ARCHITECTING QUERY COMPILERS FOR DIVERSE WORKLOADS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ruby Y. Tahboub

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Tiark Rompf, Chair

Department of Computer Science

Sunil Prabhakar

Department of Computer Science

Milind Kulkarni

School of Electrical and Computer Engineering

Christopher W. Clifton

Department of Computer Science

Approved by:

Voicu Popescu

Head of the Departmental Graduate Program

To my daughter Yasmeen

ACKNOWLEDGMENTS

It gives me great pleasure to express my gratitude to a large number of people who contributed to my success over the years of my Ph.D. journey.

I would like to express my gratitude to my advisor Prof. Tiark Rompf for introducing me to his exciting research area, helping me refining my work and becoming a better scholar. I enjoyed working with Tiark, and will always be thankful for his insights and advice. I am also grateful to my former advisor Prof. Walid Aref who helped me in my early Ph.D. years and taught me to appreciate good systems research. I specially like to thank Grégory Essertel for our successful collaborations. Special thanks to my colleagues in our research group: Fei Wang, Xilun Wu, James Decker and Guannan Wei.

I would like to extend my sincere thanks to my advisory and examining committee Prof. Chris Clifton, Prof. Milind Kulkarni, and Prof. Sunil Prabhakar. Many thanks are due to all of the Computer Science staff who work tirelessly to support our daily operations. My gratitude and appreciation to Dan Trinkle and his technical support team, Tammy Muthing and the business office team, Pam Graf for giving us the best work environment, and Pat Morgan for planning great events.

My sincere gratitude goes to my parents who raised me to value education and never quit. I am thankful for my supportive husband, Mohammad Abu Khater. Finally, most thanks go to my (almost two years) daughter Yasmeen who teaches me something new every single day!

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Tahboub, Ruby Y. PhD, Purdue University, May 2019. Architecting Query Compilers for Diverse Workloads . Major Professor: Tiark Rompf.

To leverage modern hardware platforms to their fullest, more and more database systems embrace compilation of query plans to native code. In the research community, there is an ongoing debate about the best way to architect such query compilers. This is perceived to be a difficult task, requiring techniques fundamentally different from traditional interpreted query execution. In this dissertation, we contribute to this discussion by drawing attention to an old but underappreciated idea known as *Futamura projections*, which fundamentally link interpreters and compilers. Guided by this idea, we demonstrate that efficient query compilation can actually be very simple, using techniques that are no more difficult than writing a query interpreter in a high-level language. We first develop LB2: a high-level query compiler implemented in this style that is competitive with the best compiled query engines both in sequential and parallel execution on the standard TPC-H benchmark.

Query engines process a variety of data types and structures including text, spatial, graphs, etc. Several spatial and graph engines are implemented as extensions to relational query engines to leverage optimized memory, storage, and evaluation. Still, the performance of these extensions is often stymied by the interpretive nature of the underlying data management, generic data structures, and the need to execute domain-specific external libraries. On that basis, compiling spatial and graph queries to native code is a desirable avenue to mitigate existing limitations and improve performance. To support compiling spatial queries, we extend the LB2 main-memory query compiler with spatial predicates, indexing structures, and spatial operators. To support compiling graph queries, we extend LB2 with graph data structures and

operators. The spatial extension matches the performance of hand-written code and outperforms relational query engines and map-reduce extensions. Similarly, the graph extension matches, and sometimes outperforms, low-level graph engines.

1. INTRODUCTION

The era of modern hardware and big data analytics has changed the dynamics in which data is processed and managed. Main-memory platforms have successfully eliminated slow disk access and exposed compute performance as a new bottleneck, aggravated by the decline of Moore's law. In particular when it comes to relational database systems, the "Volcano" iterator evaluation model [1], which is the prevalent design of query engines due to its simplicity and expressiveness, falls short under these new dynamics due to its excessive use of function calls to dispatch individual records of data from producers to consumers in a query plan. Query compilation, i.e., generating specialized low-level code for a given query, is emerging as a necessity for data processing systems to eliminate the interpretive overheads of existing designs and to keep up with new environmental constraints.

1.1 Problem Statement

A typical relational database management system (RDBMS) processes incoming queries in multiple stages (Figure 1.1.1): In the front-end, a parser translates a given SQL query into a logical plan. The query optimizer rewrites this plan into a more efficient form based on a cost model and emits a physical plan ready for evaluation. The back-end is usually an interpreter that executes the optimized plan over the stored data, operator by operator, producing record after record of the computed query result.

Taking a slightly broader view, a database system fits – almost literally – the textbook description of a compiler: parser, front-end, optimizer, and back-end. The one crucial difference is that the last step, generation of machine code, is typically missing in a traditional RDBMS.

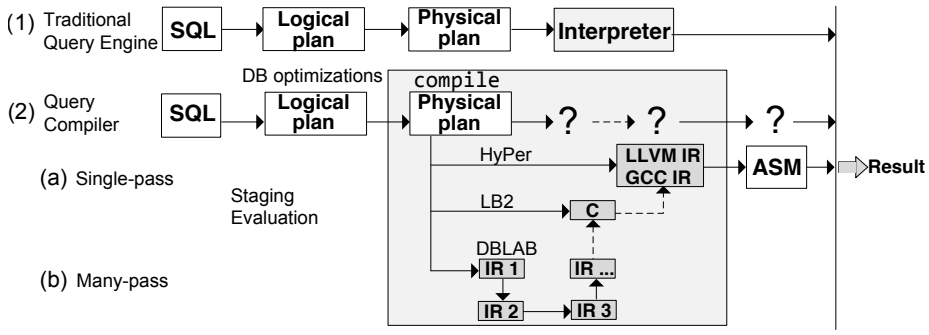


Fig. 1.1. Illustration of (1) query interpreter (2) query compilers (a) single-pass compiler (b) many-pass compiler.

For the longest time, this was a sensible choice: disk I/O was the main performance bottleneck. Interpretation of query plans provided important portability benefits, and thus, engineering low-level code generation was not perceived to be worth the effort [2]. But the circumstances have changed in recent years: with large main memories and storage architectures like Non-volatile Memory (NVM) on the one hand, and the demand for computationally more intensive analytics, on the other hand, query processing is becoming increasingly CPU-bound. As a consequence, query engines need to embrace full compilation to remain competitive, and therefore, compiling query execution plans (QEPs) to native code, although in principle an old idea, is seeing a renaissance in commercial systems (e.g., Impala [3], Hekaton [4], Spark SQL [5], etc.), as well as in academic research [6–8]. Given this growing attention, the database community has engaged in an ongoing discourse about how to architect such query compilers [6, 7, 9]. This task is generally perceived as hard and often thought to require fundamentally different techniques than traditional interpreted query execution. The most recent contribution to this discourse from SIGMOD'16 [9] states that "it is fair to say that creating a query compiler that produces highly optimized code is a formidable challenge," and that the "state of the art in query compiler construction is lagging behind that in the compilers field."

1.1.1 Spatial Workloads

Modern location-based applications rely extensively on fast spatial processing. Large parts of spatial workloads runtime – whether evaluated as an extension to a relational or clustered back-end – is spent on processing spatial data types, evaluating spatial predicates, and traversing spatial indexing structures. The high-level implementation of these spatial structures increases the interpretive overhead of processing spatial workloads, which also increases latency and lowers throughput. Hence, to improve the performance of spatial applications, spatial computing systems must specialize indexing structures, and embrace runtime compilation to generate native code as in state-of-the-art relational engines.

The core of realizing efficient spatial computing system lies in optimizing spatial data structures and predicates evaluation, (i.e., minimizing the per-record access and processing time). Spatial data types are in the two-dimensional space or higher. Thus, spatial extensions use spatial libraries for predicate computations and build spatial indexing structures to facilitate data storage and access. However, the code of external libraries is opaque to the spatial back-end and incurs runtime overhead in expensive function calls. Similarly, spatial indexes are implemented in high-level form using the in-data management structures that unify various types of structures behind a common interface, e.g., SP-GiST [10] in PostgreSQL/PostGIS. Overall, the interpretive overhead of spatial processing increases the latency of spatial workloads and lowers the overall system throughput.

1.1.2 Graph Workloads

It is often desirable for graph and relational data to co-exist and be processed together, which naturally suggests representing graphs as relations on top of an existing RDBMS. The unique advantage of such an approach is that the core data management operations are all provided by the RDBMS and a graph extension would only need to implement front-end algorithms and functionality. However, the execution

pattern of graph workloads is often dominated by long-running loops over the graph structure where each iteration performs computations on the adjacency of some vertices (e.g., set intersection during a triangle counting operation). The performance of relational graph extensions is therefore often stymied by the interpretive nature of typical relational engines and the lack of specialized data structures for adjacency relations. Instead, internal RDBMS data structures are often implemented in a generic form to unify various types of data layouts behind a common interface.

To tackle the challenges of relational graph extensions, several stand-alone graph engines have been developed (e.g., high-level Neo4j [11], low-level Snap-Ringo [12]) that process graphs in native adjacency structures. While graph processing in stand-alone systems is more performant and also often more expressive in describing graph operations (e.g., shortest paths, centrality, etc.) than relational queries, there exists a high development cost in terms of core data management and loss of interoperability with front-end and back-end systems that integrate with RDBMS.

Realizing efficient relational graph processing requires combining relational evaluation with specialized graph structures and operations, which is an uphill battle due to the difficulty of modifying the internals of a mature RDBMS (typically several million lines of code). Besides, the key performance challenge in RDBMS is the interpretive overhead associated with processing data in high-level form (e.g., generic libraries for hash maps) rather than generating optimized low-level code (as precisely described in Neumann's work [6]). Even supporting a minimal operational compiled path for graph queries entails writing thousands of lines of low-level code (e.g., programmatic LLVM API) that permeates large parts of the query engine code. *En masse*, the complexity of extending RDBMS with graph processing and compilation does not add up linearly; in fact, it *multiplies!*

This dissertation emphasizes that i) building highly efficient query compilers can be simple and elegant. Neither low-level coding nor the added complexity of multiple compiler passes are necessary. ii) The underlying idea of constructing simple but

highly efficient query compilers not only applies to purely relational queries but carries over to diverse workloads including spatial and graph queries.

1.2 Overview

1.2.1 Revisiting How to Architect Query Compilers

Large-scale data processing owes much of its success to declarative query languages. SQL and its cousins enable succinct statements of intent, at an abstraction level that allows pervasive and automatic optimizations of query terms before touching any data. Thus, data processing is fundamentally a language and compiler problem.

Query compilation itself is not a new idea. Historically, the very first relational database, IBM's System R [2], was initially designed around a form of templated code generation. However, before the first commercial release, the code generator was replaced by interpreted execution [13], since at the time, the benefits of code generation were outweighed by its complexity and issues such as code portability, cost of maintenance, and prevailing I/O intensive workloads. Indeed, compiling code for query evaluation pipelines (QEPs) is a non-trivial task. First, code generation mechanisms need to consider database-level optimizations, compiler-level optimizations and handling non-traditional data types. Second, code generators need to be extensible and expressive. Third, the generated code should be sufficiently portable, i.e., be easily mapped to a variety of target platforms.

Most emerging query compilers are either simplistic template expanders or interact with low-level frameworks like LLVM [14] that provides an even lower-level entry point into an existing compiler toolchain. But how do we get from physical query plans to this level of executable code? This is, in fact, the key architectural question. HyPer [6] uses the programmatic LLVM API and achieves excellent performance, at the expense of a rather low-level implementation that permeates large parts of the engine code. LegoBase [7] and DBLAB [9] are engines implemented in a high-level language Scala that generates efficient C code. The latter two systems add significant complexity in

the form of multiple internal languages and explicit transformation passes, which the authors of DBLAB [9] claim are necessary for optimal performance.

In Chapter 2, we revisit how query compilers are being architected and demonstrate that neither low-level coding nor the added complexity of multiple compiler passes is necessary. We present a principled approach to derive query compilers from query interpreters and show that these compilers can generate excellent code in a single pass. We present LB2, a new query compiler developed in this style that is competitive with HyPer and DBLAB.

1.2.2 Supporting Compilation in Spatial Query Engines

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 [15], or a map-reduce cluster computing framework, e.g., GeoSpark [16] and Simba [17]. The spatial predicates are often supported using an external library (e.g., JTS [18] or Geos [19]) 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 Chapter 3, we discuss compiling spatial queries into low-level code. The keys challenges are spatial query evaluation performs expensive spatial predicates and relies on efficient traversal of indexing structures. The key idea to address these challenges is to facilitate building optimized data structures using *programmatic specialization*, the

same technique LB2 already uses for generating efficient code for relational queries. In particular, we extend LB2 with spatial predicates, spatial indexing structures, and spatial operators.

1.2.3 Compiling Graph Queries

The increasing demand for graph query processing has prompted the addition of support for graph workloads on top of standard relational database management systems (RDBMS). Although this appears like a good idea — after all, graphs are just relations — performance is typically suboptimal since graph workloads are naturally iterative and rely extensively on efficient traversal of adjacency structures that are not typically implemented in RDBMS. Adding such specialized adjacency structures is not at all straightforward, due to the complexity of typical RDBMS implementations. The iterative nature of graph queries also practically requires a form of runtime compilation and native code generation which adds another dimension of complexity to the RDBMS implementation, and any potential extensions.

In Chapter 4 we demonstrate how the idea of the first Futamura projection, which links interpreted query engines and compilers through specialization, can be applied to compile graph workloads in an efficient way that simplifies the construction of relational engines that also support graph workloads. We extend the LB2 main-memory query compiler with graph adjacency structures and operators. We implement a subset of the `Datalog` logical query language evaluation to enable processing graph and recursive queries efficiently.

1.3 Contributions

The key research contributions of this dissertation are summarized as follows:

1. In the spirit of explaining key compilers results more clearly, we draw attention to an important but not widely appreciated concept known as Futamura projections, which fundamentally links interpreters and compilers through *spe-*

cialization. We propose to use the first Futamura projection as the guiding principle in the design of query compilers (Chapter 2).

2. We show that viewing common query evaluator architectures (pull-based, Volcano; and push-based, data-centric) through the lens of the first Futamura projection provides key insights into whether an architecture will lead to an efficient compiler. We use these insights to propose a novel data-centric evaluator model based on callbacks, which serves as the basis for our LB2 engine (Chapter 2).

3. We discuss the practical application of the first Futamura projection idea and show how to derive high-level and efficient query compilers from a query interpreter. We implement a range of optimizations in LB2. Among those are row-oriented vs. column-oriented processing, data structure specialization, code motion, parallelization, and generation of auxiliary index structures including string dictionaries. The set of optimizations in LB2 includes all those implemented in DBLAB [9], and some *en plus* (e.g., parallelization). But in contrast to DBLAB, which uses up to five intermediate languages and a multitude of intricate compiler passes, LB2 implements all optimizations in a single generation pass, using nothing but high-level programming (Chapter 2).

4. We support parallelism in LB2, LB2-Spatial, and LB2-Graph using per-thread data structures in OpenMP (Chapters 2-4).

5. We compare the performance of LB2 with HyPer and DBLAB. We show that LB2 is the first system implemented in a high-level language that approaches HyPer performance in a fully TPC-H compliant setting, both sequential and on up to 16 cores. By contrast, LegoBase [7] and DBLAB [9] do not support parallelism, and they need to resort to non-TPC-H-compliant indexing and precomputation to surpass HyPer on TPC-H queries. With all such optimizations turned on, LB2 is competitive with DBLAB as well (Chapter 2).

6. We support spatial query compilation in LB2. Moreover, we describe staging spatial predicates, specializing indexing structures, implementing spatial operators and compiling spatial applications (Chapter 3).

7. We support graph query compilation in LB2. We describe specializing graph data structures and graph operators. Furthermore, we implement a subset of the `Datalog` logical query language to express graph queries succinctly (Chapter 4).

8. We compare the performance of the spatial extension with handwritten code, spatial RDBMS PostGIS [15] and a Spark spatial extension Simba [17]. We also compare the performance of the graph extension with Snap [12] and EmptyHeaded [20] graph engines (Chapter 4).

The key idea in this dissertation is the practical realization of the first Futamura projection, which essentially states that the ability to specialize a query interpreter to a given query is identical to a query compiler. Guided by the first Futamura projection idea, we develop the LB2 query compiler presented in Chapter 2. LB2 is a fully compiled query engine that performs on par with, and sometimes beats the best compiled query engines on the standard TPC-H benchmark. Specifically, LB2 is the first query engine built in a high-level language that is competitive with HyPer [6], both in sequential and parallel execution. LB2 is also the first single-pass query engine that is competitive with DBLAB [9] using the full set of non-TPC-H-compliant optimizations. In Chapters 3-4, we demonstrate that the underlying idea of constructing simple but highly efficient query compilers not only applies to purely relational queries but carries over to diverse workloads including spatial and graph workloads. In Chapter 3, we extend LB2 with spatial compilation by adding spatial data types, predicates, indexing structures and spatial operators. In Chapter 4, we extend LB2 with graph query compilation by adding graph adjacency structures and graph operators.

The LB2 query compiler (Chapter 2) was implemented by Grégory Essertel. Moreover, the trie data structure and trie-based triangle count operator (Chapter4) was implemented by Xilun Wu.

1.3.1 Publications

The work in this dissertation is based on the following peer-reviewed published papers and a technical report.

1. **Ruby Y. Tahboub**, Grégory M. Essertel, Tiark Rompf. *How to Architect a Query Compiler, Revisited*, in the Proceedings of the 2018 ACM International Conference on Management of Data ACM (SIGMOD'18).

2. **Ruby Y. Tahboub**, Tiark Rompf. *On Supporting Compilation in Spatial Query Engines (Vision Paper)*, in the Proceedings of the 2016 ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL GIS'16).

3. **Ruby Y. Tahboub**, Tiark Rompf. *How to Architect a Query Compiler for Spatial Workloads*, Technical Report, 2019.

4. **Ruby Y. Tahboub**, Xilun Wu, Grégory M. Essertel, Tiark Rompf. *Towards Compiling Graph Queries in Relational Engines*, to appear in the 2019 International Symposium on Data Base Programming Languages (DBPL'19).

Specifically, the relational query compilation work in Chapter 2 is based on the first publication. The spatial query compilation work in Chapter 3 is based on the second publication and the third technical report. Lastly, the graph compilation work in Chapter 4 is based on the fourth publication.

1.4 Related Work

Compiled Query Engines. The recent changes in architecture and the push towards main memory databases have revived interest in developing efficient query compilation methods. The timeline in Figure 1.2 shows a selection of query engines that employ a form of query compilation since System R. The illustration broadly classifies the compilation method used. An arrow from A to B denotes that system B is built on or extends system A. Daytona [21] is a propriety system developed by AT&T that generates query code which enables running large queries efficiently. Rao et al. [22] compiled queries to Java bytecode by removing virtual functions from iterator evaluation. HIQUE [23] performs query plan level optimizations and uses templates to generate code. The HyPer [6] query compiler is a pivotal work that presented the data-centric evaluation model that enables generating efficient code for main-memory databases. Another important line of works employed the lightweight modular staging platform (LMS) [24] to compile domain-specific languages (e.g., SQL, machine learning, linear algebra, etc.) to low-level code using a high-level language as in Delite [25, 26] and its DSLs OptiQL, OptiML [27] and OptiMesh. Similarly, Legobase [7], the "SQL to C in 500 lines" compiler [28], Flare [29], a native compiler back-end for SparkSQL, LB2-Spatial [30] and the LB2 [31] single-pass query compiler generates code using LMS. Moreover, DBLAB [9] uses multi-pass DSL transformations, DBToaster [32] supports compiling recursive delta view queries. Tupleware [8] compiles user-defined functions. Voodoo [33] compiles portable query plans that can run on CPUs and GPUs. Weld [34] provides a common runtime for diverse libraries, e.g., SQL and machine learning. Butterstein et al. [35] compiles query subexpressions into machine code in Postgres. Similarly, the work in [36] uses program specialization and LLVM to generate query code in Postgres. To address the issue of processing raw heterogeneous data RAW [37] uses compilation to generate scanning code based on the type of raw data it is loading, H2O [38] generates code for layout-aware access operators and ViDa [39] compiles data structures and query operators. Tung-

sten in SparkSQL [5] generates Java code. Commercial examples are Hekaton [4], DryadLINQ [40], Impala [3] and Spade [41].

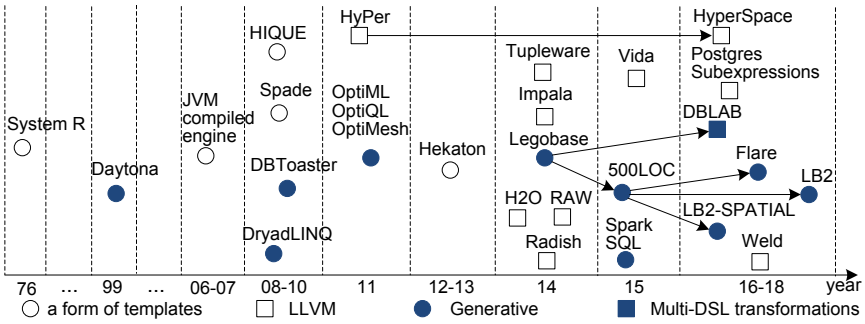


Fig. 1.2. The evolution of query compilation.

Spatial Processing Several well-known relational databases are extended with indexing structures, spatial types, etc. to support spatial processing. For instance, PostGIS [15] uses SP_GiST's [10] R-tree, Oracle Spatial [43] adds QuadTree, Microsoft SQL [44] adds a hierarchical grid index, IBM DB2 [45] uses a grid index where each cell is indexed using a btree. However, slow data loading remains a major disadvantage in relational engines specially spatial datasets are typically large. AT-GIS [46] is a single-node parallel spatial processing system that integrates parsing and spatial query processing using the proposed associative transducers (ATs) computational abstraction. MonetDB [47] (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, large join queries are suboptimal since it requires maintaining the entire dataset in memory. GeoCouch [48] is a NoSQL spatial processing engine. A comprehensive overview on spatial indexing structures are surveyed in [49, 50].

