Scalable Big Spatial Data Processing with SQL Query Compilation and Distributed Morsel-driven Parallelism

Rahul Sahni, Xiaozheng Zhang, Sudip Chatterjee and Suprio Ray
University of New Brunswick, Fredericton, Canada
Email: {rahul.sahni, xz.zhang, sudip.chatterjee, sray}@unb.ca

Abstract—**The rapid rise in spatial data volumes from diverse sources necessitate efficient spatial data processing capability. Although most relational databases support spatial extensions of SQL query features, they offer limited scalability. Traditional relational database query processing follows a pull-based (or tuple-at-a-time) model of query processing. This is not efficient for processing large volumes of data. A number of specialized spatial data processing systems were developed that extend cluster computing frameworks, such as Spark and Hadoop. However, these systems are characterized by limited or no support for spatial SQL query execution. The few systems that support SQL querying, suffer from the overheads of the pull-based model.**

We present a compilation-based distributed SQL query processing system. It follows a data-centric query compilation approach that takes a SQL query and generates distributed C++ (UPC++) based physical query plans. The generated code is compiled and executed on a distributed in-memory high performance framework based on the Partitioned Global Address Space (PGAS) paradigm. We also introduce morsel-driven parallelism for scalable spatial query execution in a distributed runtime. We conduct experimental evaluation of our system with two real-world datasets on a number of spatial query workloads. Experimental results demonstrate that our system performs significantly better than a leading spatial big data system Apache Sedona and distributed parallel relational database Citus.

I. Introduction

The volume of spatial data is rising due to many factors, including the spread of GPS-enabled mobile devices and sensors, geo-social media, advances in remote sensing and satellite imaging, and improving storage capacity. In response to the challenges of spatial big data several specialized systems were proposed that extended cluster computing frameworks Hadoop and Spark. They include Apache Sedona [1] (previously, GeoSpark), SpatialSpark [2], Simba [3], LocationSpark [4], Hadoop-GIS [5] and SpatialHadoop [6]. These systems enabled practitioners to process spatial data by leveraging a cluster of machines. However, they are still in early stages of development and hence they have several limitations. According to Yu et al. [7], some of these systems support only point objects or MBR-based spatial query processing. Moreover, most of them either do not support spatial SQL or offer limited support for SQL that does not conform to ANSI-standard SQL standards [8].

In contrast, SQL is the *lingua franca* for relational database management systems (RDBMS) and most commercial and open-source relational databases offer spatial functionalities. Increasingly, SQL is becoming a top language for data analysis [9]. While RDBMS are widely used for enterprise data management, partly due to the popularity of SQL, they suffer from performance bottlenecks due to their focus on minimizing disk. RDBMS engines follow an iterator-based "tuple-at-a-time" model, which is also known as the Volcano model [10] or pull-based model. This is inherently inefficient in terms of performance due to processing each tuple by making repeated calls to the *next()* function call for each relational operator in the query plan from an input tuple stream. Driven by advances in computer architecture, modern machines are equipped with large main memory and several processing cores. Since larger amounts of data can fit in the main memory than was previously possible, optimizing code for memory usage and adopting CPU-efficient techniques have become more important. Therefore, in recent years, query compilation or push-based execution model has attracted considerable attention from the research community [11] Query compilation is different from traditional interpretation-based query processing, as it allows for the generation of query-specific and data-centric code.

Although query compilation can offer significant performance benefits, re-architecting an SQL query engine to incorporate data-centric compilation is challenging because of the associated complexity. Consequently, existing RDBMSs have either not adopted it at all, or they support a very limited form of query compilation. For example, currently PostgreSQL supports just-in-time (JIT) query compilation for tuple materialization and expression evaluation only. This requires building PostgreSQL from source with the flag `--with-llvm` or building it with OMR JitBuilder support [12]. Adapting query compilation techniques for spatial workloads entails additional complexities. To our knowledge, only one previous research made an endeavour [13], where they identify why existing query compilation techniques are not quite effective for spatial queries. They propose a generative query compilation approach, LB2-Spatial, which transpiles a spatial SQL query into a source program (in Scala or C) and it then gets compiled into native code and executed. To our understanding, LB2-Spatial focuses on MBR-based spatial query execution (essentially, the Filter step of the 2 step Filter-Refinement process). Moreover, LB2-Spatial is based on a single node. Such an approach is not scalable, particularly, in view of rapidly growing data volume.

Besides a lack of support for SQL for spatial data, another issue with many of the big spatial systems is that they do not handle processing skew dynamically. Spatial data is characterized by both (i) data skew, caused by uneven distributions of tuples, and (ii) processing skew, caused by variation in computation time in the Refinement step due to the difference in object sizes and complexity. Existing approaches support spatial partitioning that can handle data skew, but they do not handle processing skew very well. LocationSpark introduced a scheduler to deal with spatial query skew resulted by high popularity of certain data partitions. However, LocationSpark does not handle processing skew, and it does not support SQL. Apache Sedona supports standard spatial SQL, but it also does not handle processing skew.

In this paper, we present a scalable relational data system that supports execution of standard spatial SQL over a cluster of machines. Our system, CasaDB [14], performs compilation of SQL queries following a data-centric approach. It takes a SQL query as input and generates UPC++ code (a high performance computing or HPC extension of C++) following a distributed HPC programming model called Partitioned Global Address Space (PGAS) [15]. The UPC++ code is compiled into native code and executed on a PGAS runtime distributed over a number of machines. To support parallel query processing while addressing processing skew, our system introduces a distributed morsel-driven parallel query scheduler. It is inspired by the morsel-driven parallelism [16], which is based on the idea of scheduling small fragments of input data, called *morsels*, to worker thread. The worker thread processes a morsel over an entire operator pipeline until the next pipeline breaker in a compiled query plan.

For regular non-spatial data, each morsel contains the same number of tuples. However, the original morsel-driven parallelism approach [16] will not work for spatial data. For a spatial workload, each morsel should correspond to a spatial partition or *tile*. Moreover, the morsel-driven approach needs to be adapted to handle spatial data processing skew. Therefore, we propose two approaches for morsel-driven parallel spatial data processing: *Monolithic Tile-based Morsel-driven Parallelism (MTMP)* and *Granular Tile-based Morsel-driven Parallelism (GTMP)*. In both of these approaches, each spatial tile is treated as a morsel, but in the GTMP we introduce granularity at a sub-morsel level that are called *granules*. The original morsel-driven parallelism was introduced for a single node multicore machine and for regular (non-spatial) workload. To our knowledge, our system is the first to incorporate distributed spatial query compilation and distributed morsel-driven spatial query execution. As a result, our approach can handle processing skew by dynamically handling spatial partitions over a cluster on machines.

