of PISTON. We execute the Crosses query with trajectory datasets Dataset-10mi and Dataset-100mi. We vary the query interval 200, 400, 600, 800 and 1000. The results for Dataset-10mi can be seen in Figure 7.16, and for Dataset-100mi in Figure 7.17. In both cases, single-threaded PISTON is 2 to 3 orders of magnitude faster than INLJ2I-TF (TB-tree & STR). We were not able to evaluate INLJ2I-TF with Dataset-1bi, because TB-tree takes too long to load the data from trajectory dataset and create index. The excellent speedup of PISTON over INLJ2I-TF could be attributed to the facts that the STR index search step in Figure 7.8 is eliminated and that the Point-in-polygon test is a constant time operation in PISTON. Furthermore, PISTON significantly reduces the number of Point-in-polygon tests that need to be performed.

Memory usage: Figure 7.19 shows the memory usage of PISTON against TB-tree & STR. With the smallest dataset Dataset-10mi, PISTON's memory consumption is higher than that TB-tree & STR. But with the moderate sized Dataset-100mi PISTON requires half as much as memory. With the largest dataset Dataset-1bi, TB-tree & STR approach requires more memory than the machine's available RAM capacity. Due to the preprocessing step

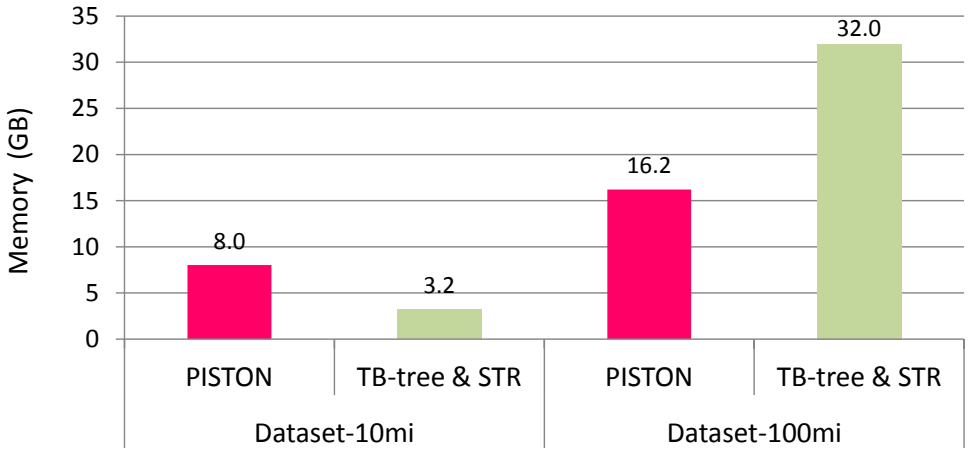


Figure 7.19: Memory usage: PISTON vs TB-tree & STR

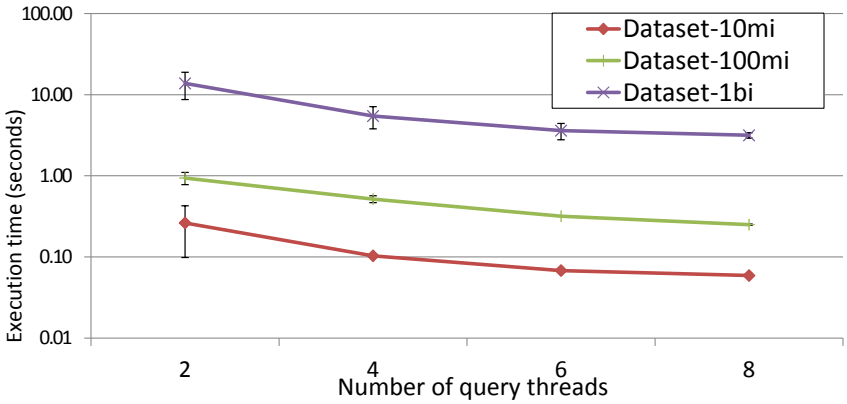


Figure 7.20: Execution times of query Crosses with different number of worker threads (interval length 1000)

PISTON'S spatial index requires more memory than the spatial index STR, that stores only object MBRs. On the other hand, due to using compressed bitmap PISTON's trajectory index is much more memory efficient than TB-tree. This suggests that with PISTON's memory usage scales well with Big Data.

Index creation time: To compare the overhead associated with PISTON against TB-tree & STR approaches, we report the time to create the trajectory index from the trajectory dataset, as well as, the time to create the spatial index from the polygon dataset. Here the polygon dataset remains the same, but trajectory dataset is changed i.e. Dataset-10mi and Dataset-100mi were used. In the case of PISTON the spatial index creation time (including the setup steps for polygons) takes longer than the trajectory index creation time. With TB-tree & STR approach, the time taken to create the trajectory index TB-tree dominates the overall time. As can be seen in Figure 7.18, with Dataset-10mi the overall index creation time is more than an order of magnitude longer with TB-tree & STR compared to PISTON. With Dataset-100mi, TB-tree & STR takes more than two orders of magnitude longer than PISTON. In fact, we could not use TB-tree to generate index for Dataset-1bi, because it takes very long.