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 [51] did not use an index. SpatialHadoop [52] 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 [53] supports SJMR in addition to a specialized pathology image analysis module. Parallel SECONDO [54] integrates Hadoop with the SECONDO [55] database that supports spatial processing. GeoMesa [56] supports indexing and querying of spatiotemporal data on Accumulo [57]. MD-HBase [58] extends HBase [59] (key-value store) with multidimensional indexes to support range and kNN queries. Accumulo and HBase are based on Google's BigTable [60]. 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 [5] computing framework has become popular for its main-memory execution model and expressive front-end. GeoSpark [16], SparkGIS [61], Stark [62], LocationSpark [63], Simba [17], Magellan [64], SpatialSpark [65] 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. [66] presented a multiway spatial join algorithm with Spark (MSJS). However, the performance of Spark extensions inherits Spark's internal bottlenecks in main-memory processing, the overhead of distributed datasets (RDDs) operations, and communication through Spark's runtime system [29]. The spatial extension in this work overlaps the operations in these systems without incurring runtime overhead (i.e., only generates code for evaluation).

Relational Graph Processing In SQLGraph [67] and Grail [68] graphs are stored using relational tables. A syntactic layer translates graph queries from domain-specific API to procedural SQL. Integrating graph processing inside RDBMS greatly benefit relational-graph workloads. Graphite [69], SAP HANA Graph [70] and GRFusion integrate graph processing inside RDBMS. GRFusion processes graphs natively as views. The previous systems focus on efficient graph traversals.

Single-machine Shared-memory Ligra [71] and Galois [72] are shared-memory graph frameworks where Ligra is known for switching between different implementations of operators and Galois is known for its task scheduler. Examples of semi-external memory and SSD engines that apply techniques as the sliding window algorithm to minimize disk I/O are GraphChi [73], X-Stream [74] and Grid-Graph [75]. EmptyHeaded [20] compiles graph join queries with strong theoretical guarantees. Lux [76] leverages multi-GPUs for efficient graph processing.

Specialized and Distributed Specialized graph engines implement native graph model with a rich set of graph operations. Titan [77] and Neo4j [11] are stand-alone graph engines that provide rich graph queries, e.g., paths, subgraphs, etc. Ringo [78] and GraphGen [79] convert relational graph data to a native graph representation. On scaled-out graph analytics, Teradata Aster [80] provides a SQL front-end where graph functions are executed using a specialized graph engine. GraphBuilder [81] is a Hadoop-based graph construction library that extracts graphs from unstructured data. Giraph [82] is a Hadoop-based iterative graph processing system. GraphLab [83] is an asynchronous parallel framework where evaluation is based on a Gather-Apply-Scatter (GAS) model and supports operates on both distributed and shared-memory platforms. Powergraph [84] is a distributed GraphLab. Naiad [85] is a main-memory distributed data-flow computation system that supports graph streaming algorithms. Graph-X [86] is Spark-based graph system that provides a graph abstraction layer for graph computations using relational operators such as join and group-by.

Graph Query Languages are surveyed in [87]. Recent examples include Oracle's PGQL [88], Cypher [89], and GCore [90]. Green-Marl [91] is a DSL for expressing graphs algorithms intuitively. The Green-Marl compiler generates efficient graph operations code. Examples of `Datalog` evaluation engines SociaLite [92] developed for social networks analysis, LogicBlox [93], Soufflé [94] implements program specialization techniques to compile `Datalog` programs, BigDatalog [95] is a Spark-based distributed `Datalog` engine. The Datalography engine [96] is built on the top of Giraph,

and Myria [97] to support graph analytics. G-SPARQL [98] is a SPARQL language for querying RDF graphs.

Graph Mining Numerous graph systems focus on graph mining tasks, e.g., Motif Counting (MC), Frequent Subgraph Mining (FSM), etc. Arabesque [99] provides a graph exploration model referred to as *embedding*. In each exploration round, the existing embeddings are expanded. The embeddings are refined using a filter operation. ScaleMine [100] is a parallel FSM system. DistGraph [101] is a distributed mining system that optimizes communication costs. RStream [102] is a single machine graph mining system that extends the GAS evaluation model. G-thinker [103] offers an API for graph mining algorithms.

1.5 Hypothesis

Our proposed thesis hypothesis states that *it is possible to efficiently compile relational, spatial and graph queries in high-level language using a single-compiler pass. In particular, we architect a single-pass, multi-stage generative query compiler that can compile diverse query workloads.*

2. REVISITING HOW TO ARCHITECT A QUERY COMPILER

This Chapter is based on the paper *How to Architect a Query Compiler, Revisited* which appeared at the Proceedings of the 2018 ACM International Conference on Management of Data SIGMOD'18 [31].

The performance of a typical relational database management system (RDBMS) on analytic workloads is primarily determined by two facets: the query evaluation paradigm, e.g., iterator style [1] or data-centric [6], and characteristics of the runtime, i.e., whether the back-end code is interpreted, compiled, or a combination. The recent shift to large main-memory platforms has eliminated I/O as the key bottleneck, and exposed compute performance as a new limiting factor for many workloads, which triggered a radical rethinking of query evaluation and runtime architecture.

In this Chapter, we present the LB2 query compiler: a high-level query compiler implemented with using the practical realization of the first Futamuara projection that links interpreters and compilers through specialization. The LB2 query compiler (Chapter 2) was implemented by Grégory Essertel.

2.1 Futamura Projections

In 1970, at a time when the hierarchical and network models of data [104] were *du jour*, Codd's seminal work on relational data processing appeared in CACM [105]. One year later, in 1971, Yoshihiko Futamura published his paper "Partial Evaluation of Computation Process–An approach to a Compiler-Compiler" in the Transactions of the Institute of Electronics and Communication Engineers of Japan [106]. The fundamental insight of Codd was that data could be profitably represented as high-level relations without explicit reference to a given storage model or traversal strategy. The

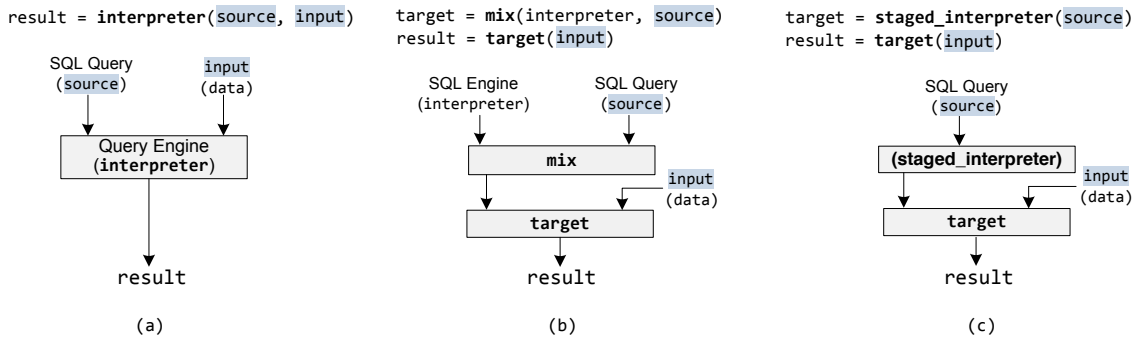


Fig. 2.1. (a) Query interpreter (b) applying the first Futamura projection on a query interpreter (c) the LB2 realization of the first Futamura projection.

fundamental insight of Futamura was that compilers are not fundamentally different from interpreters, and that compilation can be profitably understood as *specialization* of an interpreter, without explicit reference to a given hardware platform or code generation strategy.

To understand this idea, we first need to understand the *specialization* of programs. In the most basic sense, this means to take a generic function, instantiate it with a given argument, and simplify. For example, consider the generic two-argument power function (written in `Scala`) that computes x^n:

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

If we know the exponent value, e.g., `n = 4`, we can derive a specialized, *residual*, power function:

```
def power4(x:Int): Int = x * x * x * x
```

This form of specialization is also known as *partial evaluation* [107].

Specializing Interpreters. The key idea of Futamura was to apply specialization to interpreters. Like `power` above, an interpreter is a two-argument function: its arguments are the code of the function to interpret, and the input data this function should be called with. Figure 2.1a illustrates the case of databases: The query engine evaluates a SQL query (static input) and data (dynamic input) to produce the result.

The effect of specializing an interpreter is shown in Figure 2.1b: if we have a program specializer, or *partial evaluator*, which for historical reasons is often called `mix`, then we can specialize the (query) interpreter with respect to a given source program (query). The result is a single-argument program that computes the query result directly on the data and runs much faster than running the query through the original interpreter. This is because the specialization process strips away all the "interpretive overhead," i.e., the dispatch that interpreter performs based on the structure of the query. In other words, through specialization of the interpreter, we are able to obtain a *compiled* version of the given program!

This key result–partially evaluating an interpreter with respect to a source program produces a compiled version of that program–is known as the first Futamura projection. Less relevant for us, the second and third Futamura projections explain how *self-application* of `mix` can, in theory, derive a compiler generator: a program that takes any interpreter and produces a compiler from it.

Codd's idea has been wildly successful, spawning multi-billion-dollar industries, to a large extent thanks to the development of powerful automatic query optimization techniques, which work very well in practice due to the narrow semantic model of relational algebra. Futamura's idea of deriving compilers from interpreters automatically via self-applicable partial evaluation received substantial attention from the research community in the 1980s and 1990s [108], but has not seen the same practical success. Despite partial successes in research, fully automatic partial evaluation has turned out to be largely intractable in practice due to the difficulty of binding-time separation [107]: deciding which expressions in a program to evaluate directly, at specialization time, and which ones to residualize into code.

Programmatic Specialization. Even though the magic `mix` component in Figure 2.1b has turned out to be elusive in practice, all is not lost. We just have to find another way to implement program specialization, perhaps with some help from the programmer. Going back to our example, we can implement a self-specializing power function like this:

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

What is different? We changed the type of `x` from `Int` to `MyInt`, and assuming that we are working in a high-level language with operator overloading capabilities, we can implement `MyInt` as a symbolic data type like this:

```
// symbolic integer type
class MyInt(ref: String) {
  def *(y: MyInt) = {
    val id = freshName();
    println(s"int $id = $ref * ${y.ref};");
    new MyInt(id)
} }
// implicit conversion Int -> MyInt
implicit def constInt(x: Int) =
              new MyInt(x.toString)
```

When we now call `power` with a symbolic input value:

```
power(new MyInt("in"),4)
```

it will emit the desired specialized computation as as a side effect:

```
int x0 = in * 1;    int x1 = in * x0;
int x2 = in * x1;   int x3 = in * x2; // = in * in * in * in
```

In the following, we visualize the intermediate states of generating code for the `power` function discussed earlier. The following code shows the `power` function and `MyInt` class implementation:

```
def power(x:Int, n:Int): Int =
  if (n == 0) 1 else x * power(x, n - 1)
// symbolic integer type
class MyInt(ref: String) {
  def *(y: MyInt) = {
    val id = freshName();
    println(s"int $id = $ref * ${y.ref};");
    new MyInt(id)
} }
// implicit conversion Int -> MyInt
implicit def constInt(x: Int) = new MyInt(x.toString)
```

The `power` function is recursive, with the bottom case reached when `n` becomes zero. Following the standard call-by-value evaluation rules, steps 1-8 show how `power` is expanded for `n = 4, 3,..., 1`.

```
1 ⟦ power(new MyInt("in"), 4) ⟧
2 ⟦ if (4 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 3)) ⟧
3 ⟦ new MyInt("in") * power(new MyInt("in"), 3) ⟧
4 ⟦ new MyInt("in") * (
     if (3 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 2)) ⟧
```

```
5 ⟦ new MyInt("in") * (new MyInt("in") * power(new MyInt("in"), 2)) ⟧
6 ⟦ new MyInt("in") * (new MyInt("in") * (
    if (2 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 1)) ⟧
7 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * power(new MyInt("in"), 1))) ⟧
8 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (
    if (1 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 0)))) ⟧
```

When `power` is invoked with `n = 0`, we reach the bottom of the recursion, i.e., no further function calls will be pushed on the call stack. It is the time to go up and perform the remaining evaluation actions, which will generate code.

```
9 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") *
    power(new MyInt("in"), 0)))) ⟧
10 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") *
     if (0 == 0) 1 else power(new MyInt("in"), -1)))) ⟧
```

The base case of `power` returns the value `1`. Next, the multiplication operator in `MyInt("in") * 1` emits the first line `int x0 = in * 1`, as side effect, and returns `MyInt("x0")`.

```
11 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") * 1))) ⟧
12 int x0 = in * 1
 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * new MyInt("x0"))) ⟧
```

Similarly, the remaining multiplication expressions in steps 13-15 generate the rest of the code.

```
13 int x0 = in * 1
   int x1 = in * x0
 ⟦ new MyInt("in") * (new MyInt("in") * new MyInt("x1")) ⟧
14 int x0 = in * 1
   int x1 = in * x0
   int x2 = in * x1
 ⟦ new MyInt("in") * new MyInt("x2") ⟧
15 int x0 = in * 1
   int x1 = in * x0
   int x2 = in * x1
   int x3 = in * x2
 ⟦ new MyInt("x3") ⟧
```

Once the full code is generated, the result is stored in the variable `x3`, and the final return value will be `MyInt("x3")`.

Binding each intermediate result to a fresh variable guarantees a proper sequencing of operations. Based on this core idea of introducing special data types for symbolic or *staged* computation, we suddenly have a handle on making the first Futamura projection immediately practical. As with `power` and `MyInt` above, we just need to

implement a query interpreter in such a way that it evaluates a concrete query on symbolic input data. The result, illustrated in Figure 2.1c, is a practical and directly implemented realization of the first Futamura projection without `mix`. The remainder of this Chapter will explain the details of this approach and present our query compiler LB2.

2.2 Structuring Query Evaluators

It is important to note that the first Futamura projection itself does not capture any kind of program analysis or optimization. This poses the question how to generate *optimized* code. A key observation is that the shape of specialized code follows the shape of the interpreter that is specialized. Hence, we can derive the following design principle: *By engineering the source interpreter in a certain way, we can control the shape of the compiled code.*

In the following, we review popular query evaluation models with an eye towards specialization, and present our improved data-centric execution model.

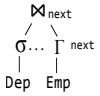
2.2.1 The Iterator (Volcano) Model

The Iterator (Volcano) Model is based on a uniform `open()`, `next()` and `close()` interface for each operator. Figure 2.2a-b shows a QEP, and the operator interface in Volcano [1]. Evaluation starts when the root operator (e.g., hash join) invokes `next()` to probe offspring operators for the next tuple. Subsequent operators (e.g., select) repeatedly invoke `next()` until a scan (or materialized state) is reached. At this point, a tuple is pipelined back to its caller and the operator's code is executed. Thus, the mode of operation can be understood as *pull-based*. Although the iterator model is intuitive and allows pipelining a stream of tuples between operators, it incurs significant overhead in function calls, which one might hope to eliminate using compilation.

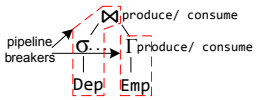
```
// Data schema: Dep(dname: String, rank: Int), Emp(eid: Int, edname: String)

select * from Dep, (select edname, count(*) from Emp group by edname) as T
where rank < 10 and dname = T.edname
```

```
Print(
  HashJoin(
    Select(Scan("Dep"))
        (x => x("rank") < 10),
    Agg(Scan("Emp"))
        (x => x("edname"))(0)
          ((agg,x) => agg + 1)))
```

(a) Query in SQL and query plan



(b) Volcano



(c) Data-centric

```
abstract class Op {
  def open(): Unit
  def next(): Record
  def close(): Unit
}

class Select(op: Op)(pred: Record => Boolean)
    extends Op {
  def open() = op.open
  def next(): Record = {
    var rec = null
    do {
      rec = op.next
    } while (!isNull(rec)
         && !pred(rec))
    rec
  }
  def close() = {...}
}
```

(d) Volcano interface and select

```
abstract class Op {
  def open(): Unit
  def produce(): Unit
  def consume(rec:Record): Unit
  def close(): Unit
}
class Select(op: Op)(pred: Record => Boolean)
    extends Op {
  var parent = null
  def open() = {
    op.parent = this
    op.open
  }
  def produce(): Unit = op.produce
  def consume(rec: Record) = {
    if (pred(rec))
      parent.consume(rec)
  }
}
```

(e) Data-centric interface and select

Fig. 2.2. Query Evaluation models.

We follow Futamura's idea and specialize the Volcano `Select` operator in Figure 2.2d to a given query, as illustrated in Figure 2.3b. However, the specialized code is inefficient. Each operator checks that the pulled record is not a `null` value, even though this check is really necessary only inside the `Scan` operator. These `!isNull(rec)`

conditions cannot be specialized away since they impose a control-flow dependency on *dynamic* data.

2.2.2 The Data-centric (Produce/Consume) Model

The data-centric (Produce/Consume) Model as introduced in HyPer [6], leads to better compilation results, and is therefore used by most query compiler developments, including LegoBase [7], DBLAB [9], and Spark SQL [5]. In this model, the control flow is inverted. Data records are pushed towards operators, improving data and code locality. Operators that materialize tuples (e.g., aggregates and hash joins) are marked as pipeline breakers. As shown in Figure 2.2e, the operator interface consists of methods `produce` and `consume`. The producer's task is to obtain tuples, e.g., from a scan or other interfacing operator in the query pipeline. The consumer carries out the operation, e.g., evaluating the predicate in `Select`.

Viewing the data-centric model through the lens of Futamura projections delivers the key explanation of why it leads to better compilation results than the Volcano model: the inter-operator control flow does not depend on dynamic data, and hence specialization can fully remove the dispatch on the operator structure, leading to the tight residual code shown in Figure 2.3c. In contrast to the Volcano model, there are no `null` records since each operator in the pipeline invokes `consume` only on valid data.

2.2.3 Data-centric Evaluation with Callbacks

For developers, the data-centric evaluation model is somewhat unintuitive since it spreads out query evaluation across `produce` and `consume` methods. But given that we have identified the desired specialization result, we can think about whether we can achieve the *same* specialization from a different API.

Figure 2.4a walks through the hash join evaluation in the data-centric model. First, the executor invokes `HashJoin.open` to initialize the hash join branches (i.e., `left` and `right` operators) followed by `HashJoin.produce`. Second, the `produce` method

invokes `produce` on each branch (i.e., `left.produce` and `right.produce`). Thus, control moves to `left.produce` and perhaps invokes `produce` on that branch a few times until a scan or a pipeline breaker is reached. At this point, `consume` performs its actions and invokes `parent.consume` to dispatch a record to its consumer. When the control eventually returns back to `HashJoin.consume`, the record is added to the hash table. In the following step, `right.produce` is invoked to process the records from the right branch. As the example illustrates, it is not always clear how the `produce` and `consume` methods are behaving: `consume` will be called by actions triggered by `produce`. This becomes even more complex when the operator possesses multiple children. In this case `consume` behaves differently depending on which intermediate operator is pushing the data.

As our first contribution, we show that we can achieve the same functionality and specialization behavior by refactoring the `produce` and `consume` interface into a single method `exec` that takes a *callback*. This new model is directly extracted from the desired specialization shown in Figure 2.3c. Figure 2.4b shows how the hash join operator can be implemented using this new interface. The `exec` method does not require additional state. Each join branch is invoked using a different callback function; first the left, and then the right. This avoids the key difficulty in the `produce`/`consume` model, namely the conflation of phases in `consume`, and also the need to maintain parent in addition to child links. A variant of this model was presented as part of a *functional pearl* at ICFP '15 [28]. Intuitively, the statement `op.exec(cb)` can be read as follows: *operator op, generate your result and apply the function* `cb` *on each tuple.*

2.3 Building Optimizing Compilers

Having identified the general approach of deriving a query compiler from an interpreter (the first Futamura projection) using programmatic specialization as described in Section 2.1, and having identified the desired structure of our query interpreter in

```
// Schema: Dep(dname: String, rank: Int)

// Query plan
Print(
  Select(
    Scan("Dep")))(x => x.rank < 10)
```

(a)

```
// Specialized data-centric evaluation
for (rec <- data) {
  if (rec.rank < 10)
    println(rec.dname+","+rec.rank)
}
```

(c)

```
// Specialized Volcano evaluation
var nextRec = 0 // scan state
val size = data.length
while (true) {  // print loop
  val rec = {
    var recSel = null
    do {        // select loop
      recSel = if (nextRec < size)
        data(nextRec++) else null
    } while (!(selRec == null) &&
     !(recSel.rank < 10))
    recSel
  }
  if (rec == null) break
  println(rec.dname + "," + rec.rank)
}
```

(b)

Fig. 2.3. Specializing select query (a) in Volcano (b) and Data-centric (c)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                  // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)         // Step 3
      hm += (lkey(rec), rec)
    else                // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Fig. 2.4. Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2).

Section 2.2, we are now faced with the task of actually making it happen. Recall, obtaining a compiled query target from an interpreted query engine requires identifying three components: the staged interpreter, the static input and the dynamic input. Figure 2.5 shows LB2's query evaluator, essentially an interpreter, that implements the data-centric evaluation with callbacks. We can now start specializing this query engine to emit source code in the same way we specialized the `power` function using the symbolic data type `MyInt` in Section 2.1. *But where in a query engine should code generation be placed to minimize changes to the operator code?*

Pure Template Expansion. As a first idea we could perform coarse-grained code generation at the operator level. Each operator is specialized as a string with placeholders for parameters. We show the aggregate operator as example:

```
class Agg(op: Op)(grp: GrpFun)(init: String)(agg: AggFun) extends Op { // operator template
  def exec(cb: String => String) = s"""
    val hm = new HashMap()
    ${op.exec { tuple =>
      s"""val key = ${grp(tuple)}
      hm.update(key, $init) { curr => ${agg("curr", "tuple")} }
    }"""}
    for (tuple <- hm) { ${cb("tuple")} }"""
}
```

At runtime, this query evaluator performs a *direct mapping* from an operator to its code, i.e., substitutes `op.exec` with the operational code of the child operator `op`. Template expansion is easy to implement and removes some of the interpreter overhead. Still, the approach is criticized as inflexible [9]. First, a query is generated exactly as written, with generic and inefficient data structure implementations. Second, cross-operator optimizations, data layout changes, etc., are all off-limits. Third, string templates are inherently brittle, and rewriting the core engine code as templates may introduce variable name clashes, type mismatches, etc., that may cause both subtle errors and hard crashes in generated code.

