On Supporting Compilation in Spatial Query Engines (Vision Paper)

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

ABSTRACT

Today's 'Big' spatial computing and analytics are largely processed in-memory. Still, evaluation in prominent spatial query engines is neither fully optimized for modern-class platforms nor taking full advantage of compilation (i.e., generating low-level query code). Query compilation has been rapidly rising inside in-memory relational database management systems (RDBMSs) achieving remarkable speedups; how can we bring similar benefits to spatial query engines?

In this research, we bring in proven Programming Languages (PL) approaches e.g., partial evaluation, generative programming, etc. and leverage the power of modern hardware to extend query compilation inside spatial query engines. We envision a *fully compiled* spatial query engine that is efficient and feasible to implement in a high-level language. We describe LB2-Spatial; a prototype for the first fully compiled spatial query engine that employs generative and multi-stage programming to realize query compilation. Furthermore, we discuss challenges, and conduct a preliminary experiment to highlight potential gains of compilation. Finally, we sketch potential avenues for supporting spatial query compilation in Postgres; a traditional RDBMS and Spark SQL; a main-memory cluster computing framework.

CCS Concepts

•**Information systems** → *Geographic information systems;*

Keywords

Spatial Query Compilation

1. INTRODUCTION

A typical spatial query combines two aspects: data operators, e.g., scan, filter, join, etc. and spatial predicates, e.g., intersects, contains, etc. While the first part is generic and forms the backbone of any data management system, supporting operations on spatial (or geometric) types requires specialized libraries and efficient data access methods. Spatial data are by definition in 2D, 3D, or a higher-dimensional space. Hence, sorting and hashing techniques to implement join operators, for instance, are not applicable. As a result, spatial query processing relies extensively on multi-dimensional indexes to run queries efficiently.

Most spatial query engines are implemented as an extension to an RDBMS, e.g. PostGIS [4], or a map-reduce cluster computing framework, e.g. Spatial Spark and Simba [8, 23]. The spatial predicates are often supported using an external library (e.g., JTS [2] or geos [1]) and the query engine is extended with spatial data types and indexes. Hence, a competitive query evaluation in the underlying database engine is crucial for the performance of spatial queries. In reality, most query engines are interpreter-based (i.e., they run pre-compiled code in contrast to generating specialized low-level target code per query at runtime) and primarily optimized for disk I/O. On the other hand, recent evolution in big memory platforms has eliminated the need to store data in disk. Instead, both data and indexes now fit in-memory. The shift to in-memory databases has revived query compilation, which was originally debuted but then abandoned in System R [10], and brought in ideas from the programming languages (PL) and compilers community to optimize query engines as in HyPer [17] and Legobase [16].

In this vision paper, we present a generative programming approach to support compilation in spatial query engines. We bring in an important result from PL's partial evaluation theory called Futamura Projections [13]. In the first Futamura Projection, specializing the code of an interpreted query engine with respect to a given query is equivalent to a compiled version of that query. Executing compiled code (in contrast to interpreted) is known to be more efficient at runtime. To the best of our knowledge, this work is the first to extend compilation to spatial query engines.

We review the necessary background on how to specialize an interpreted query engine to a compiler in Section 2. In Section 3 we describe LB2-Spatial; a prototype for the first main-memory compiled spatial query engine, discuss challenges in Section 3.1 and present a preliminary experiment, in Section 3.2, to identify potential gains of compilation. We present two use cases on how to support spatial query compilation in Postgres [5] and Spark SQL [9] in Section 4. Finally, Section 5 concludes the paper.

2. BACKGROUND

This section provides the necessary background to comprehend the prerequisites of query compilation: the distinctions between interpreters and compilers, partial evaluation, generative and multi-stage programming.

SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA
 ISBN 978-1-4503-4589-7/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2996913.2996945

