Architecting a Query Compiler for Spatial Workloads

Ruby Y. Tahboub Tiark Rompf
Purdue University, West Lafayette, Indiana
{rtahboub,tiark}@purdue.edu

ABSTRACT

Modern location-based applications rely extensively on the efficient processing of spatial data and queries. Spatial query engines are commonly engineered as an extension to a relational database or a cluster-computing framework. Large parts of the spatial processing runtime is spent on evaluating spatial predicates and traversing spatial indexing structures. Typical high-level implementations of these spatial structures incur significant interpretive overhead, which increases latency and lowers throughput. A promising idea to improve the performance of spatial workloads is to leverage native code generation techniques that have become popular in relational query engines. However, architecting a spatial query compiler is challenging since spatial processing has fundamentally different execution characteristics from relational workloads in terms of data dimensionality, indexing structures, and predicate evaluation.

In this paper, we discuss the underlying reasons why standard query compilation techniques are not fully effective when applied to spatial workloads, and we demonstrate how a particular style of query compilation based on techniques borrowed from partial evaluation and generative programming manages to avoid most of these difficulties by extending the scope of custom code generation into the data structures layer. We extend the LB2 main-memory query compiler, a relational engine developed in this style, with spatial data types, predicates, indexing structures, and operators. We show that the spatial extension matches the performance of specialized library code and outperforms relational and map-reduce extensions.

CCS CONCEPTS

• **Information systems** → *Database management system engines*; • **Software and its engineering** → *Domain specific languages*;

KEYWORDS

Query Compilation; Staging; Code Generation; Spatial Query Processing; Futamura Projections; Extensible Compilers

ACM Reference Format:
Ruby Y. Tahboub Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389701

1 INTRODUCTION

The proliferation of location-enabled devices and the popularity of emerging Internet of Things (IoT) applications have resulted in large amounts of spatial data that need to be efficiently processed. Spatial data types, e.g., geometric, geographic, etc., lie in a two or higher-dimensional space, meaning that sorting and hashing techniques typically implemented in relational query engines are not directly applicable. As a consequence, additional indexing structures, e.g., R-tree, k-d tree, etc. are needed to access spatial data efficiently.

Moving beyond relational processing, spatial engines often extend existing relational query engines, e.g., PostGIS [9], Oracle Spatial [8], etc. or cluster computing frameworks, e.g., GeoSpark [72], Simba [70], etc. with spatial data types, predicates, indexing structures, and operators as illustrated in Figure 1. The key advantage of extending an existing data management back-end is leveraging the engineering effort that went into building sophisticated management layers for memory, storage, and query evaluation.

The performance of spatial workloads is primarily affected by data access and spatial predicate evaluation. Time spent in evaluating spatial predicates makes up a large part of spatial processing runtime. For instance, the spatial range join query shown in Figure 1 performs the `ST_Contains` predicate on pairs of data records when testing for spatial containment. For convenience, spatial engines rely on external libraries, e.g., JTS [5], Geos [3], etc. for evaluating spatial predicates. However, these high-level libraries are opaque to the query

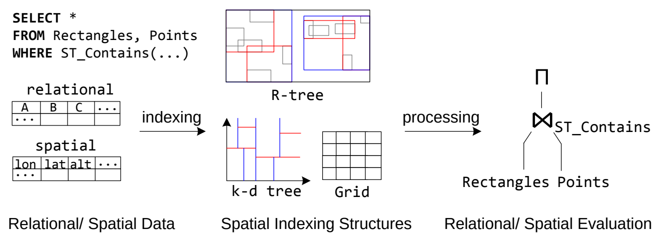


Figure 1: Spatial processing overview: a relational back-end is extended with spatial indexing structures and evaluation.

evaluator which incurs runtime overhead in expensive function calls, and also prevents further code optimizations that target both the predicate and query plan implementation, e.g., inlining, loop fusion, etc.

For the case of *relational spatial processing*, the spatial indexing structures are typically implemented in a generic form to support instantiating various types of structures behind a common interface. For instance, the SP-GiST [11] indexing framework in PostgreSQL/ PostGIS is used to instantiate a quad-tree, k-d tree, or trie. Processing data in generic structures is expensive due to function calls (usually virtual) needed to resolve which data structure to use. Similarly, the performance of *spatial Spark extensions* is suboptimal, in general, since Spark is internally designed for distributed shared-nothing environments. As a consequence, spatial datasets need to be explicitly partitioned to avoid scanning the entire data under Spark's distributed execution model. Data partitioning and two-level indexing are expensive tasks especially when performed inside Spark.

In summary, both spatial relational and spatial Spark extensions incur substantial runtime overhead. *What can be done to speed up spatial computations?* It is natural to look at query compilation – the translation of high-level queries to native code – which has seen a renaissance in relational query engines. However, applying query compilation techniques *effectively* to spatial workloads is far from straightforward.

Choosing a Query Compilation Approach. Spatial processing cost is rather dominated by evaluating *coarse-grained* spatial predicates that are individually computationally expensive and accessing complex spatial data structures that index diverse types of spatial data, e.g., points, rectangles, polygons, etc. Architecting a spatial query compiler needs to go beyond removing the interpretive overhead from query evaluation and focus on evaluating complex spatial predicates and generating efficient spatial indexing structures while taking into consideration the cost of code generation and compilation. The question then is: *how to choose a query compilation approach for spatial workloads?*

Low-level query compilation implemented in HyPer, pre-implements data structures in a high-level language such as C++ and encodes the query evaluation in LLVM assembly to minimize the base cost of code generation and compilation that takes place at runtime. However, building a spatial extension with spatial indexing structures layer using this style of code generation is challenging due the low-level implementation that permeates large parts of the engine code [64]. Furthermore, taking an in-depth look at HyPer's code generation adopted in systems like Spark [28] already reveals that pre-implementing data structures in a language other than C++, especially Java for interoperability with common cluster computing frameworks, results in generating suboptimal code due to various issues including JVM overhead, the specialization level of data structures and query evaluation did not entirely remove interpretive overhead associated with processing Spark distributed plan [28].

On the same vein, adopting DBLAB's approach for compiling spatial queries would require adding new transformation passes tailored towards high-dimensional spatial indexing structures since DBLAB's compiler is engineered to compile linear data structures, e.g., hash tables to specialized arrays.

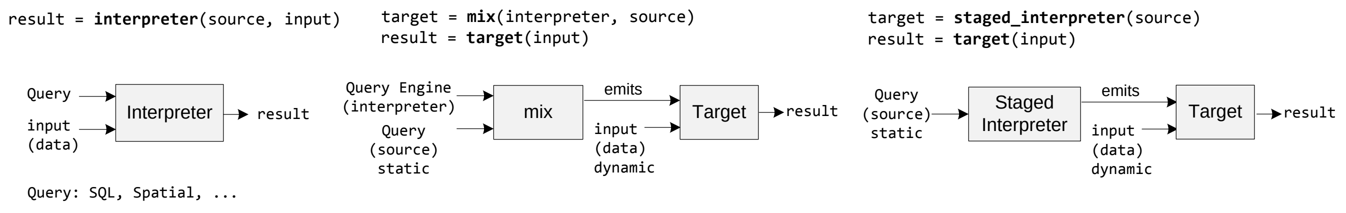
Therefore, we believe the compilation approach based on programmatic specialization presented in the work of the LB2 compiler [64] (Section 2.1) is the most suitable among others as it (i) enables generating specialized data structures while implementing the query engine in a high-level language, i.e., an interpreter-based implementation similar to existing spatial query engines (ii) allows pregenerating as much or as little data structures code, targeting short and long-running queries within the same query compiler (Section 2.2).

The key idea to address the previous challenges is to specialize spatial predicate implementation (Section 3.2) in addition to facilitating building optimized data structures (Section 3.3) using *programmatic specialization*, the same technique LB2 already uses for generating efficient code for relational queries. We thus extend the LB2 [64] main-memory query compiler with spatial compilation[1]; in particular, spatial predicates, indexing structures, and spatial operators.

This paper makes the following contributions:

- We discuss the unique challenges of compiling spatial workloads and build a spatial extension to the LB2 query compiler (Section 2).
- We describe staging spatial predicates, compiling optimized spatial indexing structures, supporting parallelism and applications in LB2-Spatial (Section 3).
- We compare the performance of the spatial extension on small and large datasets with low-level libraries code, spatial RDBMS PostGIS, and two Spark spatial

[1]Compiling relational queries is extensively discussed in [39, 40, 45, 57, 64]. In this work, we give a general overview about relational queries but primarily focus on compiling spatial workloads.



```
result = interpreter(source, input)
```

```
target = mix(interpreter, source)
result = target(input)
```

```
target = staged_interpreter(source)
result = target(input)
```

(a) Query Interpreter (b) The first Futamura projection (c) The practical realization of the First Futamura Projection without mix

Figure 2: Specializing query engines using the practical realization of the first Futamura Projection [64].

extension Simba and GeoSpark. The experiments validate that the performance of low-level libraries can be achieved in LB2-Spatial extension (Section 4).

Finally, we review related work in Section 5 and Section 6 concludes the paper.