We have conducted a thorough experimental evaluation with two different real-world spatial datasets: TIGER [17] California and OpenStreetMap [18] (OSM). On a workload involving several spatial joins. According to prior research [19], GeoSpark (i.e. Apache Sedona) showed the best performance among 5 Spark based spatial data systems. Hence, in our experimental evaluation, we compare CasaDB (spatial) with Apache Sedona. Our results demonstrate that on spatial query workloads CasaDB performs 10x to 151x better than Sedona. We have also conducted an experimental evaluation of CasaDB against a modern distributed parallel relational database Citus [20]. Here also CasaDB performs 2x better than Citus.

The main contributions of this paper are as follows:

- We propose an SQL query compilation based scalable spatial data processing system, which can generate code for single node or a distributed runtime based on PGAS paradigm.
- We introduce two algorithms for morsel-driven parallel processing of spatial queries: MTMP and GTMP.
- We present a few features to improve parallel spatial query performance, including global and local indexing.
- We present extensive experimental results involving two real-world datasets against Apache Sedona and Citus.

The remainder of this paper is organized as follows. In Section II we provide a background. We describe our system approach in Section III and the query processing algorithms in Section IV. Then we outline the experimental evaluation of our system in Section V. The related work is discussed in Section VI. Finally, we conclude the paper in Section VII.

II. Background

In this section we provide some background information on query compilation and distributed query processing using partitioned global address space (PGAS).

A. Query Compilation

Many of the database engines were developed when I/O was dominating the overall processing time and the memory was low. They typically use the *tuple-at-a-time*/Volcano model [10], which is not efficient in the context of modern machines with large main memory. The key concept behind query compilation is to process the data in a tight loop and perform operator fusion. In this way, the code and data reside in the CPU cache, therefore, producing better performance. Instead of iterating over every operator in the QEP tree for each tuple, the query compilation technique generates high-level/low-level code. Neumann et al. [11] showed how operator fusion can improve performance significantly by combining multiple operators in a tight loop until there is a pipeline breaker, such as join. In this work, we choose this approach of query compilation for spatial data. In addition, we generate code that can be executed in distributed environments.

B. Partitioned Global Address Space and UPC++

Although a number of specialized data systems have been proposed, which are based on Hadoop and Spark, these systems suffer from the overhead of the underlying frameworks. In this work, we have aimed to leverage high-performance computing (HPC) framework to support distributed parallel execution of SQL queries. Specifically, we utilize UPC++, which is a PGAS programming framework designed for HPC that augments C++ with distributed runtime capabilities.

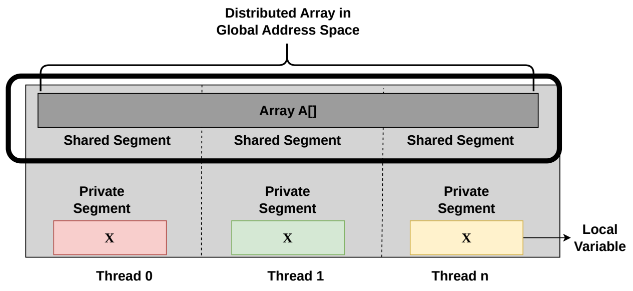


Fig. 1: PGAS memory organization

Partitioned global address space (PGAS) [15] is a parallel programming model for developing high-performance applications on computer clusters. It provides a global address space partitioned among the cluster nodes. PGAS programs are based on a single program, multiple data (SPMD) model running on a cluster. At run time, a PGAS program consists of multiple processes executing the same code on different nodes. Each process has a rank, which is the identifier of the node it runs on. The processes can access a global address space partitioned into local address spaces for each process. Local addresses can be accessed directly. Remote addresses belonging to different processes are accessed using API calls, also known as remote procedure calls (RPC). Figure 1 illustrates the memory organization of a PGAS system. PGAS contains two segments, a *shared segment* where the data is shared globally and a *private segment* where the data is private to each thread. The array $A[]$ is a shared distributed data structure. Each process can access it. GASNet [21], a PGAS network protocol, provides both synchronous and asynchronous versions of reads and writes.

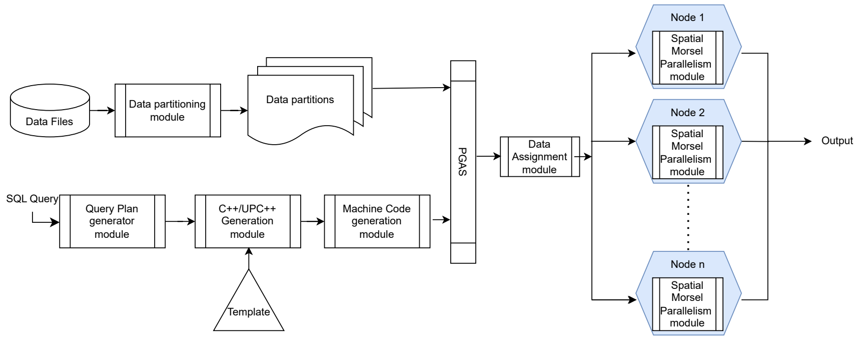
UPC++ [22], [23] is a C++ library that supports the Partitioned Global Address Space (PGAS) programming model. UPC++ is designed for writing efficient, scalable parallel programs on distributed-memory parallel computers.

III. Our approach

The high-level architecture of our system, CasaDB, is shown in Figure 2. The query compilation engine of CasaDB consists of several modules: Data partitioning module, Data assignment module, Query plan generator, Code generation module and machine code generation module. The Data partitioning module is responsible for partitioning the data files using QuadTree partitioning algorithm, which divides the spatial domain of the dataset into four quadrants recursively. This can deal with data skew more efficiently as compared to uniform grid partitioning scheme. Once the data is partitioned, it is the responsibility of Data assignment module to assign data partitions to the nodes in the cluster. It makes sure that nodes read only the data they will be processing. The Query plan generator module takes in SQL query and it leverages Apache Calcite [24] to generate optimized physical query plan. The physical query plan in then ingested by Code generation module, which generates query-specific, data-centric C++/UPC++ code. This module follows the push-based model [25] to generate high-level code and ensures that the tuples stay in the CPU registers until a pipeline breaker is reached. This pipeline breaker is the boundary until which a tuple can move through the operators without materialization. This way of code generation helps the data to be near the L1 cache, therefore exploiting the CPU registers and getting the best performance from the processing units. Finally, the Machine code generator module produces optimized machine code, which is then sent to all of the nodes in the cluster. Each of these nodes support an independent Spatial Morsel-driven Parallelism Module. This module supports morsel-driven parallelism in each of the nodes, which helps them to efficiently process each pipeline independently. Finally, the generated UPC++ is compiled into native code, which runs on the PGAS runtime, to produce the query output.

