Building a Fast In-memory Database Using Efficient Query Compilation and a Tuned Data-centric IR

by

Nithin Ivan

Master of Science in Software Systems, BITS-PILANI, Rajasthan, India, 2014

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Suprio Ray, Ph.D., Computer Science
Examining Board:	Ken Kent, Ph.D., Computer Science, Chris Rouse, Ph.D., Electrical and Computer Engineering Virendra Bhavsar, Ph.D., Computer Science

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

July, 2022

Abstract

Traditional database systems convert an SQL (Structured Query Language) query into a plan-tree consisting of relational algebra operators and interpret it to return the result. This design was adequate in the past, as the major bottleneck was reading and writing data to disk, which offset the overheads of interpretation. Modern databases with access to faster storage systems and huge amounts of main memory expose these overheads of the interpretation model. The tree of operators can be compiled into a lower level and executed directly. In this thesis, a new compilation-based, in-memory database system is introduced, which uses a push-down approach to generate query-specific code. This generated code is compiled and executed to return the result. The system described in this thesis has been implemented with a template-based code generation framework written in Java and supports multiple target back-end systems like C++, UPC++, and also a tuned data-centric IR. The new system was compared to a typical postgreSQL installation, and the performance of the generated IR code was found to be significantly better.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Suprio Ray. The completion of this research would not have been possible without Dr. Ray. His expert academic and research guidance has always motivated me to expand my boundaries and adopt a systematic approach to work. I would also like to thank Dr. Kenneth Kent for providing this amazing opportunity to work on an industry-led project at IBM CAS–Atlantic. A debt of gratitude is also owed to Dr. Mark Stoodley, from IBM Runtime Laboratories, for providing his expert insights and feedback at different stages of the research project. This research was conducted within the Centre for Advanced Studies – Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for facilities of CAS–Atlantic in completing this research.

I would also like to extend my sincere thanks to Senior Research Assistant Stephen MacKay, for his precious feedback, and also Research Assistant Deverne Jones, for keeping the systems up and always "Counting on me." I sincerely thank my team members at the CAS–Atlantic lab, for their help during my research—Hassan Arafat, for his expert guidance on C/C++ and Saumya Verma for all the discussions and help with the IR. Last but not the least, a big thank you to my family, for believing in me and providing all the support that I needed.

Table of Contents

List of Tables

List of Figures

Abbreviations

CPU Central Processing Unit

DAG Directed Acyclic Graph

DBMS Database Management System

DFS Depth First Search

JDBC Java Database Connectivity

JIT Just-in-time

MLIR Multi-level Intermediate Representation

RACO Relational Algebra Compiler

RDBMS Relational Database Management System

SF Scale Factor

SPIR Standard Portable Intermediate Representation

SQL Structured Query Language

SSA static single-assignment

TPC-H Transaction Processing Council Ad-hoc/decision support benchmark

Chapter 1

Introduction

A database management system (DBMS) is generally divided into two major subsystems—the Compile-time System (CTS) and the Run-time System (RTS). The input to the system enters through the CTS, and is translated into executables that the RTS interprets to return the results. The input commands to a DBMS fall into several different categories related to Schema creation and modification, DBMS statistics, transactions and queries. There are two major approaches to querying a DBMS: Interpretation and Compilation. This thesis presents systems and experiments that deal entirely with the compilation approach to query processing.

1.1 Motivation

The interpretation model converts the query string into a calculus-based internal representation. There could be a rewrite of this representation at various levels. Typical rewrite steps include un-nesting nested queries, pushing down selections, and introducing indices. The query is then interpreted. The first function, "interpret", takes a simple Structured Query Language block and extracts the various clauses before calling "eval". The query is then evaluated recursively by first producing the cross product of the entries in the 'from' clause. After this processing, the predicate

(if applicable) is applied, and a 'result' tuple is constructed and added to the result set 'R'. If the query has a filter, the tuples are added only when the 'where' predicate evaluates to true. This is not the most efficient method for processing a query.

The compilation model also begins with a query rewrite, using techniques like loop un-nesting for performance gains. After the rewrite, the plan-generation phase begins. An optimal plan is constructed after considering a multitude of query plans. While a rewrite is performed on a calculus-based representation of the query, the plan generation constructs an algebraic representation of the query, using well-known operators like selection and join. After the plan generation, the code-generation phase generates and passes the code to the RTS to be executed.

Compiling queries to machine code is an efficient method of query execution. One often overlooked problem with compilation is the time it takes to generate machine code. There have been attempts to improve the query compilation using different techniques.

1.2 Thesis Statement

The speed of query evaluation can be significantly improved using data-centric code-generation and compiling the generated code into an intermediate representation, tuned for relational algebraic expressions.

1.3 Research Questions

To support the thesis statement, the following research questions (RQs) have been pursued:

- **RQ1:** What are the various code-generating query-processing systems and which one offers best performance when handling big data?

- **RQ2:** Can we design a query processing system that generates query-specific code that performs better than the interpreted model and existing code-generating systems?

- **RQ3:** What will be the impact of using a new intermediate representation, on the overall query processing time of the system built as part of RQ2?

1.4 Thesis Contributions

In this thesis, the research questions have been explored by:

- Carrying out a literature survey of the existing query processing systems that work on code-generation, assessing their performance for big data loading and identifying possible areas of improvement (Section 2.5);

- Designing and building a full scale query execution engine that can generate code for multiple back-ends, like C++ and UPC++, from any given SQL query (Section 3.1);

- Designing and implementing a code-generation engine, based on the push-down model, emphasizing data-locality with a template-based approach (Section 3.2.1);

- Integrating Apache Calcite with the system to perform SQL query parsing and optimization, before invoking the code-generation engine (Section 3.1.1);

- Integrating Apache Arrow with the system to create a fully functional in-memory database system capable of executing SQL queries on large data-sets, loaded in main memory (Section 2.1.5);

- Exploring the possibility of a new MLIR-based intermediate representation,

that can further reduce the compile-time overhead and improve the performance of the system (Section 3.2.3);

- Experimental evaluation and analysis of CasaDB using TPC-H benchmark (Section 4); and
- Implementing proof-of-concept code, using MLIR which, opens the door for an improved design of CasaDB with an Intermediate Representation (Section 5.1).

Chapter 2

Background and Related Work

This chapter explains the concepts behind the code-generation and compilation framework, and the major components used through out this research work. The benchmarks and frameworks used for implementation of the proposed system are also introduced. Existing systems that are similar to our query execution engine are also described in detail, along with the major differences in their design and implementation.

2.1 Background

This section discusses the concepts of query execution, models of query execution and Apache Calcite[4].

2.1.1 Query Processing

Query processing refers to the process of translating a user query into an equivalent and efficient executable plan for a database system and executing it to return the results [31]. A typical SQL query processing workflow can be summarised in the following steps:

- *Parsing and Translation:* The query, in the form of an SQL statement, is parsed by an SQL parser to build a relational algebra form (syntax tree). The parser checks the syntax, identifies keywords and verifies relations, while the query is translated into an internal form and then to the relational algebraic expression.

- *Optimization:* Any relational algebraic expression can be evaluated in several different ways. An annotated expression with details of the evaluation strategy, known as an evaluation plan, is generated by the system. The optimizer chooses the most optimal plan among all the equivalent plans, based on the cost-estimate of plans derived from statistical information in the DBMS catalog and other system factors like network latency, CPU consumption data, and disk I/O. The statistical information includes the number of tuples, size of a tuple, and the number of blocks.

- *Execution:* Finally, the query execution engine takes the physical query plan and executes it to return the result.

Figure 2.1 is a diagrammatic representation of the query processing flow.

2.1.2 Compilers and Intermediate Representations

Compilers are essential components of many development tool-chains, that increase developer productivity. A compiler is typically thought of as a monolithic black box that takes in a high-level source program and outputs a semantically identical low-level one. Internally, it is organised into layers; what flows between internal layers is referred to as intermediate representations (IRs). There are numerous IRs and the

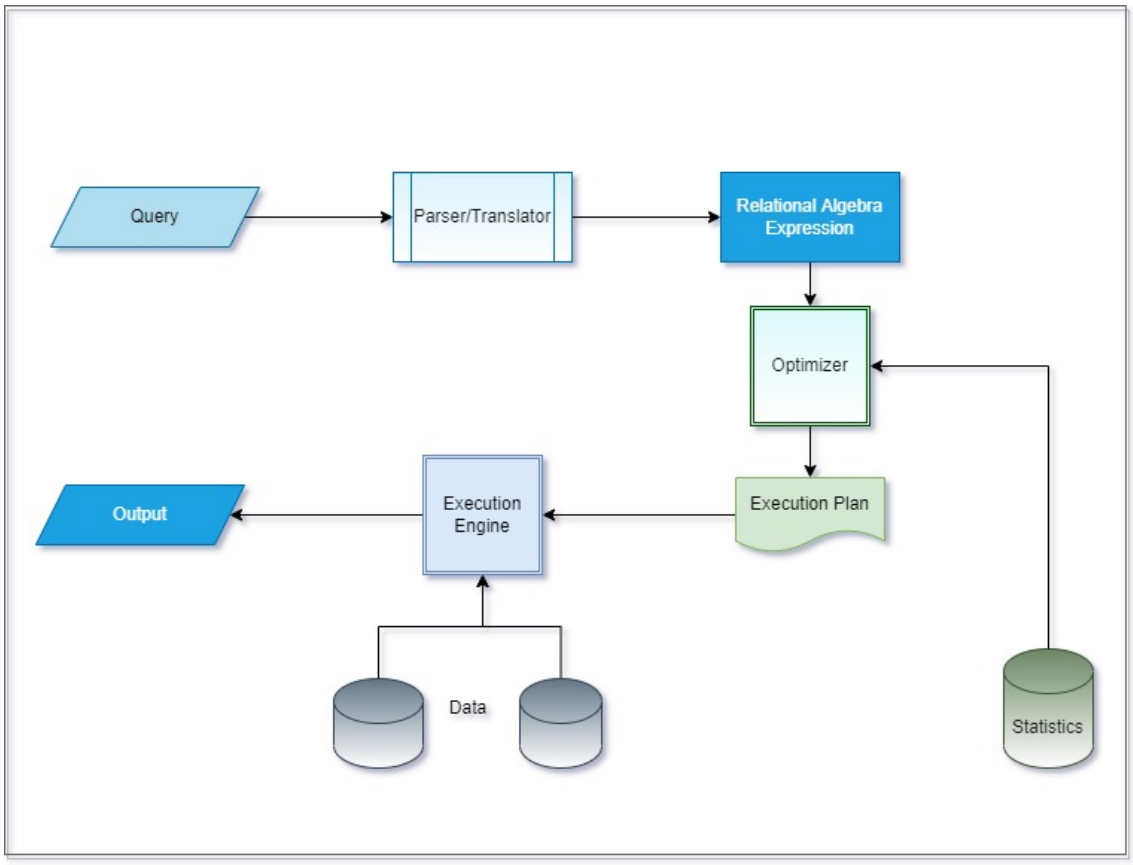

Figure 2.1: Query Processing Steps

most common ones are MLIR [28] and LLVM [27]. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies and is not an acronym. The programming constructs vary to a great degree between different programming languages, and the different hardware available have their own separate instruction sets. Compilers, which stand in between the source code and the target hardware connection, have limited performance options, since they must juggle several parameters, to ensure that optimizations applied are advantageous in the majority of circumstances. This is a particularly high standard to attain, especially for compilers that handle general-purpose programming languages like C++. As a result, only a few transformations can be done globally e.g.,dead code removal, constant folding, canonicalization; with the rest being pushed down to a second target-specific optimization step.