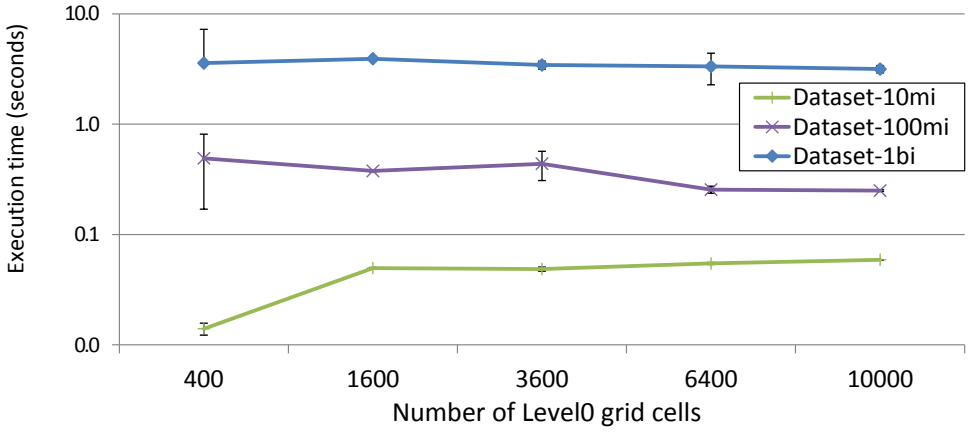


Figure 7.21: Execution times of query Crosses with different number of Level0 grid cells (interval length 1000, 8 cores)

7.6.1.3 Multi-Threaded Scalability Of PISTON

So far PISTON was used in a single-threaded setup. To evaluate the multi-threaded performance of PISTON we executed the Crosses query with 2, 4, 6 and 8 cores. We use all three trajectory datasets: Dataset-10mi, Dataset-100mi and Dataset-1bi. The query interval length was 1000. Figure 7.20 shows the execution times (in log scale) with different number of query execution threads. As can be seen, with all three datasets PISTON shows near linear speedup with the number of threads from 2 to 6. From 6 threads to 8 threads the reduction in execution time slows down. However, with 8 threads the query execution time is less than a second for both Dataset-10mi and Dataset-100mi; and for Dataset-1bi it is about 3 seconds. Due to the small query execution times, adding more threads offers diminishing returns. Note, that the 8-thread performance of PISTON with Dataset-100mi and interval 1000 represents a **speedup of 14,640X** over INLJ2I-TF, which is quite significant. Note that, we limit the evaluation to 8 cores even though the machine has more cores. Because, the query execution times with Dataset-10mi and Dataset-100mi are already less than 1 second and using more cores does not improve the performance significantly.

7.6.1.4 Handling skew

The distribution of the trajectories and the spatial objects can be highly skewed. In all previous experiments, we chose PISTON's Level0 grid dimension to be 100x100 (10000 cells). To show how PISTON manages skew, we vary the number of Level0 grid cells to 400, 1600, 3600, 6400 and 10000 (corresponding to dimensions 20x20, 40x40, 60x60, 80x80 and 100x100). We executed the Crosses query with 8 cores for query interval length 1000 using all three trajectory datasets: Dataset-10mi, Dataset-100mi and Dataset-1bi. Figure 7.21 shows the execution times (in log scale). As can be seen, for the two larger datasets the query performance remains relatively stable with the varying number of grid cells. For the smallest dataset of Dataset-10mi, the query execution time is the lowest when the number of grid cells is 400. This is expected because for this dataset, the max execution time is less than 0.1 second and the extra overhead of processing the cells is the lowest when the number of grid cells is the least. PISTON's trajectory index uses an adaptive load-balancing algorithm similar to in [102]. Its spatial index does not replicate objects, but rather splits objects along tile boundaries. As a result PISTON does a good job of handling the skew.

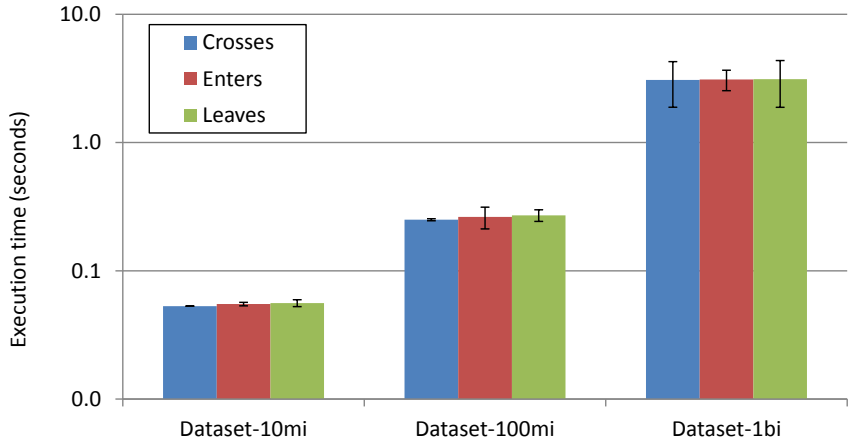


Figure 7.22: Execution times of Crosses with different predicates (interval length 1000)