A. Code Generation

We use the compilation based technique [11] to generate the C++ / UPC++ code for an input SQL query. The main idea behind this technique is that, during query processing the tuples should remain in the registers as long as possible, and they should only be spilled to memory when needed (e.g. at a *pipeline breaker*). An important concept in the push-based data-centric model is the pushing of data towards operators, instead of pulling tuple at a time from the data stream as in the Volcano model. The push-based model introduces two separate functions, *produce()* and *consume()*, to generate the code, as shown in Fig. 3. It is data-centric and keeps the tuples in the CPU registers as long as possible, and only materializes the tuples at pipeline breakers. It pushes the data towards the operators which results in much better code and data locality. On the other hand, the Volcano model [10], uses the $next()$ to get the next tuple and it keeps iterating on the tuple stream and processing them one by one, this introduces a lot of function calls and materialization points, which make it inefficient. One thing to note about the compilation based technique is that the *produce()* and *consume()* function calls on the operators are not present in the final generated C++/UPC++ code, instead we use these functions to generate parts of the final generated code. A physical query plan generated by CasaDB for a *Spatial Join query* is shown in Figure 4(b). CasaDB has a set of physical operators and they all correspond to Relational Algebra (RA) operators. Each of the these physical operators have *produce()* and *consume()* defined on them. Hence, calling *produce()* on any of these operators will call *produce()* on its child and so on, and calling *consume()* will emit the operator specific code. The *CTableScan* operator emits code related to reading tuples from a table, whereas *CIndexScan* is responsible for emitting code related to index scan. The operator *CSpatialJoin* emits code that performs the actual spatial join operation and spatial predicate evaluation. The operator *CApply* performs the projection operation and emits the code related to that. Finally, the *CStore* operator is responsible for materializing the results and it emits code related to that. All of these operators have their pre-defined C++/UPC++ templates, which are used to generate data-centric code. As can be seen in Figure 4(b), the root of the tree *CStore* calls *produce()* on its child *CApply*, and the *CApply* calls *produce()* on its child, and



Data Files
Data partitioning module
Data partitions
SQL Query
Query Plan generator module
C++/UPC++ Generation module
Machine Code generation module
Template
PGAS
Data Assignment module
Node 1
Spatial Morsel Parallelism module
Node 2
Spatial Morsel Parallelism module
Node n
Spatial Morsel Parallelism module
Output

Fig. 2: CasaDB architecture

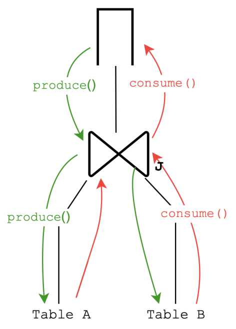


Fig. 3: Produce-Consume model

this continues until we reach the leaf node. When *CTableScan* determines that it has no child to call *produce* on, it will call *consume()* on its parent with the emitted code, which is highlighted in the Figure 4. Then the parent of *CTableScan* will call *consume()* on its parent, and it will continue until we reach the root node. There is a *CompilerState* object in the Code generation module that knows where to place the emitted block of code in the final generated code.

B. Data Partitioning

In a typical relational distributed query execution system, regular (non-spatial) data is statically partitioned using Range or Hash partitioning so that each node in the cluster gets equal size of relevant data. Spatial joins use the spatial boundary of spatial partitions or tiles to perform computation. So, such type of partitioning scheme will not work with spatial data. We also need to efficiently deal with the tuple distribution skew, which is caused by the variation in the number of tuples in each partition, and also need to deal with processing skew, which is due to the complexity and difference in the size of spatial objects. We use a spatial declustering scheme that tries to create balanced spatial partitions and to deal with the inherent skew in spatial data. This is based on a recursive partitioning of the spatial domain, resembling QuadTree partitioning. In each round of the algorithm, our declustering approach finds the tile with the most number of objects and partitions it into four equal size quadrants. This process continues until the total number of tiles generated so far is below a maximum threshold. This threshold can be configured to generate partitions with 512, 1024, 2048 and 4096 (or another number of) tiles. We evaluate the performance with each of these configurations in Section V. This partitioning of data is handled by the data partitioning module mentioned above.

C. Tile Assignment

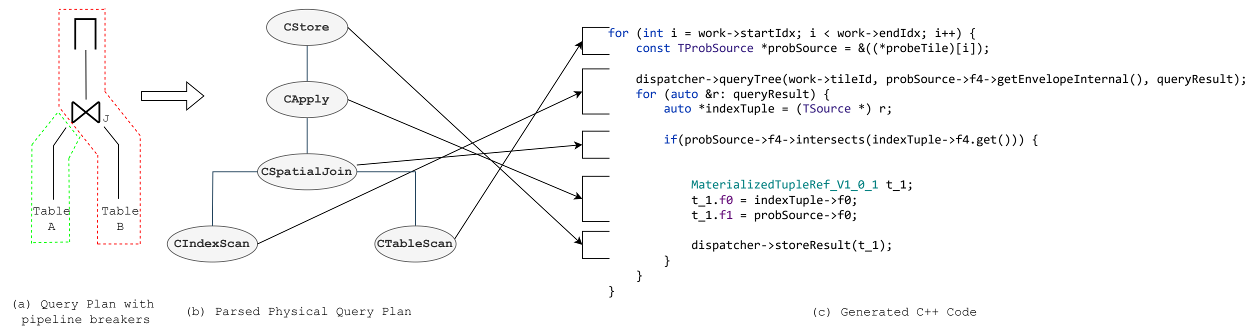
After partitioning the data, we need to make sure that we only read data that is required by the node. We achieve this by range-partitioning the number of tiles and then assigning a range of tiles to each of the nodes. This ensure that each node only deals with the tiles it is assigned. This assignment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

$$nt <= \left\lceil \frac{number\ of\ tiles}{number\ of\ nodes} \right\rceil \tag{1}$$

This tile distributions plays a major role in the final Join operation in the Spatial Morsel-driven parallelism module, as we only need to look at the corresponding tiles to which a tuple from probe table belongs too while looking at indexed tuple. This essentially reduces our search space by a margin.