IRs, as the name implies, are just representations of programs. They are designed to make transformations easier. An IR can be in one of three formats: in-memory for quick analysis and transformation, byte-code for persistence and interchange, or textual for human inspection and debugging. Depending on the use case, there are design choices to be made in terms of what form is prioritised and how much compatibility is provided. For example, LLVM IR is designed for compiler internal usage and opts for efficient in-memory processing and weak compatibility [27], whereas SPIR-V (Standard Portable Intermediate Representation) is designed for hardware driver consumption and opts for rapid byte-code form processing and good compatibility [19].

The first version of LLVM was published in 2003 [27]. It is highly mature and has a fantastic ecosystem, after over 20 years of development. There are a variety of front-end programming languages and back-end hardware targets supported. To support their own stacks, several software and hardware companies provide their own customised branch of LLVM. The practise of decoupling and modularization is the most significant aspect of LLVM. It is only natural that LLVM has spawned a wealth of fantastic libraries and tools. Intel C/C++ compilers have adopted LLVM [21] and a new LLVM based compiler for Dotnet has been announced [12]. The most important piece in this ecosystem is the IR. It helps the systems represent programs using control flow and basic blocks containing static single-assignment-formed instructions. Its completeness allows it to be self-contained, allowing it to be separated from other representations and serve as the only point of contact between front-ends and back-ends. This completely separates front-ends and back-ends. LLVM decouples and modularizes compilers with LLVM IR and libraries. It also brings the UNIX-like simplicity to compilers. However, it is not a good fit in several domains due to cer-

tain design aspects. For example, it does not have a hard guarantee on stability and compatibility [30], and LLVM IR itself is a monolithic central IR. MLIR decouples compilers and IRs further by breaking down monolithic IRs into mixable dialects [32]. It enables defining and converting abstractions at different levels and helps in transitioning from monolithic to modular. It also helps different systems to have their own customized solutions. MLIR is discussed in detail, later in Section 2.2.

2.1.3 JIT compilation

Just-in-time (JIT) compilation, also known as run-time compilation, is a method of running computer programs, that includes compilation at run time, rather than before the program is launched. It means that the code is only built when it is required. JIT compilers can be used for performance optimization as they reduce the execution time. A JIT compiler starts operating after the program starts and compiles code, which is commonly referred to as generating code "on the fly" or "Just In Time" [44]. Many programming languages such as Java, and C# [8] have a built-in JIT compiler, to turn platform-neutral bytecode into native code at runtime, to avoid interpretation overheads. JIT compilers optimize code by leveraging run time information to remove common sub-expressions, reduce memory accesses in register allocations, and perform data-analysis and register operations by translating from stack operations [47]. As JIT compilation occurs during the execution of the application, dynamic portions, that do not change throughout execution, such as user input, translates into compile-time constants. Compile-time constants, can be used by JIT compilers to optimise loops, leverage loop unrolling and also to specialize functions using constant propagation.

2.1.4 Apache Calcite

Apache Calcite [4] is a dynamic data management framework. Many major open-source data processing systems, such as Apache Hive [20], Apache Storm [46], Apache Flink [15], Druid [13], and MapD [11], use Calcite as a core software framework for query processing, optimization, and query language support. Calcite's architecture includes a query optimizer that is modular and extensible with hundreds of built-in optimization rules, a query processor that can handle a variety of query languages, an adapter architecture that is designed for extensibility, and support for heterogeneous data models and stores (relational, semi-structured, streaming, and geo-spatial). Calcite's modular, embeddable, and extensible architecture makes it an appealing candidate for use in big-data frameworks. It is a live project that is constantly adding support for new data sources, query languages, and query processing and optimization techniques. Calcite optimizes queries by repeatedly applying planner rules to a relational expression. A cost model guides the process, and the planner engine generates an alternative expression that has the same semantics as the original but at a lower cost. Figure 2.2 shows the architecture for Apache Calcite.

2.1.5 Apache Arrow

Apache Arrow [3] defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format also supports zero-copy reads for lightning-fast data access without serialization overhead. A critical component of Apache Arrow is its in-memory columnar format, a standardized, language-agnostic specification for representing structured, table-like datasets in-memory. This data format has a rich data type system (included nested and user-defined data types) designed to support the needs of analytic database systems and data frame libraries. An in-memory database loads the entire data into memory, during the startup phase.

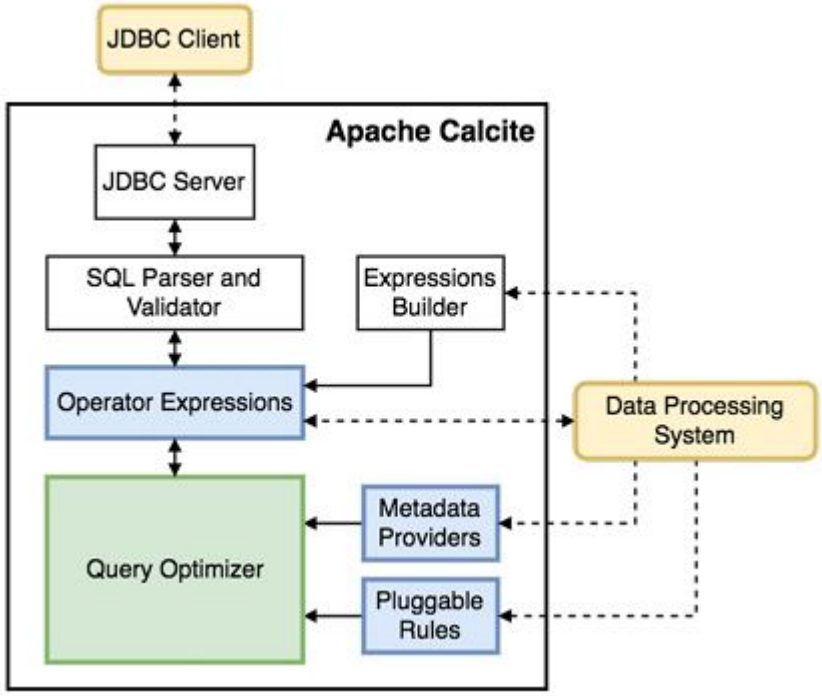

Figure 2.2: Calcite Architecture [4]

CasaDB uses Apache Arrow to load and store the data in memory in a columnar format.

2.1.6 Eclipse OpenJ9

Eclipse OpenJ9 [38] is a high performance, scalable, enterprise-grade Java Virtual Machine (JVM) implementation. Contributed to the Eclipse project by IBM, the OpenJ9 JVM underpins the IBM SDK, Java Technology Edition product that is a core component of many IBM Enterprise software products. The OpenJ9 JVM combines with the Java Class libraries from OpenJDK to create a complete JDK tuned for footprint, performance, and reliability that is well suited for cloud deployments.

2.1.7 PostgreSQL

PostgreSQL [42] is an advanced, enterprise-class, and open-source relational database system. PostgreSQL supports both SQL (relational) and JavaScript Object Nota-

tion (JSON) (non-relational) querying. It is a highly stable database backed by more than 20 years of development by the open-source community. PostgreSQL is used as a primary database for many web applications as well as mobile and analytics applications. PostgreSQL has been proven to be highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate. There are active PostgreSQL clusters in production environments that manage many terabytes of data, and specialized systems that manage petabytes.

2.1.8 Query Execution and Code-Generation

In most early database systems, query engine design followed a volcano-style [18] iteration model. It worked reasonably well before the advent of main-memory database systems. As the major bottleneck shifted away from disk read/write (with the main-memory systems), the inefficient volcano style gave way to *vectorization* [52] and *data-centric code generation* [34] style of query engines. Thomas Neumann [34] studied the model in detail and uncovered the aspects affecting execution speed and also proposed an efficient modern approach to query compilation. In his landmark paper on query compilation in 2011, he described the three design aspects of this model that affects the overall performance of a DBMS. In a volcano iterator query engine, every physical algebraic operator generates a tuple stream from its input and enables iteration over this tuple stream by repeatedly invoking the operator's *next* function. Although this is a very concise and simple interface that allows for the easy combination of arbitrary operators, it is not suited for modern systems due to several inherent technical design flaws. First, the *next* function will be called millions of times (depending on the number of records). Second, a virtual call or a call via a function pointer is commonly used to call *next*. As a result, the call is much more expensive than a standard function call, and degrades the branch prediction on modern systems. Third, code-locality and keeping a record of tuple-location within

the system is inefficient, which leads to performance degradation, when compressed relations are used in the system.

The datacentric code-generation model [35] describes a system where each relational operator implements a push-based interface, named *produce* and *consume*. The operators themselves do not always work directly on the tuples, instead the produce/-consume calls generate code for a given query. They can alternatively be described as operator methods that are invoked during a depth-first traversal of the query plan tree, with produce being invoked during the first and consume being invoked in the second, after all children have been processed. The code generated is tailored to the query's data types and combines all operators from a pipeline of non-blocking relational operators into a single (possibly nested) loop, that can be compiled into efficient machine code. The major advantage of this model is that the generated code minimizes memory access. It loops over all tuples of a materializing operator and loads each individual tuple into CPU registers, processes the tuple and stores it into the next materializing operator, forming a pipeline.

The system implemented as part of this research, follows the datacentric approach and generates code in C++ and UPC++. This dynamic code-generation allows for very efficient execution plans.

2.2 Multilevel Intermediate Representation

MLIR [28] is a reusable and extensible compiler infrastructure. There are several well-known algorithms in the compiler-design field, that handle static analysis, code-generation, transformations and many other aspects. There are also several technology platforms like LLVM [27] compiler infrastructure and JVM [29] that enables

massive code reuse across the systems. These systems offer a single level of abstraction to interface with the respective systems, which enables a straightforward integration. At the same time, this approach causes code fragmentation and increases the cost of building a domain specific Intermediate Representation and compilers. There is also an issue of different levels of abstractions; different problems are optimally modelled at different levels of abstraction. As an example, a C++ source code analysis is very complex at the LLVM IR level and many programming languages develop their own IR to solve domain-specific problems. MLIR greatly helps in building domain-specific IRs to tackle programming language design and implementation challenges. There are three major elements of MLIR design that enables it to solve compiler engineering problems better than any existing frameworks:

- Standardizing the Static Single Assignment (SSA)-based IR data structures;
- Providing a declarative system for defining IR dialects; and
- Providing a wide range of common infrastructure like documentation, parsing and printing logic, location tracking, multi-threaded compilation support, and pass management.