Programmatic Specialization. In order to avoid the problems of coarse templates, we can push specialization further down into the structures that make up the query engine in Figure 2.5, in particular the `Record` and various `HashMap` classes. The

key benefit of this approach is that the main engine code in Figure 2.5 can *remain unchanged*!

The generated code is the same as for operator templates, but all the code generation logic is now confined to `Record` and `HashMap`, with a much smaller surface exposure to bugs related to string manipulation. The implementation is as follows:

```
class Record(fields: Seq[(String, String)]) {
  val name = freshName()
  println(s"""val $name = new Record(${fields map ...  })""")
  def apply(field: String) = s"""$name.$field"""
  def update(field: String, v: String) = println(s"""$name.$field = $v;""")
}
class HashMap() {
  val name = freshName()
  println(s"""val $name = new HashMap()""")
  def apply(key: String) = s"""$name($key)"""
  def update(key: String, v: String)(up: String) =
    println(s"""$name($key) = $up($name.getOrElse($key,$v))""")
}
```

Still, the generated code uses unsophisticated `Record` and `HashMap` implementations, which will not exhibit optimal performance. When targeting C code, we will likely use an equivalent library such as `GLib`, which has high performance overhead.

Optimized Programmatic Specialization. For optimum performance, we want to implement specialized internal data structures instead of relying on generic libraries. To achieve this, we take the specialization idea one step further and push code generation even further down to the level of primitive types and operations. This means that not even our `Record` and `HashMap` implementations need to mention code generation directly, and as we will show in Sections 2.3.1-2.3.2, this will enable a range of important optimizations while retaining a high-level programming style throughout. This leads us to use *exactly* the `MyInt` class shown in Section 2.1, and, while it would be possible to build an entire query engine in this way, it pays off to use an existing code generation framework that already implements this low-level plumbing.

Lightweight Modular Staging (LMS). LB2 internally uses a library-based generative programming and compiler framework LMS [24] to encapsulate the code gener-

```
type Pred   = Record =>
    Rep[Boolean]
type KeyFun = Record => Record
type OrdFun = (Record,Record)
    => Rep[Int]
type AggFun = (Record,Record)
    => Record
type GrpFun = Record => Record

class Scan(table: Buffer)
    extends Op {
  def exec(cb: Record =>
      Unit) = {
    for (tuple <- table)
      cb(tuple)
} }
class HashJoin(left: Op,
    right: Op)(lkey: KeyFun)
(rkey: KeyFun) extends Op {
    /* code in Figure 2.4 */
    }
```

```
class Select(op: Op)(pred: Pred)
    extends Op {
  def exec(cb: Record => Unit) = {
    op.exec { tuple =>
      if (pred(tuple)) cb(tuple)
} } }
class Sort(op: Op)(ordFun:
    OrdFun) extends Op {
  def exec(cb: Record => Unit) = {
    val res = new FlatBuffer()
    op.exec { tuple => res +=
        tuple }
    res.sort(ordFun)
    for (tuple <- res)
      cb(tuple)
} }
```

```
class Agg(op: Op)(grp: GrpFun)
    (init: Record)(agg:
        AggFun) extends Op {
  def exec(cb: Record =>
      Unit) = {
    val hm = new HashMap()
    op.exec { tuple =>
      val key = grp(tuple)
      hm.update(key, init) {
        curr => agg(curr,
            tuple)
    } }
    for (tuple <- hm)
      cb(tuple)
} }
```

Fig. 2.5. LB2 Query Evaluator (data-centric with callbacks).

ation logic. LMS maintains a graph-like intermediate representation (IR) to encode *high-level* constructs and operations. Moreover, LMS provides a high-level interface to manipulate the IR graph. LMS distinguishes two types of expressions; present-stage expressions that are executed normally and future-stage expressions that are compiled into code. LMS defines a special type constructor `Rep[T]` to denote future-stage expressions, e.g., our `MyInt` corresponds to `Rep[Int]` in LMS, and given two `Rep[Int]` values `a` and `b`, *evaluating* the expression `a+b` will *generate* code to perform the addition. LMS provides implementations for all primitive `Rep[T]` types, i.e., strings, arrays, etc. In addition, LMS also provides overloaded control-flow primitives, e.g., `if (c) a else b` where `c` is a `Rep[Boolean]`.

LMS is a full compiler framework, which supports a variety of intermediate layers, but we only use it as a code generation substrate for our purposes. To draw a comparison with HyPer, LB2 is implemented in Scala instead of C++ and uses LMS instead of LLVM. While LLVM [14] operates on a lower level than LMS, the difference

is not fundamental, and it is important to note that abstractions similar to those we propose can also be built on top of LLVM in C++, using standard operator overloading and lambda expressions, which have been available since C++11.

Preliminary Example. In order to better understand how a query is compiled using the optimized programmatic specialization, consider the following aggregate query and the query execution plan (QEP) (refer to Figure 2.5 for operators implementation).

```
select edname, count(*)
from Emp
group by edname
```


```
Print(
    Agg(Scan("Emp"))
        (x => x("edname"))(0)
          ((agg,x) => agg + 1))
```

In LB2, like in any other database, the query is represented by a tree of operators. Evaluation starts with the root operator in the QEP (i.e., `Print`). Calling `Print.exec` with an empty callback will call its child operator's `Agg.exec` method with a callback that encodes evaluation actions to be carried out on the records that `Agg` produces: in this case, just printing out the result. After that, `Agg.exec` calls `Scan.exec` with a callback tailored to the aggregate operation and the callback it received from `Print`. Based on Futamura projections discussed in Section 2.1, the result of executing a staged query interpreter is a *residual* program that implements the query evaluation on record fields where all abstractions are optimized away and indirect control flow removed.

For the remainder of this section, we discuss how interesting performance optimizations for data structures, storage layout, memory management, etc., are implemented in LB2 while retaining the single code generation pass architecture.

2.3.1 Row or Column Layout

Typically, query engines either support row-oriented or column-oriented storage. Each data layout works better in one situation than the other, so it is attractive to be able to support both within the same query engine.

In LB2, the `Record` class is the entry point to query engine specialization. `Record` is an abstract class that contains a schema (a sequence of named `Field` attributes) and supports lookup of `Value`s by name. It is important to note that the schema is entirely *static*, i.e., only exists at code generation time, while subclasses of `Value` carry actual `Rep[T]` values, i.e., dynamic data that exists at query evaluation time. Subclasses of `Record` can support either flat, row-oriented, storage through a base pointer (class `NativeRec`), or abstract over the storage model entirely by referencing individual `Value`s directly from a record in a *scalarized* form, mapped to local variables in generated code (class `ColumnRec`).

```
abstract class Field { def name: String }          // models an attribute's name and type
case class IntField(name: String) extends Field    // Int type attribute
case class StringField(name: String) extends Field // String type attribute

abstract class Value                               // models an attribute's value
case class IntValue(value: Rep[Int]) extends Value { ... }  // operations elided
case class StringValue(value: Rep[String], length: Rep[Int]) extends Value { ... }

abstract class Record { def schema: Seq[Field]; def apply(name: String): Value }

case class NativeRec(pt: Rep[Pointer], schema: Seq[Field]) extends Record {
  def apply(name: String) = getField(schema,name).readValue(pt, getFieldOffset(schema,name))
}
case class ColumnRec(fields: Seq[Value], schema: Seq[Field]) extends Record {
  def apply(name: String) = fields(getFieldIndex(schema,name))
}
```

Likewise, LB2 abstracts over `Record` storage through an abstract class `Buffer`, with implementation classes for row-oriented (`FlatBuffer`) and column-oriented storage (`ColumnarBuffer`):

```
abstract class Buffer(schema: Seq[Field]) {
  def apply(x: Rep[Int]): Record
  def update(x: Rep[Int], rec: Record): Unit
}
case class FlatBuffer(schema: Seq[Field], size: Long) extends Buffer(schema) ...
case class ColumnarBuffer(schema: Seq[Field], size: Long) extends Buffer(schema) {
  val cols = schema map { fld => Column(fld,size) }
  def apply(x: Rep[Int]) = ColumnRec(cols map { c => c(x) }, schema)
  def update(x: Rep[Int], y: Record): Unit =
    cols foreach { c => c.update(x, y(c.field.name)) }
}
case class Column(field: Field, size: Long) {
  val buf: Rep[Pointer]  = malloc(size * field.size)
  def apply(x: Rep[Int]) = field.readValue(buf, x)
  def update(x: Rep[Int], y: Value): Unit = field.writeValue(buf, x, y)
```

```
}
```

The internal implementations are unsurprising, and exactly what one might write in a high-level query interpreter without much concern for low-level performance. But it is important to note that there is never any code like `new Record(...)` or `new Value(...)` being generated. Hence, `Value`, `Record`, `Buffer`, etc., objects are generation-time-only abstractions that are completely dissolved as part of the symbolic evaluation. The generated code consists only of the operations on the `Rep[T]` values hidden inside the `Value` and `Buffer` subclasses, i.e., only raw `malloc`s and pointer reads/writes.

A pipeline of operators accesses records uniformly in either row or column format. A pipeline breaker materializes the intermediate `Records` inside a buffer or higher-level data structure, at which point a format conversion may occur.

2.3.2 Data Structure Abstractions

Query engines use map data structures to implement aggregate and join operations. For aggregations, a `HashMap` accumulates the aggregate result for a grouping key and for hash joins, a `HashMultiMap` collects all records for a given key. As discussed earlier, implementations from a generic library (e.g., `Glib`) tend to be inefficient due to expensive function calls, resizing, etc. Moreover, we want to make different low-level implementation choices for different parts of the engine (e.g., open addressing vs. linked hash buckets). Based on the `Buffer` abstractions (Section 2.3.1), we can build a considerable variety of fast hash table implementations with little effort.

The code example below illustrates the one we use for aggregates (see Figure 2.5). Class `HashMap` defines the interface, and subclass `LB2HashMap` provides an implementation based on open addressing on top of `ColumnarBuffer`. Method `update` locates the key and updates the aggregate value. Method `foreach`, which is invoked for Scala expressions of the form `for (rec <- hm)`, traverses all stored values. The hash map is fully specialized for key and value types.

```
abstract class HashMap(kSche: Seq[Field], vSche: Seq[Field]) {
  def update(key: Record, init: Record)(up: Record => Record): Unit
```

```
  def foreach(f: Record => Unit): Unit
}
class LB2HashMap(kSche: Seq[Field], vSche: Seq[Field]) extends HashMap {
 val size = defaultSize
 val agg = new ColumnarBuffer(vSche, size)
 val keys = new ColumnarBuffer(kSche, size)
 val used = new Array[Int](size)
 var next = 0
 def update(k: Record, init: Record)(up: Record => Record) = {
   val idx = defaultHash(k) % size
   if (isEmpty(keys(idx))) {
     used(next) = idx; next += 1
     keys(idx) = k; agg(idx) = up(init)
   } else agg(idx) = up(agg(idx))
 }
 def foreach(f: Record => Unit) = {
   for (idx <- 0 until next) {
     val j = used(idx)
     f(merge(keys(j), agg(j)))
} } }
```

It is important to note that this code is the same as one would write in a library implementation. However, as with `Buffer`s and `Record`s, the `HashMap` abstraction is completely dissolved at code generation time, leaving only low-level array and index manipulations.

The hash join operator uses a `HashMultiMap`, which differs from this code in its interface, and also in its internal implementation (we use linked buckets instead of open addressing). The elided code follows the same high-level of abstraction, and if we wish, we can pull out some aspects of the two hash map classes into a common base class, again without any runtime cost.

2.3.3 Data Partitioning and Indexing

Query engines use statistics and metadata to determine when an index can be used to speed up query execution. We assume that these decisions are made during the query planning and optimization phase. To add index capabilities to LB2, we provide a corresponding set of indexed query operators in the same style as those defined in Figure 2.5. The code below shows an index join. The operator interface

is extended with a `getIndex` method, which enables `IndexJoin.exec` to find tuples that match the join key:

```
// Index join operator that uses index created on the left table
class IndexJoin(left: Op, right: Op)(lkey: String)(rkey: KeyFun) extends Op {
  def exec(cb: Record => Unit) = {
    val index: Index = left.getIndex(lkey) // obtain index for left table
    right.exec { rTuple => // use index to find matching tuples
      for (lTuple <- index(rkey(rTuple))) cb(merge(lTuple, rTuple))
} } }
```

LB2 realizes sparse and dense index data structures for primary and foreign keys on top of the abstractions presented in Sections 2.3.1 and 2.3.2, behind a uniform `Index` interface. The `IndexEntryView` class enables iterating over index lookups via `foreach`. Method `exists` is used by `IndexSemiJoin` and `IndexAntiJoin` operators.

```
abstract class IndexEntryView {
  def foreach(f: Record => Unit): Unit
  def exists(f: Pred): Rep[Boolean]
}
```

```
abstract class Index(schema: Seq[Field]) {
  def apply(k: Record): IndexEntryView
}
```

In addition to query evaluation, LB2 generates data loading code for different storage modes, which we extend to create index structures. These can serve as additional access paths on top of underlying data, or as primary partioned and/or replicated data format, e.g., when there are multiple foreign keys.

Date Indexes. LB2 represents dates as numeric values to speed up filter and range operations. If metadata about date ranges is available, it will enable further shortcuts. LB2 breaks down dates into year and month and uses existing abstractions to index dates based on year or month. Adding a date index is similar to creating an index on a primitive type. Hence, we elide further details.

String Dictionaries. Another form of indexing is compressed columns and dictionary encodings. LB2 implements string dictionaries to optimize string operations, e.g., `startsWith`. Individual columns can be marked as dictionary compressed in the database schema. Building on `StringValue` and `ColumnarBuffer` discussed earlier in Section 2.3.1, LB2 defines an alternative string representation class `DicValue` and a class `StringDict` that stores and provides access to compressed string values.

```
class DicField(name: String) extends Field {
  def dict: StringDictionary = ...
}
class DicValue(idx: Rep[Long]) extends Value {
  def startsWith(p: DicValue) = p.idx <= idx
  ...
}
```

```
class StringDict(attr: String, size: Long) {
  val store = Column(StringField(attr),size)
  def convert(string: StringValue): DicValue
  def get(idx: DicValue): StringValue
}
```

Consider the simple case of compressing a single string column. At loading time, the `StringDict` is loaded from memory. When the loader reads a string, it creates a `StringValue` and uses the `StringDict` to convert it into its `DicValue` compressed form: the index where the `StringValue` is stored inside the `StringDict`.

While string dictionaries may speed up most string operations, their use requires some care. The comparison of two strings in compressed form is only valid if they share the same dictionary. One solution is to use a single global dictionary for all string attributes, however it is not easy to maintain. Another solution is to group the columns that are the most likely going to be used together within one dictionary. In the event of a comparison between string belonging to two different dictionaries, the fallback is to extract the `StringValue` and perform the operation on the string representation. Moreover, some operations generate strings at runtime (e.g., `substring`). In that case, uncompressed strings need to be used as well. Conceptually, LB2 can support both implementations. The `DicField` class keeps track of the dictionary associated with a given attribute, which allows LB2 to generate compressed string operations where possible and uncompressed operations otherwise. Finally, string dictionaries do not add new query operators, i.e., they operate transparently as part of the data representation layer.

2.3.4 Code Layout and Code Motion

Hoisting memory allocation and other expensive operations from frequently executed paths to less frequent paths can speed up evaluation dramatically, especially in hash join and aggregate queries where memory may be pre-allocated in advance.

Let us recall the design principle from Section 2.2, that the shape of the generated code follows how the interpreter is written. Based on this insight, we may reorder evaluation actions in a certain way that moves expensive operations off the hot path of query execution. In particular, we can extend LB2's callback interface to enable hoisting data structure allocation. In Figure 2.6, we demonstrate how a small change to the `exec` method signature enables data structures hoisting. Method `exec` in Figure 2.6-a1 takes a single argument, allocates data structures and invokes `parent.exec`. In Figure 2.6-a2, `exec` is re-implemented as zero-parameter method that separates the data structure allocation and query execution by performing the allocation first and only then *returning* a function that executes the operator's main data path. The rest of the example in Figure 2.6-b1 and b2 shows how client programs can inject code, e.g., timing, inbetween memory allocation, and the main evaluation loop. Figure 2.6-c1 and c2 show the respective generated code.

2.3.5 Parallelism

Query engines realize parallelism either explicitly by implementing special *split* and *merge* operators [109], or internally by modifying the operator's internal logic to orchestrate parallel execution. LB2 uses the latter, and generates code for OpenMP [110].

Interestingly, the same pattern we used in Section 2.3.4 to achieve code motion can be used to structure query engine code for parallel execution. The callback signature for `exec` we defined earlier works well in a single-threaded environment, but multi-threaded environments require synchronization and thread-local variables. Thus, LB2 defines a new class `ParOP` with a modified `exec` method that adds another callback level.

For state-less operators such as `Select`, the parallel implementation is very similar to the single threaded one; it is possible to use a wrapper to transform a single threaded pipeline into a parallel one. Assuming we have a parallel scan operator `ParScan`, we can perform a parallel selection like this:

```
val parSelect = parallelPipeline(op => Select(op)(t => t("rank") < 10))
parSelect(ParScan("Dep"))
```

```
def exec(cb: Record => Unit) = {
  val hm = new HashMap()

  op.exec { tuple =>
    hm.update(grp(tuple), ...)(...)
  }
  for (tuple <- hm) cb(tuple)
}
```

(a1) Aggregate skeleton

```
def exec = {
  val hm = new HashMap(); val dataLoop = op.exec
  (cb: Record => Unit) => {
    dataLoop { tuple =>
      hm.update(grp(tuple), ...)(...)
    }
    for (tuple <- hm) cb(tuple)
} }
```

(a2) Optimized aggregate skeleton

```
// execute aggregate

time { Print.exec(t => ()) }
```

(b1) Aggregate client program

```
// execute aggregate
val query = Print.exec
time { query(t => ()) }
```

(b2) Optimized aggregate client program

```
struct timeval start, end;
gettimeofday(&start, NULL);
// data struct allocation
int* agg = malloc(...); ...
// processing
for (int i = 0; i < N; i++) {
  // ... compute aggregate ...
}
for (int i = 0; i < next; i++) {
  // ... print records ...
}
gettimeofday(&end, NULL);
```

(c1) Generated code

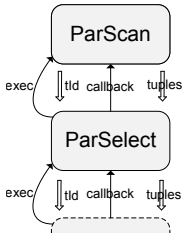
```
// data struct allocation
int* agg = malloc(...); ...
struct timeval start, end;
gettimeofday(&start, NULL);
// processing
for (int i = 0; i < N; i++) {
  // ... compute aggregate ...
}
for (int i = 0; i < next; i++) {
  // ... print records ...
}
gettimeofday(&end, NULL);
```

(c2) Generated code

Fig. 2.6. From a query plan to optimized C code including data structures specialization and code motion.

The definition of `ParOp` and `parallelPipeline` is as follows:

```
class ParOp {
  type ValueCallback = Record => Unit
  type DataLoop = ValueCallback => Unit
  type ThreadCallback = Rep[Int] => DataLoop => Unit
  def exec: ThreadCallback => Unit
}
def parallelPipeline(seq: Op => Op) =
  (parent: ParOp) => new ParOp {
    def exec = {
      val opExec = parent.exec
      (tCb: ThreadCallback) => opExec { tId => dataloop =>
        tCb(tId)((cb: ValueCallback) =>
          seq(new Op { def exec = dataloop }).exec(cb) })
} } }
```



The communication between operators is illustrated in the drawing above. The downstream client of `parSelect` initiates the process by calling `exec`, which `parSelect` forwards upstream to `ParScan`. `ParScan` starts a number of threads, and on each thread, calls the `exec` callback with the thread id `tId` and another callback `dataloop`. This will allow the downstream operator to initialize the appropriate thread-local data structures. Then the downstream operator 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.

While the `parallelPipeline` transformation covers the simpler state-less operators, some extra work is required for pipeline breakers. The callback interface makes sophisticated threading schemes possible, in an elegant manner. For operators such as `Agg`, LB2's parallel implementations split their work internally across multiple threads, accumulating final results, etc. By using callbacks in a clever way, we can delegate some of the synchronization effort to specialized parallel data structures. In the case of `Agg`, LB2 uses a `ParHashMap` abstraction, which internally has to enforce thread safety. Multiple implementations are possible, using synchronization primitives, partitioning, or an internal lock-free design.

```
abstract class ParHashMap(nT: Long, kSche: Seq[Field], vSche: Seq[Field]) {
  def apply(tId: Rep[Int]): HashMap
  def merge(init: Record, agg: AggFun): Unit
  def partition(tId: Rep[Int]): DataLoop
}
```

This design is powerful enough to encapsulate all subtle issues related to multi-threading. Similar to the code motion idea, we use the callbacks to re-organize the operator code into different parts:

```
class Agg(grp: GrpFun)(init: Record)(agg: AggFun) extends ParOp {
  def exec = {
    val opExec = op.exec
    val hm = new ParHashMap(nbThread, grp.schema, agg.schema)
    (tCb: ThreadCallback) => {
      opExec { (tId: Rep[Int]) => (dataLoop: DataLoop) => // parallel section starts
        val lHm = hm(tId): HashMap
        dataloop { tuple => // computes partial result for this thread
          val k = grp(tuple)
          lHm.update(k, init) { c => agg(c, tuple) }
      } }                                                  // parallel section ends
      hm.merge(init, agg) // merge the results accross threads
      parallelRegion { tId => tCb(tId)(hm.partition(tId)) } // restart a pipeline in parallel
} } }
```

The distinct parts are the global initialization of the data-structure, the local initialization for each thread, the computation, and finally the merging between threads, followed by the beginning of a new parallel pipeline.