D. Index Organization

Indexing can speedup the Join operation. In the Filter phase of the Join operations, a spatial index can filter out a significant number of tuples. Since the Refinement step dominates the entire Join operation so, we need to effectively filter out as many tuples as possible before we get to actual spatial predicate evaluation. Ideally, we build an index on the smaller table and then use the same index on all of the nodes, but in our system index creation is decided very carefully and it depends on the type of Join operation being performed. Spatial Joins can be classified into a few categories. Queries that find object pairs satisfying some spatial predicates, such as `ST_CONTAINS` are called *Spatial Join Query*. A query that finds object pairs satisfying a particular distance unit is called *Distance Join Query*, `ST_DISTANCE` predicate is a good example of this. A query that finds object pairs that are within a particular range (e.g. a circular region with a given radius) of each other is called *Range Join query*, and `ST_DIWITHIN` can be used for this. The *kNN Join Query* finds the top-k nearest object pairs that satisfy a spatial predicate. For *SpatialJoinQuery*, CasaDB (spatial) could use the tile-based approach to perform the join, where in the Filter step objects belonging to the same tile are processed together, and it can also use tile-wise index for filtering. In case of *Distance Join Query*, using index is not useful, because we need to calculate the distance between each object pair, and then check if they satisfy the mentioned distance unit or not. For *RangeJoinQuery*, using tile-wise index may also not be helpful, because we could be checking whether an object is within the radius of a query object located in some other tile. So for this case, we build the index on the entire table (Global Index). Building index dynamically enables CasaDB to fully utilize the degree of parallelism (DoP) through our spatial Morsel-drievn parallelism approach. Fig. 5 shows both types of spatial index.



```
for (int i = work->startIdx; i < work->endIdx; i++) {
    const TProbSource *probSource = &((*probeTile)[i]);

    dispatcher->queryTree(work->tileId, probSource->f4->getEnvelopeInternal(), queryResult);
    for (auto &r: queryResult) {
        auto *indexTuple = (TSource *) r;

        if(probSource->f4->intersects(indexTuple->f4.get())) {


            MaterializedTupleRef_V1_0_1 t_1;
            t_1.f0 = indexTuple->f0;
            t_1.f1 = probSource->f0;

            dispatcher->storeResult(t_1);
        }
    }
}
```

Fig. 4: Code generation

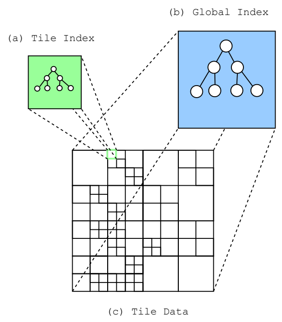


Fig. 5: Index organization

E. Morsel-Driven Parallelism

To overcome the limitations of parallelism in *exchange* operators in Volcano model, Morsel-driven parallelism [16] was introduced. It achieves parallelism by running the operator pipelines in parallel on separate threads and can even change the degree of parallelism mid-execution. Unlike traditional parallelism approach where data is equally divided amongst all the threads, here the data is divided into small fixed-sized chunks, called *morsels*. The dispatcher spawns a fixed number of machine-dependent threads and each of these threads is assigned a morsel. Once a morsel is done processing, the worker is assigned another morsel. Spatial workloads are very different from regular workloads. Some of the tuples in a spatial workload could take more time to process than other tuples because of the complexity of the geometric object they represent. In contrast, the time to process different tuples from a regular query workload (involving non-spatial data) is consistent on average. The original morsel-driven parallelism [16] works well for regular workloads, but for tile-based spatial workload it is not that straightforward, especially when it comes to "defining" a morsel and also the inherent processing skew in spatial-workloads. We propose two different algorithms based on morsel-driven parallelism for parallel spatial query processing: *Monolithic Tile-based Morsel-driven Parallelism (MTMP)* and *Granular Tile-based Morsel-driven Parallelism (GTMP)*. In both of these algorithms, each tile is treated as morsel, but in the *GTMP* we introduce granularity at a sub-morsel level, these are called "granules" and instead of processing a morsel, we process its granules. The scheduling of these work units involve a dispatcher assigning a morsel (in MTMP) or granule (in GTMP) to a worker thread. Once a thread is done processing its assigned morsel/granule, it receives the next morsel/granule until all of the morsels/granules are processed. In our experimental evaluation, we found that sometimes MTMP struggles with data skew, especially when both of the tiles in a spatial join operation have a lot of tuples (i.e. a large morsel), the worker assigned to this morsel takes a lot of time, while the other workers are done with their morsels and the overall process is waiting for this worker to finish. GTMP handles this scenario gracefully by breaking each morsel into granules and processing them efficiently such that the other waiting workers can take up the granules from the big morsel for processing.

1) Monolithic Tile-based Morsel Parallelism (MTMP):

In this algorithm, all the tuples in a tile make one morsel, essentially we are processing one tile at a time. So, if we have used a partitioning scheme that divides the data into 512 tiles, for example, then the number of morsels will be 512. The number of tuples in a morsel or the morsel size in this approach is dynamic, compared to the original morsel-driven approach, where they set the morsel size to a fixed number, for instance, 10,000 tuples. The morsel size in our approach is equal to the number of tuples in a particular tile. So, the morsel size is dynamic but the number of morsel is static. This approach was our first attempt at introducing morsel-driven parallelism in our spatial query engine. It has three components, *MTMPWork*, *MTMPWorker* and *MTMPDispatcher*. *MTMPWork* defines the work that needs to be done. It contains the morsel, id of the tile the morsel belongs to and the kind of processing needs to be done on the morsel, which is defined by *WorkState*. This *WorkState* indicates the *MTMPWorker* the kind of work that needs to be done on the morsel. The work could involve processing the tiles, or waiting for all the other workers to finish their processing when all the morsels are assigned to the workers, or the final state when all the processing is done. Here, the morsel is all of the tuples inside a particular tile. *MTMPWorker* does the actual processing of the morsel. As soon as such a *MTMPWorker* is spawned it asks for *MTMPWork*, and on the basis of the *WorkState* of the *MTMPWork*, it performs the actual work. A worker is alive as long as there is some work to process. As can be seen in Algorithm 2 *MTMPDispatcher* is responsible for managing the *MTMPWorker* and assigning the *MTMPWork* to them. It neatly divides the dataset into morsels and finally give them to the workers to perform the actual work when

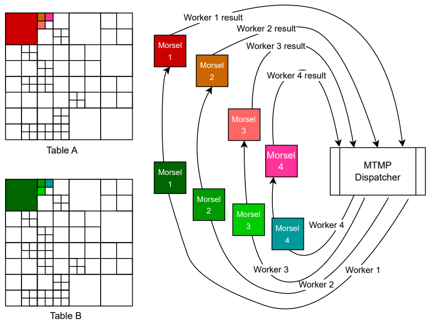


Fig. 6: Monolithic Tile-based Morsel Parallelism

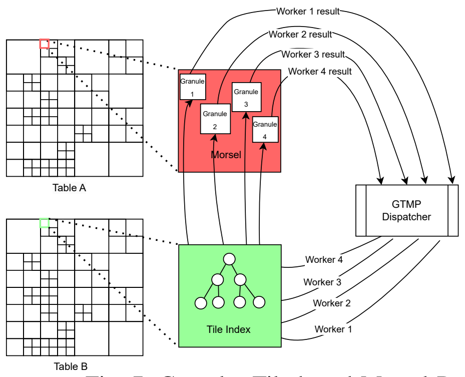


Fig. 7: Granular Tile-based Morsel Parallelism

they ask for. It also keeps track of the overall state of the actual processing we are doing and stores the final result. One key thing about this approach is that it does not need index to process *Spatial Join Operations* since the *MTMPWorker* knows which tile it is processing and it has access to the morsel in that particular tile from both of the tables. Fig. 7 shows how this works and Algorithm 2 shows how work is assigned to the workers. In Section 4 we will show how this approach works in different kinds of joins.