MLIR is a relatively young but reasonably mature ecosystem. It has already enabled profound changes to the compilers [21] [12], most notably infrastructurization. Infrastructurization happens when the technology has matured enough to be part of the infrastructure itself, rather than being a different entity. Being part of the infrastructure means the technology is mature enough and is widely adopted. MLIR is a compiler infrastructure that aims to provide reusable and extensible fundamental components to facilitate building domain-specific compilers. It has no fixed standard IR, but instead, it promotes the development of domain specific dialects. What MLIR provides is just infrastructure to define operations and form logical groups of them based on functionality. MLIR also provides universal patterns or passes

that can apply to suitable operations without hard-coding them. MLIR's dialect ecosystem [32] is in a growing phase, but there are already several stable and mature dialects. For example, there is an LLVM dialect to convert to/from IRs in other systems. Then there are middle level abstractions like Linalg, Tensor, Vector, SCF for structured code generation. There is Affine, Math, and Arithmetic for low level computation and AI framework dialects like TensorFlow, TFLite, MHLO, Torch, and TOSA for importing ML model graphs into MLIR.

An intermediate representation is any representation of a program "between" the source and target languages. MLIR is an intermediate representation system between a language (like C) and the compiler-backend (like LLVM). It allows code reuse between different compiler stack of different languages and other performance and usability benefits. There are 3 major advantages of using MLIR:

- Source code location tracking by default.
- All functions run on multiple cores by default (uses OpenMP [2]).
- Better code reuse for compiler stack (for new library and hardware) and hence, optimizations done by other languages can be reused.

There are no predefined types or instructions. This allows developers to define our own data-types and instruction abstraction. There are also no predefined operations. This means that we can model our problems at a different abstraction level, defining our own operations and providing implementation through dialects. An object in MLIR is a dialect that can be thought of as a class, from the point of view of implementation. A dialect has 3 parts: function name, list of input parameters, and list of output parameters. This helps us in modularizing the generated IR from the CasaDB code-generator. By default, all functions run on all cores and each operand has a source code memory address attribute, so that errors directly point to the line of source code in which the error originated.

2.3 UPC++

UPC++ [50] is a C++ library that supports Partitioned Global Address Space (PGAS) programming [1]. It is designed for writing efficient, scalable parallel programs by making use of distributed-memory. The UPC++ uses one-sided Remote Memory Access (RMA) and Remote Procedure Call (RPC). UPC++ is designed to support exascale high-performance computing, and the library interfaces and implementation maximize scalability. In UPC++, all communication operations are syntactically explicit, which encourages programmers to consider the costs associated with communication and data movement.

2.4 TPC-H Benchmark Queries

TPC-H is a benchmark for decision support, transaction processing, and database performance [5], [36]. It consists of ad-hoc business analytics queries, depicting decision support systems that examine large amounts of data, execute complex queries, and provide answers to critical business questions. The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric and reflects multiple aspects of the capability of the system to process queries such as database size, query processing power and query throughput.

2.4.1 Schema and Scale Factors

The Transaction Processing Council Ad-hoc/decision support benchmark (TPC-H) benchmark suite consists of a 3rd Normal Form (3NF) schema consisting of 8 tables. 3rd normal form is a database schema design approach for relational databases, which uses normalizing principles to reduce the duplication of data, avoid data anomalies,

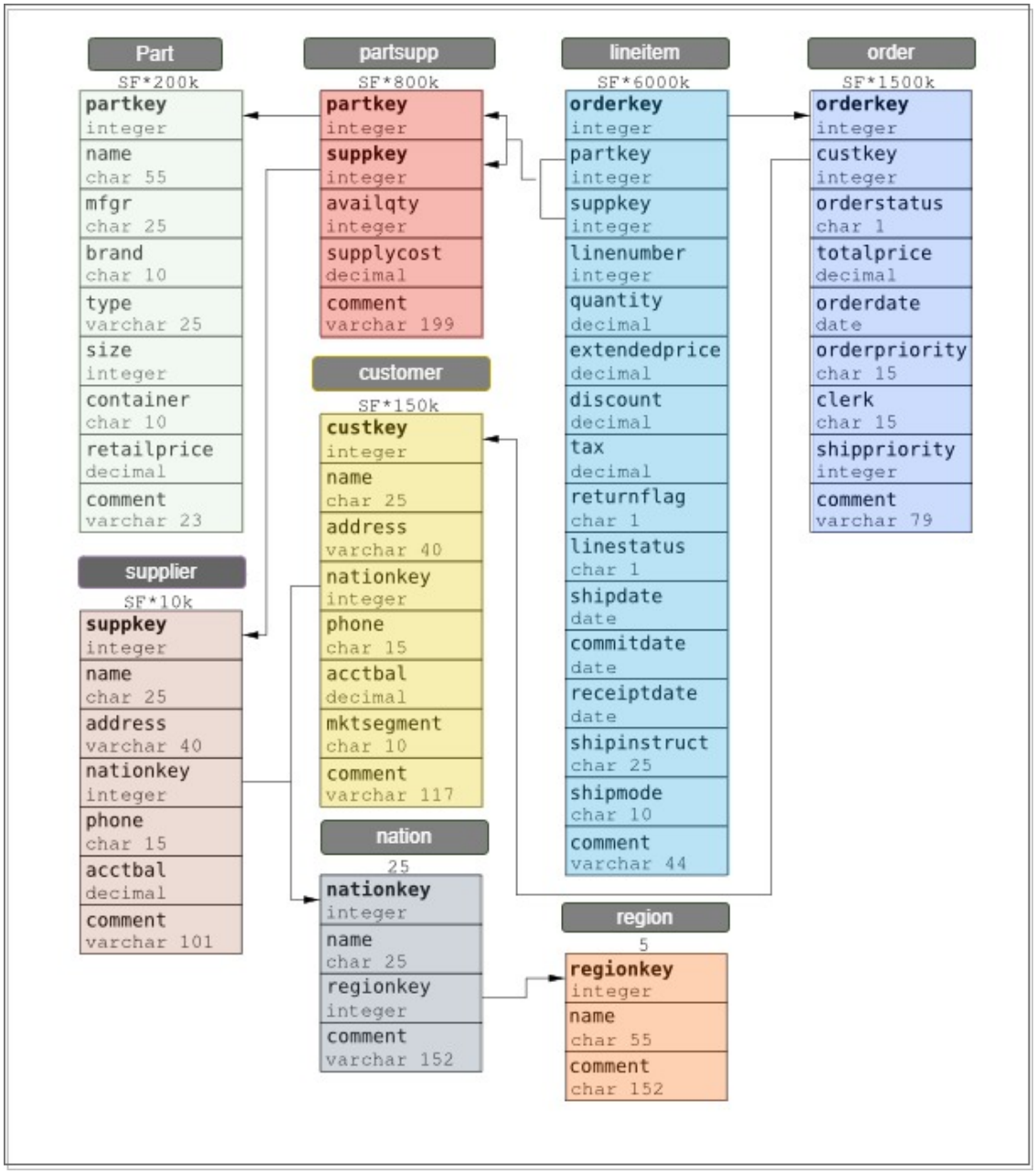

Figure 2.3: TPC-H Schema

ensure referential integrity, and simplify data management [51]. The benchmarks can be run using pre-determined database sizes, referred to as "scale factors". Each scale factor corresponds to the raw data size of the data warehouse. A scale-factor of 1 corresponds to a raw data size of 1 GB and is commonly denoted as "SF-1". Out of the eight tables, six grow linearly with the scale factor and are populated with data that is uniformly distributed. The relationships between columns in these tables are illustrated in Figure 2.3, taken from the TPC-H website [49]. TPC-H comes with various data set sizes to test different scaling factors.

2.4.2 Queries

TPC-H benchmark suite consists of 22 queries. A list of five queries were chosen for evaluation of the execution engine as part of this research. The TPC-H queries used in this research are listed below.

Pricing Summary Report Query (Q1): This query (Listing 2.1) reports the amount of business that was billed, shipped, and returned. It provides a summary pricing report for all *lineitems* shipped as of a given date. The given date is within 60–120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of *lineitems* in each group is included.

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
```

```
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as
    sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
    from
        lineitem
    where
        l_shipdate <= date '1998-12-01' - interval '90' day
    group by
        l_returnflag,
        l_linestatus
    order by
        l_returnflag,
        l_linestatus
```

Listing 2.1: TPCH-Q1

Shipping Priority Query (Q3): This query, shown in Listing 2.2, retrieves the 10 unshipped orders with the highest value. The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of *l_extendedprice * (1-l_discount)*, of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 un-shipped orders exist, only the 10 orders with the largest revenue are listed.

```
    select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
```

```
        o_shippriority
    from
        customer,
        orders,
        lineitem
    where
        c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-15'
        and l_shipdate > date '1995-03-15'
    group by
        l_orderkey,
        o_orderdate,
        o_shippriority
    order by
        revenue desc,
        o_orderdate
    limit 10
```

Listing 2.2: TPCH-Q3

Order Priority Checking Query (Q4): The Order Priority Checking Query, shown in Listing 2.3, counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

```
 select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
```

```
        o_orderdate >= date '1993-07-01'
        and o_orderdate < date '1993-10-01'
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority
```

Listing 2.3: TPCH-Q4

Forecasting Revenue Change Query (Q6): This query, shown in Listing 2.4, quantifies the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range in a given year. It considers all the *lineitems* shipped in a given year with discounts between DISCOUNT-0.01 and DISCOUNT+0.01. The query lists the amount by which the total revenue would have increased if these discounts had been eliminated for *lineitems* with l_quantity less than quantity.

```
select
        sum(l_extendedprice * l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1995-01-01'
```

```
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
```

Listing 2.4: TPCH-Q6

Discounted Revenue Query (Q19): The Discounted Revenue Query, shown in Listing 2.5, reports the gross discounted revenue attributed to the sale of selected parts handled in a particular manner. Parts are selected based on the combination of specific brands, a list of containers, and a range of sizes.