Table 7.5: Evolving polygons dataset generation settings

Parameter	**Settings**
Space domain	Texas, 1251 km x 1183 km
Number of objects	1000
Time duration	1000
Distribution	Random, Skewed, Gaussian

7.6.1.5 Different Spatio-Temporal Predicates

In this section we evaluate the performance of PISTON with different spatio-temporal predicates: Crosses, Enters, Leaves. Figure 7.22 shows the execution times of the queries with PISTON for different predicates. The queries were executed with 8 threads and the interval length was 1000. All three trajectory datasets were used. The results show that the performance of the queries were very similar regardless of the predicate for any trajectory dataset.

7.6.2 Dataset With Evolving Polygons

In all previous experiments the polygon objects were fixed. They did not move or change their sizes. There are many scenarios where the objects could evolve over time, such as in urban and agricultural land-use. Other use cases include political boundary changes and geographical changes due to glacial movements, deforestation or water-level changes etc. Next we evaluate PISTON with datasets in which the polygons also evolve with time in terms of changes in location and size. The trajectory datasets remain the same as in Table 7.2.

7.6.2.1 Experimental Setup

To generate the moving polygons dataset, we utilized the GSTD tool [119]. It can generate synthetic data for moving point or rectangular objects that follow one of several possible distributions. While generating polygons dataset with GSTD we used the spatial dimensions of Texas to exactly match with the trajectory dataset's spatial dimensions. The configuration parameters are shown in Table 7.5. GSTD was used to generate evolving polygon datasets for three different distributions: Random, Skewed and Gaussian. Visual representations of the datasets are shown in Figure 7.25.

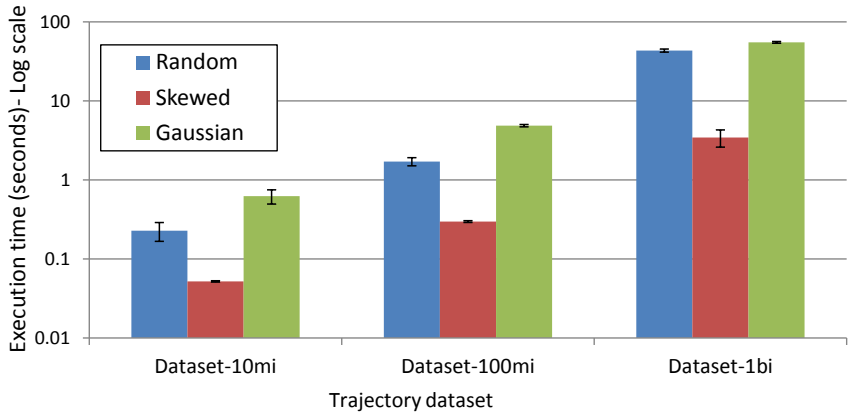


Figure 7.23: Execution times of query Crosses with different distributions for evolving polygons (single threaded)

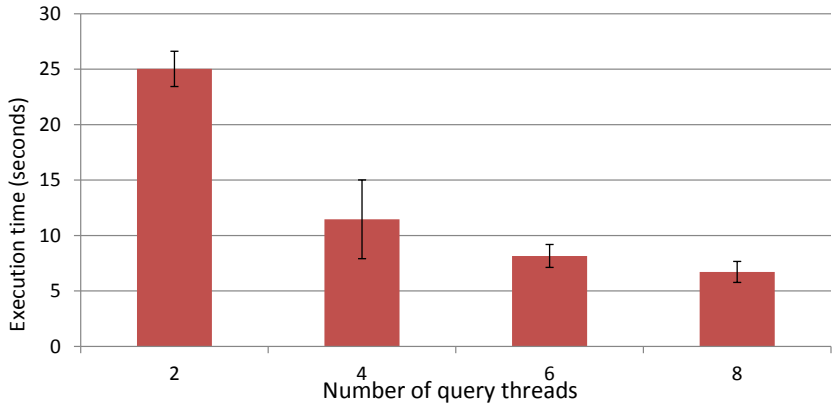
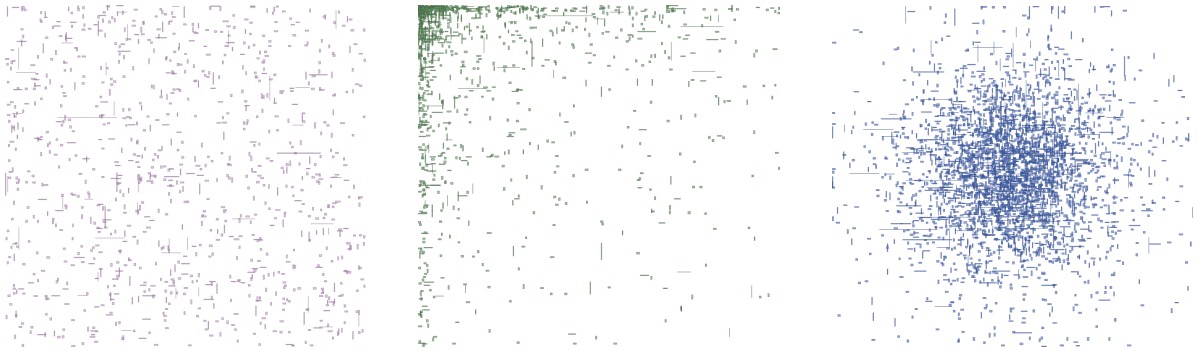