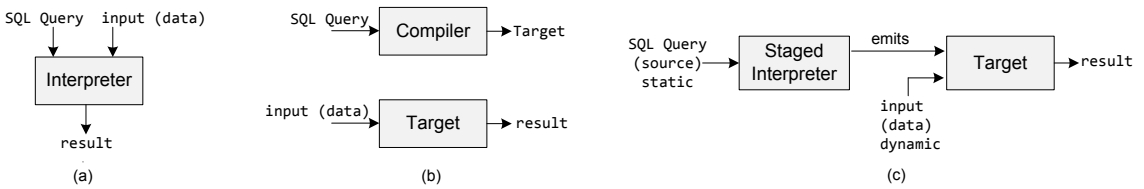

Figure 1: (a) Interpreter (b) Compiler (c) The first Futamura projection

2.1 Partial Evaluation

A plain query *interpreter* first parses a given SQL query into a parse tree, i.e., an Abstract Syntax Tree (AST) of algebraic operators. It then evaluates the parse tree expressions with respect to the input data and produces results all in a single stage, as illustrated in Figure 1a: $result = interpreter(query, input)$. An interpreter is relatively easy to implement but slow at runtime, due to the dispatch overhead of repeatedly looking up which expression to evaluate next. On the other end of the spectrum is *compilation*; a two-phase execution. In the first stage, a source program (i.e. query, without input data) is parsed by a *compiler* and target code is emitted. In the second stage, the generated target code evaluates the input, and directly produces the result. The staged nature of compilers removes interpretation overhead from the runtime and enables applying optimizations on different stages as illustrated in Figure 1b: $target = compiler(query)$, $result = target(input)$. Despite the benefits of compilation, query engines favored interpreters over compilers since I/O was the prime bottleneck, mostly shadowing the compute overhead. It is easier to traverse and evaluate parse trees in a single pass as in the iterator evaluation [14], maximizing pipelining and minimizing disk reads.

Program *specialization* studies the instances where input values are always known then generates an optimized code that applies input as constants instead of passing input as parameters. *Partial evaluation* [15] studies program specialization with respect to input parameters.

The Futamura projections [13] theory is an example of applying specialization on interpreters. The first projection states that a specialized interpreter with respect to input is equivalent to a compiler. Consider Figure 1c, a staged interpreter is a query engine where the code is bootstrapped with special annotations. The staging process enables applying transformation and optimizations in future stages when more information becomes available. The staged interpreter and the static input i.e., query, together comprise a compiler that emits a target code. Next, a dynamic input is evaluated against the target to produce the result.

2.2 Generative and Multi-stage Programming

Generative programming is a form of programming where a program written in high-level abstractions composes another specialized program. That is, instead of writing down a low-level optimized program, generative programming provides customized components, domain-specific metaprograms and optimizations as a library to generate compiled code. Along the same lines, multi-stage programming separates computations into stages based on the frequency of execution and information availability.

LMS (*Lightweight Modular Staging*) [19] is a multi-stage

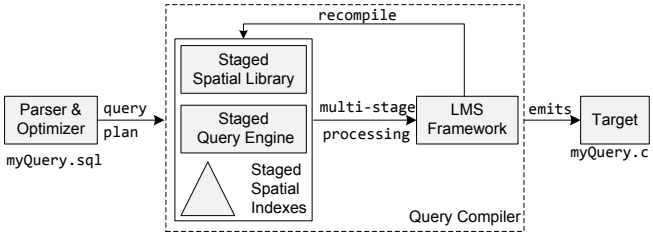

Figure 2: The architecture of a compiled spatial query engine.

generative programming framework that provides runtime compilation and code generation in Scala. At the core of LMS is a graph-like intermediate representation (IR) that encodes constructs and transformations. LMS provides a high-level interface to add and manipulate existing and user defined IR nodes. The LMS approach leverages the Scala type system to distinguish future execution stages (from the present stage) where further optimizations can be applied. A special type constructor called `Rep[T]` (where T is a type e.g., Integer) is introduced to tag expressions computed in a future stage. In other words, present-stage computations are executed right away and future stage expressions `Rep[T]` are inserted into an IR graph to be optimized in subsequent optimization passes. The following code shows a piece of our staged query interpreter's code. The `Rep` constructor is applied only to the fields of `Record` objects. Thus, the main interpreter function `execOp` will be specialized with respect its input, a tree of query operators `op` of type `Operator`, and a callback function `yld` which is to be invoked for each record. Thus, according to the Futamura Projections, the result of executing `execOp` on a given `op` will be a specialized program that *implements* the manipulations of record fields in low-level code, but has all the overhead of a `Record` abstraction and indirect control flow removed.

```
class Record(fields: List[Rep[String]], schema: List[String])
def execOp(o: Operator)(yld: Record => Unit) = o match {
  case Scan(name, schema, delim, externalSchema) => ...
  case Join(left, right) => ...
  ...
}
```

Finally, LMS applies compiler level optimizations e.g., function inlining, dead code elimination, loop unrolling, etc. before generating optimized code either in Scala or C.

3. LB2-SPATIAL: A COMPILED SPATIAL QUERY ENGINE

We envision a generative programming approach to support compilation in spatial query engines. The prototype

described in this section is based on LB2 [3]; a compiled main-memory query engine forked from Legobase [16]. The distinction between LB2 and Legobase lies in the implementation of push-based evaluator and low-level compilation details that are out of this paper's scope. Figure 2 gives a high-level architecture of a compiled spatial query engine. First, the query parser and optimizer are extended to identify spatial constructs and produce a feasible execution plan. Next, the spatial library functions, query evaluator and spatial indexes (e.g., R tree, k-d tree, Quad-tree, etc.) are annotated with `Rep` types to enable staging. The LMS framework constructs the IR graph and initiates a number of consecutive transformation passes. In each pass, all of the available predefined or user-defined optimizations are applied and a new IR is generated. Towards the end of each pass, the set of available optimizations are re-examined. Finally the LMS compiler generates compiled query code i.e., optimized C code and invokes just-in-time JIT compiler to compile C sources into executable target. The query input i.e., data is evaluated against the target code.