```
select
        sum(l_extendedprice* (1 - l_discount)) as revenue
from
        lineitem,
        part
where
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#12'
                and p_container in ('SM CASE', 'SM BOX', 'SM PACK',
   'SM PKG')
                and l_quantity >= 1 and l_quantity <= 1 + 10
                and p_size between 1 and 5
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#23'
                and p_container in ('MED BAG', 'MED BOX', 'MED PKG',
    'MED PACK')
                and l_quantity >= 10 and l_quantity <= 10 + 10
```

```
                and p_size between 1 and 10
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#34'
                and p_container in ('LG CASE', 'LG BOX', 'LG PACK',
'LG PKG')
                and l_quantity >= 20 and l_quantity <= 20 + 10
                and p_size between 1 and 15
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
```

Listing 2.5: TPCH-Q19

2.5 Related Work

In recent years, Query compilation [48] and Intermediate Representations (IR) for relational algebra has become a hot topic for research. The pull-based iterator model of query execution is not very efficient when the data-set is huge and better approaches have been proposed that eliminates interpretation overheads by compiling the queries into specialized programs.

Hyper [23] uses datacentric code-generation techniques along with LLVM IR to achieve excellent performance on query execution with very large data-sets. Early in 2010, Krikellas et al. [26] proposed a template based approach for code generation from SQL query plans. They proposed and implemented a system called HIQUE,

where the idea was to use a collection of highly efficient code templates and dynamically instantiate them to create query and hardware-specific source code. The source code was compiled and dynamically linked to the database server for processing. They argued that code generation diminishes the bloat of higher-level programming abstractions necessary for implementing generic, interpreted, SQL query engines.

Weld [40] was proposed as a common run-time for high performance data analytics. The researchers put forth issues related to larger systems, where multiple libraries are combined together using function calls. It works great, when the code deals with small data-sets,but when the data size grows, the cost of data-movement becomes a barrier. Applications become quite slow, even though the computing power is substantial. Weld addresses this issue and proposes a new API and managed Intermediate representation for cross-library optimizations. Data-movement and parallelism optimizations are the key. The system works on separately developed, existing libraries written in multiple languages. It has only two parallel operators, known as loops and builders. Weld has been implemented and tested with Python and Scala, and requires integration with each library that is required to be part of the optimization. This could be a tedious task, but Weld supports incremental integration, where parts of the existing library (functions) can be moved to weld.

Funke et al. introduced FlounderIR [17] to mitigate longer compile-time. The proposed system integrates machine-level code-generation with a query compiler. It also demonstrates the abstractions introduced via FlounderIR and the translation of the IR to machine-code tailored to the underlying hardware.

Another recent related work by Chen et al. introduced HorseIR [10] and compared a relational DBMS to a dynamic language processor and proposed to use array-based

programming language optimizations to gain performance, while executing complex queries. HorseIR is an intermediate representation that falls between the query and the compiled code. The query is first converted to an optimized query plan and then to the array-based IR (HorseIR), which allows the application of compiler optimization strategies like dead-code elimination, and loop fusion. The vector is the key in-built data-structure in HorseIR, It corresponds to columns in the database and has well-defined built-in functions to operate on the data. The type system allows both explicit and wild-card datatypes.

2.6 Summary

In this chapter *RQ1* was pursued by surveying existing query processing models like iterator-model and push-based models. The different components that will be used in pursuing *RQ2* were also described in this chapter. Query optimization and planning, Intermediate Representations, along with bench-marking of query processing systems were also referred to in this chapter. Existing similar systems like Hyper, Weld and HIQUE were also mentioned.

Chapter 3

CasaDB: Push-based In-Memory Query Processing

3.1 Design and Implementation

Over the years, the slow I/O capabilities of traditional secondary storage devices forced data management systems to adopt design strategies such as cache pools, concurrency control, and disk-oriented algorithms to hide the I/O performance gap. Today, there are systems capable of accommodating terabytes of memory, albeit at a substantial price premium over conventional or solid-state storage alternatives. This means that big data operations can be performed entirely in-memory and the performance of such a query engine will primarily be dominated by its memory access and CPU usage, as the disk I/O is eliminated [6].

The volcano-style iterator model, which had high interpretation overhead, has been replaced by the datacentric, run-time code-generation model [24]. This model leads to very efficient and compact code, that accesses memory as rarely as possible. This is achieved by using *pipelines*. The code loops over all tuples of a materializing operator, loading each tuple into CPU registers, processing each tuple when needed

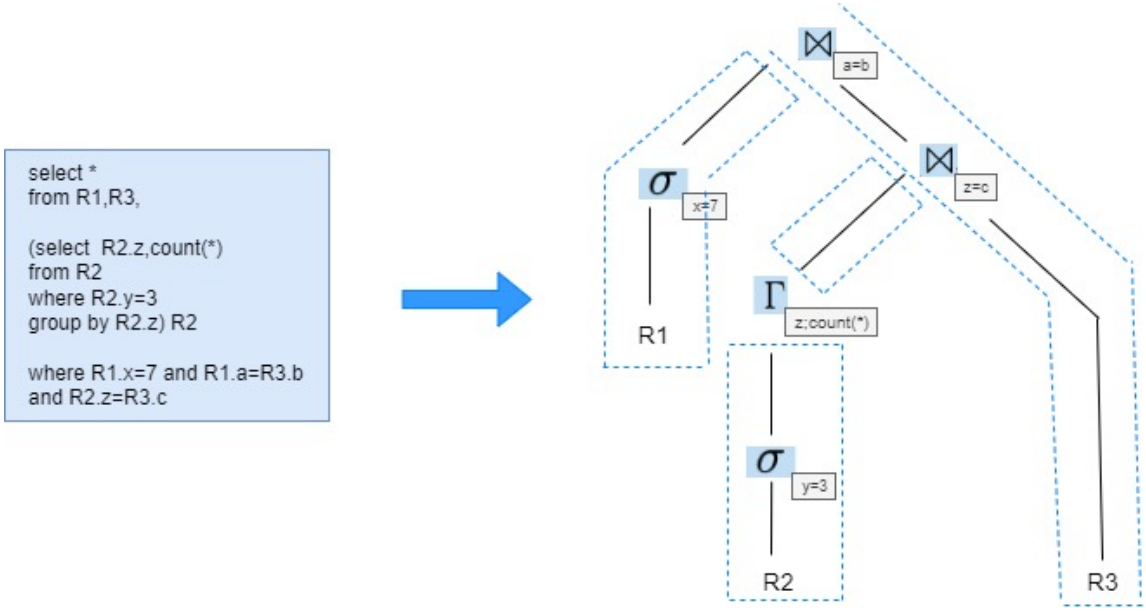

Figure 3.1: Query Plan with visible Pipelines [35]

and storing them into the next materializing operator. This sequence, referred to as a pipeline, consists of multiple operators, and each operator can be a part of multiple pipelines [35]. Figure 3.1 depicts an example query plan with visible pipeline boundaries.

Consider the query and the pipelines in Figure 3.1. The first fragment combines the table scan for relation R1, the selection and filter of x=7, and the build part of join(z=c) into one code fragment. The query execution code is designed to be datacentric, where each code fragment performs all actions possible on its part of the execution pipeline, before materializing the result into the next pipeline breaking operator. This makes query compilation complicated, as the individual operator logic is spread out over multiple code fragments with an irregular structure. In the iterator model, everything is a simple "next" call, but in this case, the complex operator logic directly affects the code generation. The iterator model has a nice, simple interface, but the usage of virtual function calls and frequent memory accesses degrades the performance. Code-generation is a much more complex process, but it can generate near optimal assembly code, with only the necessary instructions and also keep all

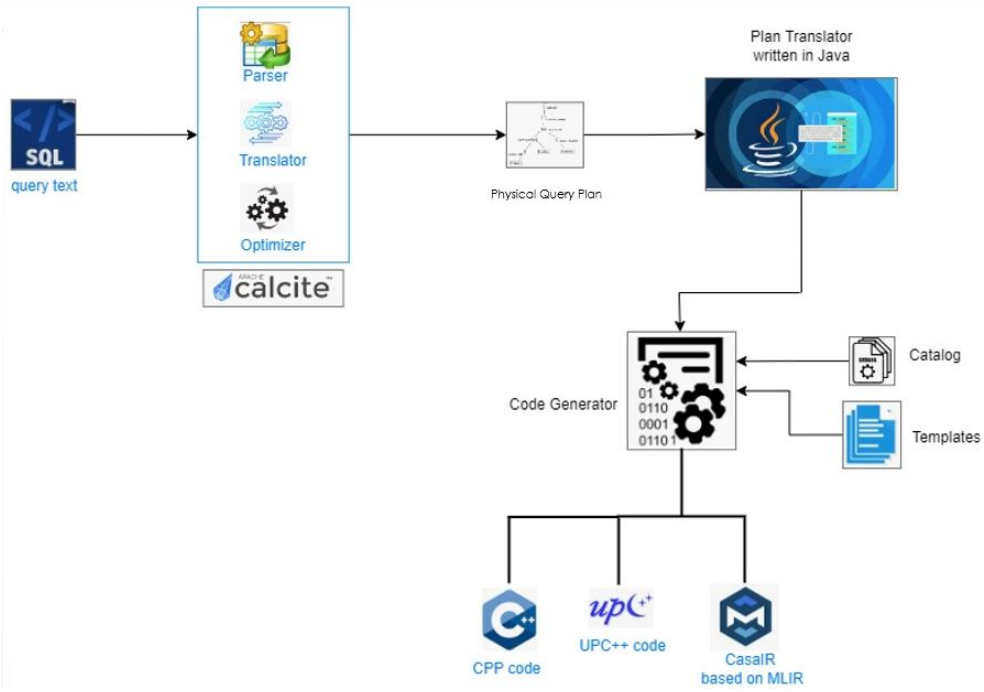

Figure 3.2: CasaDB - Architecture

relevant values in CPU registers, increasing the performance.

As part of this thesis, RQ2 was pursued by designing and implementing a query processing engine called CasaDB, named after our research centre, Centre for Advanced Studies–Atlantic, Canada (CASA). CasaDB uses a datacentric, push-based model along with query-specific code-generation, to achieve fast query processing.

Figure 3.2 shows the components and the architecture of CasaDB. CasaDB uses Apache Clacite for parsing, translating and optimizing the input query. It generates an optimized query plan which is used by the "Plan Translator" to generate and modified query plan which will be used by the code-generator to generate code in the specified language. The details of the individual components and the architecture is described in the following sections.

3.1.1 Query Parsing and Optimization

There are four different stages in the query execution process. The query is lexically analyzed using a lexer (LEX), parsed by a parser (YACC, BNF) and validated against known metadata of the database. The second stage involves compiling a query tree, using logical operators (logical plan), which is passed as an input to the third stage, query optimization. A query optimizer investigates several access strategies before choosing the optimal query plan, in which operators are converted to physical operators and is called a physical plan. The fourth and the last stage converts the physical plan to query specific executions to return the result. Figure 3.3 shows the various stages as described above.