2.3.6 Comparison with a Multi-Pass Compiler

We contrast LB2's implementation with a recent multi-pass query compiler, DBLAB [9]. Both systems aim to implement a query interpreter in a high-level language once and control query compilation by modifying parts of the query engine. In LB2, we modifed the core abstractions underneath the main engine code to add code generation. In DBLAB, the query engine code itself is *transformed* to lower-level code, through multiple intermediate stages. For each optimization, an analysis pass identifies pieces of code to be re-written followed by one or more re-writing passes. DBLAB offers a flag per optimization that can be configured for each query. We compare how key optimizations are implemented in both systems.

Data Layout. The default data layout in DBLAB is row-oriented. DBLAB supports column-oriented layout on a best effort basis using a compiler pass that converts arrays of records to a record of arrays where possible. In LB2, operators decide which storage

layout to use by instantiating one of several implementation classes, e.g., `FlatBuffer` or `ColumnarBuffer` as discussed in Section 2.3.1. These decision can be made based on input from the query optimizer.

Data Structure Specialization. DBLAB introduces a number of intermediate abstraction levels (referred to as stack-of-DSLs) and defines optimizations that can be performed at each level. Specializing high level data structure, e.g., hash maps, into native arrays takes one analysis pass and up to three re-writing passes. In LB2, the same transformation is achieved by implementing hash maps as generation-time abstractions, which ensures that only native array operations are generated. Adding a new hash map variant requires a high-level implementation in LB2, using normal object-oriented techniques (see Section 2.3.2). For example, LB2 uses open addressing for aggregates and linked hash buckets for joins. In DBLAB, all analysis and transformation passes are specific to a linked-bucket hash table implementation. Adding a variant based on open addressing would require an entire new set of analysis and transformation passes.

Index Structures. DBLAB's makes indexing decisions not based on query plans, but on a lowered version of the query engine code after inlining the operators. A first analysis extracts hash join patterns and evaluates whether an index can be instantiated. In some sense, this appears to be putting the cart before the horse, because high-level query plan structure needs to be reverse-engineered from comparatively low-level code. A second analysis rule determines the type of index, and a third rule determines whether the hash map is collision-free on the hash function and hence can use one dimensional arrays. Finally, a rewrite rule updates the generated code to use the index and inserts index creation code into the loading phase. This automatic index inference in DBLAB is a *global* approach that always creates indexes without reasoning about the index cost or whether a non-index plan could be more efficient. LB2 does not attempt to infer indexes automatically and instead delegates such decisions to the query optimizer.

For string dictionaries, DBLAB performs a compiler pass to create string dictionaries on all string attributes and hoist the allocation statements to loading time. The rewrite rule identifies the type of string operation and updates the code accordingly. Again, LB2 does not attempt to infer string dictionary usage from low-level code, but assumes that this information is already available. Instead of transforming code, LB2 uses corresponding implementation classes as discussed in Section 2.3.3.

Code motion. DBLAB performs detailed analysis to collect data structures allocation statements along with the dependent statements. After that, a rewrite rule moves allocation to loading time. Hence, hoisting is sensitive to order. As illustrated in Section 2.3.4 LB2's callback interface seamlessly hoist data structures allocation outside the operator code, with small changes to the operator interface.

Parallelism. LB2 implements parallelism as shown in Section 2.3.5. DBLAB does not implement parallelism, although the paper [9] discusses some ideas in this direction.

In summary, we believe that there are certain drawbacks to multi-pass query compilers. Most importantly, optimizations require a number of analysis and rewrite passes, i.e., database engineers have to become real compiler experts. The hard part about writing compilers is not to get a transformation working on a few examples, but to ensure correctness for *all* possible corner cases and for interactions with other parts of the system, which may be changed independently. In comparison, LB2's single-pass approach facilitates query compilation using techniques that are no more difficult than writing a query interpreter in a high-level language.

2.4 Experimental Evaluation

In this section, we evaluate the performance of LB2 on the standard TPC-H benchmark with scale factor SF10. We compare LB2 with Postgres [111] and two recent state-of-the-art compiled query engines: Hyper [6], and DBLAB [9]. HyPer

implements compilation using LLVM and DBLAB is a multi-pass query compiler that generates C.

Configurations. We present three sets of experiments. The first set evaluates the performance of LB2 with only those optimizations that are compliant with the official TPC-H rules. The second set focuses on comparing optimizations that replicate data and create auxiliary indexes (Section 2.3) with their counterparts in DBLAB. Finally, the last set evaluates parallelism in LB2 and HyPer when scaling up the number of cores (DBLAB only runs on a single core). We run each query five times and record the median reading. For DBLAB and LB2, we use `numactl` to bind the execution to one CPU. HyPer provides a flag for the same purpose `PARALLEL=off`.

Query plans in LB2 and DBLAB are supplied explicitly while HyPer and Postgres implement a cost-based query optimizer. Since it is difficult to unify query plan across all systems, we report two sets of results for LB2. The line `LB2 (DBLAB plan)` uses DBLAB's plans and `LB2 (hyper plan)` uses HyPer's plans to the extent possible but at least with the same join ordering. We choose not to turn indexing off in HyPer to allow the query optimizer to pick the best plan. Also, DBLAB replaces the outer join in Q13 with a hard-coded imperative array computation using side effects that is neither expressible in SQL, nor in their internal query plan language.

DBLAB offers close to 30 configuration flags to enable/disable optimizations. In the first experiment, we use the `-compliant` option (a subset of compliant configurations per query). In the second experiment, we use the compliant configuration with the `hm-part` flag, `DBLAB/LB 4` and `DBLAB/LB 5` configurations respectively to enable indexing, date indexing with partitioning and string dictionaries as described in [9, 112]. We compare each of these configurations to the corresponding one in LB2.

Performance Evaluation Parameters. The first experiment (i.e., TPC-H compliant runtime) and the third experiment (i.e., parallelism) use the absolute runtime as a key evaluation parameter. The second experiment evaluates the impact of individual optimizations that are pre-computations. In this case, we also report the overhead

introduced by all these pre-computations relative to the loading time of LB2 without any index generation (fastest loading).

Experimental Setup. All experiments are conducted on a single machine with 4 Xeon E7-88904 CPUs, 18 cores and 256GB RAM per socket (1 TB total). The operating system is Ubuntu 14.04.1 LTS. We use Scala 2.11, Postgres 9.4, HyPer v0.5-222-g04766a1 [1], GCC 4.8 with optimization flag `-O3` and GLib library 2.0. We tried different versions of GCC and Clang with very similar results. We use the version of DBLAB released by the authors as part of the SIGMOD '16 artifact evaluation process [112][2].

2.4.1 TPC-H Compliant Runtime

In the first experiment, we compare LB2 with Postgres, DBLAB, and HyPer on a single core under the TPC-H compliant settings. Figure 2.7 reports the absolute runtime for all TPC-H queries. We follow [9] in evaluating DBLAB without any indexing and use the same configuration for LB2. We did not disable primary key indexing in HyPer, as doing so appears to lead to very suboptimal query plans. In the plans reported here, HyPer employed one or more index joins in Q2, Q8-Q10, Q12, and Q21. Postgres is a Volcano-style interpreted query engine that is representative of wide-spread traditional systems.

Overall, LB2 outperforms Postgres and DBLAB in all queries where query plans are matched. Furthermore, LB2 and HyPer's performance is comparable. On a query by query analysis, LB2 outperforms DBLAB in aggregate queries Q1 and Q6 by 70% and 4% respectively. On join queries Q3, Q5, Q10, etc. LB2 is 3×-13× faster than DBLAB. Similarly, LB2 is 5×-13× faster in semi join and anti join queries Q4, Q16, Q21 and Q22. In Q13, DBLAB replaces the outer join operator with a hard-coded

[1]At the time of writing, HyPer binaries are no longer publicly available. We use a version obtained in late 2015.

[2]The generated C files for TPC-H queries submitted in [112] do not include an equivalent configuration for (`compliant+index`) and also use nonstandard string constants, e.g., in Q3, Q7, etc. For a uniform comparison, we re-generated the C files using [113].

SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	241581	6665	33835	7980	30158	23485	32577	29953	64429	33237	7113	38040	31425	22154	23183	13450	155541	91241	29548	65681	299691	11766
DBLAB	809	219	6575	7977	3049	232	3598	5494	32313	4175	104	3847	(1079)	526	933	2961	3795	4771	6353	763	13851	1328
LB2 (DBLAB plan)	476	140	491	573	576	223	914	1941	4540	1230	52	691	4012	377	373	595	2001	1294	2623	569	1887	235
HyPer	613	61	1193	872	991	233	982	637	2029	1021	138	504	3718	342	261	1274	580	3764	2003	439	1658	190
LB2 (HyPer plan)	476	141	494	573	649	222	706	862	2806	1378	51	691	4012	246	189	593	1994	1289	2965	833	1793	234

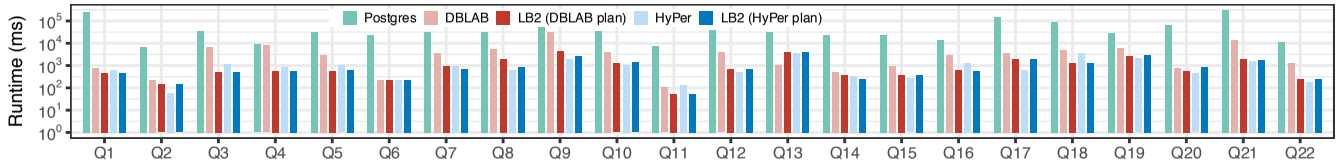


Fig. 2.7. The absolute runtime in milliseconds (ms) for DBLAB, LB2 (with DBLAB's plans), HyPer, LB2 (with HyPer's join ordering plans) in TPC-H SF10. Only TPC-H compliant optimizations are used.

imperative array computation that has no counterpart in the query plan language. Hence, a direct comparison for this query is misleading, and we do not attempt to recreate an equivalent "plan" in LB2.

The performance gap between LB2 and DBLAB can be attributed, in part, to a number of implementation details. First, LB2 implements a generation-time string abstraction (explained in Section 2.3.2) that optimizes commonly used string operations while DBLAB primarily relies on C-strings and only optimizes one instance (the `startsWith` operation). Second, the systems make different decisions related to hash join implementations, e.g., the hash function and number of keys used while creating a hash table. LB2's hash table is always custom-generated while DBLAB sometimes relies on GLib C data structures and also uses function calls for some string operations as opposed to inlining and specialization in LB2. Furthermore, there are different trade-offs related to handling composite keys, i.e., either creating a very selective hash table that combines more than one key or using only a single key and relying on the join condition to filter unqualified tuples (essentially a short linear search). We observe that DBLAB opts for smaller hash tables at the expense of more extensive search, while LB2 chooses more precise hashing. Finally, there are differences in optimizing data layout, e.g., pertaining to row-oriented vs. column-oriented layout for internal data structures (Section 2.3).

Comparing the performance of LB2 and HyPer, we observe that LB2 is faster by at least 2×-3× in Q3, Q11, Q16, and Q18. Also, LB2 is 25%-50% faster than HyPer in Q1, Q4, Q5, Q7, Q14, and Q15. On the other hand, HyPer is faster than LB2 by 2×-3× in Q2 and Q17. This performance gap is, in part, attributed to (1) HyPer's use of specialized operators like GroupJoin and (2) employing indexes on primary keys as seen in Q2, Q8-Q10, etc. Finally, while both LB2 and HyPer generates native code (LB2 generates C and HyPer generates LLVM), differences in implementation may result in faster generated code. One example is floating point numbers. HyPer uses proper decimal precision numbers, whereas LB2 and DBLAB use double precision floating point values.

2.4.2 Index Optimizations

The second experiment focuses on evaluating three advanced optimizations that were used by DBLAB to justify a multi-pass compiler pipeline [9]; primary and foreign key indexes, date indexes, and string dictionaries. In their full generality, these optimizations are not compliant with the TPC-H rules [114] since they incur precomputation and potentially a duplication of data. While a subset of these optimizations that indexes data uniformly across all queries would be allowed by the TPC-H spec, we do not evaluate this setting for consistency with previously published DBLAB configurations [9, 112]. The results of this experiment are shown in Figures 2.8 and 2.9. The first table and graph give the absolute runtime of the TPC-H queries with different levels of indexing enabled: primary/foreign key, date columns, and string dictionaries. The second graph shows the overhead on loading time associated with creating these indexes.

Primary and Foreign Key Indexes. The results for this configuration are shown in line `DBLAB/LB2-idx`. Recall that DBLAB analyzes intermediate code to decide on which indexes it will create. LB2 makes those decisions based on the query plan. For the purpose of this experiment, we have tuned LB2's decision rules to lead to the same decisions as DBLAB. We observe that DBLAB's index cost is greater than LB2's in all queries with Q5, Q12, and Q3 as the top three. In these queries, a hash map index is created on a sparse key. While both systems follow a similar compromise in allocating larger space to optimize access time, LB2's column-oriented layout contributes to lowering index creation and access cost. This observation is the main reason for the better performance for LB2 over DBLAB. On a query by query analysis for the absolute runtime, LB2 outperforms DBLAB in join query Q3 by 3× and by 15%, 80% in Q10 and Q5 respectively. Similarly, LB2 is 2×-4× faster in semi join and anti join queries Q4, Q22, Q16, Q21. On the other hand, DBLAB is faster than LB2 in Q7 and Q9 by 3% and 13% respectively. As discussed in Section

SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBLAB-idx	827	144	2341	631	380	236	123	332	900	765	100	1167	(1379)	272	919	4648	266	4680	247	312	1408	208
LB2-idx	521	87	685	316	207	223	127	81	1021	663	15	489	3895	272	440	1157	144	183	69	84	347	104
DBLAB-idx-date	928	101	422	217	207	117	78	292	846	703	85	1242	(1337)	289	40	5355	218	200	311	414	626	173
LB2-idx-date	485	85	152	162	212	80	108	81	1024	455	17	294	3895	26	54	1153	138	204	69	83	406	104
DBLAB-idx-date-str	497	97	138	222	225	82	79	23	510	603	130	362	(4507)	16	41	555	13	198	15	401	494	176
LB2-idx-date-str	487	74	166	185	170	79	85	35	641	408	16	184	3904	10	56	774	7	184	11	85	384	104

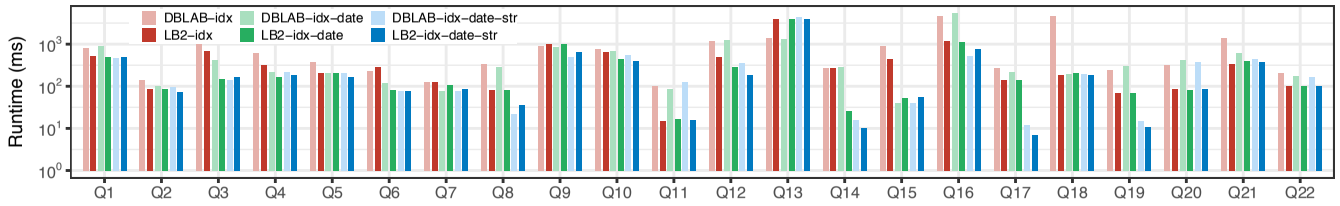


Fig. 2.8. The absolute runtime in milliseconds (ms) after enabling non-TPC-H-compliant indexing, date indexing and string dictionary in SF10 using DBLAB plans.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBLAB-compliant	3.8	1.5	3.8	4.1	3.6	3.3	7.4	6.2	7.3	5.9	2.2	3.3	4.8	3.4	2.2	3.5	3.3	2.2	3.8	3.6	1.8	3.5
LB2-compliant	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
DBLAB-idx	4	1.7	5.6	5.5	5.6	3.5	5.6	6	5.5	5.5	2.9	3.6	5.2	3.8	2.5	3	4.7	2.8	6.2	5.9	3	5.4
LB2-idx	1	0.9	1.3	1.2	1.7	1	3.2	4	4.1	1.2	1.3	1	1	1	1	0.8	2.7	1.2	6.7	3.6	2.1	1.7
DBLAB-idx-date	3.9	2.1	4.4	4.4	3.9	6.2	4.1	4.2	4.2	4.5	2.2	3.7	4.7	4.2	2.4	3.2	3.3	2.7	4.8	4.5	2	3.8
LB2-idx-date	1.2	0.9	1.7	1.2	1.6	1.2	2.3	3.2	3.9	1.2	1	1.1	0.8	1.1	1.1	0.7	2.2	1.1	6.2	3.1	1.6	1.2
DBLAB-idx-date-string	3.8	3	4.3	4.1	4	4.4	4.6	4.4	4.5	4.7	4.2	4.7	37.4	4.8	2.4	4.6	3.2	2.6	5.3	5	2	5.5
LB2-idx-date-string	1.7	1.2	1.9	1.5	1.9	1.4	2.6	3.5	4.3	1.4	1.3	1.3	1	1.6	1.4	1.2	2.8	1.7	5.5	3.3	2	1.7

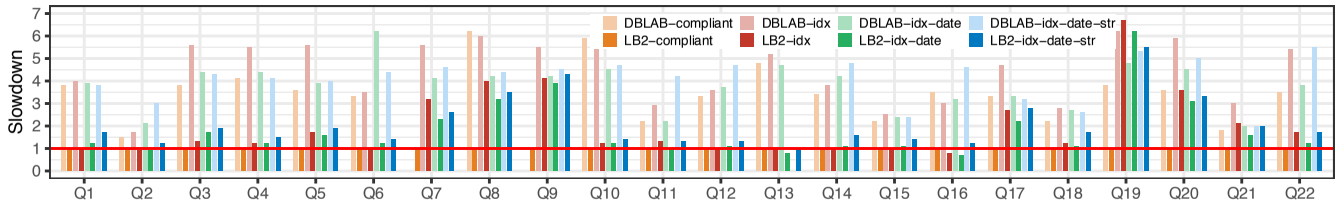


Fig. 2.9. Overhead in loading time introduced by index creation, date indexing and string dictionary on DBLAB and LB2 in SF10 using DBLAB plans (slowdown relative to LB2-compliant).

2.4.1 the performance gap is attributed to a variety of implementation details that are different between the two systems.

Finally, we can notice that using index inference may result in access paths that are slower than hash joins, e.g., Q16. Indexes are built on leaf nodes whereas hash tables are built on interior nodes with the smaller intermediate result. Therefore, a query optimizer with a cost model would avoid slower access paths and using an index every time it is available may not be the optimal solution for the query. For Q13 DBLAB is much faster than LB2 due to their use of imperative computation outside of query plans (as noted above). Curiously, enabling index optimizations causes a slowdown in DBLAB for Q13.

Date Indexes. The date indexing optimization is used when a table is filtered on a date attribute. The table is partitioned by year and month on the given attribute and the index is scanned only on the dates that satisfy the predicate. This optimization is always beneficial in both systems.

String Dictionaries. DBLAB creates string dictionaries to speed up commonly used string operations: `equality`, `startsWith`, `endsWith` and `like`. In LB2, a string dictionary is used only for the first three operations. We decided that tokenizing strings in order to optimize the `like` operation (as in DBLAB) is difficult in the general situation. A simple example is `like %green%`. This should match the word `greenway` but tokenizing over spaces, punctuation, etc. would not be correct. The line `DBLAB/LB2-idx-date-str` in the Figure 2.8 shows runtime when using the string dictionary optimization in DBLAB and LB2. Queries 3, 8, 12, 17, 19 are only using equality operations. LB2 is 35%-95% faster in Q12, Q17, Q19 whereas DBLAB is 20%, 50% faster in Q3 and Q8 respectively. Moreover, Q2 and Q14 uses `startsWith` and `endsWith`. LB2 is 30% and 60% faster in these queries. Finally, Q9, Q13 and Q16 use `like` that LB2 does not optimize. However, DBLAB does not optimize it for Q13 either. When generating the C code for Q13 in this experiment, the DBLAB executable raised a segfault that we manually fixed.

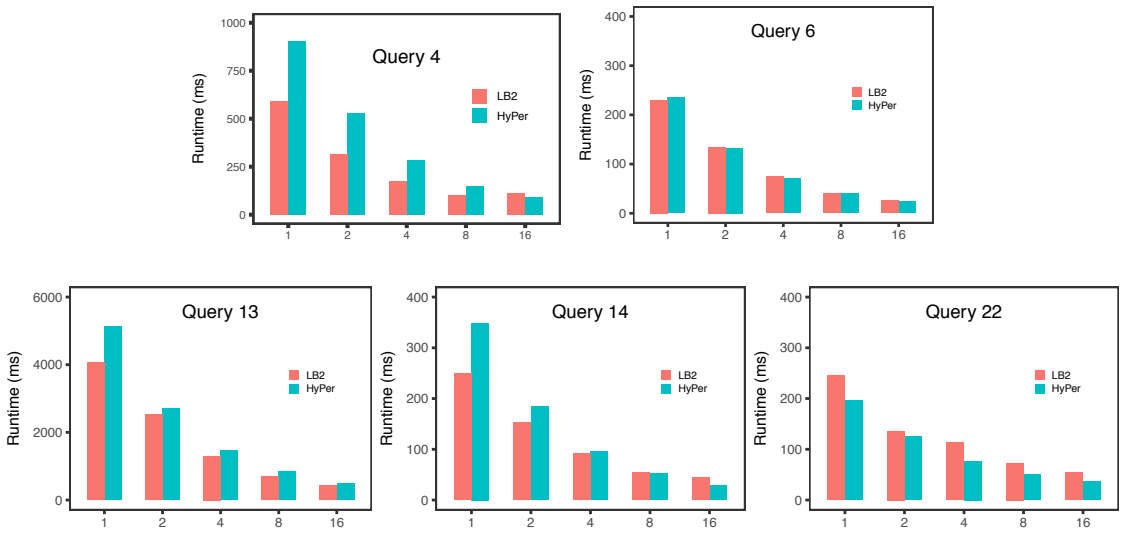


Fig. 2.10. The absolute runtime in milliseconds (ms) for parallel scaling up LB2 and HyPer in SF10 on 2, 4, 8 and 16 cores.