Compilation offers mechanisms to eliminate unnecessary code (e.g. expensive validation checks, configuration variables, etc.) that query engines perform during execution. For instance, consider building an R-tree for points (in contrast to minimum bounding rectangles MBRs), in this case, points are treated as rectangles adding extraneous boundary checks while processing each point. LMS reduces evaluation overhead by re-compiling engine codes effected at runtime.

In summary, compilation benefits spatial query engines with improving the performance of backbone data operators and further optimizing spatial operations and indexing.

3.1 Challenges

In this section, we highlight some of the challenges encountered while extending compilation to main-memory spatial query engines e.g., code generation approach, spatial indexing and diverse programming models.

3.1.1 Code Generation and Optimizations

The choice of code generation mechanism is critical to realize compilation inside query engines while lowering development overhead. For instance low-level generators e.g., Low-Level Virtual Machine (LLVM) has been utilized to in query compilation [17, 12]; however, low-level code generation approaches are difficult to implement and may hinder productivity. The multi-stage generative programming in this work follows the *direct approach* (also called *cogen by hand* [11]) where a programmer is involved in the loop to specialize engine code with respect to input. The core specialization task is relatively easy [18] yet, the programmer's domain knowledge can be useful in reasoning which subset of parameters to multi-stage and whether IR level operations should be defined.

Additional challenges on the compilation frameworks level include handling the *phase-ordering problem*; where there are several ordering for applying database and PL optimizations. Furthermore, mitigating the performance cost on calling un-staged code (e.g, external spatial libraries) and the choice of generated code e.g., Java bytecode or low-level C.

3.1.2 Spatial Indexing

Spatial indexes make up the core of spatial query engines. Spatial queries employ several types of indexing depending on the nature of query e.g., R-trees are superior for indexing MBRs and k-d trees are suitable for range and KNN. Compiling nonlinear data structures imposes several data layout and memory management challenges. For instance, knowing the data structure size a priori at compilation time helps with allocating the right amount of memory, except, this is not always the case. Furthermore, trees should be flattened as linear arrays. Hence, careful data referencing is crucial to avoid unnecessary levels of indirection. Interestingly, under certain conditions, a well-implemented data structure e.g., grid may outperform spatial trees [20].

3.1.3 Supporting Diverse Programming Models

The generative programming approach, as discussed so far, enables compiling spatial queries on single-threaded platforms. However, spatial computations are becoming massively parallel and there is an interest in supporting distributed and parallel platforms e.g., NUMA, GPU, Spark, etc. Without loss of generality, the core challenge is embodied in generating a portable spatial compiled query code that can run on multiple platforms.

General-purpose Domain-Specific Languages (DSLs) compiler frameworks e.g., Delite [22] provides DSLs e.g., OptiQL to develop programs that can be compiled into multiple portable targets. Extending spatial primitives and indexing to OptiQL, for instance, is one path to extend spatial compilation into diverse programming models.

3.2 A Preliminary Experiment

We conduct a preliminary experiment in order to gain insights about how the performance of compiled spatial queries compare to optimized code. We implement R-tree index and window join query (in around 300 lines of high-level Scala code) inside LB2-Spatial. Consider example Query 1 that performs index window join query on tables `T1`, `T2` with schema `[<id:Integer>, <P:Point>]`. A spatial index is built on `T2.P`. We perform the query in (i) PostGIS [4]; a generic spatial RDBMS and (ii) a specialized spatial indexing framework [7, 21] (SIF for brevity) that implements in-memory spatial join where the code is pre-compiled, hand-written and hand-optimized in C++.

Query 1.

```
select id1, id2
from T1, T2
where ST_Contains(ST_MakeBox2D (
      ST_Point( ST_X(T1.p)-xDel, ST_Y(T1.p)-yDel),
      ST_Point( ST_X(T1.p)+xDel, ST_Y(T1.p)+yDel)),
      T2.point)
```

The absolute runtime of window join query (excluding index building time) is 280ms in PostGIS, 12.09ms in SIF and 1.4ms in LB2-Spatial. Therefore, realizing spatial compilation has the potential to bridge the performance gap between spatial query engines and specialized frameworks.

4. EXAMPLES

Realizing a fully compiled spatial query engine is the ultimate goal of this research. Still, there is a dire need to extend spatial query compilation into existing systems. In this section we discuss how our vision is extensible; we describe viable approaches to support spatial query compilation in Postgres and Spark SQL.