The primary focus of this research is the query execution engine and our initial inclination was towards an open source database system, which can be modified to include the different components that we designed. PostgreSQL [42] also known as Postgres, which is a free and open-source relational database management system (RDBMS), was the first choice and efforts were made to modify the source code to include JIT compilation using the Eclipse OMR's JitBuilder compiler framework [37] [45], as a pilot. It was quickly realised that effort was being spent on understanding and altering the humongous code-base that had evolved over decades, instead of the research area that was the primary focus. HYPER and MonetDB were other systems that were considered. HYPER [23] is closed-source and forked versions were not flexible enough to be instrumented while MonetDB [7] had a code-base that was too large to be altered, within the desired time-frame. The storage model in MonetDB is a significant deviation from traditional database systems and uses the decomposed storage model (DSM), which represents relational tables using vertical fragmentation, by storing each column in a separate table. Moving forward with MonetDB meant that any system that will be developed will have to use the same vertical fragmentation which will limit our ability to create a modular and extensible query

execution engine. A decision was made to build a minimal database system from the ground up, using open source components interleaved with the newly designed code-generating, push-down model. This gave us flexibility to choose components and implement our own. It also meant choosing the best performing open-source components for those areas of a database that are not the primary focus of this research. This included the Structured Query Language parser and translator, the query optimiser and also internal data representation/storage. The project started with Relational Algebra Compiler (RACO) [33] as the query parser/translator and the optimiser, after an extensive survey of available systems. RACO is part of a distributed query execution engine developed using partitioned global address space (PGAS) [1]. RACO was simple enough to setup and instrument quickly, but it was evident that this part of the system will have to be rewired, as RACO did not support all the features of SQL (e.g: order by, group by). Apache Calcite [4] is an industrial standard query parser and optimiser with very good performance. A proof-of-concept implementation using Apache Calcite proved to be faster and easier to setup and after several experiments, Calcite was chosen as the query parser and optimiser for the new system.

3.1.1.1 Role of Clacite

The focus of this project was on the last stage of query processing, execution of the physical plan in the most efficient way. Using Calcite to handle the three stages before execution, allows the CasaDB execution engine be incrementally extended, by including support for all standard physical operators in SQL. It also allows future expansion as new operators are added to the standard SQL.

Apache Clacite serves as the lexer/parser, validator and query optimizer in the CasaDB architecture, allowing us to concentrate our efforts solely on the query execution engine.

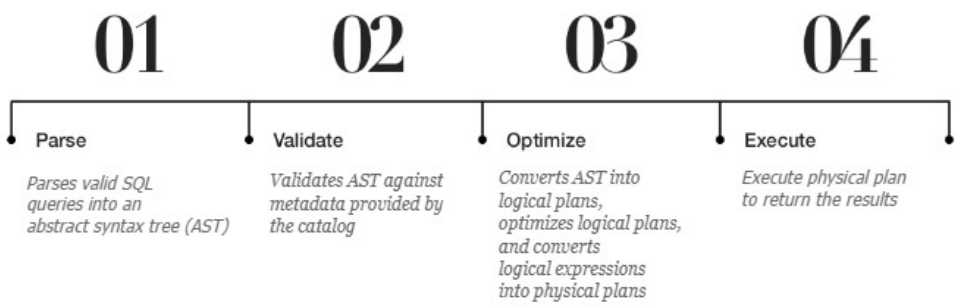

Figure 3.3: Query Processing - Stages

Calcite performs multiple tasks and emits a physical plan in the form of a tree. The nodes of the tree are RelNode objects that represents an operator (e.g., EnumerableAggregate) along with all the necessary attributes like the columns involved, any arithmetic operators or conditional operators.

Consider the query given in Listing 2.2, the *shipping priority query*, which retrieves the shipping priority and potential revenue, defined as the sum of *l_extendedprice * (1-l_discount)*, of the orders having the largest revenue among those that had not been shipped as of a given date.

For a scale-factor of 1, Calcite generates an optimized query plan as depicted in Figure 3.4.

3.1.2 Plan Translator

The physical plan in the form of a "plan-tree" is fed into the "CasaDB PlanTranslator", which applies the datacentric principles to identify materialization points in the tree, before translating the plan to the CasaDB equivalent. The translator accesses metadata, stored in the database catalog, and the physical plan from Calcite, to identify the necessary translation operations. These operations convert the "Calcite-specific" operators to CasaDB equivalent operators. CasaDB operators follow a generic model where each operator requires a set of attributes to be populated.

Table 3.1 lists the attributes required by all the CasaDB operators. The values for these attributes are extracted from the operator nodes in the Calcite generated plan and inserted into an equivalent operator node in the translated CasaDB query plan.

There are three components to be handled when converting Calcite plans to CasaDB plans:

- *Schemas:* Calcite data types are richer than CasaDB data types, so the conversion from Caclite schema to CasaDB relational schema does not reduce the semantic coverage.

- *Scalar expressions:* Scalar expressions are expressions consisting of scalar operands, such as columns, scalar variables, or constants, and operators on those operands, including mathematical operators or functional calls. These expressions are needed in several SQL statements, e.g., a filter condition in a FILTER statement like "a $<$ 1." Scalar expressions in Calcite are represented in expression trees, and are converted into CasaDB expression trees through a post-order traversal over Calcite scalar expression trees to convert expression nodes from leaves to roots.

- *Relational expressions:* Relational expressions are expressions where operands are relations and operators are relational operators, like FILTER, PROJECT, and JOIN. Calcite plans are Directed Acyclic Graphs, where source nodes are LOAD operators on input tables, sink nodes are STORE operators that save result relations into output destinations; and intermediate nodes are relational operators to carry out the transformation. A Depth First Search traversal is performed on the Calcite plan, starting from STORE nodes, and converts each Calcite relational operator.

CasaDB introduces a new operator "APPLY", which acts as the materialization

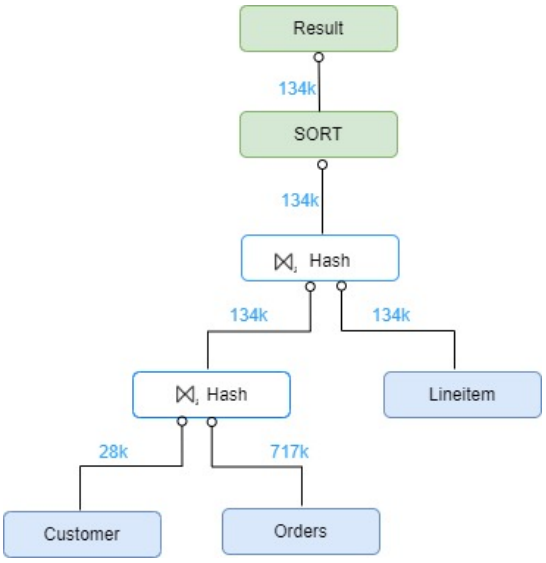

Figure 3.4: Query Plan - TPC-H Q3

Attribute	Description	Examples
Name	Name of the operator	CGROUPBY, CSELECT
ID	Unique Identifier generated internally	140451421575056
Parent	ID of the Parent node	
Input	ID of the Input node	
Right	ID of Right child for multi-input operators like Join	NULL or ID
Grouping_List	List of column numbers for GroupBY	[”0”,”2”,”3”]
Emitters	List of columns emitted by this operator(produce)	"l_orderkey": [{ "child": "UnnamedAttributeRef(0, None)" }]

Table 3.1: CasaDB operator attributes

operator within the system. The translator operates in two passes. In the first pass, it identifies the positions where the “APPLY” operator has to be inserted. In the second pass, it converts Calcite operator nodes to the equivalent CasaDB operator nodes, adds, inserts materialization operators at relevant positions and creates a new plan.

3.2 Query Execution

Once the translator emits the CasaDB compatible query execution plan, the query execution phase is initiated. It involves two stages: code generation and execution.

3.2.1 Code Generation

The code generation in CasaDB is performed by the code-generation layer, which follows a Push-down (produce-consume) model to generate query specific C++ or UPC++ code. Using this strategy, proposed by Neumann [34], optimized and efficient code is generated, which is specific to each query that is being processed. The generated code is near-optimal and it follows the principle of leaving data in CPU registers as much as possible, to reduce data movement to increase the performance of generated code.

In the code generator, all operators provide a uniform interface of *produce()* and *consume()* methods. This setup purely exists only in the code-generator and not in the final generated code. At the query compiler level, this is a very simple and intuitive interface, that is very similar to that of the iterator model. Each operator offers two functions:

- produce()
- consume(attributes, source)

The "produce" method forces the operator to produce its result rows. The results are pushed towards the **consumer** operators by invoking their respective *consume()* functions. Consider TPC-H query 6 (Listing 2.4) and the mapping of the operators to the corresponding generated code fragments as shown in Figure 3.5.

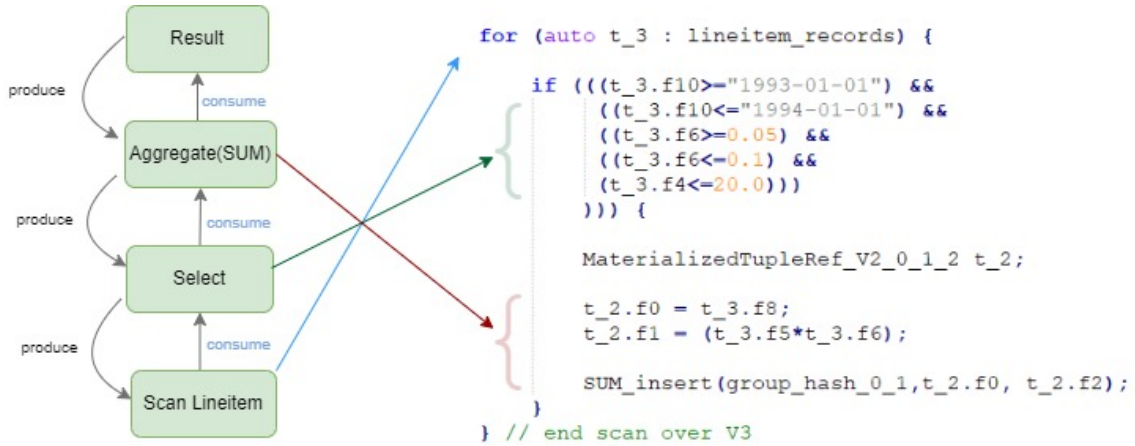

Figure 3.5: Code Generation for Q6

The code generation main module calls the produce function for the *"Result"* operator. Since, *"Result"* does not have any rows in the beginning, it in turn calls the produce() method of *"Aggregate"* followed by *"Select"* and *"Scan"*. Since, *"Scan"* is a terminal operator (leaf node) and can actually consume tuples from the data layer, it ingests the tuples and calls consume() on *"Select"*. This process repeats until the final code-generation operator that requested tuples at the start, receives it.

CasaDB works primarily with the operator tree, and uses "produce" and "consume" functions associated with each operator in the tree to generate code. Produce generates loops over the tuples present for an operator. The child of each operator invokes consume, surrounded by a loop over the operator's tuples. Produce is more like a "pull" action for the tuples from an operator, while consume "push"es them towards the next operator. CasaDB handles expressions by attaching a produce function to each expression, which generates code for evaluating that expression.

This can be further explained using pseudo-code for an operation. Listing 3.1 shows the pseudo-code for a scan filter operation.