2.4.3 Parallelism

In this experiment, we compare the scalability of LB2 with HyPer. DBLAB does not support parallelism. We pick five queries that represent aggregates and join variants. Figure 2.10 gives the absolute runtime for scaling up LB2 and HyPer for Q4, Q6, Q13, Q14, and Q22 in SF10. Overall, the speedup of LB2 and HyPer increases with the number of cores, by an average 4×-5× in Q22 and by 5×-11× in Q4, Q6, Q13, and Q14.

At a closer look, LB2 outperforms HyPer in semi join Q4 by 50% with 2 to 8 cores. In outer join Q13, LB2 is 10%-20% faster than HyPer up to 16 cores. On the other hand, the performance of LB2 and HyPer is comparable in aggregate query Q6. Finally, HyPer outperforms LB2 in anti join Q22 by 10%-50% with 2 to 16 cores. In summary, the parallel scaling in LB2 and HyPer lie within a close range. However, the difference in implementation and parallel data structures can result in faster code. In terms of implementation, LB2 generates C code and realizes parallelism in a high-

level style using OpenMP. HyPer generates LLVM and uses Pthreads for fine-grained (Morsel-driven [115]) parallelism.

Conclusion The results of our experiments show that LB2 can compete against state-of-the-art query compilers. However, LB2's design is simpler than both DBLAB and HyPer; it is derived from a straightforward query interpreter design and does not require multiple compiler passes or additional intermediate languages.

2.4.4 Code generation and Compilation

In this experiment, we analyze the overhead of code generation and compilation using GCC in LB2 and DBLAB with the compliant and optimal configurations. The results are illustrated in Figure 2.11. The y-axis value shows both code generation and GCC compilation for each configuration.

Code generation time increases with the number of operators and subqueries, e.g., in Q2, Q5, Q8, and Q21. Compilation times are constant for any dataset size, and can often be amortized if queries are precompiled and used multiple times.

At a closer look, LB2 is faster than DBLAB in TPC-H compliant mode for code generation by an average of 3%-80%. On the other hand, LB2 is slower for Q1, Q4, Q11-Q13, Q15, and Q18-Q19 by an average of 15%-2.6×. For GCC compilation, LB2 is between 40% slower and 75% faster than DBLAB. In the optimal mode, LB2 is slower for Q15 and Q17-Q21 by up to 40% and faster for the remaining queries by up to 3.8×. For GCC compilation, LB2 is between 30% slower and 35% faster than DBLAB.

2.4.5 Productivity Evaluation

Table 2.1 summarizes the line of code (LOC) that have been added to the compliant LB2 compiler for each optimization. The original core engine of HyPer is reported to consist of around 11k lines of code [6] that uses low-level LLVM APIs to produce code in LLVM's intermediate language.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LB2-compliant	243	650	381	221	444	59	519	663	447	427	302	356	197	147	322	286	171	366	228	387	398	211
LB2-compliant GCC	175	558	421	336	664	180	436	507	498	439	320	283	203	195	228	292	214	368	349	367	449	229
LB2-opt	367	736	420	274	566	139	537	693	468	358	379	261	226	248	420	309	435	502	526	512	686	276
LB2-opt GCC	245	578	289	248	385	180	421	462	408	338	302	259	214	222	340	274	237	332	402	365	467	231
DBLAB-compliant	178	670	514	187	701	90	620	1199	745	458	266	258	164	154	124	485	182	251	198	467	410	271
DBLAB-compliant GCC	260	401	488	363	699	228	605	785	772	434	300	391	254	311	292	497	376	545	598	606	486	273
DBLAB-opt	605	2040	856	658	1783	363	1409	2671	868	700	460	632	432	417	333	772	313	425	473	492	497	403
DBLAB-opt GCC	237	324	382	333	508	207	513	551	532	392	261	321	267	294	266	348	303	342	390	424	390	316

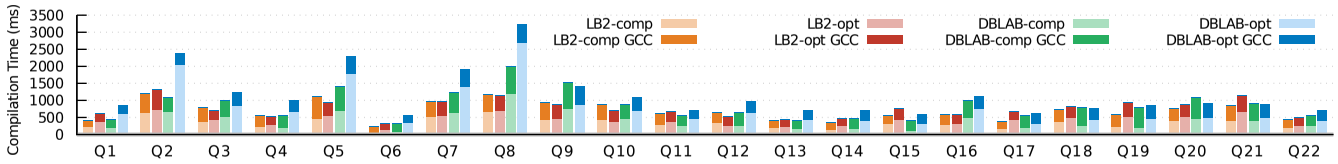


Fig. 2.11. Code generation and compilation for LB2 and DBLAB.

DBLAB is based on a specifically developed compiler framework called SC [9], with about 30k lines of code. Most or all of the facilities provides by this framework appear necessary. While the DBLAB developers claim that SC can serve as a *generic* compiler framework, there is no evidence that it is used anywhere else. By contrast, LB2 uses the LMS compiler framework [24], which has been used by a variety of groups in academia and industry for applications ranging from generating JavaScript to C, CUDA, and VHDL. LMS is only 7k lines in total, and of that, LB2 uses only a small fraction. While LMS does provide sophisticated support for multiple intermediate languages and transformation passes, LB2 does not use any of these facilities. Removing this functionality from LMS leads to a core framework of only 2k lines that are enough to support LB2.

Table 2.1.
Lines of code needed to add various optimization to LB2.

	LB2	DBLAB
Base	1800	3700
Index data structures	200	505
Compliant Indexing compilation	80	318
Non-compliant String Dictionary	150	456
Non-compliant Date Indexing	50	400
Memory Allocation Hoisting	30	186
Other	600	1000
Total modified	1100	2800

2.5 Discussion

In this section, we draw some general insights and aim to clarify some specific points of discussion, notably regarding single-pass vs. multi-pass compilers.

Templates Expansion vs. Many Passes: A False Dichotomy. The authors of DBLAB [9] motivate their multi-pass architecture by claiming that "all existing query compilers are template expanders at heart" and that "template expanders make

cross-operator code optimization impossible". But this argument confuses *first-order*, context-free, template expansion with the richer class of potentially more sophisticated single-pass compilers, which can very well perform context-dependent optimizations. Especially in the context of Futamura projections, we have shown that the structure of the query interpreter that is specialized into a query compiler has a large effect on the style of generated code, and can achieve all the optimizations implemented in DBLAB.

Are Many Passes Necessary? Staying with the specialization idea of Futamura projections, we can specialize away further abstraction levels without auxiliary transformation passes. Hence, we demonstrate that for standard relational workloads, no additional intermediate languages besides the query plan language and the generated C level are necessary.

Do Many Passes Help or Hurt? Multiple passes help if one step of expansion/specialization can expose information to an analysis that is not present in the source language. For database queries, the source language of query plans is already designed to contain all relevant information. If some information is missing, it can likely be added to the execution plan language. Hence, we find that additional abstraction levels in between relational operators and C code do not provide tangible benefits. To the contrary, some information may get lost and has to be arduously recovered. Multiple transformation passes depend on analysis of imperative code that manipulates low-level data structures, which is notoriously difficult.

In contrast, the database community has already solved the query optimization problem for interpreted engines, and cost-based optimizers that produce good plans are available. In particular, deciding when an index can be used to speed up a join query is readily solved by looking at the query plan. Trying to make such a decision by analyzing low-level code generated from a physical plan (as DBLAB does) seems overall backward, and unlikely to scale to realistic use cases.

When to use Multiple Intermediate Languages? We have argued that we do not need stacks of multiple intermediate languages for compiling relational query

plans. So when do we need them? A key use case is in combining multiple front-end query languages (DSLs), e.g., SQL and a DSL for machine learning or linear algebra (as in Delite's OptiQL and OptiML [27]). First, domain-specific optimizations need to be applied on each DSL independently, e.g., arithmetic simplification, join ordering, etc. After that, a series of compiler transformations may be needed to translate both DSLs into a common intermediate core, where loop fusion and other compiler level optimizations can be performed before code generation.

In a relational system, the query plan DSL is essential. We have to perform cost-based optimization of join ordering before we can think about generating code. The situation is similar in a linear algebra DSL, where we have to perform arithmetic optimizations before switching representations from matrices and vectors to loops and arrays.

2.6 Conclusions

We advocate that query compilation need not be hard and that a low-level coding style on the one hand and the added complexity of multiple compiler passes and intermediate languages, on the other hand, are unnecessary. Drawing on the old but under-appreciated idea of Futamura projections, we have presented LB2; a fully compiled query engine, and we have shown that LB2 performs on par with, and sometimes beats the best compiled query engines on the standard TPC-H benchmark. Specifically, LB2 is the first query engine built in a high-level language that is competitive with HyPer [6], both in sequential and parallel execution. LB2 is also the first single-pass query engine that is competitive with DBLAB [9] using the full set of non-TPC-H-compliant optimizations. In conclusion, we demonstrate that highly efficient query compilation can be simple and elegant, with not significantly more implementation effort than an efficient query interpreter.

3. ARCHITECTING A QUERY COMPILER FOR SPATIAL WORKLOADS

This Chapter is based on the paper *On Supporting Compilation in Spatial Query Engines (Vision Paper)* which appeared at the Proceedings of the 2016 International Conference on Advances in Geographic Information Systems SIGSPATIAL GIS'16 [30] and the technical report *How to Architect a Query Compiler for Spatial Workloads* [116].

Modern location-based applications rely extensively on 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 Chapter, 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 structure layer. We extend the LB2 query compiler, with spatial data types, predicates, indexing structures, and operators. We show that the spatial extension matches the

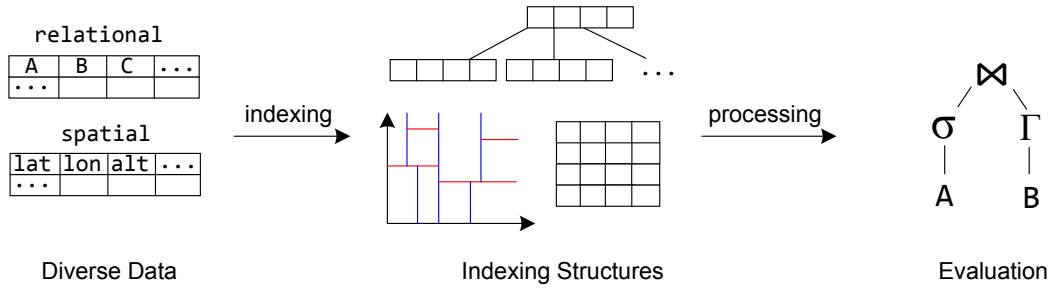


Fig. 3.1. Overview of a spatial extension.

performance of specialized library code and outperforms relational and map-reduce extensions.

Spatial processing is commonly realized as an extension to a relational database management system (RDBMS), e.g., PostGIS [15], Oracle Spatial [43], etc., or a cluster computing framework, e.g., GeoSpark [16], Simba [17], etc. Figure 3.1 gives an overview of such spatial extensions that add spatial data types, predicates, indexing structures, and operators. 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.

Bottlenecks in Existing Spatial Engines Time spent in evaluating spatial predicates makes up a large part of spatial processing runtime. For instance, a spatial range join operator 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 [18], Geos [19] for evaluating spatial predicates. However, these high-level libraries are opaque to the query 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 in-database data structures are typically implemented in a generic form to unify various types of structures behind a

common interface. For instance, the SP-GiST [10] indexing framework in PostgreSQL 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. Additionally, most relational engines are interpreters that run precompiled code to process data in high-level form, i.e., `Record` structures (in contrast to query compilation, i.e., generating specialized low-level target code per query plan at runtime as demonstrated in recent query compilers [6, 7, 9, 31]).

Similarly, the performance of *spatial Spark extensions* is suboptimal, in general, due to the following issues [29]. Spark is internally designed for distributed shared-nothing environments. At runtime, Spark generates multiple code regions for its resilient distributed datasets RDDs. Traversing separate regions at runtime is expensive due to various reasons, e.g., main-memory data structures, JVM overhead, etc. Moreover, spatial datasets need to be explicitly partitioned to avoid scanning the entire data under Spark's distributed execution model. Data partitioning is expensive especially when performed inside Spark due to internal overheads of RDD processing. 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. We discuss some of the unique challenges next.

3.1 Challenges of Compiling Spatial Workloads

Over the previous years, several approaches have been proposed for state-of-the-art query engines to compile relational queries to native code. For instance, HyPer uses low-level code generation [6] that mixes generated LLVM for isolated *pipelines* within a query plan with pre-written `C++` kernels for data structures and individual operators. LegoBase [7] and DBLAB [9] are high-level query compilers that use multi-

ple intermediate languages and compiler passes to generate optimized code. The LB2 query compiler [31] demonstrates that a simple one-pass compiler phase is sufficient to efficiently compile relational queries.

However, at the time of writing this dissertation, none of the previous query compilation techniques have been applied to architect a spatial query compiler *yet* since (i) spatial processing is fundamentally different from relational workloads in terms of data dimensions, indexing structures and query evaluation (ii) implementing any of the previous compilation approaches in spatial engines would not necessarily result in significant speedups, unless the generated spatial indexing structures and spatial query evaluation code are *carefully* specialized to eliminate interpretive overhead.

In particular, as we discuss below, the architecture pioneered by HyPer and adopted by many other systems including Spark SQL, which uses on-the-fly-generated code only for the high-level control structure but relies on pre-written libraries of data structures is unlikely to scale to efficient spatial workloads.

Identifying the Dominating Query Evaluation Cost Relational data is typically dense, and sorting and indexing methods, e.g., hash tables or B-tree variants, are sufficient to achieve efficient evaluation. As a consequence, works on relational query compilation have primarily focused on removing the interpretive overhead inherent in traversing a query execution plan and evaluating fine-grained logical predicates that are individually inexpensive. On the other hand, spatial processing cost is rather dominated by evaluating *coarse-grained* spatial predicates that are individually computationally epensive and accessing complex spatial data structures that index diverse types of spatial data, e.g., points, rectangles, etc.

Choosing a Query Compilation Approach As discussed earlier, spatial data are multi-dimensional, diverse and sparse, i.e., potentially covering a large space with a less regular structure. Therefore, we argue that 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 in-

dexing structures. The question then is: *how to choose a query compilation approach for spatial workloads?*

At first glance, a compilation approach similar to HyPer's where data structures are pre-implemented in a high-level language, i.e., `C++` and the query evaluation encoded in LLVM assembly would appear as an attractive approach to compile spatial workloads. However, taking an in-depth look at this style of code generation as it is adopted in systems like Spark [29] 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.

Furthermore, 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.

For the rest of this Chapter, we demonstrate that the underlying idea of constructing simple but highly efficient query compilers not only applies to purely relational workloads but carries over to processing spatial workloads. Given an optimized spatial query plan, the key idea to address the previous challenges is to specialize spatial and predicate implementation (Section 3.2.2) in addition to facilitating building optimized data structures (Section 3.2.3) using *programmatic specialization*, the same technique LB2 already uses for generating efficient code for relational queries. We thus extend the LB2 [31] main-memory query compiler with spatial compilation. In particular, spatial predicates, indexing structures, and spatial operators.

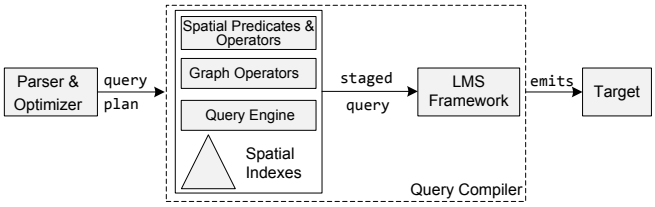


Fig. 3.2. Extending LB2 with spatial processing.

3.2 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 [117], applying adaptive query processing techniques [118], 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 LB2 with spatial data types, spatial predicates, spatial indexing structures, and spatial operators.

3.2.1 LB2-Spatial Overview

The LB2-Spatial extension compiles spatial queries into optimized native code. Figure 3.2 shows a high-level architecture of LB2-Spatial. The front-end accepts SQL queries, spatial queries (adopting the syntax in [15,17]), 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 select and join operators: rectangle range, distance, and kNN (each operator is implemented without using an index, with an index, serial and parallel). The index-based kNN operator implements the branch-and-bound algorithm from Roussopoulos et al. [119]. Similar to LB2, the data structures used by spatial operators, e.g., spatial indexes are implemented using LB2's high-level abstractions, e.g., various array buffers that generate low-level code.

As discussed earlier in Chapter 2, 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 (written 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 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 leverages the extensible Spark SQL [5] front-end and Catalyst optimizer as follows. The SQL grammar rules are extended with spatial keywords, e.g., within, kNN, etc. needed to form spatial queries. Similarly, the Catalyst optimizer is extended with rules that identify the spatial constructs and also performs optimizations as pushing down spatial predicates. Since Spark SQL is implemented in Scala, LB2-Spatial only needs to map Spark's optimized query plan into LB2's back-end operators and data structures[1].

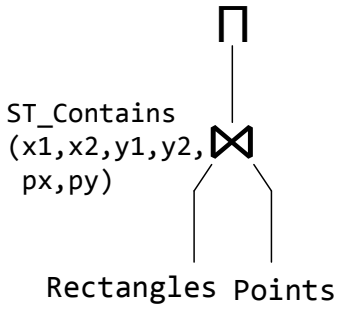
Query Example Throughout the remainder of this Section, we are going to use the range join query illustrated in Figure 3.3a (expressed in SQL and spatial API) as a running example for compiling spatial queries. Given two tables, `Rectangles` and `Points`, the range join query finds the points located inside the rectangle areas.

Figure 3.3b shows the straightforward implementation of 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 6-10 shows the actions that are performed

[1]The Spark SQL version used is 2.2

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

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)

Fig. 3.3. (a) Rectangle range join query in SQL and QPlan (b) the implementation of NestedLoopRangeJoinOp in LB2-Spatial.

on the `left` operator where the records obtained from `Rectangles` table are inserted into a buffer to prepare for the joining operation. Moreover, Lines 11-18 encodes the join operation where a pair of rectangle and point values are extracted in Lines 13-16 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 3.2.2-3.2.5, we discuss the elements of compiling spatial queries: staging spatial predicates, specializing indexing structures, supporting parallelism in shared-memory and spatial applications.

3.2.2 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 [18] in Java, Geos [19] in C++, etc. are used for spatial predicates' evaluation. However, using libraries for spatial predicates evaluation result in generating suboptimal code. First, the generic library interface adds 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 3.4a shows a staged implementation for a simplified[2] `ST_Contains` predicate used in range queries that tests whether a point (x, y) is located inside a minimum bounding rectangle (x1, x2, y1, y2). The `Rep` constructor denotes that all parameter values are future stage (i.e., only known at runtime). Figure 3.4b shows a partial code for a filter operation that uses the staged `ST_Contains` to check whether a point is contained inside a rectangle. Figure 3.4c shows the generated code in `Scala` (LB2-Spatial generates `Scala` and `C`).

[2]typically a compile-time abstractions Rectangle and Point are used.

```
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")
// ...

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)

Fig. 3.4. Compiling spatial predicates (a) staging `ST_Contains` predicate using `Rep` type constructor (b) application code that uses `ST_Contains` (c) generated code in `Scala`.

3.2.3 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, indexes are created based on the key attributes specified in the query plan. Finally, indexes are made available for the query evaluator.

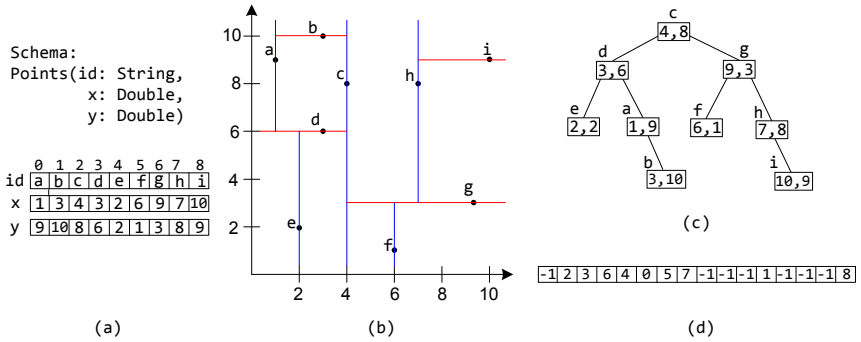


Fig. 3.5. Specializing k-d tree index in LB2 (a) data in column-layout (b)-(c) standard k-d tree implementation using pointers and (d) using a flat array.

Data 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 when 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 [10] 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 (instead of using generic libraries) and uses data schema to specialize access to indexes, and hence eliminating dispatch overhead. Moreover, LB2 flattens tree structures into arrays to optimize data access and layout.

Figure 3.5 demonstrates the specialization of the k-d tree index in LB2-Spatial. Data is stored in a column-oriented layout for optimized storage and access. Figure 3.5b-c shows the space layout, and the standard k-d tree using pointers. The flattened k-d tree is shown in 3.5d 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 [120].