2 QUERY COMPILATION OVERVIEW

In this Section, we review the generative programming code generation approach used in LB2-Spatial and discuss the compilation overhead in LB2-Spatial and recent state-of-the-art query engines.

2.1 Generative Query Compilation

Most query engines are interpreters that evaluate a query plan with respect to input data and produce results as illustrated in Figure 2a.

```
result = interpreter(source, input)
```

The fundamental idea of compiling SQL to low-level code lies in removing the interpretive overhead associated with query evaluation. In other words, applying a form of partial evaluation to remove the dispatch which the interpreter performs on the structure of the query.

The first Futamra projection [32], illustrated in Figure 2b, states that specializing an interpreter (e.g., query engine) with respect to a static input (e.g., query) is identical to a query compiler.

```
target = mix(interpreter, source)
result = target(input)
```

As discussed in the literature [23], fully automatic partial evaluation ("mix" in Figure 2b) is largely intractable due to the difficulty of binding time separation [38] (i.e., deciding which expressions to specialize and generate in the residual code). Hence, the programmatic specialization discussed in [64] delegates staging the query interpreter to programmers and generative programming frameworks as illustrated in Figure 2c.

```
target = staged_interpreter(source)
result = target(input)
```

The programmatic specialization approach for writing self-specializing code is summarized as follows. First, a programmer identifies the parameters that need to be specialized or

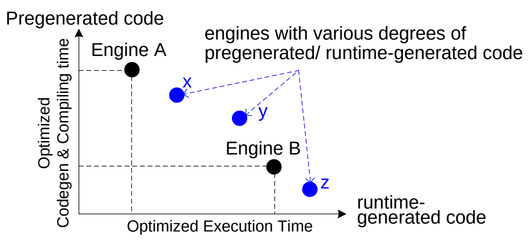


Figure 3: The trade-off between optimizing execution time and compilation overhead in exiting query compilers.

generated in the residual code. Second, a code generation framework, e.g., the Lightweight Modular Staging (LMS) is used to emit evaluation code for residual expressions. LMS is a library-based generative programming and compiler framework that uses generative programming abstractions, operator overloading and other features in regular general-purpose languages to generate code. The following example shows specializing the power function for a fixed value of n using LMS.

```
def power(x: Rep[Int], n:Int): Int =
  if (n == 0) 1  else x * power(x, n - 1)
```

The parameter x is annotated using the Rep constructor to denote that the staged expression x is a symbolic integer type with operations overloaded to emit code. The generated code for power(x,4) is as follows.

```
int x0 = x * 1; int x1 = x * x0;
int x2 = x * x1; int x3 = x * x2;
```

Figure 2c shows the practical realization of the first Futamura projection. The query engine (i.e., interpreter) is staged with Rep annotations and specialized with respect to a given SQL query. The compiled target is used to evaluate data at runtime and produce the query result.

2.2 Compilation Overhead

Code generation and compilation incur additional overhead at runtime, typically hundreds of milliseconds to a second, in high-level query compilers. For long-running queries, the compilation overhead is negligible to the overall execution time. However, for shorter queries, a pre-compilation strategy, e.g., using pregenerated data structures and individual

```
1  class Agg(op: Op)(grp: GrpFun)(init: Record)(agg: AggFun) extends Op {
2   def exec(cb: Record ⇒ Unit) = {
3     val hm = new HashMap()
4     op.exec { tuple ⇒
5       val key = grp(tuple)
6       hm.update(key, init) {
7         curr ⇒ agg(curr, tuple)
8     } }
9     for (tuple <- hm)
10       cb(tuple)
11  } }
```

(a)

```
1 class HashMap() {
2   val name = freshName()
3   emit(s"""val $name = new HashMap()""")
4   def apply(key: Key) = s"""$name($key)"""
5   def update(key: Key, v: Value)(up: Value ⇒ Value) =
6     emit(s"""$name($key) = $up($name.getOrElse($key,$v))""")
7 }
```

(b)

```
1 abstract class HashMap(kSch: Seq[Field], vSch: Seq[Field]) {
2  def update(key: Record, init: Record)(up: Record ⇒ Record): Unit
3  def foreach(f: Record ⇒ Unit): Unit
4 }
5 class LB2HashMap(kSch: Seq[Field], vSch: Seq[Field]) extends HashMap {
6  val size = defaultSize
7  val agg = new ColumnarBuffer(vSch, size)
8  val keys = new ColumnarBuffer(kSch, size)
9  val used = new Array[Int](size)
10  var next = 0
11  def update(k: Record, init: Record)(up: Record ⇒ Record) = {
12    val idx = defaultHash(k)
13    if (isEmpty(keys(idx))) {
14      used(next) = idx; next += 1
15      keys(idx) = k; agg(idx) = up(init)
16    } else agg(idx) = up(agg(idx))
17  }
18  def foreach(f: Record ⇒ Unit) = {
19    for (idx <- 0 until next) {
20      val j = used(idx)
21      f(merge(keys(j), agg(j)))
22 } } }
```

(c)

Figure 4: (a) The aggregate operator in LB2 that uses (b) an abstraction to generate a hash map using a pre-compiled data structure or (c) generates a fully-specialized hash map implementation in-place.

operators, may be a beneficial trade-off for minimizing compilation overhead, and hence the end-to-end execution time.

Figure 3 demonstrates how code generation approaches implemented in recent state-of-the-art query engines address compilation overhead. For instance, Engine *A* (similar to Hyper [45]), generates code that links with pre-compiled data structures. Although the generated code is not fully specialized, the compiling overhead is significantly reduced. On the other hand, Engine *B* (similar to Legobase [39] and DBLAB [57]), is engineered towards producing highly optimized code, i.e., with all function calls inlined, data structures specialized, etc. at the expense of compilation time.

The generative query compilation approach adopted in this work is inherently flexible in the amount of pregenerated and runtime-generated code. In other words, the spatial extension does *not* need to pick one extreme or the other but can pick any point on the spectrum, i.e., compile queries that link with pre-compiled indexing structures or generate all the query code at runtime. However, for long-running spatial join queries data structures specialization is more critical than relational queries. Hence, the performance profile of Engine *Z* is more desirable in such cases.

2.3 The LB2 Query Compiler

LB2 is a high-level relational query compiler that uses LMS with guidance from the first Futamura projection [33] to compile queries into optimized low-level code.

The code in Figure 4a shows the `Aggregate` operator that uses an abstraction (Line 3) to generate a hash map data structure. The `HashMap` class illustrated in Figure 4b generates only the hash map object and method calls. The hash map implementation is a linked code that is either pregenerated or an existing library (similar to Engine A in Figure 3). Furthermore, LB2 provides another hash map class, shown in 4c, that uses buffer abstractions to generate specialized hash map code [64] (similar to Engines *B* and *Z* in Figure 3).

3 ARCHITECTING A SPATIAL QUERY COMPILER

Over the previous decade, spatial query engines have devoted a tremendous effort to support large workloads by building spatial indexes [49], applying adaptive query processing techniques [16], exploiting advances in distributed and parallel computing, etc. We add to the ongoing effort and compile spatial queries to low-level code as in pioneering relational databases. Our goal is to architect a spatial query compiler, in a high-level language, by extending the LB2 query compiler with spatial data types, spatial predicates, spatial indexing structures, and spatial operators.

3.1 LB2-Spatial Overview

The LB2-Spatial extension compiles spatial queries into optimized native code. Figure 5 shows a high-level architecture of LB2-Spatial. The front-end accepts SQL queries, spatial queries (adopting the syntax in [9, 70]), and a domain-specific API for spatial operations in `Scala`. The back-end is extended with spatial predicates, indexing structures (R-tree, k-d tree and 2D grid) and the following predicates and join operators: range, distance, and *k*NN (each operator is implemented with an index, without using an index, serial and parallel).

Finally, LB2-Spatial adds two types of spatial indexing structures to support both short and long-running queries. Fully specialized indexing structures implemented using

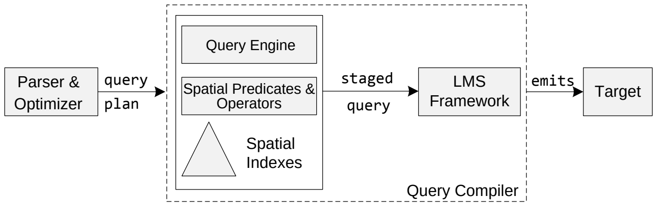


Figure 5: Extending LB2 with spatial processing[2].

LB2's high-level abstractions, e.g., various array buffers that generate low-level code, and pre-compiled indexing structures that are linked as external libraries.

As discussed earlier (in Section 2 and [64]), evaluating a query plan with respect to a staged query evaluator (using LMS `Rep` annotations) produces a staged query. The LMS framework builds an intermediate representation (IR) graph that encodes high-level constructs and operations. The result of executing the graph is a source program (generated in `Scala` or `C`) that implements the query evaluation without the interpretive overhead of processing static input (i.e., query pipeline). Finally, the generated code is compiled into a target and executed.