```
Scan.produce() {

  output << "for (tuple in T) {"
    parent.consume()
  output << "}"

}

Filter.produce() {
// Generate loop over result, by generating a loop over result of
 it's child.

  child.produce();
}

Filter.consume() {
// Call in a loop over the child's result

  output << "if (";
    condition.produce()

  output << ") {"
    parent.consume()
  output << "}"

}
```

Listing 3.1: Pseudo code for operators (Scan and Filter)

Operators like "Join" and "Group-by" are handled in a different way. These operators store some information in a hash-table and later, return to run a loop over it. Pseudo-code for a hash-based join operation is shown in Listing 3.2

```
Join.produce() {

  // generate a loop over one side
  left_child.produce()

  // generate a loop over the other side
  right_child.produce()

}

Join.consume(src) {

  if (src == left_child) {
    // loop over the left child's result

    output << "add t into HashTable"

  } else {
    // This is called in a loop over the right child's result

    output << "for (t' in HashTable[t]) {"
      parent.consume(this)
    output << "}"

  }
}
```

Listing 3.2: Pseudo code for Hash-Join

Similarly for a Group-by operator, a second loop is used to go over each bucket in the hash-table and output the result of the aggregation function. Listing 3.3 shows the pseudo code for a group-by operator.

```
groupBy.produce() {
  // First, generate a loop over child's result
    child.produce()

  //loop each bucket and then output the result of aggregation
 functions

  output << "for (bucket in HT) {"
    output << "t = compute(bucket)"
    parent.consume()
  output << "}"
}

groupBy.consume() {
  output << "insert t into HashTable[group by columns of t]"
}
```

Listing 3.3: Pseudo code for Group-by

3.2.1.1 Template-based Implementation

In the template-based approach followed in CasaDB to generate C++ code, we use Jinja [22] Templates to define the different sections of the pipelined code. Jinja is a fast, extensible templating-engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document. Jinjava [14] is the Java version of the Jinja templating-engine.

As an example, consider the materialization of a table in the pipelined generated code. We must define a class that represents the tuple during materialization including the fields and the methods needed (e.g., print). It could be made into a template as shown in Figure 3.6. The {{ }} represents text that will be substituted while

```
    // can be just the necessary schema
class {{tupletypename}} {
  // Invariant: data stored in _fields is always in the representation
  // specified by _scheme.

  public:
  {% for ft in fieldtypes %}
      {{ft}} f{{loop.index - 1}};
  {% endfor %}

  static constexpr int numFields() {
    return {{numfields}};
  }

  // size of all fields in struct removing only end padding
  static size_t fieldsSize() {
      const {{tupletypename}} _t;
      return

      {% if numfields == 0 %}
          0;
      {% elif numfields == 1 %}
          sizeof(_t.f0);
      {% else %}
          ((char*)&_t.f{{numfields-1}}) + sizeof(_t.f{{numfields-1}}) - ((char*)&_t);
      {% endif %}
  }

  // debugging function to get storage sizes
  static void print_representation() {
      const {{tupletypename}} _t;

      {% if numfields == 0 %}
          // noop
      {% elif numfields == 1 %}
      std::cout << _t.fieldsSize() << std::endl;
      {% else %}
      {% for i in range(1, numfields) %}
      std::cout << (((char*)&_t.f{{i}}) - ((char*)&_t.f{{i-1}})) << ",";
      {% endfor %}
      std::cout << (_t.fieldsSize() - (((char*)&_t.f{{numfields-2}}) - ((char*)&_t)));
      std::cout << std::endl;
      {% endif %}


  }
```

Figure 3.6: Materialization Template

rendering the document. Using this template, the generated code can be rendered, by supplying the values to be substituted in a HashMap, with key being the name of the variable. The generated code is shown in the Figure 3.7.

3.2.2 Compilation and Execution

The code generated in the previous phase is highly efficient, data-centric and query specific. This means that when compiled and executed, the performance is very close to that of hand-written code for a particular query. The code-generation can be configured for multiple back-end targets. Currently C++ and UPC++ are supported in CasaDB.

Compilation of the generated code is a major overhead in query processing. For long-running queries that deal with a huge amount of data and processing, the compilation time is not very significant. An important aspect at this stage is that the compilation of the generated code is independent of the data-size. As an example, the CasaDB-generated C++ code of TPC-H query 6, took 4 seconds to compile, when experiments were being executed for a scale-factor of 1. The compilation time is independent of the scale-factor and the percentage of time spent on compilation becomes negligible, as the data-size increases to terabytes and beyond. When the scale-factor increases, only the number of rows in each table in the TPC-H benchmark increases. The CasaDB startup will take care of loading the data into memory and the number of records are not hard-coded in the generated code, which means the same code will be generated for the same query for any scale-factor.

CasaDB uses Clang++ [16] compiler to compile the generated code. There is also a provision to cache recent builds using CCache [9], which enables faster compilation and execution of the generated code.

```
class MaterializedTupleRef_V2_0_1_2 {
  // Invariant: data stored in _fields is always in the representation
  // specified by _scheme.

  public:

      std::array<char, MAX_STR_LEN> f0;

      double f1;

      double f2;



  static constexpr int numFields() {
    return 3;
  }

  // size of all fields in struct removing only end padding
  static size_t fieldsSize() {
      const MaterializedTupleRef_V2_0_1_2 _t;
      return


          ((char*)&_t.f2) + sizeof(_t.f2) - ((char*)&_t);

  }

  // debugging function to get storage sizes
  static void print_representation() {
      const MaterializedTupleRef_V2_0_1_2 _t;



      std::cout << (((char*)&_t.f1) - ((char*)&_t.f0)) << ",";

      std::cout << (((char*)&_t.f2) - ((char*)&_t.f1)) << ",";

      std::cout << (_t.fieldsSize() - (((char*)&_t.f1) - ((char*)&_t)));
      std::cout << std::endl;



  }
```

Figure 3.7: Rendered Code - Q6

3.2.3 CasaIR

All of the key ideas from MLIR (Section 2.2), will greatly improve the performance of CasaDB when a fully functional IR is used to model the the problems in relational algebra. Due to the overhead of compiling the generated high-level code, a new Intermediate Representation (IR) for CasaDB has been designed. This MLIR–based CasaIR will be at a higher level than LLVM and will be progressively lowered to LLVM before execution. As part of this research, proof-of-concept, hand-written code has been created, using a combination of different dialects in MLIR [32] like memref, scf, arith and the standard built-in MLIR dialect. This will serve as a precursor to the CasaIR, which will be standard IR for CasaDB. A modified mini version of the TPC-H Query 6 has been used for evaluation. The results are discussed in the next chapter. CasaDB will be enhanced to generate the new IR, from an optimized query plan. This work is being carried out as part of a different project.

3.2.3.1 Query 6 using MLIR

The modified mini version of TPC-H Query 6 used for this experiment, contains an arithmetic operation, along with an aggregation and a filter that includes an arithmetic comparison. There is no support for strings by default, in MLIR dialects, hence the parts of Query 6 that deal with strings have been modified. This is shown in Listing 3.4. MLIR is only a framework for creating IRs, and does not provide any standard types or operations. We will be using some of the more common dialects that have been already created using MLIR and is available to be included in a custom dialect. All of the common available dialects deal with vectors, tensors, control-flow, and arithmetic operations. Hence, "String" as a data-type is not available in any of the common dialects. The ideal way is to create a new dialect with "String" as a type having all related operations and attributes within and use it along with other available dialects. This will be followed when CasaDB is enhanced

to generate its own IR.

```
select
    sum(l_extendedprice * l_discount)
from
    lineitem
where
    l_discount <  0.07
```

Listing 3.4: Modified TPCH-Q6

The first major step is to load the data in the lineitem table. MLIR dialects do not have a straightforward method to deal with file I/O. To circumvent this, an external pre-compiled C++ library was created, which accepts a memref object and loads it with the data read from lineitem.tbl. The main function in the IR is shown in Listing 3.5

The second step is to filter tuples that satisfy the condition $l\_discount < 0.07$. This was achieved using the *scf* dialect, that provides the "for loop" and "if-then-else" constructs. A single loop is used, which handles both the filter and the aggregation in the query.

```
func.func @main() {

  %lqt = memref.alloc() : memref<6001215x2xf64>
  %re = call @loadData_q6(%lqt) : (memref<6001215x2xf64>) -> (i64)
  %a0 = arith.constant 0: index
  %ai = arith.constant 9: index
  %at = arith.constant 1: index
  %rei = arith.index_cast %re: i64 to index
  // External function to note-down timestamp
  call @time_exec(): () -> (i64)
  call @vecsum(%lqt, %rei) : (memref<6001215x2xf64>, index) -> ()
```

```
  call @time_exec(): () -> (i64)
  return
}
```

Listing 3.5: Main Function (Intermediate Representation)

The MLIR–based code is compiled to an executable by using the mlir-optimizer, the mlir-translator and finally clang++ [16]. The external pre-compiled C++ function is linked before executing. These steps are incorporated into a bash script (Listing 3.6).