Figure 7.24: Execution times of query Crosses with different number of threads for Dataset-1bi (interval length 1000, Gaussian distribution)

7.6.2.2 Different Distributions

To compare the execution times of spatio-temporal topological join query for Crosses predicate with the different distributions, we executed the query with three polygon datasets generated by the GSTD tool as described in Section 7.6.2.1. We varied the trajectory datasets by using Dataset-10mi, Dataset-100mi and Dataset-1bi. Figure 7.23 shows the execution times of these queries. In all cases the combination of a given trajectory dataset and the polygon dataset with Gaussian distribution took the longest times. On the other hand, it look the shortest times with polygon dataset having Skewed distribution. To understand this result, the trajectory dataset Dataset-10mi is visualized in Figure 7.26. In the case of polygons generated with Skewed distribution (Figure 7.25(b)), the majority of the polygons are clustered in the top-left corner of the region. When this dataset is joined with the trajectory dataset, most of them are filtered out in the partial trajectory MBR based filtering process. Therefore very few polygons are actually joined with the trajectories. On the other hand, with Gaussian distribution, the majority of the polygons are situated about the center of the region (Figure 7.25(c)), where most of the trajectories are also clustered. Therefore, many of the polygon-trajectory pairs pass the partial trajectory MBR based filtering into the actual predicate evaluation stage and hence they require more computation relative to the previous case.



(a) Random distribution (b) Skewed distribution (c) Gaussian distribution

Figure 7.25: Distribution of polygon objects

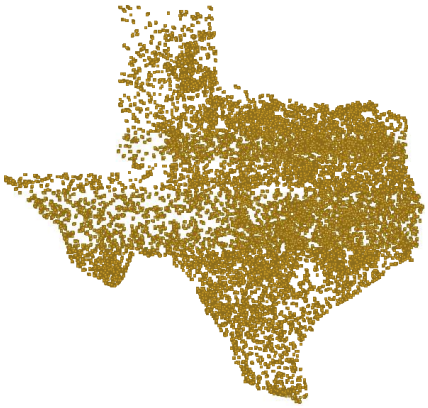


Figure 7.26: Trajectory dataset Dataset-10mi

7.6.2.3 Multi-Threaded Scalability Of PISTON

Figure 7.24 shows the multi-threaded scalability with PISTON for the polygon dataset generated with Gaussian distribution. The trajectory dataset is the Dataset-1bi. As before, PISTON shows near linear speedup as the number of threads are varied from 2 to 6.

7.7 Chapter Summary

With the explosive growth of spatio-temporal data and the increasingly popularity of Location-Based Services (LBS), there is growing impetus to deal with more complex spatio-temporal queries. Whereas spatial join received wide attention from the research community, it is not the case with spatio-temporal topological join. We have demonstrated that such queries can be long running even with a moderately large trajectory dataset.

We introduce PISTON, a parallel in-memory spatio-temporal query processing system that consists of a high performance spatial index and a trajectory index. With extensive evaluation, we have demonstrated that our system achieves several orders of magnitude better performance than the applicable algorithms with existing trajectory and spatial index.

Chapter 8

Conclusion

With the growing popularity of Web Mapping, Location-Based Services (LBS) and the advent of big-data spatial analytics, the spatial support in the major relational databases has become increasingly important. At the same time emerging technological trends bring new opportunities, as well as some challenges. Modern processors come equipped with multiple processing cores, which can be utilized to parallelize the compute intensive spatial joins common in spatial analysis tasks. Processing spatial computation in a cluster of multicore machines is in itself an interesting research problem. Such clusters are built from multiple generations of hardware where processing heterogeneity is a common phenomenon. The onset of the Cloud computing brings additional set of challenges, as the infrastructure itself is not fixed. Processing heterogeneity in the Cloud is considered norm rather than an exception. We have introduced a query execution infrastructure, Niharika, to parallelize the spatial join queries in a heterogeneity-aware manner in the Cloud. The experimental evaluations suggest that with Niharika spatial join queries involving polylines and polylines achieve near-linear speedup with the number of cores. We also indentified the issue of processing skew due to spatial object properties that limits the parallel performance of spatial join queries involving polygons.