Front-end and Query Optimization. Spatial Spark extensions, e.g., Simba and GeoSpark, provide a SQL front-end as traditional RDBMS, and a programmatic front-end that offers a spatial processing API within a programming language, e.g., `Scala`, `Python`, etc. LB2-Spatial follows the same approach and provides SQL and programmatic front-ends.

Query Example. Throughout the remainder of this Section, we are going to use the point-rectangle range join query (using `ST_Contains` predicate) illustrated in Figure 6a (expressed in SQL and spatial API) as a running example for compiling spatial queries. Given two tables, `Rectangles` and `Points`, the point-rectangle range join query finds the points located inside the rectangle areas.

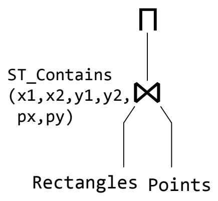
Figure 6b shows the straightforward implementation of point-rectangle range join using nested loops (in the data-centric model with callbacks). Query evaluation starts with `Print` operator calling the `exec` method of its child `NestedLoopRangeJoinOp`. The `NestedLoopRangeJoinOp` in turn executes its own `exec` method (Lines 6-19) where it calls the `exec` methods of `Scan(Rectangles)` and `Scan(Points)` operators (i.e., `left.exec` and `right.exec`). At a closer look, Lines 7-10 show the actions that are performed on the `left` operator where the records obtained from the `Rectangles` table are inserted into a buffer to prepare for the join operation. Moreover, Lines 11-18 encode the join operation where a pair of rectangle and point values are extracted in Lines 13-16

[2]The spatial extension is the realization of a vision presented in [65] where we describe supporting compilation in spatial query engines.

```
// Rectangles(rid: Int, x1: Double, x2: Double, y1: Double, y2: Double)
// Points (pid: Int, px: Double, py: Double)

SELECT *
FROM Rectangles, Points
WHERE ST_Contains(x1,x2,y1,y2,px,py)

Print(
 NestedLoopRangeJoinOp(
  Scan(Rectangles), Scan(Points))
   (x ⇒ x("x1"))(x ⇒ x("x2"))
   (x ⇒ x("y1"))(x ⇒ x("y2"))
   (x ⇒ x("px"))(x ⇒ x("py")))
```



(a)

```
1  class NestedLoopRangeJoinOp(left: Op, right: Op)
2  (x1Fun: key1, x2Fun: key2, ..., pxFun: keyx, pxFun: keyy)
3  extends Op {
4    val len = var_new(0L)
5    val buffer = new ColumnarBuffer(schema, defaultSize)
6    def exec(cb: Record ⇒ Unit) = {
7      left.exec { rec ⇒
8        buffer(len) = rec
9        len = len + 1
10      }
11     right.exec { rec ⇒
12       for ( i<-0 until len) {
13         val x1 = key1(buffer(i))
14       // val x2 ... val y1 ... val y2 ...
15         val px = keyx(rec)
16         val py = keyy(rec)
17         if(ST_Contains(x1,x2,y1,y2,px,py))
18           cb(merge(buffer(i),rec))
19  }}}}
```

(b)

Figure 6: (a) Point-rectangle range join query in SQL and spatial API (b) the implementation of NestedLoopRangeJoinOp in LB2-Spatial.

from the corresponding records, and the spatial predicate `ST_Contains` is evaluated. Finally, the statement in Line 18 is essential as it invokes the callback of the caller's `exec` method to stream the joined records to the parent operator `Print`.

Next, in Sections 3.2-3.5, we discuss the elements of compiling spatial queries: staging spatial predicates, specializing indexing structures, supporting parallelism in shared-memory and spatial applications.

3.2 Staging Spatial Predicates

Spatial predicates encode the spatial relation between spatial types, e.g., overlap, containment, etc. Spatial query engines extensively use spatial predicates (usually provided by external libraries) to implement various spatial operations, e.g., nearest neighbors, ranking, etc. For convenience, external libraries, e.g., JTS [5] in `Java`, Geos [3] in `C++`, etc. are used for spatial predicates' evaluation. However, using libraries for evaluating spatial predicate would result in generating suboptimal code. First, the generic library interface adds

```
// ST_Contains predicate
def ST_Contains(x1: Rep[Int],
  x2: Rep[Int], y1: Rep[Int],
  y2:Rep[Int],  x: Rep[Int],
  y: Rep[Int]): Rep[Boolean]= {

  ((x1 <= x) && (x2 >= x) &&
        (y1 <= y) && (y2 >=y))
}
```

(a)

```
// Scanner code ...
val rec = Record(fields, schema)
val x1 = rec("x1")
val x2 = rec("x2")
val y1 = rec("y1")
// ...
// invoking ST_Contains
if(ST_Contains(x1, x2, y1, y2,px,py))
  println("ST_Contains")
```

(b)

```
// Scanner code ...
// x11-x16 represent x1,x2,y1,y2,px,py
val x18 = x11 <= x15
val x20 = if (x18) {
  val x19 = x12 >= x15; x19
} else false
val x22 = if (x20) {
  val x21 = x13 <= x16; x21
} else false
val x24 = if (x22) {
  val x23 = x14 >= x16; x23
} else false
val x27 = if (x24) {
  val x25 = println("ST_Contains"); x25
} else ()
x27
```

(c)

Figure 7: Compiling spatial predicates (a) staging `ST_Contains` predicate using `Rep` type constructor (b) application code that uses `ST_Contains` (c) generated code for `ST_Contains` predicate in `Scala`.

nontrivial interpretive overhead in function calls to process parameter types and choosing the right overloaded function. For instance, a spatial shape could be a point, line, polygon, etc. Second, the library code appears as a black box for the query engine and hence cannot be further optimized. To address the previous shortcomings, LB2-Spatial implements spatial predicates inside the query engine. The implementation effort is equivalent to staging the code of an existing open source library with `Rep` type constructor.

Figure 7a shows a staged implementation for a simplified[3] `ST_Contains` predicate used in range queries that tests whether a point (x, y) is located inside a rectangle (x1, x2, y1, y2). The `Rep` constructor denotes that all parameter values are future stage (i.e., only known at runtime). Figure 7b shows a partial code for a spatial predicate operation that uses the staged `ST_Contains` to check whether a point is contained inside a rectangle. Figure 7c shows the generated code in `Scala` (LB2-Spatial generates `Scala` and `C`). Furthermore, LB2-Spatial implements a subset of the commonly used spatial predicates (e.g., `ST_Contains`, `ST_Intersects`, etc.) Additional predicates can be incrementally added as needed.

[3]Typically a compile-time abstractions `Rectangle` and `Point` are used instead of (x1,x2,y1,y2) and (x1,y1)

3.3 Data Loading and Indexing Structures

Data Loading. Data is processed *in-situ* without an explicit preloading phase as follows. The query optimizer uses the available meta-data, and statistics to produce an optimized query plan. At loading time, the spatial indexing structures are created based on the key attributes specified in the query plan. Finally, the spatial indexes are made available for the query evaluator.

Spatial Indexing Structures. Spatial data are multi-dimensional in nature. Hence, spatial engines implement various types of indexing structures to access data efficiently. For instance, R-trees are used for indexing minimum bounding rectangles (MBRs), k-d trees are suitable for range, and nearest neighbor queries, grids perform best when data is not skewed, etc. General indexing frameworks, e.g., SP-GiST [11] used in PostGIS provides high-level abstractions to support most commonly used tree indexes, e.g., R-tree, k-d tree and tries. Traversing generic trees tend to be inefficient due to expensive function calls (usually virtual). Thus, LB2-Spatial generates code for indexes (to support peak performance for long-running queries) and uses data schema to specialize access to indexes, and hence eliminating dispatch overhead. Moreover, LB2-Spatial flattens tree structures into arrays to optimize data access and layout.

Figure 8 demonstrates flattening the k-d tree index in LB2-Spatial. Figure 8a shows spatial data stored in a column-oriented layout for optimized storage and access. Figure 8b-c shows the space layout, and the standard k-d tree using pointers. The flattened k-d tree is shown in 8d where the values inside the flat array reference the original data without duplication. In main-memory spatial processing, flat data structures have shown good performance [59].

All runtime-generated indexing structures in LB2-Spatial are implemented using array abstractions that generate optimized code similar to LB2's [64]. Moreover, index-based operations, e.g., range, kNN, etc. are implemented as an index-based method, i.e., similar to normal interpreter code that is called by spatial operators. For instance, Figure 9 shows the implementation of an index range join operator that uses a spatial index (in contrast to the nested loop version shown in Figure 6). Line 6 obtains the index built on the `Points` (right) table. Lines 9-14 shows the actions performed on the `Rectangles` (left) table where a rectangle is extracted (Line 10) and the `RangeRectangle` method implemented inside the index finds the points that lie inside the rectangle. Furthermore, the callback is invoked in Line 13 to stream the result to the parent operator. Note the structure of basic operators, and index-based operators are almost the same. The difference lies in invoking the implementation provided by the index operation.