2) Granular Tile-based Morsel Parallelism (GTMP): This algorithm introduces the concept of a granule, which is the basic unit of processing. In MTMP, the basic unit of processing is a morsel. In GTMP, a morsel is composed of granules. The number of granules depends on the morsel size of the corresponding tile. So, the number of granules can be a dynamic entity, in the same way the morsel size can be dynamic in MTMP. One of the key differences between both the approaches is that GTMP uses index for different join operations. It needs index because a morsel (i.e. a tile) is divided into granules and each of these granules can be processed by different workers hence, we need a way to evaluate the granules within a morsel with those in the other table in the same morsel. For *Spatial Join*, GTMP uses tile-wise index, and for *Range Join* it uses a global index on the table because the query object could be present in some other tiles. It also has a similar *GTMPWork* component, which contains the tile id, granule and the *WorkState*. The worker here is called *GTMPWorker* and its functioning is a bit different from its counterpart in the MTMP, as it uses index (global or tile-wise Fig. 5). The dispatcher in GTMP is called *GTMPDispatcher* and it is responsible for handling the *GTMPWorker* and creating granules from each morsel and assigning them to the workers. Algorithm 3 shows how work is assigned to the workers and Figure 7 provides a holistic view of this approach. We show how this algorithm works with different kinds of join categories in a subsequent section.

3) Distributed MTMP/GTMP: Each node in the cluster has its own MTMP or GTMP Dispatcher and Worker. The Dispatcher spawns as many workers as the machine-dependent threads are supported in a node. The master node in the cluster runs the Data Assignment module (Figure 2), and is responsible for dividing the tiles equally and assigning it to all the nodes including itself. Once each of the nodes loads its needed data, it processes the data in morsel/granule-driven fashion independently. Finally, when the Dispatchers of all the nodes complete their tasks, the master node aggregates the results and produces the final output.

Algorithm 1: GTPMWorker/MTMPWorker

```
isAlive = true;
while isAlive do
    get next available Work from Dispatcher, by calling
      getWork() ;
    get WorkState from Work;
    if WorkState == build then
        noOp;
    else if WorkState == waitingBuild then
        noOp;
    else if WorkState == probe then
        Iterate the morsel/granule and do the processing according
          to the spatial predicate. Store result in global data
          structure using Dispatcher;
    else if WorkState == waitingProbe then
        Wait for all of the workers to finish the probing
    else
        isAlive = false
    end
end
```

Algorithm 2: Pseudo-Code for getWork() MTMPDispatcher

```
Input: dispatcherState, tileIdQueue
Output: MTMPWork
if dispatcherState == probingMorsels then
    if tileIdQueue.isEmpty() then
        dispatcherState = doneProbingMorsel;
        return MTMPWork with JobState::waitingProbe;
    end
    currentTileId, get current processing tile from
      tileIdQueue.front();
    probeMorsel, get morsel for currentTileId;
    assign probeMorsel to MTMPWork with
      JobState::probe and also include the
      currentTileId;
    return MTMPWork
else if dispatcherState == doneProbingMorsels then
    if all workers are done probing then
        return MTMPWork with JobState::done;
    else
        return MTMPWork with JobState::waitingProbe;
    end
else
    return MTMPWork with JobState::done;
end
```

Algorithm 3: Pseudo-Code for getWork() GTMPDispatcher

```
Input: dispatcherState, tileIdQueue, granuleSize,
       granuleStartIndex, granuleEndIndex
Output: GTMPWork
if dispatcherState == probingMorsels then
    if tileIdQueue.isEmpty() then
        dispatcherState = doneProbingMorsel;
        return GTMPWork with JobState::waitingProbe;
    end
    currentTileId, get current processing tile from
      tileIdQueue.front();
    probeMorsel, get morsel for currentTileId;
    granuleStartIndex = granuleEndIndex;
    granuleEndIndex = granuleStartIndex +
      granuleSize;
    if granuleEndIndex >= probeMorsel.length()
      then
        granuleEndIndex = probeMorsel.length();
        tileIdQueue.pop()
    end
    Slice a granule from probleMorsel using
      granuleStartIndex and granuleEndIndex and
      assign it to GTMPWork with JobState::probe and also
      include the currentTileId;
    if granuleEndIndex >= probeTuples.length()
      then
        granuleStartIndex = granuleEndIndex = 0;
    end
    return GTMPWork
else if dispatcherState == doneProbingMorsels then
    if all workers are done probing then
        return GTMPWork with JobState::done;
    else
        return GTMPWork with JobState::waitingProbe;
    end
else
    return GTMPWork with JobState::done;
end
```

IV. Query processing

In the following sections we discuss how Monolithic Tile-based Morsel-driven Parallelism (MTMP) and Granular Tile-based Morsel-driven Parallelism (GTMP) support a few different types of Joins. The other Join categories are omitted due to space constraints.

A. Spatial Join processing

Spatial Join finds object pairs from two tables which satisfy a spatial predicate like, `ST_INTERSECTS`, `ST_CROSSES` and others. In a non-indexed Join operation, there are two nested loops, where the loops are for iterating one of the two tables involved in the join. We compare the tuples from the outer (loop) table with each of the tuples in the inner (loop) table and then do spatial predicate evaluation (Refinement step) on both of the tuples. If it (i.e. tuple pairs) successfully passes the predicate evaluation, we add it to the final result. In the case of an index-based Join operation, we have a pre-built index on smaller table. The outer loop is to iterate the bigger table and then inside that loop we do index lookup using the MBR of the outer-loop tuple (Filter step). We iterate the objects returned by the index lookup and perform the Refinement to get the final output. Partitioning of dataset into tiles helps in the additional filtering of the tuples before the Refinement step in the sense that, we do not have to iterate the entire table. Instead, we just need to iterate the tuples belonging to the same tile because it is ensured that the MBR of the tuples inside a tile are close to each other, if not touching or overlapping. Essentially, partitioning the dataset into tiles, helps in reducing the search space significantly, without using an index.