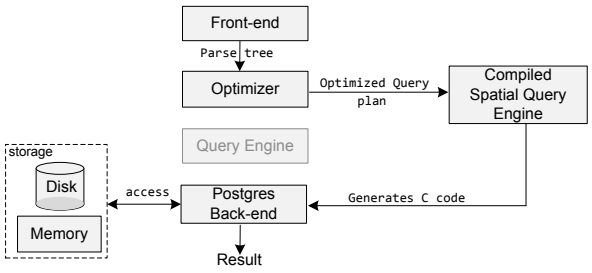

Figure 3: Supporting compiled spatial queries in Postgres.

4.1 Postgres: Traditional RDBMS

Postgres is a well-known traditional disk-based RDBMS. PostGIS supports spatial data management on top of Postgres and has become a popular choice for many GIS applications. Although Postgres/PostGIS is a disk-based system, it can benefit from compilation on warm runs when data is already in memory buffers.

In our vision, we extend a proposal to support spatial compilation in Postgres. Recall, Postgres is written in C and adopts the iterator-based evaluation. Our approach illustrated in Figure 3 keeps the front-end and optimizer intact, the optimized query plan is exported into a compiled spatial query engine (e.g., LB2-Spatial) where the query is compiled into low-level C code. In other words, we replace Postgres's query engine with LB2-Spatial. The generated code can be loaded into Postgres as dynamic library and use internal Postgres APIs to communicate with the existing infrastructure and access stored data. Several internal database details e.g., transaction management, recovery etc. will need to be realized inside LB2-Spatial.

4.2 Spark and Spark SQL

Spark SQL [9] is a declarative dataframe API and query optimizer that operates on top of Spark's Resilient Distributed Datasets (RDDs) [24] cluster computing framework. Several works extend Spark SQL with spatial data support [8, 23]. Spark SQL employs a form of query compilation [6] within its kernel, however, spatial extensions (i.e., types, operators and indexes) and Spark libraries remain pre-compiled and therefore inaccessible for Spark's built-in code generation. A full spatial query compilation support involves extending compilation into two kernels; Spark and Spark SQL in addition to spatial extensions whether embedded or external.

Spark RDDs/ Spark SQL are written in Scala and hence the generative programming approach using LMS is readily applicable. A possible limitation is Spark's underlying RDD execution model, which is based on map-reduce collections operations, and might limit performance for spatial workloads. However, code generation could be extended all the way through the RDD level, replacing Spark RDDs with an equivalent compiled backend. Equipping general-purpose compiler framework for embedded DSLs e.g., Delite [22] with spatial capabilities is promising avenue not only for Spark but also for massively parallel programming models.

5. CONCLUSIONS

We examine the state of spatial computing and analytics in main-memory platforms and recognize that spatial query engines are not leveraging the power of modern hardware. In this vision paper, we bring in ideas from the PL community to support compilation in spatial query engines. We describe LB2-Spatial; a prototype for the first compiled spatial query engine that adopts a generative programming approach and employs Lightweight Modular Staging (LMS). We further explain how to extend compilation into Postgres and SparkSQL.

Acknowledgments

We thank the anonymous reviewers for helpful pointers. This research was supported through NSF CAREER award 1553471 and NSF156420.

6. REFERENCES

[1] Geos-geometry engine, open source. https://trac.osgeo.org/geos.
[2] Jts topology suite. http://www.vividsolutions.com/jts/JTSHome.htm.
[3] Lb2. https://github.com/TiarkRompf/legobase-micro.
[4] Postgis. http://postgis.org.
[5] Postgresql. https://www.postgresql.org.
[6] Project tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal. htmlbigreddata/spatial-indexing.
[7] Spatial indexing at cornell. http://www.cs.cornell.edu/bigreddata/spatial-indexing.
[8] Spatialspark. https://github.com/syoummer/SpatialSpark.
[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD*, pages 1383–1394. ACM, 2015.
[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.
[11] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 198–214. Springer, 1994.
[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *Proceedings of the VLDB Endowment*, 8(12):1466–1477, 2015.
[13] Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
[14] G. Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.
[15] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.
[16] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.
[17] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
[18] T. Rompf and N. Amin. Functional pearl: a sql to c compiler in 500 lines of code. *Acm Sigplan Notices*, 50(9):2–9, 2015.
[19] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.
[20] D. Šidlauskas and C. S. Jensen. Spatial joins in main memory: Implementation matters! *Proceedings of the VLDB Endowment*, 8(1):97–100, 2014.
[21] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proceedings of the VLDB Endowment*, 6(14):1882–1893, 2013.
[22] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.
[23] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. 2016.
[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX*, HotCloud'10, 2010.