Pregenerated Indexing Structures. The base cost of generating spatial indexing structures at runtime is significant

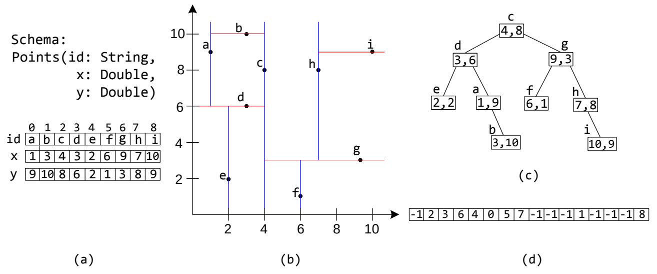


Figure 8: Flattening the k-d tree index in LB2-Spatial (a) data in column-layout (b)-(c) standard k-d tree implementation using pointers and (d) using a flat array.

```
// left: Rectangles table,
// right: spatial index on Points table
// Index range operator that uses a spatial index on the right table
class IndexRangeJoinOp(left: Op, right: Op) (idxName: String)
                       (rectFun: keyRect) extends Op {
// obtain index for right table
val index: Index = right.getIndex(idxName)
def exec(cb: Record ⇒ Unit) = {
  left.exec { lTuple ⇒
    val rectangle = rectFun(lTuple)
    // use index to find the points located inside rectangle
    index(rectangle).RangeRectangle { pnt ⇒
      cb(merge(pnt, rectangle))
}}}}
```

Figure 9: The implementation of the index point-rectangle range join operator.

for short-running spatial queries. Hence, using pregenerated data structures as external libraries would significantly reduce the end-to-end execution time. For such queries, LB2-Spatial uses external libraries for spatial indexing and generates only the necessary code for index creation and manipulating the indexed data.

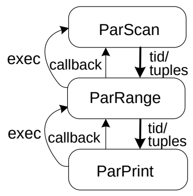
Auxiliary Data Structures. Spatial processing is best described as performing spatial computations while traversing indexing structures. The implementation of various spatial operators often use auxiliary data structures to assist traversals and maintain intermediate results. For instance, the kNN operation uses a heap to keep the list of k nearest spatial shapes during index traversal which may update the neighbors' list. In LB2-Spatial, we recognize the performance of auxiliary data structures is essential to query runtime. Therefore, all auxiliary data structures, e.g., stack, heap, etc. are implemented on optimized flattened fashion.

3.4 Parallelism

LB2 supports parallelism on shared memory systems using OpenMP [7] where blocks of parallel code are generated with OMP parallel annotations. The key elements to implement parallel evaluation are summarized as first, defining

```
1 class ParOp {
2   type ValueCallback = Record ⇒ Unit
3   type DataLoop = ValueCallback ⇒ Unit
4   type ThreadCallback =
5         Rep[Int] ⇒ DataLoop ⇒ Unit
6   def exec: ThreadCallback ⇒ Unit
7 }
```

(a)



(c)

```
8 def parallelPipeline(seq: Op ⇒ Op) =
9                     (parent: ParOp) ⇒ new ParOp {
10   def exec = {
11     val opExec = parent.exec
12      (tCb: ThreadCallback) ⇒ opExec { tId ⇒ dataloop ⇒
13       tCb(tId)((cb: ValueCallback) ⇒
14          seq(new Op { def exec = dataloop }).exec(cb) })
15 } } }
```

(b)

Figure 10: (a)-(b) Parallel operator class and parallel pipeline wrapper (c) the interactions between operators within a parallel query execution pipeline (adapted from [64]).

code generation constructs that emit various OMP annotations. Second, modifying the parallel evaluator structure, and operators internally to enable orchestrating parallel execution. Third, handling shared data structures for operators that maintain state by defining per-thread or lock-free data structures. For instance, the following code shows LB2's `parallelRegion` annotation that emits a `#pragma omp parallel` around a block of statements.

```
def parallelRegion(worker: Rep[Long] ⇒ Unit): Unit = {
  parallel_region {
    val j = ompGetThreadId
    worker(j)
}}
```

The code in Figure 10a-b shows the definition of LB2's parallel pipeline wrapper that enables generating parallel code by adding thread variables and thread callbacks (Lines 11-12). LB2-Spatial extends LB2's parallel evaluator with parallel spatial operators. Consider evaluating the following index range query.

```
SELECT *
FROM Points
WHERE ST_Contains(0,0,100,100,px,py)
```

The query pipeline in Figure 10c shows how the query evaluation starts with `ParPrint` calling the `exec` method of `ParRange` operator which in turn calls `ParScan.exec`. The `ParScan` operator is the point where threads are initiated. In other words, `ParScan` parallelizes the loop that reads data from the source. Thus, for each thread, `ParScan` calls `exec` callback with the thread id tId, and another callback dataloop. This allows the downstream operator `ParRange` to initialize the appropriate thread-local data structures (recall, `ST_Contains` uses a previously created index meaning that each thread requires a local data structure to maintain traversal state independently). Finally, the downstream operator `ParPrint` triggers the flow of data by invoking dataloop, and passing another callback upstream, on which the `ParScan` will send each tuple for the data partition corresponding to the active thread.

3.5 Spatial Applications

Traditional RDBMS provide procedural languages, e.g., PL/pgSQL in PostgreSQL to support user applications that interleave spatial processing and application code. Similarly, Spark programs naturally interleave front-end code, e.g., `Python`, `Scala`, etc. with dataframe operations. However, the performance of these applications is often suboptimal due to the limited visibility between query evaluation and user code. In other words, a query is executed independently where the application is given an iterator to process the result dataset. However, this is not an issue in LB2-Spatial since the query engine is implemented in `Scala` with LMS, the whole application code is optimized and generated (not only the operators). In the following, we demonstrate writing a spatial application using the spatial processing API.

Consider supporting a user-customized spatial-textual scoring operation that first finds the records that lie within a specific radius (facilitated by an index) and uses application code to assign a score for each record in the result based on the content of a textual attribute. Figure 11 gives an application program that performs spatial-textual processing to assign scores to tweets that were issued close to a number of cities. The application uses the tweet content to compute the score. Lines 1-11 creates an R-tree index named `tweetIdx` on `(lon,lat)` attributes of the `Tweets`. Lines 13-16 loads data into the `Cities` table which stores the geo-locations for a number of cities. The function `f` defined in Lines 22-29 encodes scoring actions to be performed on the result records obtained from `tweetIdx`. The `for` loop starting at Line 30 reads records from `Cities` table, uses the city location to probe the `tweetIdx` in order to find the tweets issued within `eps` radius from that city. Finally, the scoring code is injected using the callback function `f` which adds a score to each record in the result.

Additionally, LB2-Spatial supports writing user-defined functions (UDFs) similar to standard RDBMs. For instance, spatial predicates not already implemented in the spatial extension can be written as UDFs. Unlike traditional RDBMS, the implementation of UDFs in LB2-Spatial is *not* opaque to evaluation, and does not incur a performance penalty.

4 EVALUATION

In this section, we evaluate the performance of the spatial extension implemented in LB2. We compare the performance of LB2-Spatial with spatial library code [13, 60], a spatial extension to relational engine PostGIS [9] and two spatial Spark extensions Simba [70] and Geospark [72].

```
1  // create a record abstraction for spatial key
2  type pointRec = Record { val lon: Double; val lat: Double}
3  def pointRec(x: Rep[Double], y: Rep[Double]) =
4    new Record { val lon = x; val lat = y }
5   // Tweets schema
6  type tweetRec = Record { val tid: Long; val lon:
7     Double; val lat: Double; val tweet: String }
8  // building an R-tree index the Tweets data
9  val tweetIdx =
10    loadWithIdx[pointRec, tweetRecord] (file_tweet,
11    RTreeKey("twidx", x ⇒ pointRec(x.lon, x.lat)))
12  // Citites schema
13  type citiesRec = Record { val cname: String;
14      val lon: Double; val lat: Double }
15  // loading the Citites table
16  val Cities = load[citiesRec](file_cities)
17  // constructing a (lon, lat) key
18  val locationkey = x ⇒ pointRec(x.lon, x.lat)
19  val eps = 1.5
20  type ScoreRec = Record { val score: Long }
21  // user code for scoring records based on the tweets content
22  val f = { tuple ⇒
23    val t = tuple("tweet") // getting the tweet attribute
24    val scoreValue =
25       // code to compute a score based on the tweet text
26       // ...
27    val rec = new Record { val score = scoreValue }
28    printRecord(merge(tuple,rec))
29  }
30  // scanning Citites and probing tweetIdx
30  for ( i<-0 until Cities.length) {
32    val city = Cities(i)
33    val cityLoc = locationkey(city)
34    // invoking the distance predicate and
35    // passing the scoring code as a callback
36    tweetIdx(cityLoc).distance(f, eps)
37 }
```

Figure 11: Compiling user applications in LB2-Spatial.

We conduct three sets of experiments. The first set evaluates the performance of spatial operators in a single-core setup. We also provide experiments that focus on evaluating the effect of varying the selectivity ratio in range queries and the impact of scaling up the index size in spatial join queries. The second set of experiments evaluates parallelism in LB2-Spatial, and spatial Spark extensions when scaling up the number of cores. The third set of experiments evaluates the code generation and compilation overhead, and measures the total memory consumed by LB2-Spatial, and the Spark-based systems while performing join operations. Finally, we provide a productivity evaluation analysis that summarizes the lines of code needed to extend LB2 with spatial processing.