In case of MTMP, since each tile corresponds to a morsel, we just need to perform the Refinement step on the morsels from both of the tables belonging to the same tile. The Filter step is already done by using the tiles, and so we do not need to use index in this. We can directly go to the Refinement step. *MTMPWorker* receives a tiled id, and its associated morsels from both the tables and then it uses the non-indexed nested loop join approach to do the final predicate evaluation (Refinement). The main advantage of this algorithm is that, it achieves faster Join processing without using index.

In GTMP, morsels are divided into granules and these granules are then assigned to the *GTMPWorker*. So, a *GTMPWorker* worker could be processing a chunk of tuples (granules) in a tile (morsel), while another *GTMPWorker* could be processing a different chunk of tuples (granule) of the same tile (morsel). Hence, if we do predicate evaluation on the granules of both the tables then we will get incomplete/partial results, as we are not comparing against the entire morsel (tile). So, to get complete results, we pre-built Tile-wise index and then perform the Filter step with the granules using the index. After the Filter step, we use the result of index lookup and do the final predicate evaluation between granules and the index query result to get the final output.

B. Spatial Range Join processing

Spatial Range Join finds object pair where the objects are within a defined radius of the other object (query object) from the other table. `ST_DIWITHIN(geomA, geomB, 1)` can be used to implement queries involving radius. This spatial predicate checks if geomA is within 1 unit distance of geomB. We can use the global index in the Filter stage, and can make our query execute faster by eliminating a lot of tuples. We can not use tile-based index here because our query object and the object from the table could be outside of the tile.

In MTMP, we create morsels from the query object table and we use the pre-built global index on the other table. We then create a buffer object on the query object using the `ST_BUFFER` and the radius mentioned in the `ST_DIWITHIN`, and then use the buffered object to query the global index. The objects returned by the index is then sent through the actual predicate evaluation to get the final result. By using the global index, we can effectively filter a lot tuples from the other table.

GTMP is very similar to MTMP in this join operation. It also uses the pre-built global index and creates the granules from the query object morsels. It uses the same approach of creating buffer objects and then querying the buffer objects against the index to filter out tuples. Finally, the tuples from the index and the query granules are sent to predicate evaluation to produce the final output.

V. Experimental evaluation

In this section, we describe the experimental setup and datasets we used and the types of queries we ran to evaluate our system. Finally, we compare our system's performance with Apache Sedona (previously, GeoSpark) and Citus [20].

A. Experimental Setup

We used a cluster of 6 machines, each having a Dual-Core AMD Opteron 2222 Processor clocked at 3Ghz and 16 GB main memory. We use JDK 11 for our code generator and use UPC++ 2023.9.0 to run the distributed code. We compile generated UPC++ code using g++ version 9.4.0.

B. Datasets

We use two different datasets for our experiments, namely, TIGER [17] California and OpenStreetMap [18] (OSM). Table I and Table II provides more details about these datasets.

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

TABLE I: TIGER Dataset

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

TABLE II: OSM Dataset for UK

C. Queries

We evaluated a set of Join queries for each of the datasets, which involves spatial predicates, such as *ST_INTERSECTS*, *ST_TOUCHES*, *ST_CROSSES*, *ST_OVERLAPS* and *ST_WITHIN*. Queries for both the datasets are listed in Table III and Table IV.

Name	Query
TIGER_Q1	select arealm.id, areawater.id from arealm join areawater on ST_TOUCHES(arealm.geom, areawater.geom)
TIGER_Q2	select edges.id, arealm.id from edges join arealm on ST_CROSSES(edges.geom, arealm.geom)
TIGER_Q3	select edges.id, edges2.id from edges join edges as edges2 on ST_CROSSES(edges.geom, edges2.geom)
TIGER_Q4	select edges.id, areawater.id from edges join areawater on ST_CROSSES(edges.geom, areawater.geom)
TIGER_Q5	select areawater1.id, areawater2.id from areawater1 join areawater as areawater2 on ST_OVERLAPS(areawater1.geom, areawater2.geom)
TIGER_Q6	select pointlm.id, areawater.id from pointlm join areawater on ST_WITHIN(pointlm.geom, areawater.geom)
TIGER_Q7	select * from pointlm join areawater on ST_DWITHIN(pointlm.geom, areawater.geom, 1)
TIGER_Q8	select * from pointlm join areawater on ST_DISTANCE(pointlm.geom, areawater.geom) >= 1

TABLE III: TIGER dataset queries

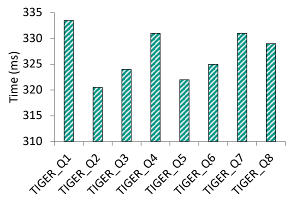


Fig. 8: Code generation time for TIGER

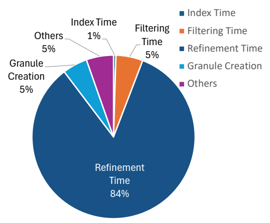


Fig. 9: Execution time breakdown for TIGER_Q3

D. Experiments

CasaDB generates UPC++ code for each of the queries, and so we measure the code generation time. Finally, we measure the execution time on 2, 4 and 6 node clusters. We then compare our system with Apache Sedona and Citus.

1) Code Generation: Code generation times for TIGER queries are listed in Figure 8. As can be seen, the code generation times are not significant compared to the actual spatial query execution times (as can be seen in the next few sections).

2) Execution time breakdown: Figure 9 shows the query execution time breakdown for TIGER_Q3 with GTMP algorithm. The refinement phase takes up 84% of the execution time, while filtering phase took only 5% of the time. Index creation took only 1%, and creation of granules for GTMP took 5% of the time. As expected the refinement phase is the most dominant phase in spatial query processing.

3) Partitioning granularity analysis: We conducted experiments involving tiles granularity. Figure 10 presents the execution time of TIGER queries on 2, 4 and 6 node clusters with 512, 1024, 2048 and 4096 tile sizes. For long-running queries (TIGER_Q2, TIGER_Q3 and TIGER_Q4), the setting of 4096 tiles performs better than all the others and scales well with increasing the nodes in the cluster, however, it is interesting to note that short running queries, such as TIGER_Q1, TIGER_Q6, the setting of 1024 tiles performs better across all nodes. This is expected because the short running queries usually process relatively fewer tuples. In case of higher tile granularity, the cost of iterating additional tiles incurs more overhead than the actual processing of tiles.