```
start_time=$(date +%s.%3N)
./bin/mlir-opt high.mlir -convert-scf-to-cf -convert-func-to-llvm -
  convert-cf-to-llvm -convert-memref-to-llvm  -reconcile-
  unrealized-casts -convert-vector-to-llvm -allow-unregistered-
  dialect -mlir-print-debuginfo  > high.llvm
./bin/mlir-translate -mlir-to-llvmir high.llvm -debug -o=high.ll >
  /dev/null 2>&1
clang++  high.ll CRunnerUtils.o /home/nithin/workspace/raco/
  c_test_environment/read_external.o
echo "Compiled.."
end_time=$(date +%s.%3N)
echo "---- Compile time = "
elapsed=$(echo "scale=3; $end_time - $start_time" | bc)
echo $elapsed
./a.out
echo "--------------"
st=$(date +%s%N)
echo $st
```

Listing 3.6: Bash Script for MLIR execution

3.3 Summary

In this chapter RQ2 and RQ3 were pursued and the design and implementation details of CasaDB were explained. CasaDB Architecture and the different components were also described in detail, along with a new MLIR-based intermediate representation (CasaIR). MLIR code to execute a modified TPC-H Q6 was also described in this chapter.

Chapter 4

Evaluation

The performance of the CasaDB, wasevaluated using the TPC-H benchmark suite of queries and the results and analysis are documented in this chapter. The experiments shown in this chapter were designed to evaluate the performance of the proposed query execution system when handling different query processing tasks: join, aggregation, tuple materialization and predicate evaluation. The results were compared to a standard PostgreSQL [41], [43] installation, that uses the "Interpreted" model of query execution

4.1 Experimental Setup

The prerequisites for running the experiments in thesis are discussed in the subsections that follow.

4.1.1 Benchmark

Selected queries from the TPC-H benchmark suite were used to measure the performance of CasaDB. Queries 1, 3, 4, 6, 19 were chosen for this evaluation. These queries are shown in listings 2.1, 2.2, 2.3, 2.4, 2.5 respectively. This set was carefully chosen to include complexities ranging from multiple joins, aggregates, and filters.

Query Operations	TPC-H Queries
Aggregates	Q1, Q3, Q4, Q6, Q19
Predicates	Q1, Q3, Q4, Q6, Q19
Joins	Q3, Q4, Q19
Group By	Q1, Q3, Q4, Q19
Sub-query	Q4

Table 4.1: TPC-H queries and operations

These queries represent most of the functions currently supported by CasaDB. The remaining queries from the TPC-H benchmark suite have substantial overlap with the operations part of the selected queries. A list of query operations and the corresponding TPC-H queries chosen for the experiments are listed in table 4.1.

4.1.2 Hardware and Software

The evaluation experiments were conducted on a four-core AMD Opteron CPU, with an aggregate memory of 20 GB RAM. The generated C++ code was compiled using g++, with maximum optimization level (-O3) and several utility functions were implemented to support hash-tables with different types of keys, handling multiple group-by, Strings, etc. Apache Arrow version 0.12.5 and Calcite version 1.28.0 were used for these experiments. Calcite along with CasaDB was compiled using Openjdk [39] version 11.0.15. CasaDB injects timing information into the generated code, so that the performance can be measured effectively during execution. Each pipeline group execution is timed.

4.2 Results

In all the experiments, the queries were executed 12 times. The results for the first two runs were ignored and the average execution time from the subsequent 10 runs were reported. The first set of experiments were performed to analyse the overall performance of CasaDB. The overall execution time is shown in Figure 4.1.

Figure 4.1: Overall Execution time : CasaDB (SF-1)

CasaDB is designed to be an in-memory database, which means that the data is loaded into memory when the database is started. Once the data is loaded into memory, each query execution can just refer to the loaded data and perform the operations, in memory. Before running experiments for a scale-factor, CasaDB startup was performed as the initial step, which reads the catalog file containing the details of the TPC-H schema, and loads the data into memory using Apache Arrow. A shared memory pointer is used by all the generated code to access the data stored in memory. The same process was followed for all scale-factors and the CasaDB startup time is excluded from query execution timing calculations.

For each query chosen from the TPC-H benchmark, the time taken for query plan and code-generation, compilation and execution of the generated code was recorded.

4.2.1 Analysis

As expected, the code-generation phase of CasaDB (which includes plan generation and optimization by Calcite) is fast and takes the least time. This is represented as the green section of the bar in Figure 4.1. Eclipse OpenJ9 was used with the flag "-XSHAREDCLASSES", which enabled the code-generation time to be lowered to around 105 ms, on average. Q3 consists of multiple joins and it takes the most time for planning and code-generation. This is due to the involvement of three tables, along with an aggregate and multiple group-bys. The planner and optimizer will have a greater set of logical plans to work with and code-generation layer also has to do extra work while generating code for multiple joins. Q4 contains a sub-query which has to be re-written during the optimization as it contains a correlated sub-query along with multiple predicates and and aggregate. As a result, Q4 also takes a few milliseconds for the code-generation phase. Q6, Q1, and Q19 do not contain multi-joins or complex sub-queries and hence take similar time for planning and generation of code.

Execution time is the least for Q6, as it contains no join or condition evaluation. It is a straightforward aggregation along with arithmetic calculations. Q1 takes the longest for providing the result as it contains 7 different aggregations and a group-by along with a filtering condition. Q19 also takes substantially less time for execution as it contains more conditional filters and complex aggregations.

The second set of experiments were carried out to compare the performance of CasaDB to that of an existing database system that uses the interpreted model of query execution. PostgreSQL [42] is a powerful open-source database with over 30 years of development, and uses the interpreted model by default. A comparison of the execution times for the chosen TPC-H queries for a scale-factor 1, is shown in Figure 4.2. All the queries in this set were executed 12 times. The results for the first two runs were ignored and the average execution time from the subsequent 10

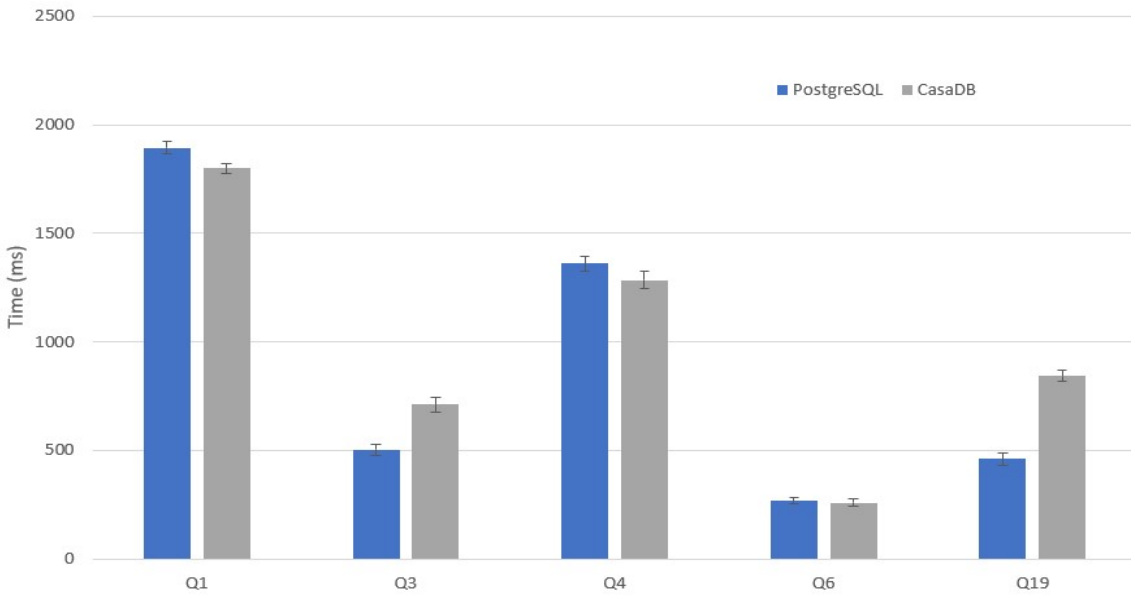

Figure 4.2: CasaDB - PostgreSQL: Execution times, SF-1

runs were used.