The experiments focus on evaluating the absolute query runtime without including data loading and indexing construction time. The rationale behind this decision is first, both PostGIS and spatial spark extensions do not optimize loading time. Second, Spark extensions perform expensive data partitioning phase during loading time to avoid scanning the entire data under Spark's distributed execution model. For

Table 1: Spatial datasets that are used in experiments.

Dataset	Geometry	#Records	Size(GB)
Tweets	Point	1 billion	32
OSM Nodes	Point	200 million	4.3
OSM Rectangles 1	Rectangle	1 million	0.05
OSM Rectangles 2	Rectangle	114 million	14
OSM Buildings	Polgons	1 million	0.19
Random	Point	10 million	0.5

Table 2: Queries that are used in experiments.

Point-Rectangle Range Predicate	`SELECT * FROM Points WHERE ST_Contains(ST_PolygonFromEnvelope(x1,x2,y1,y2), Points.pointshape);`
Point-Rectangle Range Join	`SELECT * FROM Points, Rectangles WHERE ST_Contains(Rectangles.rectangleshape, Points.pointshape);`
Rectangle-Polygon Range Join	`SELECT * FROM Rectangle, Polygons WHERE ST_Intersects(Rectangles.rectangleshape, Polygon.polygonshape);`
Distance Join	`SELECT * FROM Points1, Points2 WHERE ST_DWithin(Points1.pointshape, Points2.pointshape, distance);`
kNN Join	`SELECT * FROM Points1 AS P1 KNN JOIN Points2 AS P2 ON POINT(P2.x, P2.y) IN KNN(POINT(P1.x, P1.y), k);`

the single-core setup, we show that LB2-Spatial outperforms spatial spark extensions and PostGIS in spatial join queries by 2×-299×. For scaled-up execution, LB2-Spatial is 10×-20× faster than spatial Spark extensions.

Datasets and Queries. Table 1 shows the spatial datasets we use in the experiments section. The tweets dataset consists of one billion geo-tagged tweets located inside the United States. The tweets were collected over the period from January 2013 to December 2014. We cleaned the dataset from invalid records that did not include an accurate geo-location. Furthermore, we only kept the longitude and latitude attributes and dropped the remaining attributes. We added a serial numeric attribute to identify data records. The Open Street Map (OSM) consists of 200 million points, two rectangles datasets one million and 114 million respectively, and one million polygons obtained from a performance evaluation study that compares the performance of several spatial Spark extensions [51]. The last dataset is synthetic and consists of ten million randomly generated points using the uniform distribution.

Table 2 shows the queries used in the experiments. We chose a subset of queries that evaluate the performance of index-traversal operations (e.g., range join and distance join) and operations that maintain a state e.g., kNN join. The syntax of kNN join is adapted from [70]. We run each query five times and record the median reading. To guarantee local execution, we run the queries in a single NUMA node using numactl `-m` and `-C` options.

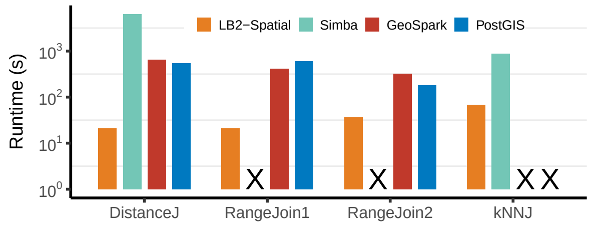
Environment. All experiments are conducted on a single NUMA machine with 4 sockets, 24 cores in a Xeon(R) Platinum 8168 CPU per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu1 16.04.9. We use Scala 2.11, GCC 5.4 with optimization flag `-O3`. We use `Scala` 2.11, PostgreSQL 10.4, PostGIS 2.2, GeoSpark 2.0, and Simba with Spark 2.1.

4.1 Single-Core Spatial Join

In the first experiment, we compare LB2-Spatial with Simba, GeoSpark, and PostGIS in point-rectangle range join (using `ST_Contains`), rectangle-polygon range join (using `ST_Intersects`), distance join, and kNN join queries using only a single-core. The goal of this experiment is to show the performance gained when compiling long-running spatial queries to low-level code. For point-rectangle range join we use the OSM points, and rectangles 1 datasets as follows. A spatial index is built on the left points table of size 200 million, and the size of the right table is one million[4]. For the rectangle-polygon range join query, a spatial index is build on the left table (rectangles 2 dataset) of size 114 million. Moreover, we set up the value of k to 5. For distance join and kNN join queries, we use the tweets dataset where a spatial index is built on the left table. The size of the index is 200 million and 10 million respectively. Moreover, the size of the right table is one million. For the kNN join query, we reduced the index size to 10 million as in [51, 70] to avoid runtime issues (i.e., Simba's kNN algorithm creates duplicate points during the kNN processing which eventually fills up the heap space causing a runtime crash [51]). The points R-tree index used in point-rectangle join, distance join and kNN join is runtime generated whereas the rectangles R-tree index used in the rectangle-polygon range join query is pregenerated.

Figure 12 gives the absolute runtime for four spatial join queries: distance join, range join (point-rectangle and rectangle-polygon) and kNN join. Overall, LB2-Spatial outperforms Simba, GeoSpark, and PostGIS in all join queries. Moreover, PostGIS outperforms Simba and GeoSpark in index distance join query and rectangle-polygon range join. The performance gap between PostGIS and spatial spark extensions is attributed in part to the fact that Spark-based systems underperform in the single-core setup[28, 43].

[4]For the single-core experiments, we limited the index size to 200 million and used a smaller right table i.e., one million, to keep the runtime of spatial Spark extensions relatively low since Spark-based systems inherit significant internal overheads due to Spark's distributed evaluation.



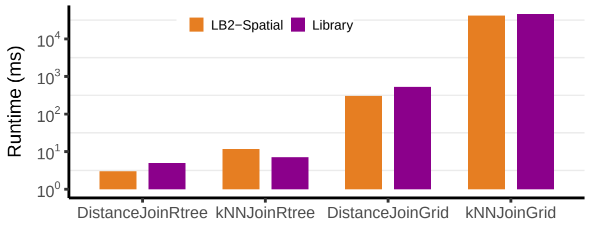
	Distance J	Range J1	Range J2	*k*NN J
LB2-Spatial	21.1	21.1	31.2	68.4
Simba	6325.8	-	-	888.2
GeoSpark	653.7	412.9	326	-
PostGIS	544.4	606	75.2	-

Figure 12: The absolute runtime in seconds for LB2-Spatial, Simba, GeoSpark, and PostGIS in index distance join, index point-rectangle range join (Range J1), index rectangle-polygon range join (Range J2), and index *k*NN join.

On a query-by-query analysis, LB2-Spatial outperforms PostGIS, GeoSpark, and Simba in the distance join by 25×, 30×, and 299× respectively. Moreover, LB2-Spatial outperforms GeoSpark and PostGIS in point-rectangle by 19× and 28× respectively. Similarly, LB2-Spatial outperforms the previous systems in rectangle-polygon range join by 2×-10× respectively. The comparable performance between PostGIS and LB2-Spatial in the polygon-rectangle range join is attributed in part to (i) the nature of the dataset where most of the polygons are contained inside the rectangles which in turn decreases the expensive intersection operation (ii) the spatial index used in this query is pregenerated and incurs additional traversal time in contrast to fully specialized indexing structures. Finally, LB2-Spatial is 13× faster than Simba in *k*NN join.

In general, map-reduce extensions similar to Simba and GeoSpark are optimized for distributed execution on large clusters. The single machine performance is suboptimal due to internal RDD overhead, JVM overhead, etc. The performance of Spark-based systems can be improved with leveraging multi-threading and appropriate data partitioning schemes that work well with the broadcast-based operations. For traditional spatial RDBMS extensions, the interpreted evaluation associated with processing data in high-level incurs significant runtime overhead.

Stand-alone Spatial Indexing Libraries. In the second experiment, we compare LB2-Spatial with stand-alone spatial indexing library code. The *Spatial Indexing at Cornell* project [13, 60] provides a spatial indexing library written in C++ for a set of common spatial indexing structures and operations. For this experiment, we extend the R-tree and grid structures from the previous library with distance join and



	DJ-R-tree	*k*NNJ-R-tree	DJ-Grid	*k*NNJ-Grid
LB2-Spatial	3	12	303	41153
Library	5	7	526	45524

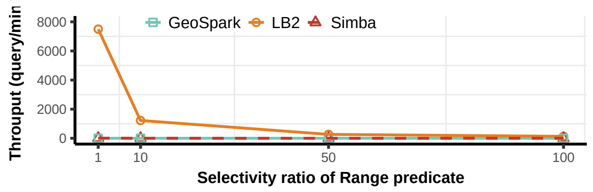
Figure 13: The absolute runtime in milliseconds for LB2-Spatial and library code in distance join and *k*NN join using R-tree and grid.

*k*NN join operations to evaluate LB2-Spatial performance with specialized code (i.e., without RDBMS overhead). We use a randomly generated dataset in this experiment since the grid index in [13] processes only positive numbers (and hence the longitude and latitude datasets are not applicable). Moreover, a spatial index is built on the left table of size 10 million[5], and the size of the right table is 1000.