```
1  // left: Rectangles table,
2  // right: spatial index on Points table
3  // Index range operator that uses a spatial index on the right table
4  class IndexRangeJoinOp(left: Op, right: Op) (idxName: String)
5                        (rectFun: keyRect) extends Op {
6  // obtain index for right table
7  val index: Index = right.getIndex(idxName)
8  def exec(cb: Record => Unit) = {
9    left.exec{ lTuple =>
10      val rectangle = rectFun(lTuple)
11     // use index to find the points located inside rectangle
12     index(rectangle).RangeRectangle{ pnt =>
13       cb(merge(pnt, rectangle))
14 }}}}
```

Fig. 3.6. The implementation of index range join operator.

All indexing structures in LB2-Spatial are implemented using array abstractions that generate optimized code similar to LB2's [31]. 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 3.6 shows the implementation of an index range join operator that uses a spatial index (in contrast to the nested loop version from Figure 3.3). 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.

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 main-

tain the list of k nearest points during index traversal which may update the neighbors list. In LB2-Spatial, we recognize the performance of auxiliary data structures is important to query runtime. Therefore, all auxiliary data structures e.g., Stack, heap, etc. are implemented optimized flattened fashion.

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 points 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.

3.2.4 Parallelism

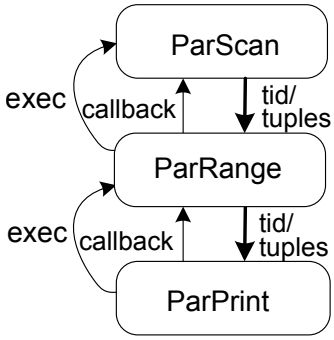
LB2 supports parallelism on shared memory systems using OpenMP [110] where blocks of parallel code are generated with OMP parallel annotations. The key elements to implement parallel evaluation are summarized as first, defining code generation constructs that emit various OMP annotations.
Second, modifying 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 3.7a-b shows the definition of LB2's parallel pipeline wrapper that enables generating parallel code by adding thread variables and thread callbacks

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

(a)



(c)

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

(b)

Fig. 3.7. (a)-(b) Parallel operator class and parallel pipeline wrapper (c) the interactions between operators within a parallel query execution pipeline (adapted from [31]).

(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 3.7c 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.

3.2.5 Spatial Applications

Traditional RDBMS provide procedural languages, e.g., PL/pgSQL to support user applications that interleave spatial processing and user-customized 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 can be optimized and generated (not only the operators). In the following, we demonstrate writing a spatial application using the spatial processing API in `Scala`.

Consider supporting a customized spatial-textual ranking operation that first filters records that lie within a specific distance (facilitated by an index) then the textual attribute is processed to assign each record a score. LB2-Spatial facilitates injecting user customized code using callbacks. Figure 3.8 demonstrates a program that performs spatial-textual ranking. Lines 1-13 creates an R-tree index named `tweetIdx` on `(lon,lat)` attributes of the table `tweets` and loads the `cities` table. The function `f` defined in Lines 19-24 encodes simple ranking actions to be performed on `tweetIdx` records. The `for` loop starting at Line 26 reads records from table cities and probes `tweetIdx`. Notice that ranking code is injected into the index distance operation in Line 30.

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 imple-

```
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 lat:
7     Double; val lon: Double; val tweet: String}
8  // building an R-tree index on table Tweets
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 lat: Double; val lon: Double;}
15      // loading table citites
16  val cities = load[citiesRec](file_cities)
17  // constructing a (lon, lat) key
18  val lonlatkey = x=>pointRec(x.lon, x.lat)
19  val eps = 1.5
20  type RankRec = Record{ val rank: Long }
21  // user code for ranking records
22  val f = { tuple =>
23    val t = record_select(tuple,"tweet")
24    val rankVal =
25        if (t.contains("keyword"))
26           10 * t.length
27        else t.length
28    val rec = new Record{val rank = rankValue}
29    printRecord(merge(tuple,rec))
30  }
31  // scanning Citites and probing tweetIdx
32  for(i<-0 until cities.length){
33    val city = cities(i)
34    val cityKey = lonlatkey(city)
35    // ranking code as callback
36    tweetIdx(cityKey).distance(f, eps)
37 }
```

Fig. 3.8. Compiling spatial applications in LB2-Spatial.

mentation of UDFs in LB2-Spatial is *not* opaque to evaluation, and does not incur a performance penalty.

3.3 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, a spatial extension to relational engine PostGIS [15] and two spatial Spark extensions Simba [17] and Geospark [16].

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 extension when scaling up the number of cores. The third experiment measures the total memory consumed by LB2, 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 due to Spark's distributed execution model. For the single-core setup, we show that LB2-Spatial outperforms spatial spark extensions and PostGIS in spatial join queries by 12×-299×. For scaled-up execution, LB2-Spatial is 10×-20× faster than spatial Spark extensions.

Datasets and Queries Table 3.1 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 rest attributes. We added a serial numeric attribute to identify data records. The Open Street Map (OSM) consists of 200 million points

and 114 million rectangles obtained from a performance evaluation study that compares the performance of several spatial Spark extensions [121]. The last dataset is synthetic and consists of one million randomly generated points.

Table 3.2 shows the queries we used in experiments. The syntax of kNN join is adapted from [17]. We run each query five times and record the median reading.

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.

3.3.1 Single-core Spatial Join

In the first experiment, we compare LB2-Spatial with Simba, GeoSpark, and PostGIS in range join, distance join, and kNN join queries using only a single-core. For range join we use the OSM points, and rectangles datasets where a spatial index is built on the left table of size 200 million, and the size of the right table is one million. Moreover, we set up the value of k to 5. For distance join and kNN join we use the tweets dataset where a spatial index is built on the left table of size 200 million. Moreover, the size of the right table is one million. For the kNN join query, we reduce the index size to 10 million as in [17, 121].

Figure 3.9 gives the absolute runtime for three spatial join queries: distance join, range join and kNN join. Overall, LB2-Spatial outperforms Simba, GeoSpark, and PostGIS in all three join queries. Moreover, PostGIS outperforms Simba and GeoSpark in index distance join query. On a query-by-query analysis, LB2-Spatial outperforms PostGIS, GeoSpark, and Simba in the distance join by 25×, 30×, and 299× respectively. Similarly, LB2-Spatial outperforms PostGIS in range join by 19× and 28× respectively. Finally, LB2 is 13× faster than Simba in kNN join.

Table 3.1.
Spatial datasets that are used in evaluating LB2-Spatial.

Dataset	Geometry	#Records	Size(GB)
Tweets	Point	1 billion	32
OSM Nodes	Point	200 million	4.3
OSM Rectangle	Rectangle	114 million	14.3
Random	Point	1 million	0.03

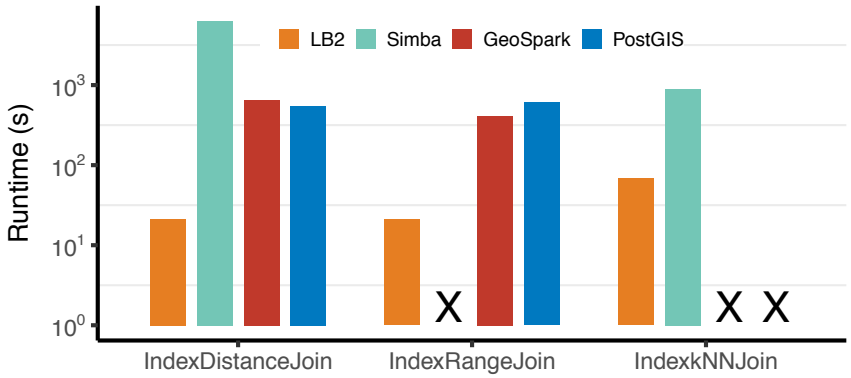
Rectangle Range	`SELECT *`
	`FROM Points`
	`WHERE ST_Contains(ST_PolygonFromEnvelope(x1,x2,y1,y2),`
	`Points.pointshape)`

Range Join	`SELECT *`
	`FROM Polygons, Points`
	`WHERE ST_Contains(Polygon.polygonshape, Points.pointshape)`

Distance Join	`SELECT *`
	`FROM Points1, Points2`
	`WHERE ST_Dwithin(Points1.pointshape,`
	`Points2.pointshape, distance)`

kNN Join	`SELECT *`
	`FROM Point1 AS P1 KNN JOIN Point2 AS P2`
	`ON POINT(P2.x, P2.y) IN KNN(POINT(P1.x, P1.y), k)`

Table 3.2.
Queries that are used in evaluating LB2-Spatial.



	Distance Join	Range Join	kNN Join
LB2-Spatial	21.1	21	68.4
Simba	6325.8	-	888.2
GeoSpark	653.7	412.9	-
PostGIS	544.4	606	-

Fig. 3.9. The absolute runtime for LB2, Simba and PostGIS in distance join, range join and kNN join.

In general, map-reduce extensions similar to Simba and GeoSpark are optimized for distributed execution on large clusters. The single machine performance is sub-optimal 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 [122, 123] provides a spatial indexing library written in `C++` for a set of common spatial indexing structures and operations. For this experiment, we

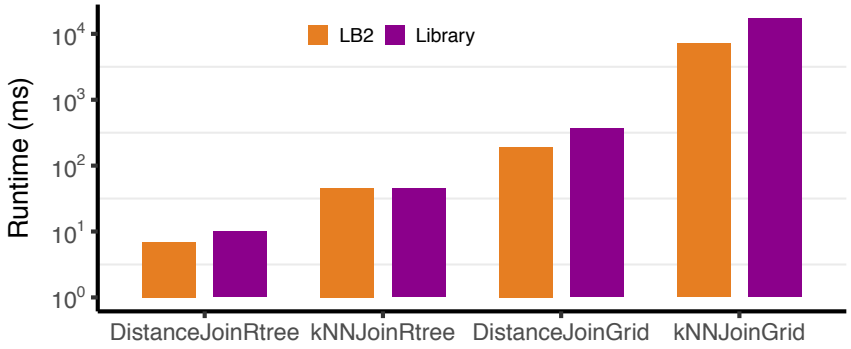
extend the R-tree and grid from the previous library with distance join and kNN join operations to evaluate LB2-Spatial performance with specialized code (i.e., without RDBMS overhead). We use a randomly generated dataset where a spatial index is built on the left table of size 1 million, and the size of the right table is 1000.

Figure 3.10 shows the absolute runtime of performing distance, and kNN join using R-tree and grid in LB2-Spatial and stand-alone indexing library. The performance of R-tree-based queries is comparable since tree indexes are useful in data pruning and hence less time is spent in computations. For the case of grid index, LB2-Spatial outperforms the library code by 1.9×, and 2.4× in distance join and kNN join respectively. The 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.

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 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 [121] and submit a batch of 100 queries as in and compute the throughput as the number of queries executed per minute [3].

Figure 3.11 shows the throughput of range query in LB2-Spatial, Simba and GeoSpark for σ = 1, 10, 50 and 100. LB2's throughput for highly selective range predicate is 7500 and 134 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.,

[3]In this experiment, a throughput value less than one is counted as zero.



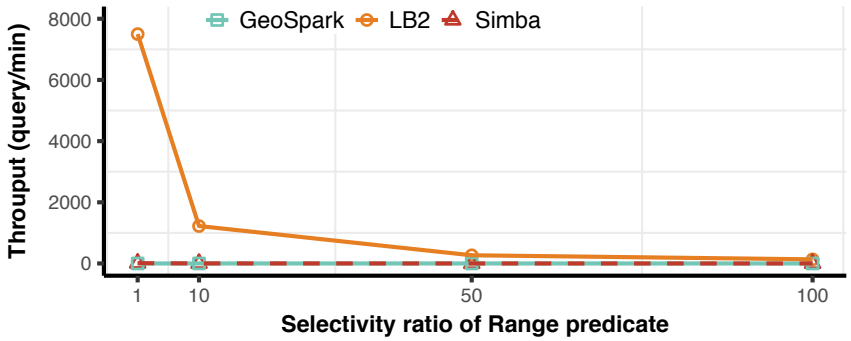
	DJ-R-tree	kNNJ-R-tree	DJ-Grid	kNNJ-Grid
LB2-Spatial	7	46	192	7155
Library	10	45	367	17108

Fig. 3.10. The absolute runtime for LB2 and handwritten code in distance join and kNN join using Rtree and grid.

9 and 2 for $\sigma = 1$, 10 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. Moreover, we exclude the spatial Spark systems since Spark does not scale up well in a single-core [29, 121].

Figure 3.12 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 (the fanout of spatial indexes is typically large where the tree height increases slowly, e.g., the R-tree node size in LB2-Spatial and Simba is 20).



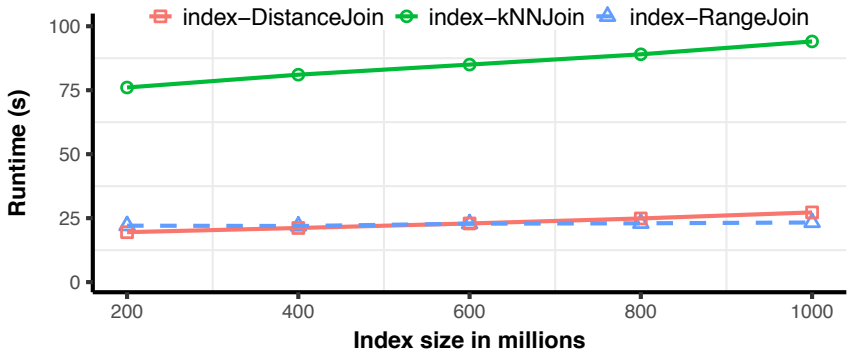
	1	10	50	100
LB2-Spatial	7500	1224	267	134
Simba	9	2	0	0
GeoSpark	0	0	0	0

Fig. 3.11. The selectivity ratio of range predicate.

3.3.2 Parallel Spatial Join Queries

In this experiment, we compare the scalability of LB2-Spatial with GeoSpark, and Simba on distance join and range join queries. We use the OSM dataset where a 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. [124]. 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 3.13 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 appears as having better speedup than LB2-Spatial at 24 cores, i.e., 8× to 11×. However, examining the absolute running

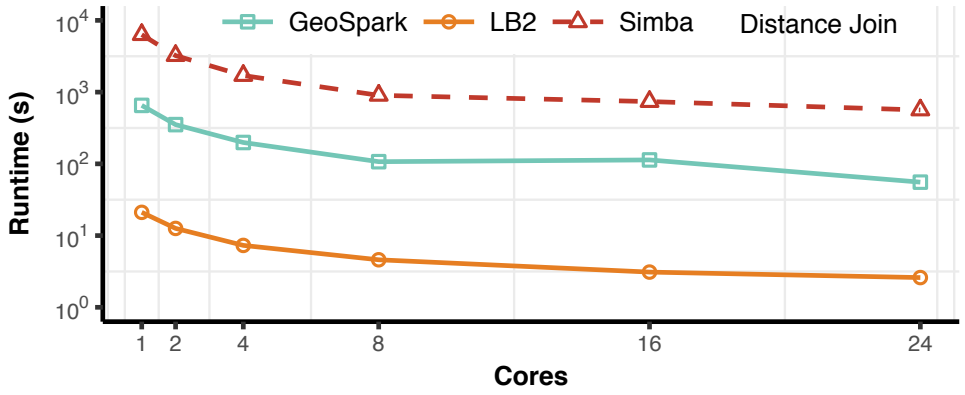


	200M	400M	600M	800M	1B
Range Join	22.07	21.93	22.83	22.98	23.31
Distance Join	19.52	21.15	22.92	24.88	24.88
kNN Join	76.05	81.06	85	89.02	94.05

Fig. 3.12. The scalability of LB2-Spatial range query with increasing the index size in a single-core.

times, LB2-Spatial is 21× 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 overheads.

Figure 3.14 gives the absolute runtime for scaling up LB2-Spatial and GeoSpark in a 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.



Runtime	1	2	4	8	16	24
LB2-Spatial	21.1	12.6	7.3	4.6	3.1	2.6
Simba	653.7	352.5	198.5	107.2	113.5	55.6
GeoSpark	6325.8	3229	1702.6	901.8	738	563.4

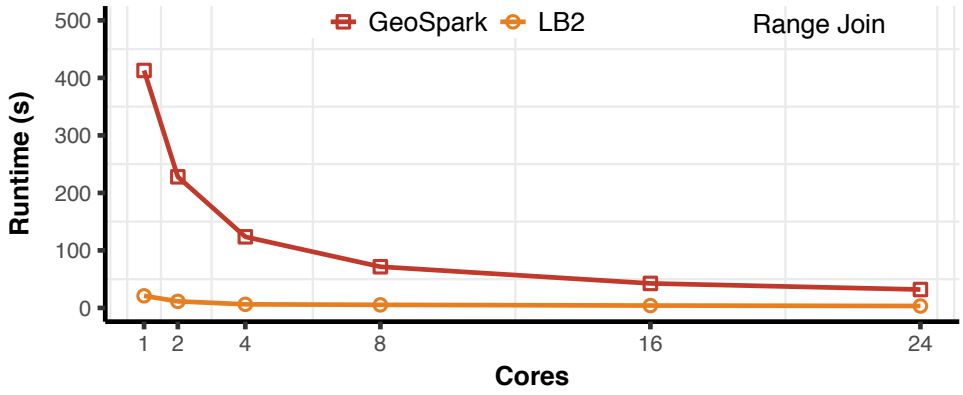
Speedup						
LB2-Spatial	1	1.7	2.9	4.6	6.8	8
Simba	1	1.9	3.3	6.1	5.8	11.8
GeoSpark	1	2	3.7	7	8.6	11.2

Fig. 3.13. The absolute runtime in seconds (s) for parallel scaling up LB2-Spatial, GeoSpark, and Simba in distance join on 2, 4, 8, 16 and 24 cores for Tweets dataset.

3.3.3 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 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[4] which gives the memory occupied by a cached RDD. We observed

[4]Memory consumption for Spark's RDDs cannot be collected programmatically [121, 125].



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

Fig. 3.14. The absolute runtime in seconds (s) for parallel scaling up LB2-Spatial, GeoSpark, and Simba in range join on 2, 4, 8, 16 and 24 cores for Tweets dataset.

that storage memory value after constructing a spatial index on 200 million records is approximately 21GB and 48GB in Simba, and GeoSpark respectively. The difference is due to index serialization in Simba [17].

Table 3.3 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 [125]. Furthermore, spatial Spark extensions perform data partitioning that requires extra storage for sampling and processing [121]. On the other hand, LB2-Spatial leverages the dataset size, when available, and incrementally in-

Table 3.3.
Total memory consumed (in GB) by LB2, 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	-

creases the data structures size otherwise. For instance, the raw size of the points and rectangles datasets used in range query is approximately 18.6GB. LB2-Spatial consumed only additional 3GB to perform this operation.

3.3.4 Productivity Evaluation

Table 3.4 summarizes the development effort in terms of line of codes (in `Scala`) needed to extend LB2 with spatial processing. The front-end work consists of first extending the Spark SQL front-end and optimizer with spatial keywords and rules. Second, mapping the optimized query plan to LB2-Spatial's operators[5]. Overall, the front-end 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, LB2-Spatial consists of 2958 lines.

3.4 Conclusions

In this Chapter, we have added spatial query compilation inside the LB2 main-memory query compiler. We support parallelism for shared memory using OpenMP

[5]We only count the lines needed for mapping spatial operators.

Table 3.4.
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

and thread-aware data structures. The spatial extension matches the performance of the library code. In single-core distance join, range join and kNN join queries, LB2-Spatial outperforms spatial spark extensions and PostGIS in spatial join queries by 12×-299×. For scaled-up execution, LB2-Spatial is 10×-20× faster than spatial Spark extensions.

4. COMPILING GRAPH QUERIES

This Chapter is based on the paper *Towards Compiling Graph Queries in Relational Engines* which is accepted at the 2019 International Symposium on Data Base Programming Languages DBPL'19 [126].

The increasing demand for graph query processing has prompted the addition of support for graph workloads on top of standard relational database management systems (RDBMS). Although this appears like a good idea — after all, graphs are just relations — performance is typically suboptimal, since graph workloads are naturally iterative and rely extensively on efficient traversal of adjacency structures that are not typically implemented in RDBMS. Adding such specialized adjacency structures is not at all straightforward, due to the complexity of typical RDBMS implementations. The iterative nature of graph queries also practically requires a form of runtime compilation and native code generation that adds another dimension of complexity to the RDBMS implementation, and any potential extensions.

In this Chapter, we demonstrate how the idea of the first Futamura projection, which links interpreted query engines and compilers through specialization, can be applied to compile graph workloads in an efficient way that simplifies the construction of relational engines that also support graph workloads. We extend the LB2 main-memory query compiler with graph adjacency structures and operators. We implement a subset of the `Datalog` logical query language evaluation to enable processing graph and recursive queries efficiently. The graph extension matches, and sometimes outperforms, best-of-breed low-level graph engines.

Graphs are Relations It is often desirable for graph and relational data to co-exist and be processed together, which naturally suggests representing graphs as relations on top of an existing RDBMS. The unique advantage of such an approach is that the

core data management operations are all provided by the RDBMS and a graph extension would only need to implement front-end algorithms and functionality. However, the execution pattern of graph workloads is often dominated by long-running loops over the graph structure where each iteration performs computations on the adjacency of some vertices (e.g., set intersection during triangle counting operation). The performance of relational graph extensions is therefore often stymied by the interpretive nature of typical relational engines and the lack of specialized data structures for adjacency relations. Instead, internal RDBMS data structures are often implemented in a generic form to unify various types of data layouts behind a common interface.

Stand-alone Graph Processing To tackle the challenges of relational graph extensions, several stand-alone graph engines have been developed (e.g., high-level Neo4j [11], low-level Snap-Ringo [12]) that process graphs in native adjacency structures. While graph processing in stand-alone systems achieves higher performance and is often more expressive in describing graph operations (e.g., shortest paths, centrality, etc.) than relational queries, there exists a high development cost in terms of core data management and loss of interoperability with front-end and back-end systems that integrate with RDBMS.

The Complexity Dimension for Building Specialized Relational Graph Engines Realizing efficient relational graph processing requires combining relational evaluation with specialized graph structures and operations, which is an uphill battle due to the difficulty of modifying the internals of a mature RDBMS (typically several million lines of code). Besides, the key performance challenge in RDBMS is the interpretive overhead associated with processing data in high-level form (e.g., generic libraries for hash maps) rather than generating optimized low-level code (as precisely described in Neumann's work [6]). Even supporting a minimal operational compiled path for graph queries entails writing thousands of lines of low-level code (e.g., programmatic LLVM API) that permeates large parts of the query engine code. *En*

masse, the complexity of extending RDBMS with graph processing and compilation does not add up linearly; in fact, it *multiplies!*

In this Chapter, we demonstrate that the underlying idea of constructing simple but highly efficient query compilers not only applies to purely relational queries but carries over to graph queries. The key challenge is that graph processing relies on efficient traversal of adjacency structures. The key idea to address this challenge is to facilitate building optimized data structures using *programmatic specialization*, the same technique LB2 already uses for generating efficient code and indexing structures for relational queries.

We extend the LB2 [31] main-memory query compiler with graph compilation, in particular, graph data structures, graph operators, and corresponding support for shared-memory parallelism. We also implement the semi-naive evaluation algorithm [127] to support graph and recursive queries.

4.0.1 Background: Datalog and Recursive Queries

Many graph operations, e.g., transitive closure, shortest paths etc. are simpler when expressed using recursion. The `Datalog` [127] logical query language enables expressing recursive and graph queries succinctly. `Datalog` is used to define rules where a rule consists of a head and body of the form *predicate(term1, term2, ...)*. Consider the following rule.