The growing main memory capacities, due to the emerging technological trends, makes server-class machines with very large RAM affordable to most enterprises. It is now possible to store the entire dataset of many applications in memory. Many spatial dataset such as the TIGER dataset can be hosted in memory. We have presented our solution, SPINOJA, to parallelize the spatial join queries in a main memory multicore system, while addressing the performance bottleneck of processing skew that we previously identified. We also introduced a novel declustering technique and a few load-balancing approaches. We demonstrated that SPINOJA significantly outperforms existing spatial join approaches in a parallel main memory setup.

LBS play important roles in many facets of our lives. With the exponential growth of spatio-temporal data, existing indexing approaches are unable to meet the very high rate of updates demanded by many LBS applications. The advent of the large RAM holds great promise to LBS as it is now possible to store in memory the active dataset which typically encompasses the past 30 to 45 days. We described our parallel main-memory spatio-temporal index, PASTIS, and spatio-temporal data processing system for update heavy workloads and to answer historical, present and predictive queries. Our extensive evaluations show the superior location update throughput of our system over existing approaches. At the same time our system can support many concurrent spatio-temporal range queries.

Although the issue of spatial join received a lot of attention, it is not the case with spatio-temporal topological join. We presented several applicable trajectory-based spatio-temporal topological join algorithms using existing

trajectory and spatial index and demonstrated that even with a moderately large dataset these queries can be quite long-running. As a solution presented, PISTON, our approach to parallel main memory trajectory-based spatio-temporal topological join. Our experimental results suggest that PISTON can achieve orders of magnitude better sppedup than the applicable algorithms.

8.0.1 Future work

In this thesis, I presented a spatial database benchmark and addressed a number of issues related to spatial and spatio-temporal data management. As a future work, I plan to extend this benchmark by including a spatio-temporal micro benchmark that would include spatio-temporal range queries.

The parallel spatial query processing systems, Niharika and SPINOJA, are currently two indepentent systems and they have very limited support for query parsing and query optimization. I am working on incorporating the techniques from these two systems into a full-fledged open-source parallel database system called Stado [115]. This would have several spatial query features, including, parallel spatial query optimizer, spatial declustering and data uploader and multi-round parallel query scheduler.

Another avenue of my future research involves extending PASTIS to a distributed main memory setup [100], particularly, in the Cloud. Cloud service providers, such as Amazon EC2, offer a number of virtual machine instance types to suit the demands of different application domains. However, either the largest memory optimized EC2 instance type may not match memory capacity of state-of-the-art hardware, or it may be too expensive. For instance, IBM enterprise server x3950 X6 [60] offers up to 12 TB of RAM, which can't be matched by EC2 instance types. To get around this problem, it is necessary to take advantage of the aggregate memory capacity of a cluster of virtual machines in a Cloud. The RAMCloud project [87] suggests an approach in which the data is kept entirely in the main memories of many commodity servers. RAMCloud relies on very low-latency network such as Infiniband, which is still not offered by Amazon EC2. I conducted a preliminary evaluation to determine the Location table insert performance of RAMCloud with the the 10 million dataset workload. This study, conducted with 4 EC2 m1.xlarge instances in a 1 Gbps network, suggests that RAMCloud's insert throughput is rather disappointing. I believe that there is an opportunity to develop a high-performance distributed in-memory LBS system that could offer significantly better insert throughput. The spatio-temporal index PASTIS has to be extended to operate in a distributed main-memory environment. A related issue is to support distributed query execution model with historical, present and predictive range queries.

Bibliography

[1] Ashraf Aboulnaga and Jeffrey F. Naughton. Accurate Estimation of the Cost of Spatial Selections. In *International Conference on Data Engineering (ICDE)*, pages 123–134, 2000.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *International Conference on Very Large Data Bases (VLDB)*, pages 922–933. VLDB Endowment, 2009.

[3] R. Acker, C. Roth, and R. Bayer. Parallel query processing in databases on multicore architectures. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2008.

[4] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–74, 2012.

[5] Ablimit Aji, Fusheng Wang, and Joel H. Saltz. Towards building a high performance spatial query system for large scale medical imaging data. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, 2012.

[6] 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. *Proceedings of the VLDB Endowment (PVLDB)*, pages 1009–1020, 2013.

[7] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. In *International Conference on Very Large Data Bases (VLDB)*, pages 1064–1075, 2012.

[8] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *International Conference on Very Large Data Bases (VLDB)*, pages 570–581, 1998.