Figure 13 shows the absolute runtime of performing distance, and *k*NN join using R-tree and grid in LB2-Spatial and stand-alone indexing library. The performance of R-tree distance join is comparable in both systems whereas the spatial library is 70% faster than LB-Spatial in the *k*NN join query. The small performance gap in the *k*NN join query is attributed in part to the uniformly distributed dataset that assists in pruning more data while performing less computations. Thus, we expect LB2-Spatial to perform better in dense datasets, e.g., tweets dataset, where more time is spent in accessing and processing data in the leaf level.

For the case of the grid index, LB2-Spatial outperforms the library code by 70%, and 10% in distance join and *k*NN join respectively. The small performance gap between the two systems is attributed in part to a number of implementation details. First, LB2-Spatial's grid is implemented as a flat array whereas the library grid index is implemented as a two-dimensional array. Hence, there is additional memory access incurred in the later system. Second, LB2-Spatial's generated code leverages compiler's level optimizations (e.g., dead code elimination, loop fusion, etc.) that are sometimes missed by general purpose compilers. Lastly, although grid structures, are significantly slower than spatial trees, grids are still commonly used for low index creation time. Furthermore, the availability of a low-cost indexing structure facilitates building two-tier indexing structures that are commonly used in distributed spatial processing.

[5]The choice of building a small index in this experiment is driven by the configurations of the R-tree in [13] which always sets the leaf size to 12 leading to a higher tree height with increasing the index size.



	1	10	50	100
LB2-Spatial	7500	1224	267	134
Simba	9	2	0	0
GeoSpark	0	0	0	0

Figure 14: The selectivity ratio of range predicate.

Range Predicate Selectivity. The selectivity of a spatial predicate determines the amount of computations performed by an operator. In this experiment, we evaluate the effect of varying the selectivity ratio of the point-rectangle range predicate using single-core. We use the OSM nodes dataset of size 200 million and build a spatial index on the points data. We follow the same approach from [51] and submit a batch of 100 queries and compute the throughput as the number of queries executed per minute. In this experiment, a throughput value less than one is counted as zero.

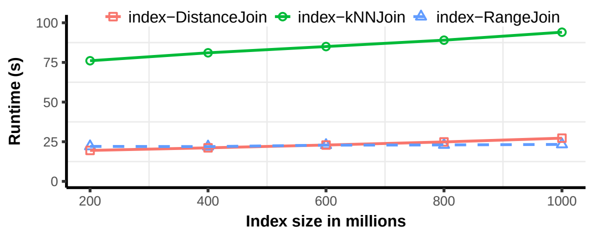
Figure 14 shows the throughput of range query in LB2-Spatial, Simba and GeoSpark for σ = 1, 10, 50 and 100. LB2-Spatial's throughput for highly selective range predicate is 7500 and 134 respectively when the range predicate does not perform any data pruning. On the other hand, the throughput of spatial Spark extensions is very low, e.g., 9 and 2 for σ = 1, 10 respectively in Simba and zero otherwise. Given the simplicity of the range predicate, the low throughput is primarily caused by Spark's runtime overhead.

Scaling up Index Size. In this experiment, we evaluate the effect of scaling up the index size in LB2-Spatial for range, distance and kNN join operators using the tweets dataset. For each join query, we build a spatial index on the left table of size 200 million, 400 million, up to one billion. The size of the right table is fixed as one million and the R-tree node size is 20. Moreover, we exclude the spatial Spark systems since Spark does not scale up well in a single-core [28, 51].

Figure 15 gives the absolute runtime in seconds for performing range, distance, and kNN join operations in LB2-Spatial. We observe the runtime increases linearly with increasing the index size. The outcome validates that LB2-Spatial does not incur system overhead beyond data loading and a proportional time for index traversal as we increased input size.

4.2 Parallel Spatial Join Queries

In this experiment, we compare the scalability of LB2-Spatial with GeoSpark, and Simba on distance join and point-rectangle range join queries. We use the OSM dataset where a



	200M	400M	600M	800M	1B
Range Join	22.07	21.93	22.83	22.98	23.31
Distance Join	19.52	21.15	22.92	24.88	24.88
kNN Join	76.05	81.06	85	89.02	94.05

Figure 15: The scalability of LB2-Spatial range query with increasing the index size in single-core.

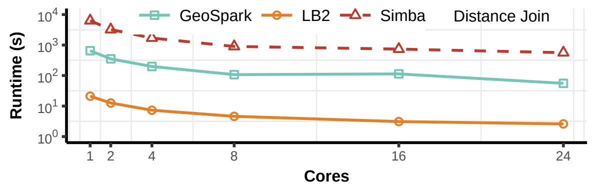
spatial index is built on the left table of size 200 million. Moreover, the size of the right table is one million. The experiment focuses on the absolute performance and the Configuration that Outperforms a Single Thread (COST) metric proposed by McSherry et al. [43]. COST compares the number of threads needed by one system to match the single-thread performance of another. We scale the number of cores up to 24 to keep the execution local within a single socket.

Figure 16a gives the absolute runtime for scaling up LB2-Spatial, GeoSpark, and Simba in distance join query. Overall, the speedup of all systems increases with the number of cores and spatial Spark systems appear as having better speedup than LB2-Spatial at 24 cores, i.e., 8× to 11×. However, examining the absolute running times, LB2-Spatial is 21× and 214× faster than GeoSpark, and Simba respectively at 24 cores. Furthermore, it takes GeoSpark over than 24 cores to match LB2-Spatial's single-core performance. What appears to be good scaling for spatial Spark extension actually reveals that the runtime incurs significant overhead.

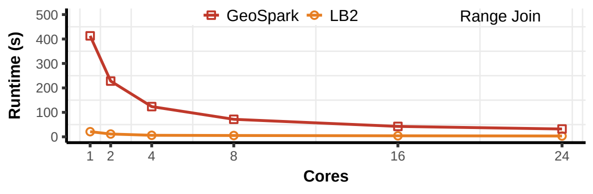
Figure 16b gives the absolute runtime for scaling up LB2-Spatial and GeoSpark in the point-rectangle range join query. LB2-Spatial outperforms GeoSpark 20× up to 4 cores, and by 10× as the number of cores reach 24. Also, it takes GeoSpark over than 24 cores to match LB2's single-core performance. The gap in performance is attributed in part to Spark's internal overhead, Java Virtual Machine (JVM) overhead, high-level data structures implementation, etc.

4.3 Code Generation and Compilation Overheads

In this experiment, we analyze the overhead of code generation and compilation using `GCC` for various spatial joins and spatial predicates operations (nested loop and index-based). The LB2-Spatial query compiler is implemented in a high-level language (`Scala`) and generates optimized `C`. As a consequence, the code generation time is affected by `Scala`'s runtime. The compilation overhead results are illustrated in



(a)



(b)

Runtime	1	2	4	8	16	24
LB2-Spatial	21.1	12.6	7.3	4.6	3.1	2.6
GeoSpark	653.7	352.5	198.5	107.2	113.5	55.6
Simba	6325.8	3229	1702.6	901.8	738	563.4
Speedup						
LB2-Spatial	1	1.7	2.9	4.6	6.8	8
GeoSpark	1	1.9	3.3	6.1	5.8	11.8
Simba	1	2	3.7	7	8.6	11.2

Runtime	1	2	4	8	16	24
LB2-Spatial	21.0	11.4	6.2	5.3	4.1	3.3
GeoSpark	412.9	228	123.6	71.6	42.6	32
Speedup						
LB2-Spatial	1	1.8	3.4	4	5.1	6.4
GeoSpark	1	1.8	3.3	5.8	9.7	12.9

Figure 16: The absolute runtime in seconds (s) for parallel scaling up LB2-Spatial, GeoSpark, and Simba in (a) distance join (left) and (b) point-rectangle range join (right) on 2, 4, 8, 16 and 24 cores for the tweets dataset.

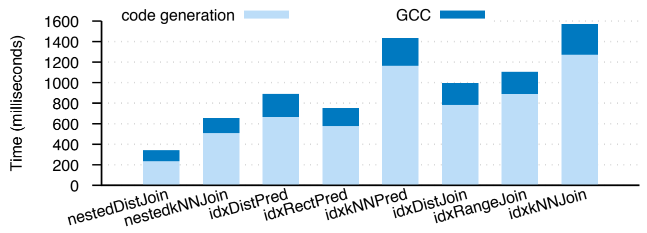


Figure 17: Code generation and compilation for spatial queries.

Figure 17 where the y-axis illustrates both and code generation and GCC compilation for each query.

Overall, the code generation time is dominated by generating specialized code for the scanner, spatial indexing structures and query evaluation. For instance, generating an index-based distance join query takes approximately 780 ms whereas a nested loop version takes 330 ms. Moreover, compiling *k*NN join operation incurs additional overhead for generating the heap data structure that maintains the list of nearest *k*. Similarly, the overhead of generating spatial predicates on a single table is comparable to spatial join since the compilation overhead is largly incurred by the data structures rather than the query evaluation. Finally, the GCC compiler with -O3 optimization adds 100 ms to 200 ms, depending on the implmentation details, code size, etc.