```
Colleagues(A,B) :- Employee(C,A), Employee(C,B)
```

In other words, *A is a colleague of B if, for some C, C is the Department of A and C is the Department of B*. Furthermore, the previous rule is a schema that is used to define propositional implications, e.g.,

```
Colleagues(Joe,Sally) :- Employee(CS,Joe), Employee(CS,Sally)
```

Facts are rules without a condition part, e.g., `Employee(Math,Ann).` `Datalog` is widely used as a graph query language [92, 95–97] due to its expressiveness and the ability

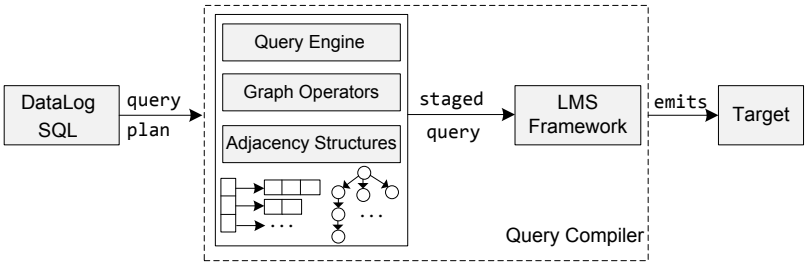


Fig. 4.1. Extending LB2 with graph processing.

to write recursive queries succinctly. For example, transitive closure can be expressed as follows:

```
Path(x,y) :- Edge(x,y)
Path(x,y) :- Path(x,z),Path(z,y)
```

The semi-naive evaluation strategy [127] is an iterative bottom-up evaluation of recursive queries. In each iteration, the least fixpoint is computed by instantiating all subgoals of the existing rules until no new tuples are discovered. Thus, it avoids repeating computations and focuses on the derived deltas from the previous iterations.

SQL-99 added `WITH RECURSIVE` clause to support recursive queries using Common Table Expressions (CTE). Recursive CTEs are incrementally evaluated and maintained (i.e., equivalent to the semi-naive evaluation). The following SQL query encodes the transitive closure operation from `src = 1`.

```
CREATE TABLE Edges (src Int, des Int);
WITH RECURSIVE TC as (
 SELECT src as dst FROM Edges WHERE src = 1
 UNION
 SELECT TC.dst FROM TC, Edges WHERE Edges.dst = TC.dst
)
SELECT * FROM TC;
```

4.1 LB2 + Graph Queries

Existing relational approaches for graph processing fall into three categories. First, translate graph operations into SQL procedures [68]. Second, adding a specialized

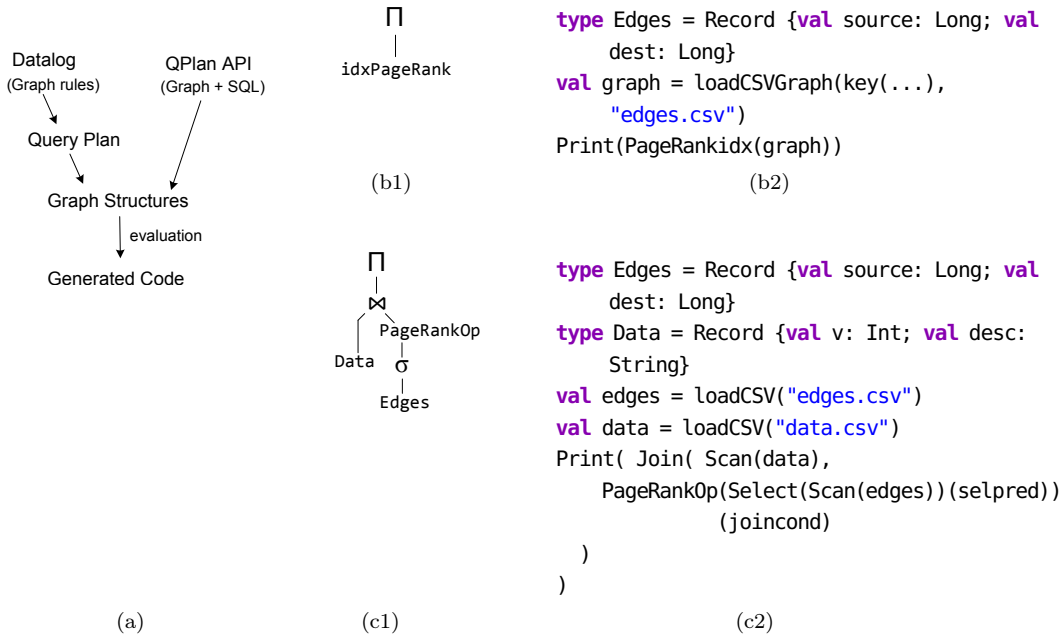


Fig. 4.2. Graph front-end in LB2 and graph query evaluation pipelines.

graph processing layer that extracts and processes graph data externally [79]. Third, extending the query engine with graph structures and operators [128] which enables processing pipelines that mix relational and graph operators. LB2 follows the last approach. (Queries implemented in any of the previous approaches could also be compiled into low-level code by extending LB2).

The extended LB2 engine compiles graph and recursive queries into optimized native code. Figure 4.1 shows a high-level architecture of the extended LB2 system. The front-end accepts SQL queries, `Datalog` rules and QPlan (a domain-specific language DSL to compose query plans). LB2 is extended with three graph structures: adjacency list, flattened adjacency list and trie. LB2 supports the following graph operations: PageRank, triangle counting, single source-destination shortest path, transitive closure, and all pairs shortest paths. Furthermore, LB2 implements shared-memory parallel operators OpenMP PageRank and OpenMP triangle counting. As discussed earlier in Chapter 2, 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 target program that implements the query evaluation without the interpretive overhead of processing static input (i.e., query pipeline).

Although the graph extension appears as if it is only wired to execute specialized queries connecting LB2 with an optimizer allows executing arbitrary recursive queries by composing a plan that employs recursive join implementation.

4.1.1 Graph Data Loading

Data in LB2 is processed *in-situ* without an explicit preloading phase. The query plan contains the necessary information to identify key attributes used to create indexing and adjacency structures. At loading time, the data loader processes relational graph data and creates the specified adjacency structures. For the case of graph-only queries, LB2 provides a data loader that can process data in compressed row storage (CRS). This data loader is most beneficial for building flat adjacency structure as it minimizes the memory allocated for preprocessing data during the graph creation phase.

4.1.2 Graph Processing

LB2 provides two front-ends for graph processing as illustrated in Figure 4.2a. Graph-only operations, e.g., traversals, shortest paths, etc., are encoded as `Datalog` rules or QPlan whereas queries that mix SQL and graph processing are written using QPlan DSL. In the case of graph-only operations, LB2 builds an adjacency structure to access the graph directly and avoid accessing the `Edge` relation while performing graph operations. In *spirit*, a graph structure resembles an index where the source vertex is the key, and the data record (i.e., the edge that contains the destination vertex and relational attributes) is the value.

In general, main-memory query compilers implement the data-centric evaluation approach [6] since it can be efficiently specialized to remove operators' interpretive overhead leading to generating efficient code. Figure 4.2b-c shows two PageRank query execution plans (QEPs). QEP (b1) is a graph-only PageRank operation that prints the result of PageRank (i.e., vertices and scores). On the other hand, QEP (c1) mixes graph and SQL operators as follows. First, the query filters the graph edges before performing PageRank. After that, the PageRank intermediate result is joined with data from another table, and the final result is printed. For both queries, the implementation of PageRank is similar, the difference is only in the operator interface. Figure 4.3a shows the `PageRankidx` operator. Line 5 obtains the previously created graph. The `exec` method invokes the `PageRank` operator implemented inside the graph structure (see Section 4.1.3). Since `PageRankidx` is a leaf operator in the query plan, there is no `parent.exec` call. Figure 4.3b shows `PageRankOp` used in QEP (c) where the `exec` operator creates a graph structure at evaluation time (due to the filter operation) and inserts the edges received from the interfacing filter operator. After that, `graph.PageRank` is invoked, and the result is passed to the join operator through the callback function `cb`.

4.1.3 Graph Data Structures

Adjacency lists are the most commonly used data structures to build and process graphs due to their optimized storage and intuitive interface. LB2 implements three graph data structures: adjacency list, flattened adjacency list, and trie, shown in Figure 4.4. For graph-only queries, the flattened adjacency list has the best runtime performance (see Section 4.2.1). However, the creation of such a graph structure is expensive since it constructs an adjacency list indirectly to obtain the neighbor sets as illustrated in Figure 4.4b-c (except when data is already stored in CRS form). The adjacency list performs well with queries that mix graph and SQL since the graph is built at runtime. Recently, tries have received attention as graph data structure [129]

as they facilitate performing multi-way joins (in contrast to two binary joins). For instance, triangle counting can be implemented directly as a three-way join. Therefore, LB2 adds a trie adjacency structure to optimize operations with multi-way join patterns. However, the performance of trie-based operations is sensitive to vertex ordering in the graph. A good vertex ordering assists the join operation to minimize the number of comparisons (the experiment in Section 4.2.1 gives insights about the performance of trie-based triangle count). The trie data structure and trie-based triangle count operator was implemented by Xilun Wu.

Internally, the specialized graph structures are implemented using LB2's generation-time abstractions (e.g., records, data buffers, etc.) that emit low-level code. In the following code, we show the extended graph adjacency list structure implemented in LB2 which is similar to any high-level graph implementation (for simplicity we omit the vertex to list index mapping and resizing steps).

```
1 abstract class Graph {
2  def insert(src: Rep[Long], dest: Rep[Long]): Unit
3  // graph operations, e.g., PageRank, triangle count
4 }
5 class LB2Graph(val size: Rep[Long]) extends Graph {
6  val cap0 = defaultSize
7  val adjList = LB2BufferFlat2D[Long](verSize, cap0)
8  val adjListCount = NewArray[Long](verSize)
9
10  def insert(vs: Rep[Long], vd: Rep[Long]) ={
11        adjList.getRow(vs).update(adjListCount(vs), vd)
12        adjListCount(vs) = adjList(vd) + 1
13 }}
```

The `LB2BufferFlat2D` class is an abstraction over (an array of arrays) `Array[Array[Long]]` that is generated as `long** adjList`. Moreover, the method `getRow` that obtains a row from the adjacency list and is generated as `long* row = adjList[vs]`.

Auxiliary Data Structures Graph operators use auxiliary data structures while processing. For instance, Dijkstra's algorithm for finding the shortest path between two vertices in a graph uses a minimum heap for distance values. All auxiliary data structures in LB2 are implemented as high-level abstractions that emit low-level code.

```
1 class PageRankidx(left: Op)
2  (gr: String) extends Op {
3
4 def exec(cb: Record => Unit) = {
5  val graph = left.getGraph(gr)
6   graph.PageRank {rec =>
7     cb(rec)
8    }
9   }
10 }
11
12
```

(a)

```
class PageRankOp(parent: Op)
  (srcKey: KeyFun)
  (destKey: KeyFun) extends Op {