[9] Petko Bakalov, Marios Hadjieleftheriou, Eamonn Keogh, and Vassilis J. Tsotras. Efficient trajectory joins using symbolic representations. In *International Conference on Mobile Data Management (MDM)*, pages 86–93, 2005.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *International Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

[11] BenchmarkSQL. http://benchmarksql.sourceforge.net.

[12] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[13] Veeravalli Bharadwaj, Debasish Ghose, Venkataraman Mani, and Thomas G. Robertazzi. Scheduling Divisible Loads in Parallel and Distributed Systems. *IEEE Computer Society*, 1996.

[14] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems a systematic approach. In *International Conference on Very Large Data Bases (VLDB)*, pages 8–19, Oct/Nov 1983.

[15] Dhruba Borthakur. Petabyte scale databases and storage systems deployed at facebook. In *International Conference on Management of Data (SIGMOD)*, 2013.

[16] Thomas Brinkhoff, Hans-Peter Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *International Conference on Data Engineering (ICDE)*, pages 40–49, 1993.

[17] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *International Conference on Management of Data (SIGMOD)*, 1994.

[18] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *International Conference on Management of Data (SIGMOD)*, pages 237–246, 1993.

[19] Thomas Brinkhoff, Hans peter Kriegel, and Bernhard Seeger. Parallel Processing of Spatial Joins Using R-trees. In *International Conference on Data Engineering (ICDE)*, pages 258–265, 1996.

[20] What is Bristlecone? http://www.continuent.com/community/lab-projects/bristlecone.

[21] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with seti. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[22] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. *IPPS/SPDP*, 1999.

[23] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel H. Saltz. Titan: a high-performance remote sensing database. In *International Conference on Data Engineering (ICDE)*, 1997.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–15, 2006.

[25] Su Chen, Christian S. Jensen, and Dan Lin. A Benchmark for Evaluating Moving Object Indexes. *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[26] Yun Chen and Jignesh M. Patel. Design and evaluation of trajectory join algorithms. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 266–275, 2009.

[27] Daniel Citron, Adham Hurani, and Alaa Gnadrey. The harmonic or geometric mean: does it really matter? *SIGARCH Computer Architecture News*, 34:18–25, 2006.

[28] Eliseo Clementini and Paolino Di Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, 90(1-4):121–136, 1996.

[29] City of Austin GIS Data Sets. ftp://ftp.ci.austin.tx.us/GIS-Data/Regional/coa_gis.html.

[30] Alessandro Colantonio and Roberto Di Pietro. CONCISE: COmpressed 'N' Composable Integer SEt. *Information Processing Letters*, 110:644–650, 2010.

[31] Alain Crolotte. Issues in benchmark metric selection. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 146–152. Springer Berlin / Heidelberg, 2009.

[32] DBT - database test suite. http://osdldbt.sourceforge.net.

[33] Michael J. de Smith, Michael F. Goodchild, and Paul A. Langley. *Geospatial Analysis: a comprehensive guide to principles techniques and software tools*. Matador, third edition, 2009.

[34] D. J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 35(6):85–98, 1992.

[35] Travis County Digital Flood Insurance Ratemap Database. ftp://ftp.ci.austin.tx.us/GIS-Data/Regional/environmental.

[36] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013.

[37] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *International Symposium on Spatial and Temporal Databases (SSTD)*, pages 189–207, 2009.

[38] M. Egenhofer, A. Frank, and J.P. Jackson. A topological data model for spatial databases. In *International Symposium on Large Spatial Databases*, 1989.

[39] M. J. Egenhofer and R. Franzosa. Point-set topological spatial relations. *International Journal of Geographic Information Systems*, 5(2):161–174, 1991.

[40] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature management in data centers: why some (might) like it hot. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[41] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):881–901, 2002.

[42] Ying Fang, Jiaheng Cao, Yuwei Peng, and Liwei Wang. Indexing the Past, Present and Future Positions of Moving Objects on Fixed Networks. In *International Conference on Computer Science and Software Engineering (CSSE)*, pages 524–527, 2008.

[43] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Symposium on Cloud Computing (SoCC)*, pages 20:1–20:14, 2012.

[44] Alvaro A. A. Fernandes, Andrew Dinn, Norman W. Paton, M. Howard Williams, and Olive Liew. Extending a deductive object-oriented database system with spatial data handling facilities. *Information and Software Technology*, 41(8):483–497, 1999.

[45] Top 300 commercial fleets. http://www.fleet-central.com/TopFleets/.

[46] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *International Conference on Management of Data (SIGMOD)*, pages 319–330, 2000.

[47] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 1999.

[48] GEOS. http://http://trac.osgeo.org/geos/.

[49] Michael L. Gonzales. Seeking spatial intelligence. *Intelligent Enterprise (InformationWeek)*, 3(2), January 2000.

[50] Jim Grey. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[51] Christophe Gurret, Apostolos Papadopoulos, Yannis Manolopoulos, and Philippe Rigaux. The basis system: a benchmarking approach for spatial index structures. In *International Workshop on Spatio-Temporal Database Management (STDBM)*, pages 152–170, 1999.