Furthermore, we analyze the compilation overhead for two high-level query compilers[6] discussed in Section 2.2. Engine *A* uses precompiled indexing structures, e.g., existing libraries. In this case, the query compiler generates only the necessary code for index creation and manipulating the

[6]The query compiler instances used in this experiment are implemented in high-level language which incurs code generation cost.

Table 3: Code generation, compilation and execution time in milliseconds for compiling point-rectangle range join and point-rectangle range predicate queries in Engine *A* (precompiled linked index), and Engine *X* (generated index without specialization).

RangeJoin	generation	compilation	execution
Engine *A*	339	115	152896
Engine *X*	564	390	72676
RangePredicate			
Engine *A*	265	55	< 1
Engine *X*	325	340	< 1

indexed data. On the other hand, Engine *X* is a point on the compilation spectrum where the query engine generates data structures without specialization, e.g., function inlining, data structures flattening, etc. Table 3 compares the performance of compiling point-rectangle range join and point-rectangle range predicate queries in the previous two engines. Overall, pre-compilation in Engine *A* reduces compilation overhead approximately by 2× in comparison with Engine *X* and the compilation overhead reported in Figure 17.

In general, the compilation overhead for spatial join queries is negligible to the end-to-end query execution time. However, for the shorter queries, e.g., range predicates, a pre-compilation strategy is a beneficial trade-off for minimizing compilation overhead.

4.4 Memory Consumption

In this experiment, we measure the total memory consumed by LB2-Spatial, Simba, and GeoSpark while performing spatial join operations. For both Spark and LB2-Spatial, we used

Table 4: Total memory consumed (in GB) by LB2-Spatial, Simba and GeoSpark while performing various spatial join operations.

	Distance Join	Range Join	kNN Join
LB2	12.3	21.1	1.3
Simba	60.8	-	2.6
GeoSpark	94.4	95.8	-

the `time` command in `Unix` with the verbose option `-v` and recorded the value of *maximum resident set size*. We also monitored the Storage tab in Spark's web UI[7] which gives the memory occupied by a cached RDD. We observed that storage memory value after constructing a spatial index on 200 million records is approximately 21 GB and 48 GB in Simba, and GeoSpark respectively. The difference is due to index serialization in Simba [70].

Table 4 gives the maximum execution memory used by LB2-Spatial and the Spark-based systems. LB2-Spatial consumes less memory than Spark-based systems (approximately 5×-7× and 4.5× less in the distance and range queries respectively). Typically, Spark-based systems consume more memory due to replication, distributed execution, and JVM [61]. Furthermore, spatial Spark extensions perform data partitioning that requires extra storage for sampling and processing [51]. On the other hand, LB2-Spatial leverages the dataset size, when available, and incrementally increases the data structures size otherwise. For instance, the raw size of the points and rectangles datasets used in range query is approximately 18.6 GB. LB2-Spatial consumed only additional 3 GB to perform this operation.

4.5 Productivity Evaluation

Table 5 summarizes the development effort in terms of line of code (in `Scala`) needed to extend LB2 with spatial processing. The front-end work consists of extending an existing SQL front-end and optimizer with spatial keywords and rules. Overall, the front-end extension was written in 277 lines. The spatial indexing structures (R-tree, k-d tree and grid) and auxiliary data structures were developed in 1087 lines. Moreover, basic, index-based, single thread and parallel operations (for R-tree, k-d tree and grid) are implemented in 1474 lines. Other 120 lines of code cover various tasks, e.g., data loading, configurations, etc. Overall, the LB2-Spatial engine consists of 2958 lines.

5 RELATED WORK

Spatial Processing. Several well-known relational databases are extended with indexing structures, spatial

[7]Memory consumption for Spark's RDDs cannot be collected programmatically [51, 61].

Table 5: Lines of code needed to extend LB2 with spatial processing.

Front-end	277
Spatial indexing and auxiliary data structures	1087
Spatial operators (basic and index-based)	1474
Other	120
Total	2958

types, etc. to support spatial processing. For instance, PostGIS [9] uses SP_GiST's [11] R-tree, Oracle Spatial [8] adds QuadTree, Microsoft SQL [30] adds a hierarchical grid index, IBM DB2 [24] uses a grid index where each cell is indexed using a B-tree [12]. Similarly, SAP HANA Spatial [10] and Vertica [14] created spatial extensions. AT-GIS [48] is a single-node parallel spatial processing system that integrates parsing and spatial query processing using the proposed associative transducers (ATs) computational abstraction. MonetDB [68] (a column-store) does not support indexing and stores spatial bounding boxes as a separate column. The performance of containment queries is comparable to the index-based implementation. However, the performance of large join queries are suboptimal since it requires maintaining the entire dataset in memory. GeoCouch [2] is a NoSQL spatial processing engine. A comprehensive overview of spatial indexing structures is surveyed in [42, 44, 46].

Hadoop + Spatial Processing. The Hadoop big data era witnessed several Hadoop-based systems that extended Hadoop with spatial partitioning, indexing, operators, etc. The Spatial Join with MapReduce (SJMR) first introduced in [73] did not use an index. SpatialHadoop [27] pioneered SJMR using two indexing levels: a global index for partitioning data across nodes and a local index for accessing data within each node. Similarly, HadoopGIS [15] supports SJMR in addition to a specialized pathology image analysis module. Parallel SECONDO [41] integrates Hadoop with the SECONDO [36] database that supports spatial processing. GeoMesa [31] supports indexing and querying of spatiotemporal data on Accumulo [1]. MD-HBase [47] extends HBase [4] (key-value store) with multidimensional indexes to support range and kNN queries. Accumulo and HBase are based on Google's BigTable [22]. The main disadvantages in Hadoop-based systems are the need to load data into HDFS and the cost of inter-job data movement.

Spark + Spatial Processing. In recent years, Spark [17] computing framework has become popular for its main-memory execution model and expressive front-end. GeoSpark [72], SparkGIS [19], Stark [37], LocationSpark [67], Simba [70], Magellan [6], SpatialSpark [71] and others extended Spark with spatial indexing (e.g., R-tree, quadtree, etc.), spatial operators (e.g., distance join, kNN join, etc.) and spatial types (e.g., points, polygons, etc.). Du et. al. [25] presented a multiway spatial join algorithm with Spark (MSJS). However, the performance of Spark extensions inherits Spark's internal main-memory processing bottlenecks, the overhead of distributed datasets (RDDs) operations, and communication through Spark's runtime system [28]. The spatial extension in this work supports spatial processing without incurring runtime overhead. Similar to spatial Spark extensions, LB2-Spatial integrates spatial indexing and operations into LB2. Furthermore, LB2-Spatial generates code for a single machine instead of clusters.

Query Compilation. The idea of Compiling SQL queries into native code was first proposed in System R [18]. However, compilation was not adapted at that time due to portability challenges. The Volcano iterator evaluation model [34] implemented in most disk-based data management systems, for its simplicity and expressiveness, suffered from large overhead due to calling `next` for each tuple. Hence, Vectorization techiques introduced in MonetDB [68], where an operation is performed on an array of tuples, reduces the Volcano's evaluation overhead significantly. Daytona [35] compiles Cymbal (Daytona's query language) into `C` code. The work of [54] compiles queries to Java bytecode by removing virtual functions from the iterator evaluation. The holistic query evaluation (HIQUE [40]) uses a template code generation approach to compile queries to `C` code.

HyPer [45] introduced the data-centric evaluation model that is used in most main-memory databases. HyPerSpace [52] extends spatial compilation into HyPer. In contrast to LB2-Spatial, HyPerSpace uses the S2 library for spatial predicates, and does not implement spatial indexing structures. Furthermore, HyPerSpace does not support join operations which extensively use spatial indexing structures for efficient performance. Voodoo [53] compiles portable query plans that can run on CPUs and GPUs. Moreover, Voodoo's intermediate algebra captures hardware optimizations, e.g., multicores, SIMD, etc. Weld [50] provides a common runtime for diverse libraries that represent computations using Weld's intermediate representation (IR). During evaluation, the IR is optimized and compiled into machine code.

Query Compilation in PostgreSQL. Butterstein et. al. [21] compiles query subexpressions into machine code. The work in [58] uses program specialization and LLVM to generate query code. Furthermore, Duta et. al. [26] compiles procedural SQL away by interpreting functions into subqueries that can be efficiently evaluated.

High-level Query Compilers. The Legobase [39] query compiler is implemented in `Scala` and uses the lightweight modular staging framework (LMS) [56] to generate optmized `C`. DBLAB [57] reimplements Legobase and uses multiple compiler passes to compile queries. The LB2 [64] query compiler expanded the work presented in ("SQL to C in 500 lines" [55]) and demonstrated that highly efficient query compilation can be performed efficiently in a single compiler pass. LB2-Graph [66] supports compiling graph workloads in LB2. Flare [28] is a back-end accelerator for Apache Spark SQL that brings relational performance on par with the best SQL engines. Lantern [69] is a machine learning framework that performs automated differentiation via delimited continuations, and uses LMS to generate efficient low-level `C++` and `CUDA` code. Flare and Lantern are integrated [29] on the code generation level to efficiently compile machine learning applications that process data in Flare. Delite [20, 62] and its domain specific languages (OptiQL, OptiML [63], and OptiMesh) use LMS to compile SQL, machine learning, linear algebra, etc. to low-level code.