4) Performance analysis of GTMP and MTMP: We evaluate the performance of GTMP and MTMP by running the TIGER queries listed in Table III on 2, 4 and 6 nodes cluster and measured the execution time of all queries. As we found in the previous section 4096 partition granularity works best for most algorithms, so for these experiments we used 4096 tiles setting. For the majority of the queries both, GTMP and MTMP have almost similar execution times apart from the queries involving the bigger tables (Edges) as shown in Fig. 10. For TIGER_Q3, GTMP is roughly 2x faster than MTMP. This is because of the high degree of data skew in the tables (Edges) involved in this query. While MTMP's Dispatcher sometimes provides large morsels to its workers, GTMP's Dispatcher breaks the large morsels into granules and gives them to its workers. As a result, GTMP's workers complete their work more quickly and then move on to their

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

TABLE IV: OSM dataset queries

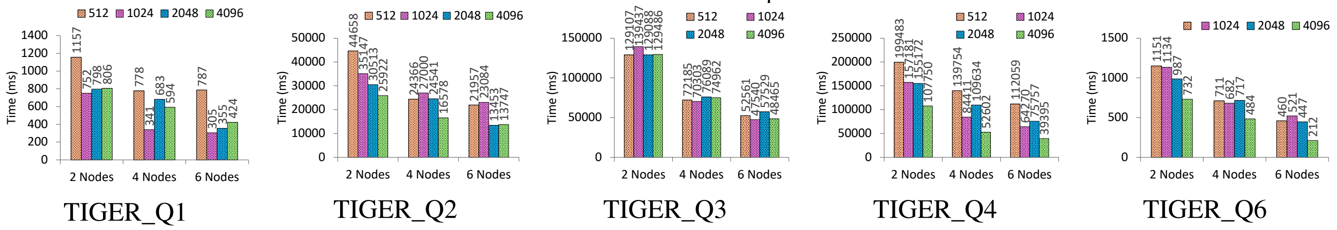


Fig. 10: GTMP performance with different partition granularity for TIGER dataset

next work. GTMP shows much more resilience with data skew than MTMP. GTMP outperforms MTMP in range join and distance join queries, it is 2.4x to 2.6x faster for TIGER_Q7 where global index is used, and it is upto 5.6x for TIGER_Q8 without global or tile-wise index.

5) Scalability of our system: Figure 12 shows the scalability of CasaDB using our best configuration, with 4096 tiles configuration and GTMP algorithm, on the cluster. For long running queries like TIGER_Q3, our systems performs 1.7x to 2.6x better on 4 and 6 nodes respectively when compared to 2 nodes, and also for TIGER_Q4, it scales well with 2x to 2.7x performance boost on the cluster. Short running queries like TIGER_Q1 also scales 1.4x to 1.9x on 4 and 6 nodes, and TIGER_Q6 scales 1.5x to 3.45x. Overall, the system shows significant scalability on increasing cluster size.

6) Comparison with Apache Sedona and Citus: We evaluate CasaDB against Apache Sedona and Citus, Fig. 13 shows execution times for TIGER queries running on 6-nodes cluster. CasaDB is significantly better than Citus DB and outperforms Apache Sedona in all of the queries in all settings. CasaDB's fastest running query on TIGER dataset was TIGER_Q6 and it took 212 ms to finish, while the same query took 414 ms to execute on Citus and roughly 32 seconds on Sedona. CasaDB's longest-running TIGER dataset query was TIGER_Q3, and it took 48.4 seconds to execute, while on Citus it took 68.4 seconds to execute and Sedona took roughly 4 mins. CasaDB is 71x faster than Sedona and 2.1x faster than Citus for TIGER_Q1. It is 151x faster for TIGER_Q6 than Sedona and roughly 2x faster than Citus. Overall, CasaDB scales well compared to Sedona and Citus and Spatial Joins are 2x to 151x faster on TIGER dataset.

We also evaluated the performance of CasaDB on OSM dataset on 6 nodes. Fig. 14 shows execution times for OSM queries running on 6-nodes cluster. Longest-running query, OSM_Q1 took roughly 19 minutes to execute on CasaDB, Citus took 28.4 minutes and Sedona took roughly 11.5 hours to execute, OSM_Q4 took 17 minutes to finish on CasaDB, while Citus took 26.7 minutes to finish and Sedona took a massive 17.6 hours to execute it. The short running query, OSM_Q3 performed best on Citus with just 10.3 seconds, while CasaDB took 1.8 minutes to finish and Sedona finished in 19 minutes. Overall, CasaDB is at least 10x to 62x faster than Sedona and atmost 1.56x times faster than Citus DB on OSM dataset.

VI. Related Work

The topic of scalable processing of spatial data has a long history. Paradise [26] is one of the earliest parallel databases developed for scalable, geo-spatial data storage and retrieval. Paradise is based on an object-relational data model that supports an extended version of SQL for spatial queries. The PostGIS spatial extension of PostgreSQL played a role in popularizing spatial SQL query processing.

The advent of MapReduce and an open-source implementation Hadoop [27], resulted in researchers introducing Hadoop-based spatial data systems, including SpatialHadoop [6] and Hadoop-GIS [5]. To take advantage of the aggregate memory pool of a compute cluster, Spark [28] has been introduced, which offers significantly better performance than Hadoop. Subsequently, several Spark-based spatial systems were proposed. They include Apache Sedona [1] (previously, GeoSpark), SpatialSpark [2], Simba [3], STARK [29], and LocationSpark [4]. As noted by Yu et al. [7], among the Spark-based systems only Sedona supports standard spatial extensions to SQL [30]. Another limitation of some of these systems is that they support only point objects or MBR-based filter oriented spatial query processing. Similar findings were reported by a study [19].

The focus of the previous systems are long-running spatial queries involving complex operations such as spatial join. A few spatial data systems has been proposed that aim at frequent updates and relatively short-running queries on points, such as spatial range and kNN queries. MD-HBase [31] is on of the earliest systems in this category, followed by systems such as DISTIL$^+$ [32]. These systems do not support SQL.