 val size = defaultSize
 def exec(cb: Record => Unit) = {
   val graph = new GraphStruct(size)
   parent.exec { rec =>
     graph.insertEdge
         (srcKey(rec), desKey(rec))
     graph.PageRank{r =>
       cb(r)
}}}
```

(b)

Fig. 4.3. PageRank operator as (a) graph-only and (b) graph + SQL.

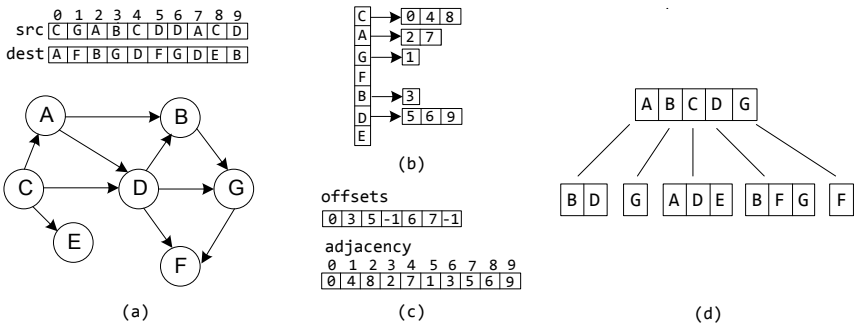


Fig. 4.4. Graph structures in LB2.

4.1.4 Recursive Queries

In standard query engines, recursive queries are composed using CTEs and evaluated using a variant of the incremental view maintenance algorithms which in essence improves on the basic semi-naive evaluation algorithm that performs a sequence of join, union, and set difference operations until a fix-point is reached.

LB2 processes recursive graph queries encoded in `Datalog` as follows. Given a query optimizer that extracts the recursion from the rules and maps the result into either a specialized operator (if a particular pattern is recognized like triangle count that

consists of two join conditions) or to an operator that unrolls recursion and perform the encoded join operation until no more new records are produced.

4.2 Evaluation

In this Section, we evaluate the performance of the graph extension implemented in LB2. We compare LB2's performance with several low-level graph processing engines. EmptyHeaded [20] is regarded as a state-of-art graph relational engine that implements specialized graph structures, exploits SIMD parallelism and generates optimized low-level code. We also picked two popular low-level shared-memory graph engines SNAP [12] and Ligra [71].

We conduct two experiments to evaluate the performance of the graph extension. In the first experiment, we evaluate the performance of graph pattern and graph analytical queries (triangle count and PageRank) in LB2, EmptyHeaded, SNAP and Ligra using standard graph benchmarks in Table 3.1. Furthermore, we illustrate the effect of graph data structure choice (trie, adjacency list, and flat array) on the performance graph queries. The second experiment compares the parallel scalability of the previous graph engines on triangle count and PageRank queries. We show the performance of LB2's graph extension is competitive with state-of-art low-level graph engines.

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 EmptyHeaded v.0.1 and SNAP 4.1.

Datasets and Configurations Table 3.1 shows the graph datasets we use in the experiments section that span three application domains: social, web and product networks. The two largest datasets are LiveJournal1 and Orkut (approximately 69M and 117M edges respectively) which represent the size of practical graph workloads.

Table 4.1.
Graph datasets that are used in evaluating LB2-Graph.

Dataset	Nodes	Edges	Description
Amazon	334,863	925,872	Product network
YouTube	1,134,890	2,987,624	Social network
Skitter	1,696,415	11,095,298	Internet topology
Orkut	3,072,441	117,185,083	Social network
LiveJournal1	4,847,571	68,993,773	Social network

Table 4.2.
Runtime of single-core triangle count (in seconds) for LB2 (using flat adjacency list and trie), SNAP and EmptyHeaded.

Dataset	LB2-Flat	LB2-Trie	SNAP	EmptyHeaded
Amazon	0.1	0.9	0.19	0.5
YouTube	0.4	0.5	0.7	0.5
Skitter	1.4	1.7	2.2	1.7
Orkut	80.9	61	87.2	60.1
LiveJournal1	13.6	26.4	18.1	24.9

For each query, we measure the runtime excluding compilation time, data loading and adjacency list (or index) creation. We run each query 5 times and pick the median measurement. To guarantee local execution, we run the queries in a single NUMA node using numactl `-m` and `-C` options.

4.2.1 Single-core Graph Pattern and Analytics Queries

Triangle counting Triangle counting is an important sub-graph pattern matching query largely used in graph structure mining and graph benchmarking in general. For

this query, we follow SNAP's implementation that treats the graph as undirected when counting triangles. Table 4.2 gives the runtime of evaluating triangle count on LB2, SNAP[1], and EmptyHeaded on five graph datasets. For small to medium size datasets (i.e., Amazon, YouTube, Skitter and LiveJournal1 where the number of edges is approximately 69M), LB2 outperforms SNAP by 7%-2.7× and EmptyHeaded by 80%-7×. In the Orkut dataset, EmptyHeaded is 35% faster than LB2-flat and comparable to LB2-Trie. Also, SNAP is slower than both of LB2 and EmptyHeaded due to the high-level implementation of its graph structure that is more expensive to access than the specialized structures in LB2 and EmptyHeaded. LB2's performance advantage over EmptyHeaded is attributed to its specialized data structures in addition to the fact that EmptyHeaded's SIMD parallel set operations work better with high density skewed data.

PageRank Another graph analytical query is PageRank, developed to rank the importance of a webpage based on the link structure of the web [130]. The applicability of PageRank computations to any graph (e.g., road, social, biology, etc.) make it an important graph workload. Similar to SNAP and Ligra, we implement the algorithm in Berkhin's survey [131]. Table 4.3 gives the runtime for 25 iterations of PageRank on LB2, SNAP, Ligra and EmptyHeaded. For fairness across all systems, we commented out the convergence computations in LB2, SNAP, and Ligra since the query is designed to run for a constant number of iterations without convergence testing that may incur additional runtime cost or cause an early exit. EmptyHeaded is the slowest among the benchmarked systems due to running naive recursion in this query which is equivalent to a simple unrolling of the join algorithm [20]. The authors pointed out that PageRank performance can be further improved with double buffering and redundant join elimination. For large datasets LiveJournal1 and Orkut, the performance of LB2 and SNAP is comparable (LB2 is 5% faster in LiveJournal1 whereas

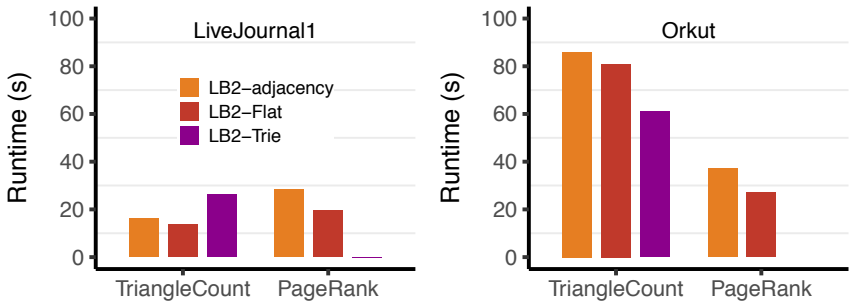
[1]SNAP provides two implementations for triangle count. We picked `GetTriangleCnt` since it can be parallelized and it also outperformed `GetTriads`.

Table 4.3.
The absolute runtime of single-core PageRank (in seconds) for LB2 (using flat adjacency structure), SNAP, Ligra and EmptyHeaded.

Dataset	LB2-Flat	SNAP	Ligra	EmptyHeaded
Amazon	0.2	0.4	0.3	2.6
YouTube	0.7	1.1	1.0	8.2
Skitter	2.5	2.8	3.2	16.6
Orkut	27.2	24.4	36.3	142.8
LiveJournal1	19.5	20.4	27.1	70.1

SNAP is 10% faster in Orkut). Furthermore, LB2 is 30%-50% faster than Ligra across all datasets.

The Effect of Adjacency List Specialization LB2 implements three graph adjacency structures: adjacency list, flattened adjacency list (using arrays that maintain the offset of vertices and neighboring edges) and trie. In this experiment, we evaluate the performance of triangle counting and PageRank using the previous three adjacency structures as illustrated in Figure 4.5. For the triangle counting query in LiveJournal1 the flat implementation is faster than the adjacency list and trie by 20% and 95% respectively whereas the trie is faster than the adjacency and flat list in Orkut by 40% and 30% respectively. For PageRank query, the adjacency list is slower than flat array version by 28% and 14% respectively. For the adjacency list, the difference in performance is a result of generating adjacency lists as an array of arrays (`long** adj`). Hence, the first array access to obtain a row from the adjacency list (`long* nbr_lst = adj[index]`) is slightly more expensive than two simple array accesses (i.e., first access to obtain the vertex offset in the edges array and second to start iterating over the vertex neighbors). Similarly, a trie similar to the one in Figure 4.5d incurs two accesses to obtain an edge. Moreover, the performance of trie-based



	LiveJournal1		Orkut	
	TriangleCount	**PageRank**	**TriangleCount**	**PageRank**
LB2-Adj	16.3	25	86	31
LB2-Flat	13.6	20	80.9	27
LB2-Trie	26.4	-	61	-

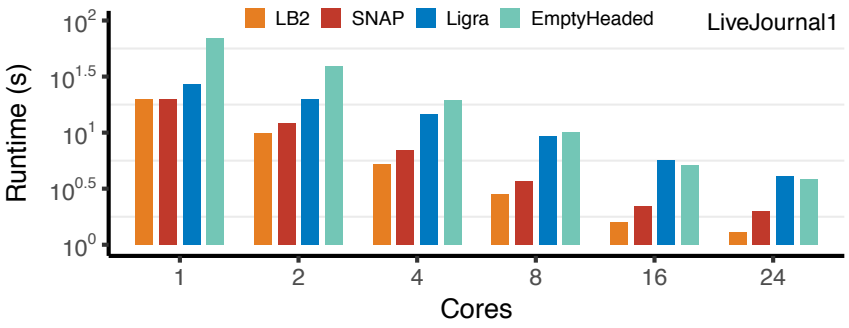
Fig. 4.5. Runtime of single-core triangle counting and PageRank using LB2 with adjacency list and flat array in LiveJournal1 and Orkut datasets.

operations is sensitive to vertex ordering in the graph which justifies the variance in performance on different datasets. Finally, the key insight gained from comparing various implementations of data structures is that optimized memory access matters in processing large graphs.

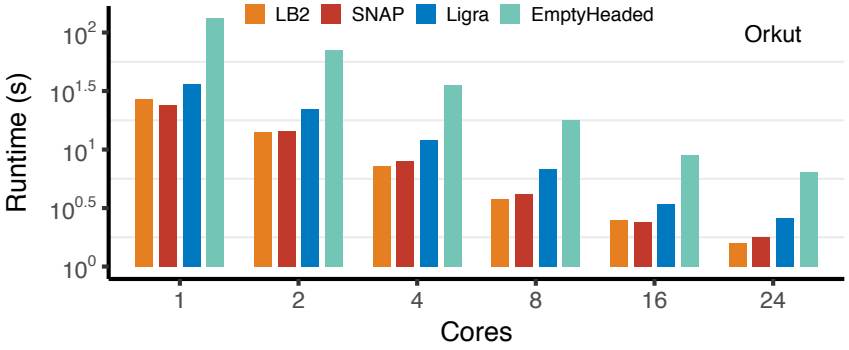
4.2.2 Parallel Graph pattern and Analytics Queries

In this experiment, we compare the multicore scalability of LB2 with SNAP, Ligra and EmptyHeaded. Figure 4.6 gives the absolute runtime for scaling up LB2, SNAP, Ligra and EmptyHeaded in PageRank on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets. We scale the number of cores up to 24 to keep the execution local within a single socket.

Overall, the speed up in all systems linearly increases with the number of cores. The systems with the fastest single core runtime (SNAP and LB2) achieves a speedup



	1	2	4	8	16	24
LB2	20	9.9	5.2	2.8	1.6	1.3
SNAP	20	12.2	7	3.7	2.2	2
Ligra	27	20	14.5	9.3	5.7	4.1
EmptyHeaded	70	39	19.5	10	5.1	3.8



	1	2	4	8	16	24
LB2-Flat	27	14	7.2	3.8	2.5	1.6
SNAP	24	14.3	7.9	4.2	2.4	1.8
Ligra	36	22	12	6.8	3.4	2.6
EmptyHeaded	132	71	35.5	17.9	8.9	6.4

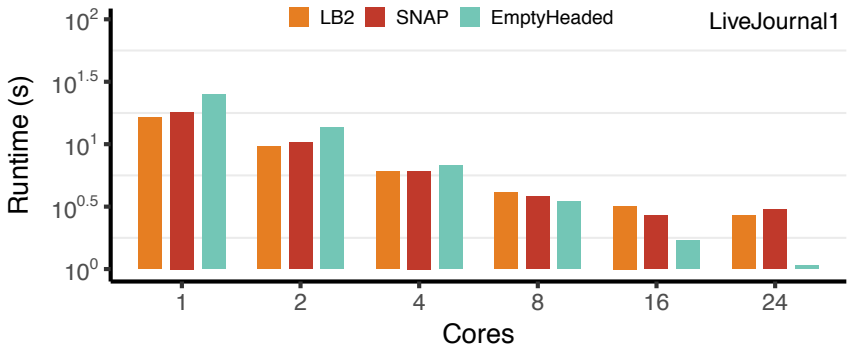
Fig. 4.6. The absolute runtime in seconds (s) for parallel scaling up LB2, SNAP, Ligra and EmptyHeaded in PageRank on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets.

of 10× and 15× respectively in LiveJournal1, and speedup of 13× and 16× respectively in Orkut. Furthermore, LB2 and SNAP outperform Ligra and EmptyHeaded in this query. At a closer look, the scaling up of LB2 is 4× (at two cores) and 2× (at 24 cores) faster than EmptyHeaded in LiveJournal1 and 5×-6× times faster in Orkut. Similarly, LB2 is average 5×-6× times faster than Ligra in LiveJournal1.

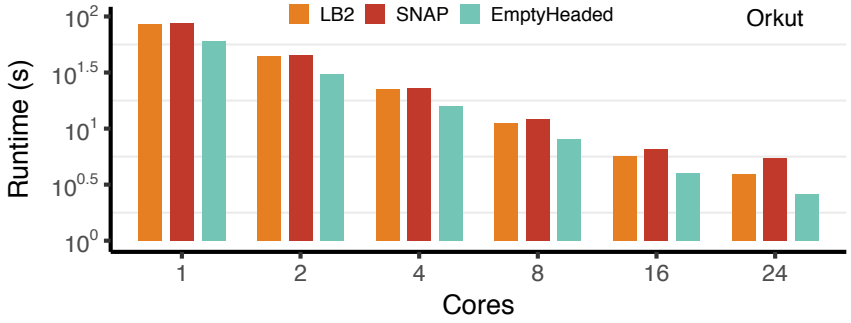
We observed that the implementation of SNAP PageRank spends time in creating a vertex lookup index that maps the vertex id into a vector index which explains, in part, the small difference in runtime numbers. In the case of EmptyHeaded, (as we discussed in the single-core experiment) that EmptyHeaded implements the naive evaluation (for this operator) which is not aggressive enough in duplicate elimination and causes the operator to perform extraneous work per iteration. EmptyHeaded would probably match LB2 if the implementation were improved. Although Ligra is implemented in C++, our observation is the high-level data structure adds runtime interpretive overhead in contrast to LB2's specialized data structure.

Figure 4.7 gives the absolute runtime for scaling up LB2, SNAP, and EmptyHeaded in triangle count on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets. Overall, EmptyHeaded's scaling up in LiveJournal1 outperforms LB2 and SNAP by an average of 8%-2.8× from 8 cores and above. Similarly, EmptyHeaded outperforms LB2 and SNAP in Orkut by an average of 40%-2×. At a closer look, the performance of LB2 and SNAP is comparable. For instance, the scaling of LB2 is 10% faster than SNAP in LiveJournal1 at core 24 whereas SNAP is faster than LB2 by 25% for the same core value. Moreover, EmptyHeaded results are attributed, in part, to the data structure, query optimizer that picks the proper layout based on data skew and SIMD parallelism.

Discussions The outcomes of the parallel scaling experiment are consistent with the single-core experiment. The performance of LB2 is attributed, in part, to optimized evaluation and data structures specialization. Altogether, our experiments show that



	1	2	4	8	16	24
LB2-Adj	16.3	9.6	6	4.1	3.2	2.7
SNAP	18.1	10.3	6.1	3.8	2.7	3
EmptyHeaded	24.9	13.7	6.7	3.5	1.7	1.07



	1	2	4	8	16	24
LB2-Flat	27	14	7.2	3.8	2.5	1.6
SNAP	24	14.3	7.9	4.2	2.4	1.8
Ligra	36	22	12	6.8	3.4	2.6
EmptyHeaded	132	71	35.5	17.9	8.9	6.4

Fig. 4.7. The absolute runtime in seconds (s) for parallel scaling up LB2, SNAP, and EmptyHeaded in triangle count on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets.

LB2 can compete against state-of-the-art graph engines. However, LB2's design is simpler as it is derived from a straightforward query interpreter design.

4.3 Conclusions

In this Chapter, we have added graph query compilation inside the LB2 main-memory query compiler. We support parallelism for shared memory using OpenMP and thread-aware data structures. The graph extension matches, and sometimes outperforms, specialized low-level graph engines.

5. SUMMARY

In this dissertation, we have demonstrated the practical application of a deep and fundamental idea known as the first Futamura projection, which essentially states that the ability to specialize a query interpreter to a given query is identical to a query compiler.

In Chapter 2, we have presented LB2; a fully compiled query engine, and we have shown that LB2 performs on par with, and sometimes beats the best compiled query engines on the standard TPC-H benchmark. Specifically, LB2 is the first query engine built in a high-level language that is competitive with HyPer [6], both in sequential and parallel execution. LB2 is also the first single-pass query engine that is competitive with DBLAB [9] using the full set of non-TPC-H-compliant optimizations.

In Chapters 3-4, we have demonstrated that the underlying idea of constructing simple but highly efficient query compilers applies not only to purely relational queries, but carries over to diverse workloads including spatial and graph queries. Chapter 2 was published and presented in the ACM 2018 International Conference on Management of Data (SIGMDOD'18). In Chapter 3, we have added 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 kNN join queries, LB2-Spatial outperforms spatial spark extensions and PostGIS in spatial join queries by 12×-299×. For scaled-up execution, LB2-Spatial is 10×-20× faster than spatial Spark extensions. In Chapter 4, we have added graph query compilation inside the LB2 main-memory query compiler. We support parallelism for shared memory using OpenMP and thread-aware data structures. Chapter 3 is currently under submission, and Chapter 4 is accepted at the 17th Symposium on Data Base Programming Languages (DBPL'19).

REFERENCES

REFERENCES

[1] G. Graefe, "Volcano-an extensible and parallel query evaluation system," *TKDE*, vol. 6, no. 1, pp. 120–135, 1994.

[2] 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," *TODS*, vol. 1, no. 2, pp. 97–137, 1976.

[3] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. a. Yoder, "Impala: A modern, open-source SQL engine for hadoop," in *CIDR*, 2015.

[4] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized oltp engine," in *SIGMOD*. ACM, 2013, pp. 1243–1254.

[5] 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 *SIGMOD*. ACM, 2015, pp. 1383–1394.

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

[7] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, "Building efficient query engines in a high-level language," *PVLDB*, vol. 7, no. 10, pp. 853–864, 2014.

[8] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.

[9] A. Shaikhha, I. Klonatos, L. E. V. Parreaux, L. Brown, M. Dashti Rahmat Abadi, and C. Koch, "How to architect a query compiler," in *SIGMOD*, 2016.

[10] "SP-GiST indexes," https://www.postgresql.org/docs/10/static/spgist.html.

[11] "Neo4j," https://neo4j.com/.

[12] "SNAP: Stanford Network Analysis Project," https://snap.stanford.edu/index.html.

[13] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A history and evaluation of system r," *Commun. ACM*, vol. 24, no. 10, pp. 632–646, Oct. 1981.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, Mar 2004, pp. 75–88.

[15] "Postgis," http://postgis.org.

[16] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *SIGSPATIAL*. ACM, 2015, p. 70.

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

[18] "JTS topology suite," http://www.vividsolutions.com/jts/JTSHome.htm.

[19] "GEOS-Geometry engine, open source," https://trac.osgeo.org/geos.

[20] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *TODS*, vol. 42, no. 4, p. 20, 2017.

[21] R. Greer, "Daytona and the fourth-generation language cymbal," in *ACM SIGMOD Record*, vol. 28, no. 2. ACM, 1999, pp. 525–526.

[22] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman, "Compiled query execution engine using jvm," in *ICDE*. IEEE, 2006, pp. 23–23.

[23] K. Krikellas, S. D. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE*. IEEE, 2010, pp. 613–624.

[24] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, 2012.

[25] 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 TECS*, vol. 13, no. 4s, p. 134, 2014.

[26] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun, "Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns," in *CGO*. ACM, 2016, pp. 194–205.

[27] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *ICML*, 2011.

[28] T. Rompf and N. Amin, "Functional pearl: a SQL to C compiler in 500 lines of code," in *ICFP*. ACM, 2015, pp. 2–9.

[29] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data," in *OSDI*, 2018, pp. 799–815.

[30] R. Y. Tahboub and T. Rompf, "On supporting compilation in spatial query engines:(vision paper)," in *SIGSPATIAL*, 2016.

[31] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *SIGMOD*, 2018, pp. 307–322.

[32] Y. Ahmad and C. Koch, "DBToaster: A SQL compiler for high-performance delta processing in main-memory databases," *VLDB*, vol. 2, no. 2, pp. 1566–1569, 2009.

[33] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo - a vector algebra for portable database performance on modern hardware," *VLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.

[34] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab, "Weld: A common runtime for high performance data analytics," in *CIDR*, 2017.

[35] D. Butterstein and T. Grust, "Precision performance surgery for PostgreSQL: Llvm—based expression compilation, just in time," *VLDB*, vol. 9, no. 13, pp. 1517–1520, 2016.

[36] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher, "Runtime specialization of PostgreSQL query executor," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2017, pp. 375–386.

[37] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki, "Adaptive query processing on RAW data," *PVLDB*, vol. 7, no. 12, pp. 1119–1130, 2014.

[38] I. Alagiannis, S. Idreos, and A. Ailamaki, "H2O: a hands-free adaptive store," in *SIGMOD*, 2014, pp. 1103–1114.

[39] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki, "Just-in-time data virtualization: Lightweight data management with vida," in *CIDR*, 2015.

[40] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *SIGOPS/EuroSys*, ser. EuroSys, 2007, pp. 59–72.

[41] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: the system s declarative stream processing engine," in *SIGMOD*. ACM, 2008, pp. 1123–1134.

[42] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun, "Go meta! A case for generative programming and dsls in performance critical systems," in *SNAPL*, ser. LIPIcs, vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 238–261.

[43] "Oracle spatial and graph," https://www.oracle.com/assets/spatial-and-graph-ds-1738135.pdf.

[44] Y. Fang, M. Friedman, G. Nair, M. Rys, and A.-E. Schmid, "Spatial indexing in microsoft SQL server 2008," in *SIGMOD*, 01 2008, pp. 1207–1216.

[45] J. R. Davis, "IBM's DB2 spatial extender: Managing geo-spatial information within the dbms," 1998.

[46] P. Ogden, D. Thomas, and P. Pietzuch, "At-gis: Highly parallel spatial query processing with associative transducers," in *SIGMOD*. ACM, 2016, pp. 1041–1054.

[47] M. Vermeij, W. Quak, M. Kersten, and N. Nes, "Monetdb, a novel spatial column-store dbms," 2008.

[48] "Geocouch," https://accumulo.apache.org.

[49] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods," *IEEE Data Eng. Bull.*, 2003.

[50] L.-V. Nguyen-Dinh, W. G. Aref, and M. Mokbel, "Spatio-temporal access methods: Part 2 (2003-2010)," *IEEE Data Eng. Bull.*, 2010.

[51] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "Sjmr: Parallelizing spatial join with mapreduce on clusters," in *CLUSTER*. IEEE, 2009, pp. 1–8.

[52] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *ICDE*, 2015, pp. 1352–1363.

[53] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *VLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[54] J. Lu and R. H. Guting, "Parallel secondo: boosting database engines with hadoop," in *ICPADS*. IEEE, 2012, pp. 738–743.

[55] R. H. Güting, T. Behr, V. Almeida, Z. Ding, F. Hoffmann, M. Spiekermann, and L. D. für neue Anwendungen, *SECONDO: An extensible DBMS architecture and prototype*.

[56] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon, "Spatio-temporal indexing in non-relational distributed databases," in *Big Data*. IEEE, 2013, pp. 291–299.

[57] "Accumulo," https://github.com/couchbase/geocouch.

[58] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *MDM*, vol. 1. IEEE, 2011, pp. 7–16.

[59] "Hbase," https://hbase.apache.org/.

[60] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *TOCS*, vol. 26, no. 2, p. 4, 2008.

[61] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *SIGSPATIAL*. ACM, 2017, p. 28.

[62] S. Hagedorn, P. Götze, and K.-U. Sattler, "The stark framework for spatio-temporal data analytics on spark," *BTW*, 2017.

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

[64] "Magellan: Geospatial analytics using spark," https://github.com/harsha2010/magellan.

[65] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *ICDEW*. IEEE, 2015, pp. 34–41.

[66] Z. Du, X. Zhao, X. Ye, J. Zhou, F. Zhang, and R. Liu, "An effective high-performance multiway spatial join algorithm with spark," *ISPRS International Journal of Geo-Information*, vol. 6, no. 4, p. 96, 2017.

[67] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie, "SQLGraph: an efficient relational-based property graph store," in *SIGMOD*. ACM, 2015, pp. 1887–1901.

[68] J. Fan, A. Gerald, S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," 2015.

[69] M. Paradies, W. Lehner, and C. Bornhövd, "Graphite: an extensible graph traversal framework for relational database management systems," in *SSDBM*. ACM, 2015, p. 29.

[70] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner, "The graph story of the sap hana database." in *BTW*, 2013.

[71] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[72] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP*. ACM, 2013, pp. 456–471.

[73] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *OSDI*. USENIX, 2012.

[74] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *SOSP*. ACM, 2013, pp. 472–488.

[75] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning." in *USENIX ATC*, 2015.

[76] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *PVLDB*, vol. 11, no. 3, pp. 297–310, 2017.

[77] "Titan," http://titan.thinkaurelius.com/.

[78] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, "Ringo: Interactive graph analytics on big-memory machines," in *SIGMOD*, 2015, pp. 1105–1110.

[79] K. Xirogiannopoulos, U. Khurana, and A. Deshpande, "Graphgen: Exploring interesting graphs in relational data," *VLDB*, vol. 8, no. 12, pp. 2032–2035, 2015.

[80] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao, "Large-scale graph analytics in aster 6: bringing context to big data discovery," *VLDB*, vol. 7, no. 13, pp. 1405–1416, 2014.

[81] T. L. Willke and N. Jain, "Graphbuilder – a scalable graph construction library for apache tm hadoop tm," in *NIPS*, 2012.

[82] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*. ACM, 2010, pp. 135–146.

[83] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *VLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[84] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *WWW*, 2008, pp. 695–704.

[85] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *SOSP*. ACM, 2013, pp. 439–455.

[86] "Graph-X," https://spark.apache.org/graphx/.

[87] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[88] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "Pgql: a property graph query language," in *GRADES*, 2016, p. 7.

[89] "Cypher graph query language." https://neo4j.com/developer/cypher-query-language/.

[90] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda *et al.*, "G-core: A core for future graph query languages," in *SIGMOD*, 2018, pp. 1421–1432.

[91] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *SIGARCH*, vol. 40, no. 1, 2012, pp. 349–362.

[92] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *ICDE*. IEEE, 2013, pp. 278–289.

[93] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *SIGMOD*. ACM, 2015, pp. 1371–1382.

[94] "Soufflé: A Datalog synthesis tool for static analysis," https://souffle-lang.github.io/.

[95] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *SIGMOD*. ACM, 2016, pp. 1135–1149.

[96] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch, "Datalography: Scaling datalog graph analytics on graph processing systems," in *BigData*. IEEE, 2016, pp. 56–65.

[97] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *PVLDB*, vol. 8, no. 12, pp. 1542–1553, 2015.

[98] S. Sakr, S. Elnikety, and Y. He, "Hybrid query execution engine for large attributed graphs," *Information Systems*, vol. 41, pp. 45–73, 2014.

[99] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *SOSP*. ACM, 2015, pp. 425–440.

[100] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: scalable parallel frequent subgraph mining in a single large graph," in *High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 61.

[101] W. Gan, J. C.-W. Lin, H.-C. Chao, and J. Zhan, "Data mining in distributed environment: a survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 7, no. 6, p. e1216, 2017.

[102] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *OSDI*.

[103] D. Yan, H. Chen, J. Cheng, M. T. Özsu, Q. Zhang, and J. Lui, "G-thinker: big graph mining made easier and faster," *arXiv*, 2017.

[104] C. W. Bachman, "The programmer as navigator," *Communications of the ACM*, vol. 16, no. 11, pp. 653–658, 1973.

[105] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[106] Y. Futamura, "Partial evaluation of computation process — an approach to a compiler-compiler," *Transactions of the Institute of Electronics and Communication Engineers of Japan*, vol. 54-C, no. 8, p. 721–728, 1971.

[107] N. D. Jones, "An introduction to partial evaluation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 3, pp. 480–503, 1996.

[108] N. Jones, C. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[109] M. Mehta and D. J. DeWitt, "Managing intra-operator parallelism in parallel database systems," in *SIGMOD Conference*, 1995, pp. 743–754.

[110] "OpenMP," http://openmp.org/.

[111] "PostgreSQL," https://www.postgresql.org.

[112] A. Shaikhha, I. Klonatos, L. E. V. Parreaux, L. Brown, M. Dashti Rahmat Abadi, and C. Koch, "DBLAB SIGMOD 2016 reproducibility," https://dl.acm.org/citation.cfm?id=2915244, 2016.

[113] "DBLAB reproducibility git repository," https://github.com/rtahboub/dblab, 2016, commit: e5d3fe2d5e8705fc827c40b07b752291f967a5a6, last accessed: 02-15-2018.

[114] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," Tech. Rep. RT0968, October 2015.

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

[116] R. Y. Tahboub and T. Rompf, "How to architect a query compiler for spatial workloads," Purdue University, Tech. Rep., 2019.

[117] B. Ooi, R. Sacks-Davis, and J. Han, "Indexing in spatial databases," 1993.

[118] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "Aqwa: adaptive query workload aware partitioning of big spatial data," *PVLDB*, vol. 8, no. 13, pp. 2062–2073, 2015.

[119] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *ACM sigmod record*, vol. 24, no. 2. ACM, 1995, pp. 71–79.

[120] D. Šidlauskas and C. S. Jensen, "Spatial joins in main memory: Implementation matters!" *PVLDB*, vol. 8, no. 1, pp. 97–100, 2014.

[121] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *PVLDB*, vol. 11, no. 11, pp. 1661–1673, 2018.

[122] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke, "An experimental analysis of iterated spatial joins in main memory," *VLDB*, vol. 6, no. 14, pp. 1882–1893, 2013.

[123] "Spatial indexing at cornell," http://www.cs.cornell.edu/bigreddata/spatial-indexing.

[124] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *HotOS*. USENIX Association, 2015.

[125] Spark, "Tuning Spark," https://spark.apache.org/docs/latest/tuning.html.

[126] R. Y. Tahboub, X. Wu, G. M. Essertel, and T. Rompf, "Towards compiling graph queries in relational engines," in *DBPL*. ACM, 2019.

[127] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *ICDE*, vol. 1, no. 1, pp. 146–166, 1989.

[128] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, "Extending in-memory relational database engines with native graph support," in *EDBT*, 2018, pp. 25–36.

[129] T. L. Veldhuizen, "Leapfrog triejoin: A simple, worst-case optimal join algorithm," 2014.

[130] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[131] P. Berkhin, "A survey on pagerank computing," *Internet Mathematics*, vol. 2, no. 1, pp. 73–120, 2005.

VITA

VITA

Ruby received her Ph.D. from the Department of Computer Science at Purdue University. Her research interests lie in the intersection area of database systems and programming languages. Ruby earned her master's degree in computer science from Purdue University. She also earned another master's degree from the University of Jordan and her bachelor of science from Jordan University of Science and Technology. Ruby completed a summer internship with Software and Services Group (SSG) in INTEL. Also, Ruby was awarded a best demo award in the ACM SIGSPATIAL 2015 and in 2016, Ruby received the Boyce Graduate teacher award from the college of science at Purdue University. She is a member of the IEEE and a member of the ACM.