[52] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, pages 357–399, 1994.

[53] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[54] Marios Hadjieleftheriou, George Kollios, J. Tsotras, and Dimitrios Gunopulos. Indexing spatiotemporal archives. *VLDB Journal*, 15(2):143–164, June 2006.

[55] Apache Hadoop. http://hadoop.apache.org/.

[56] Eric Haines. Point in polygon strategies. In Paul Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, 1994.

[57] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Conference on Innovative Data Systems Research (CIDR)*, pages 79–87, 2007.

[58] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database system. In *International Conference on Very Large Data Bases (VLDB)*, pages 562–573, 1995.

[59] Abdeltawab M. Hendawi and Mohamed F. Mokbel. Panda: A Predictive Spatio-temporal Query Processor. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 13–22, 2012.

[60] IBM servers. http://www-03.ibm.com/systems/x/hardware/enterprise/.

[61] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Maintenance of spatial semijoin queries on moving points. In *International Conference on Very Large Data Bases (VLDB)*, pages 828–839, 2004.

[62] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Transactions on Database Systems*, 32(1), 2007.

[63] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient B+-tree based indexing of moving objects. In *International Conference on Very Large Data Bases (VLDB)*, pages 768–779, 2004.

[64] Junchen Jiang, Hongji Bao, Edward Y. Chang, and Yuqian Li. MOIST: A scalable and parallel moving object indexer with school tracking. *Proceedings of the VLDB Endowment (PVLDB)*, pages 1838–1849, 2012.

[65] jTPCC - open source Java implementation of the TPC-C benchmark. http://jtpcc.sourceforge.net.

[66] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[67] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *International Conference on Management of Data (SIGMOD)*, pages 324–335, 1997.

[68] G. Leptoukh. NASA remote sensing data in earth sciences: Processing, archiving, distribution, applications at the GES DISC. In *International Symposium on Remote Sensing of Environment (ISRSE)*, 2005.

[69] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. STR: A simple and efficient algorithm for R-tree packing. In *International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

[70] Dan Lin, Christian S. Jensen, Beng Chin Ooi, and Simonas Šaltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *International Conference on Mobile Data Management (MDM)*, pages 59–66, 2005.

[71] Hung-Yi Lin. Indexing the Trajectories of Moving Objects. In *International MultiConference of Engineers and Computer Scientists*, 2009.

[72] Diego R. Llanos. TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Record*, 35(4):6–15, 2006.

[73] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *International Conference on Management of Data (SIGMOD)*, pages 247–258, 1996.

[74] Gang Luo, Jeffrey F. Naughton, and Curt J. Ellmann. A Non-Blocking Parallel Spatial Join Algorithm. In *International Conference on Data Engineering (ICDE)*, pages 697–705, 2002.

[75] Tobias Mayr, Philippe Bonnet, and Johannes Gehrke. Leveraging non-uniform resources for parallel query processing. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2002.

[76] MOTO (Moving Objects Trace generatOr). http://moto.sourceforge.net/.

[77] Jussi Myllymaki and James Kaufman. Dynamark: A benchmark for dynamic spatial indexing. In *International Conference on Mobile Data Management (MDM)*, pages 92–105, 2003.

[78] Mario A. Nascimento and Jefferson R. O. Silva. Towards historical R-trees. In *Symposium on Applied Computing (SAC)*, pages 235–240, 1998.

[79] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE Database Engineering Bulletin*, 33:46–55, 2010.

[80] Jinfeng Ni and Chinya V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering*, 19(5):663–678, May 2007.

[81] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *International Conference on Mobile Data Management (MDM)*, pages 7–16, 2011.

[82] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. Touch: in-memory spatial join by hierarchical data-oriented partitioning. In *International Conference on Management of Data (SIGMOD)*, pages 701–712, 2013.

[83] http://www.opengeospatial.org/ogc.

[84] Open Geospatial Consortium. http://www.opengeospatial.org/ogc.

[85] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *International Conference on Management of Data (SIGMOD)*, pages 326–336, 1986.

[86] The Open Source Database Benchmark. http://osdb.sourceforge.net.

[87] John Ousterhout et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Operating Systems Review*, pages 92–105, 2010.

[88] PostgreSQL Partitioning. http://www.postgresql.org/- docs/8.3/static/ddl-partitioning.html.

[89] 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. In *International Conference on Management of Data (SIGMOD)*, pages 336–347, 1997.

[90] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *International Conference on Management of Data (SIGMOD)*, pages 259–270, 1996.

[91] Jignesh M. Patel and David J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 54–61, 2000.

[92] N. Patton, M. H. Williams, K. Dietrich, O. Liew, A. Dinn, and A. Patrick. VESPA: A benchmark for vector spatial databases. In *British National Conference on Databases: Advances in Databases*, pages 81–101, July 2000.