6 CONCLUSIONS

The widespread use of location-enabled devices has resulted in volumes of spatial data that need to be efficiently processed. Several spatial engines are implemented as extensions to relational query engines or map-reduce cluster computing frameworks to leverage optimized memory, storage, and evaluation. Still, the performance of these extensions is often impeded by the interpretive nature of the underlying data management, generic data structures, and the need to execute domain-specific external libraries.

In this work, we address the system challenges for compiling spatial workloads, and we show that the idea of the first Futamura projection that links interpreters and compilers through specialization can be applied to compile spatial queries into low-level code, mainly eliminating the complexities that so far have negatively impacted the performance of spatial extensions. As a proof of concept, we add spatial query compilation inside the LB2 main-memory query compiler. We support parallelism for shared memory using OpenMP and thread-aware data structures. The spatial extension matches the performance of the library code. In single-core distance join, range join and *k*NN join queries, LB2-Spatial outperforms spatial Spark extensions and PostGIS in spatial join queries by 2×-299×. For scale-up execution, LB2-Spatial is 10×-20× faster than spatial Spark extensions. As future work, we plan to support additional spatial data types, spatial operators, high-dimensional data structures, and minimize compilation overhead.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their valuable feedback. This work was supported in part by NSF awards 1553471, 1564207, 1918483, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

[1] [n. d.]. Accumulo. https://github.com/couchbase/geocouch.

[2] [n. d.]. GeoCouch. https://accumulo.apache.org.

[3] [n. d.]. GEOS-Geometry Engine, Open Source. https://trac.osgeo.org/geos.

[4] [n. d.]. HBase. https://hbase.apache.org/.

[5] [n. d.]. JTS Topology Suite. http://www.vividsolutions.com/jts/JTSHome.htm.

[6] [n. d.]. Magellan: Geospatial Analytics Using Spark. https://github.com/harsha2010/magellan.

[7] [n. d.]. OpenMP. http://openmp.org/.

[8] [n. d.]. Oracle Spatial and Graph. https://www.oracle.com/assets/spatial-and-graph-ds-1738135.pdf.

[9] [n. d.]. PostGIS. http://postgis.org.

[10] [n. d.]. SAP HANA Spatial Reference. https://help.sap.com/viewer/cbbbfc20871e4559abfd45a78ad58c02/2.0.02/en-US.

[11] [n. d.]. SP-GiST Indexes. https://www.postgresql.org/docs/10/static/spgist.html.

[12] [n. d.]. Spatial Extender User's Guide and Reference. ftp://public.dhe.ibm.com/ps/products/db2/info/vr105/pdf/en_US/DB2Spatial-db2sbe1050.pdf.

[13] [n. d.]. Spatial indexing at Cornell. http://www.cs.cornell.edu/bigreddata/spatial-indexing.

[14] [n. d.]. Vertical Geospatial Analysis. https://www.vertica.com/wp-content/uploads/2016/09/B9697-Geospatial_Ni.pdf.

[15] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *VLDB* 6, 11 (2013), 1009–1020.

[16] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: adaptive query workload aware partitioning of big spatial data. *PVLDB* 8, 13 (2015), 2062–2073.

[17] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 1383–1394.

[18] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: relational approach to database management. *TODS* 1, 2 (1976), 97–137.

[19] Furqan Baig, Hoang Vo, Tahsin Kurc, Joel Saltz, and Fusheng Wang. 2017. SparkGIS: Resource Aware Efficient In-Memory Spatial Query Processing. In *SIGSPATIAL*. ACM, 28.

[20] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *CGO*. ACM, 194–205.

[21] Dennis Butterstein and Torsten Grust. 2016. Precision performance surgery for CostgreSQL: LLVM—based Expression Compilation, Just in Time. *VLDB* 9, 13 (2016), 1517–1520.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[23] Charles Consel and Olivier Danvy. 1993. Partial evaluation: Principles and perspectives. In *Journees Francophones des Langages Applicatifs*. 493–501.

[24] Judith R Davis. 1998. IBM'S DB2 SPATIAL EXTENDER: MANAGING GEO-SPATIAL INFORMATION WITHIN THE DBMS. (1998).

[25] Zhenhong Du, Xianwei Zhao, Xinyue Ye, Jingwei Zhou, Feng Zhang, and Renyi Liu. 2017. An Effective high-performance multiway spatial join algorithm with Spark. *ISPRS International Journal of Geo-Information* 6, 4 (2017), 96.

[26] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away. *CIDR* (2019).

[27] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*. 1352–1363.

[28] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*. 799–815.

[29] Grégory Essertel, Ruby Y Tahboub, Fei Wang, James Decker, and Tiark Rompf. 2019. Flare & lantern: efficiently swapping horses midstream. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1910–1913.

[30] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. 2008. Spatial indexing in microsoft SQL server 2008. *SIGMOD*, 1207–1216.

[31] Anthony Fox, Chris Eichelberger, James Hughes, and Skylar Lyon. 2013. Spatio-temporal indexing in non-relational distributed databases. In *Big Data*. IEEE, 291–299.

[32] Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan* 54-C, 8 (1971), 721–728.

[33] Yoshihiko Futamura. 1999. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation* 12, 4 (1999), 377–380.

[34] Goetz Graefe. 1994. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on* 6, 1 (1994), 120–135.

[35] Rick Greer. 1999. Daytona and the fourth-generation language Cymbal. In *ACM SIGMOD Record*, Vol. 28. ACM, 525–526.

[36] Ralf Hartmut Güting, Thomas Behr, Victor Almeida, Zhiming Ding, Frank Hoffmann, Markus Spiekermann, and LG Datenbanksysteme für neue Anwendungen. [n. d.]. *SECONDO: An extensible DBMS architecture and prototype.*

[37] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. 2017. The STARK framework for spatio-temporal data analytics on spark. *BTW* (2017).

[38] Neil D Jones. 1996. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)* 28, 3 (1996), 480–503.

[39] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *PVLDB* 7, 10 (2014), 853–864.

[40] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE, 613–624.

[41] Jiamin Lu and Ralf Hartmut Guting. 2012. Parallel secondo: boosting database engines with hadoop. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 738–743.

[42] Ahmed R Mahmood, Sri Punni, and Walid G Aref. 2019. Spatio-temporal access methods: a survey (2010-2017). *GeoInformatica* 23, 1 (2019), 1–36.

[43] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. *Scalability! But at what COST?* Technical Report. Microsoft Research.

[44] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. 2003. Spatio-temporal access methods. *IEEE Data Eng. Bull.* (2003).

[45] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[46] Long-Van Nguyen-Dinh, Walid G Aref, and Mohamed Mokbel. 2010. Spatio-temporal access methods: Part 2 (2003-2010). *IEEE Data Eng. Bull.* (2010).

[47] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM*, Vol. 1. IEEE, 7–16.

[48] Peter Ogden, David Thomas, and Peter Pietzuch. 2016. AT-GIS: Highly Parallel Spatial Query Processing with Associative Transducers. In *SIGMOD*. ACM, 1041–1054.

[49] B Ooi, Ron Sacks-Davis, and Jiawei Han. 1993. Indexing in spatial databases. (1993).

[50] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *CIDR*.

[51] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673.

[52] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *SIGMOD*. 2145–2148.

[53] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB* 9, 14 (2016), 1707–1718.

[54] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE, 23–23.

[55] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP*. ACM, 2–9.

[56] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

[57] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*.

[58] Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher. 2017. Runtime Specialization of PostgreSQL Query Executor. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 375–386.

[59] Darius Šidlauskas and Christian S Jensen. 2014. Spatial joins in main memory: Implementation matters! *PVLDB* 8, 1 (2014), 97–100.

[60] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. 2013. An experimental analysis of iterated spatial joins in main memory. *VLDB* 6, 14 (2013), 1882–1893.

[61] Spark. [n. d.]. Tuning Spark. https://spark.apache.org/docs/latest/tuning.html.

[62] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS* 13, 4s (2014), 134.

[63] Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*.

[64] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.

[65] Ruby Y Tahboub and Tiark Rompf. 2016. On supporting compilation in spatial query engines:(Vision paper). In *SIGSPATIAL*.

[66] Ruby Y Tahboub, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Towards compiling graph queries in relational engines. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages*. 30–41.

[67] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: a distributed in-memory data management system for big spatial data. *VLDB* 9, 13 (2016), 1565–1568.

[68] Maarten Vermeij, Wilko Quak, Martin Kersten, and Niels Nes. 2008. MonetDB, a novel spatial column-store DBMS. (2008).

[69] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *PACMPL* 3, ICFP (2019), 1–31.

[70] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. (2016).

[71] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 34–41.

[72] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*. ACM, 70.

[73] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. 2009. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE international conference on*. IEEE, 1–8.