VII. Conclusion

In this paper, we have presented a compilation-based distributed spatial query processing system, which introduces morsel-driven parallelism in distributed data systems. We have also extended the original morsel-driven parallelism approach for spatial query processing and have introduced Monolithic Tile-based Morsel-driven Parallelism (MTMP) and Granular Tile-based Morsel Parallelism (GTMP) for different types of spatial joins. Through our experiments, we determined which

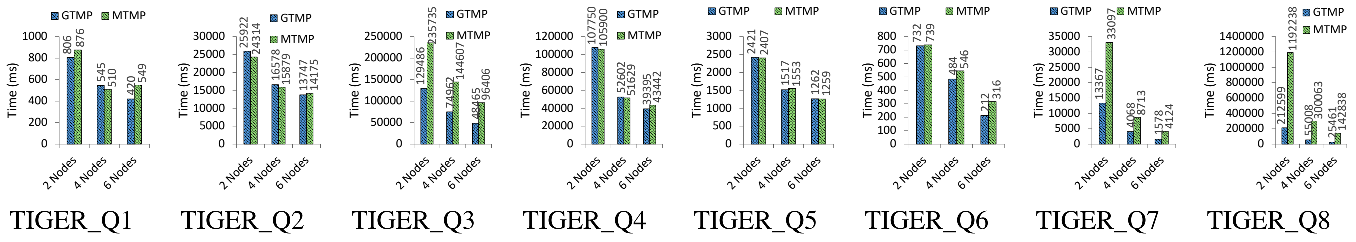


Fig. 11: GTMP vs. MTMP for TIGER queries

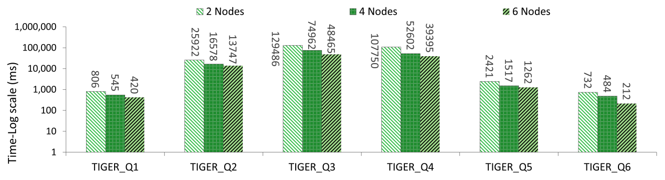


Fig. 12: CasaDB on a cluster with 2, 4, 6 nodes with TIGER dataset

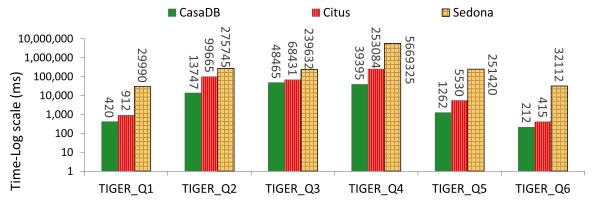


Fig. 13: CasaDB vs. Citus and Sedona on 6 nodes with TIGER dataset

tile-partitioning scheme works best for MTMP and GTMP. We showed how GTMP achieves better performance due to its skew resilience. We compared our system with Apache Sedona and Citus-Distributed Postgres. Experimental results suggest that our systems is 71x to 151x faster than Apache Sedona on TIGER dataset and 10x to 62x faster on OSM UK datasets on 6 nodes cluster. It is 1.5x to 2x faster than Citus on OSM and TIGER datasets.

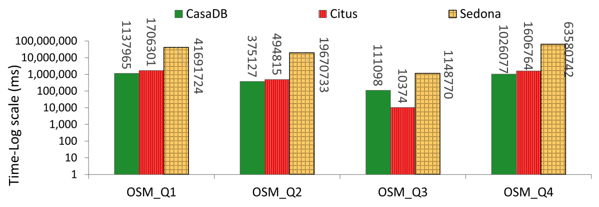


Fig. 14: CasaDB vs. Citus and Sedona on 6 nodes with OSM dataset

References

[1] "Apache Sedona," https://sedona.apache.org/.

[2] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *ICDEW*, 2015.

[3] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016.

[4] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "LocationSpark: A distributed in-memory data management system for big spatial data," *VLDBJ*, vol. 9, no. 13, pp. 1565–1568, 2016.

[5] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over Mapreduce," *PVLDB*, pp. 1009–1020, 2013.

[6] A. Eldaway and M. F. Mokbel, "SpatialHadoop: A mapreduce framework for spatial data," in *ICDE*, 2015.

[7] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: the geospark perspective and beyond," *Geoinformatica*, vol. 23, no. 1, p. 37–78, jan 2019.

[8] A. Eldawy, M. Elganainy, A. Bakeer, A. Abdelmotaleb, and M. Mokbel, "Sphinx: distributed execution of interactive SQL queries on big spatial data," ser. SIGSPATIAL, 2015.

[9] R. D. Caballar, "The Rise of SQL It's become the second programming language everyone needs to know," *IEEE Spectrum*, 2022.

[10] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993, pp. 209–218.

[11] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *PVLDB*, vol. 4, no. 9, pp. 539–550, 2011.

[12] D. Datta, M. Stoodley, and S. Ray, "Towards just-in-time compilation of SQL queries with OMR JitBuilder," in *CASCON*, 2021, p. 256–261.

[13] R. Y. Tahboub and T. Rompf, "Architecting a query compiler for spatial workloads," in *SIGMOD*, 2020, p. 2103–2118.

[14] S. Chatterjee, S. Sharma, N. Ivan, S. Verma, S. Ray, M. Stoodley, C. Zuzarte, and I. Finlay, "Compilation of sql queries for efficient distributed in-memory processing," in *CASCON*, 2023, p. 149–154.

[15] G. Almasi, "PGAS (Partitioned Global Address Space) Languages." 2011.

[16] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014, p. 743–754.

[17] http://www.census.gov/geo/www/tiger.

[18] "Openstreetmap," https://www.openstreetmap.org/.

[19] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1661–1673, jul 2018.

[20] "Citus Database," 2024. [Online]. Available: https://www.citusdata.com/

[21] D. Bonachea and J. Jeong, "Gasnet: A portable high-performance communication layer for global address-space languages," *CS258 Parallel Computer Architecture Project, Spring*, 2002.

[22] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *IPDPS*, 2014, pp. 1105–1114.

[23] "UPC++ Wiki," https://bitbucket.org/berkeleylab/upcxx/wiki/Home, 2023, accessed: 2023-03-30.

[24] Begoli, Edmon and Camacho-Rodríguez, Jesús and Hyde, Julian and Mior, Michael J. and Lemire, Daniel, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *SIGMOD*, 2018, p. 221–230.

[25] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*, 2011, pp. 195–206.

[26] J. Patel *et al.*, "Building a scalable Geo-Spatial DBMS: technology, implementation, and evaluation," in *SIGMOD*, 1997, pp. 336–347.

[27] "Apache Hadoop," http://hadoop.apache.org/.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[29] S. Hagedorn, P. Goetze, and K.-U. Sattler, "The STARK Framework for Spatio-Temporal Data Analytics on Spark," *Datenbanksysteme fur Business, Technologie und Web (BTW)*, 2017.

[30] M. Ashworth, "Information technology – database languages – sql multimedia and application packages – part 3: Spatial, standard," *International organization for standardization*, 2016.

[31] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," in *MDM*, 2011.

[32] P. Memarzia, M. Patrou, M. M. Alam, S. Ray, V. C. Bhavsar, and K. B. Kent, "Toward efficient processing of spatio-temporal workloads in a distributed in-memory system," in *MDM*, 2019, pp. 118–127.