[93] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. Dewitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *International Conference on Management of Data (SIGMOD)*, 2009.

[94] Nikos Pelekis, Babis Theodoulidis, Ioannis Kopanakis, and Yannis Theodoridis. Literature review of spatio-temporal database models. *Knowledge Engineering Review*, 19(3):235–274, September 2004.

[95] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *International Conference on Very Large Data Bases (VLDB)*, pages 395–406, 2000.

[96] PolePosition. http://polepos.sourceforge.net.

[97] R. Power. Testing geospatial database implementations for water data. In *International Congress on Modelling and Simulation (IMACS/MODSIM Congress)*, 2009.

[98] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[99] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *International Conference on Data Engineering (ICDE)*, pages 60–69, 2008.

[100] Suprio Ray. Towards High Performance Spatio-temporal Data Management Systems (© 2011 IEEE, reprinted with permission from authors). In *International Conference on Mobile Data Management - Ph.D. Colloquium*, pages 19–22, 2014.

[101] Suprio Ray, Rolando Blanco, and Anil K. Goel. Enhanced Database Support for Location-Based Services. In *International Workshop on GeoStreaming (IWGS)*, 2013.

[102] Suprio Ray, Rolando Blanco, and Anil K. Goel. Supporting Location-Based Services in a Main-Memory Database (© 2011 IEEE, reprinted with permission from authors). In *International Conference on Mobile Data Management (MDM)*, pages 3–12, 2014.

[103] Suprio Ray, Bogdan Simion, and Angela Demke Brown. Jackpine: A Benchmark to Evaluate Spatial Database Performance (© 2011 IEEE, reprinted with permission from authors). In *International Conference on Data Engineering (ICDE)*, pages 1139–1150, 2011.

[104] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. A Parallel Spatial Data Analysis Infrastructure for the Cloud. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 274–283, 2013.

[105] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. Skew-resistant parallel in-memory spatial join. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 6:1–6:12, 2014.

[106] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Symposium on Cloud Computing (SoCC)*, pages 7:1–7:13, 2012.

[107] Markus Schneider. Evaluation of spatio-temporal predicates on moving objects. In *International Conference on Data Engineering (ICDE)*, pages 516–517, 2005.

[108] Shashi Shekhar, Sanjay Chawla, Siva Ravada, Andrew Fetterer, Xuan Liu, and Chang-tien Lu. Spatial databases - accomplishments and research needs. In *IEEE Transaction on Knowledge and Data Engineering*, pages 45–55, 1999.

[109] Carter Shock, Chialin Chang, Bongki Moon, Anurag Acharya, Larry S. Davis, Joel H. Saltz, and Alan Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 1998.

[110] A Very Short History Of Big Data. http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/.

[111] Vishal Sikka, Franz Frber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhvd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *International Conference on Management of Data (SIGMOD)*, 2012.

[112] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *International Conference on Software Engineering (ICSE)*, pages 74–83, 2003.

[113] Bogdan Simion, Suprio Ray, and Angela Demke Brown. Surveying the landscape: An in-depth analysis of spatial database workloads. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 376–385, 2012.

[114] SPECweb2009. http://www.spec.org/web2009.

[115] Stado: The Open Source MPP Solution. https://launchpad.net/stado.

[116] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, and Nga Tran. C-Store: A Column-oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[117] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 storage benchmark. In *International Conference on Management of Data (SIGMOD)*, pages 2–11, 1993.

[118] Yufei Tao and Dimitris Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *International Conference on Very Large Data Bases (VLDB)*, pages 431–440, 2001.

[119] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In *International Symposium on Advances in Spatial Databases (SSD)*, pages 147–164, 1999.

[120] TIGER®, TIGER/Line® and TIGER®-Related Products. http://www.census.gov/geo/www/tiger.

[121] The Transaction Processing Performance Council. http://www.tpc.org.

[122] Arbind Man Tuladhar. *Parcel-based Geo-Information System: Concepts and Guidelines*. PhD thesis, Technische Universiteit. Delft, Netherlands, 2004.

[123] VoltDB. http://voltdb.com/.

[124] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *Sigmod Record*, 29:331–342, 2000.

[125] Darius Šidlauskas, Simonas Šaltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas aulys. Trees or grids?: Indexing Moving Objects in Main Memory. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 236–245, 2009.

[126] Darius Šidlauskas, Simonas Šaltenis, and Christian S. Jensen. Parallel Main-Memory Indexing for Moving-Object Query and Update Workloads. In *International Conference on Management of Data (SIGMOD)*, pages 37–48, 2012.

[127] Eugene Wu and Samuel Madden. Partitioning techniques for fine-grained indexing. In *International Conference on Data Engineering (ICDE)*, pages 1127–1138, 2011.

[128] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2008.

[129] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2009.

[130] Xiaofang Zhou, David J. Abel, and David Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, pages 175–204, 1998.