From the results, it is evident that CasaDB closely matches the performance of PostgreSQL and even outperforms it for 3 out the 5 chosen queries. Q3 contains multiple joins which affects the code-generation performance of CasaDB, as it supports only hash-based joins. This is currently a limitation and leads to a slightly degraded performance when there are multiple joins involved in a query like Q3. PostgreSQL has the flexibility of switching between nested-loop joins and hash-joins depending on the properties of the relations, giving it an advantage in queries involving multiple joins. Q19 consists of a single join but contains multiple sets of conditional filtering. CasaDB code-generation is fast for Q19, but the execution of the generated C++ code takes longer due to the multiple filtering conditions in the code. Deeply nested conditional statements affect the branch prediction and makes the code execute slower.

The execution timing for a scale-factor of 2 is shown in Figure 4.3. When the scale-factor increases, CasaDB performs better and catches up to PostgreSQL execution times for Q19 and Q3. This is even more evident in Figure 4.4, which shows the

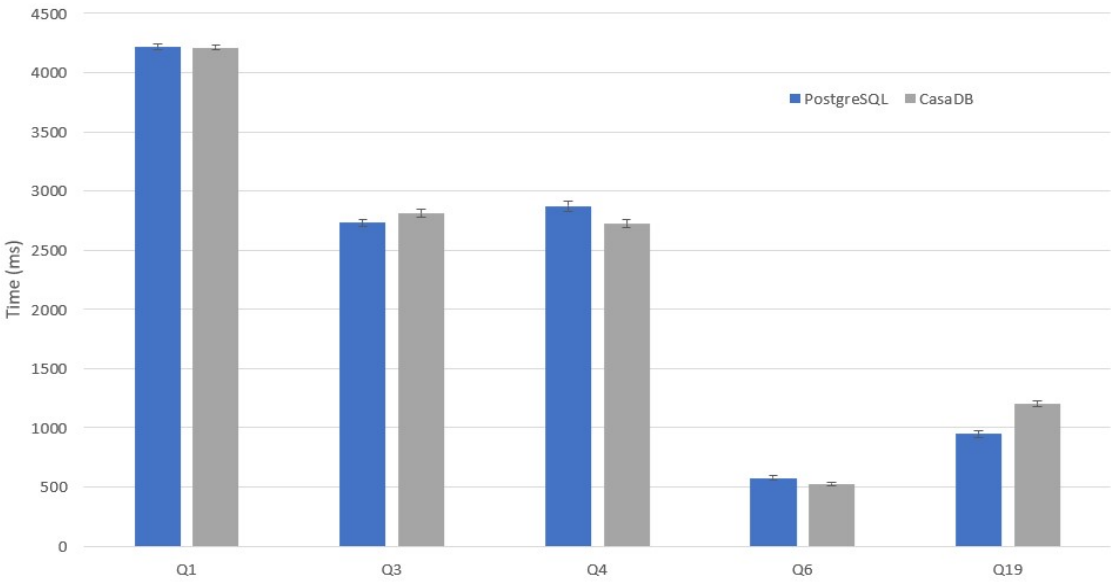

Figure 4.3: CasaDB-PostgreSQL: Execution times, SF-2

execution times for a scale-factor of 4. For Q1 and Q3 there is almost no difference in the execution times and CasaDB performs better in case Q4 and Q6 and the gap in performance is much lesser in the case of Q19, which shows that CasaDB improves in performance when the scale-factor increases. These experiments with different scale-factors proves that CasaDB is well-suited to handle larger datasets.

4.2.2 CasaIR

An MLIR based IR was used to measure the effectiveness of representing an SQL query using an IR and the performance improvements resulting from such a representation, when compared to an existing, interpreted model of query execution. The IR representation was created by-hand, for a modified version of Query 6 as shown in Listing 3.4. The query was executed with multiple scale factors and the execution time was calculated by omitting the file I/O. The same query was executed in PostgreSQL for the different scale factors with the timing option turned on. Scale factors 1, 4, 8 and 16 were used to measure the performance of the system while handling very large datasets. This experiment was executed 12 times for each scale-factor.

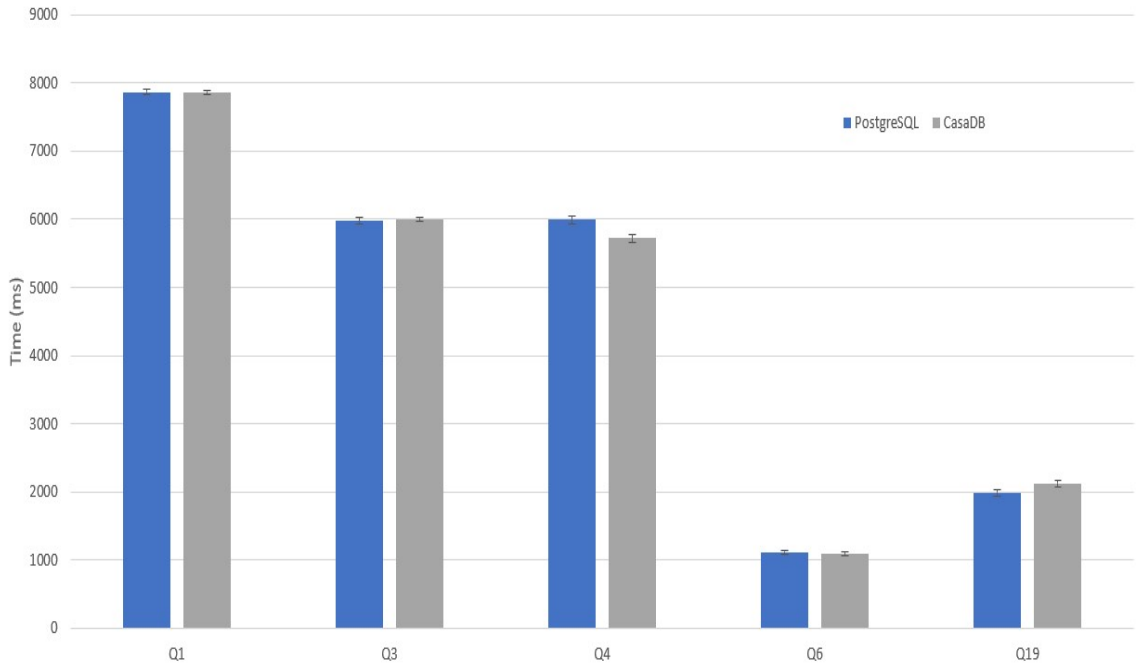

Figure 4.4: CasaDB-PostgreSQL: Execution times, SF-4

The results for the first two runs were ignored and the average execution time from the subsequent 10 runs were used. The execution times of a query directly depends on the scale-factor as it determines the size of the data to be handled. Results are shown in Figure 4.6 and the actual execution times in milliseconds are shown in Figure 4.5

From the results, it is evident that MLIR based IR has a great impact on the performance of CasaDB. Up to 20X reduction in execution time was observed for scale-factors of 1, 4, 8 and 16, when compared to PostgreSQL. The time taken for loading data from files using the pre-compiled C++ function was excluded from the timing, as we are using an in-memory database where data is loaded into the memory on startup. The time taken for plan and code generation is also added to the execution time of the MLIR-based code. From the previous experiments using CasaDB, the plan generation and optimization using Apache Calcite and C++ code generation took 112 ms on average, for TPC-H Q6. This time was added to the MLIR execu-

	MLIR	PostgreSQL
SF1	159	1033
SF4	292	4230
SF8	484	8370
SF16	850	18753

*** To make it a fair comparison, code-generation time of 112(ms) has been added to the execution time for MLIR*

Figure 4.5: Execution times (ms)

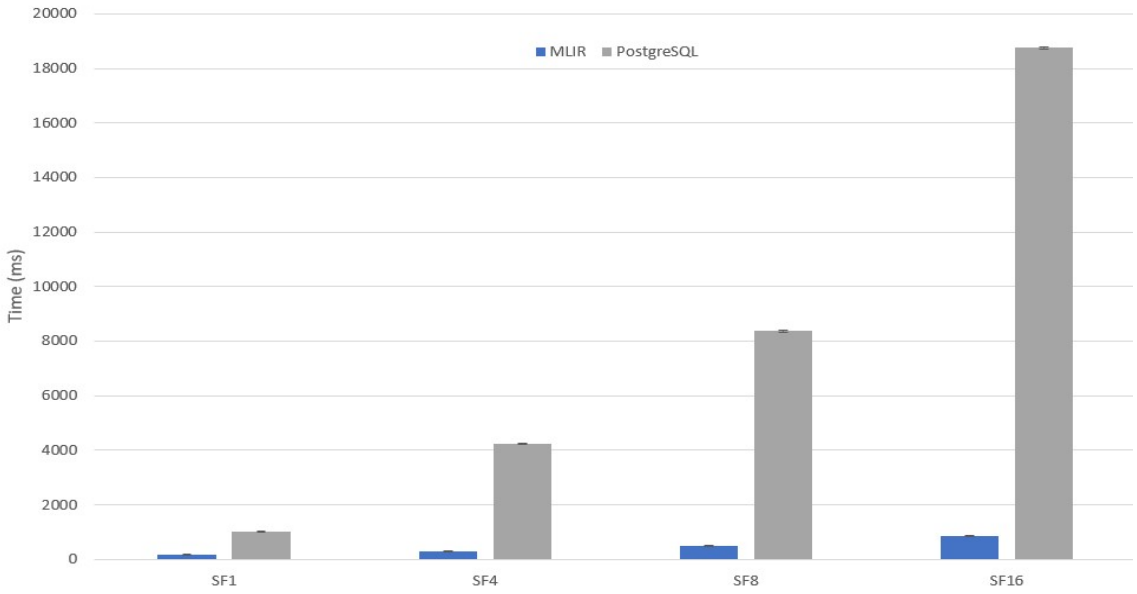

Figure 4.6: MLIR Vs. PostgresSQL TPC-H Q6

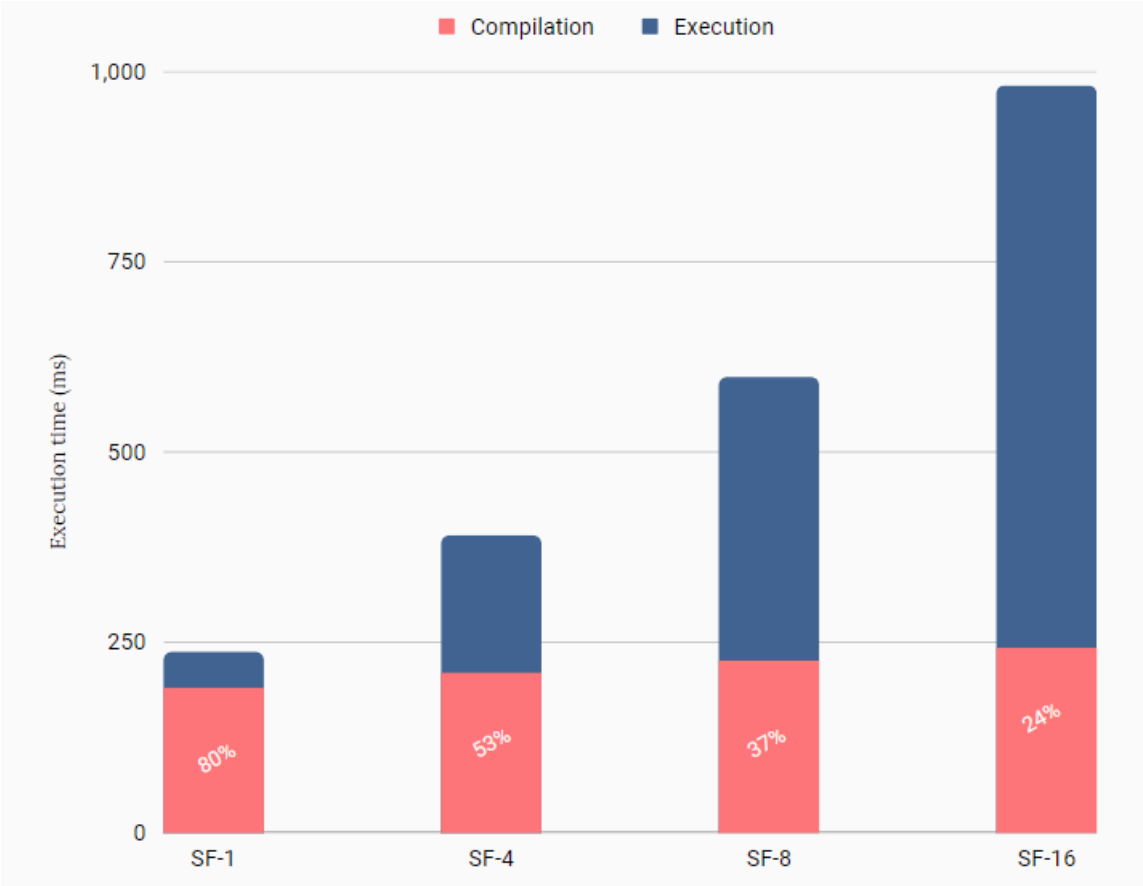

Figure 4.7: Compilation overhead reduction for high SF

tion time, as CasaDB will be eventually used to generate CasaIR. To minimize the file I/O in PostgreSQL, "shared_buffers" parameter was set, so that data is cached during the warm-up runs. The compilation overhead has been drastically reduced when an IR is used and the percentage of time spend in compiling the code also reduces drastically as the scale-factor increases. This is a very good indicator that the use of an IR will result in negligible time spent in compiling code, when handling very large datasets. For SF-1, 80% of the total time is spent for the compilation phase. For SF-4, it reduces to 53% of the total time. The percentage of time spent in the compilation phase further reduces to 37% for SF-8 and 24% for SF-16. This reduction in time spent in compilation as the data size (SF) increases, is explained in the graph shown in Figure 4.7.

4.3 Generating CasaIR

There are two possible approaches to enhance CasaDB code-generation layer to emit CasaIR. In the first approach, CasaIR can be considered as a mix of existing MLIR dialects and the same template-based approach used in generating C++ and UPC++ code can be used. Code fragments can be templatized using the MLIR dialect syntax along with Jinjava. This approach will be faster to implement, but text based rendering of code is suited for generating intermediate representations. The IR code is at a much lower level of abstraction and pre-defining templates for operations will take away some of the optimization opportunities.

The second approach will be to design CasaIR as new MLIR dialect, complete with its own set of types, attributes and operations. All procedures for creating a new dialect must be followed and the API provided by MLIR can be used to generate CasaIR. This approach will require an extension of the code-generation layer to use MLIR API to generate the new dialect. This approach will also allow MLIR optimizer to be more effective and the generated near-optimal code.

4.4 Summary

This chapter provides a detailed analysis of the experiments performed using the new push-based query execution system (CasaDB) developed as part of this thesis. The results are quite promising and provides insights into the potential performance improvements that can be achieved when a new Intermediate Representation is introduced. With the new IR, it was observed that the system performed up to 20X faster than the interpreted model of execution.

Chapter 5

Conclusions and Future Work

This thesis proposes a new push-based query processing system (CasaDB) and implements it using a combination of intuitive code-generator and translator along with open-source components like Apache Calcite and Apache Arrow. The system uses code-generation with a template-based approach and generates C++ code for specific queries. A new IR has also been introduced as part of this thesis. This IR, based on MLIR, helps in modelling relational algebra-based queries at different levels of abstraction. Experiments show that the new system with simple C++ code-generation is able to perform better than existing interpretation-based query processing systems. CasaDB outperforms PostgreSQL in 3 out of the total 5 TPC-H benchmark queries, chosen for evaluation. The design of a new IR and enhancing the code-generation layer to emit the new IR, has also been proved to increase the performance of the system.

5.1 Future Work

The MLIR-based CasaIR has proved to very efficient for representing and executing SQL queries. At present, CasaDB does not fully generate the new IR from the input SQL query. Enhancing the code-generation layer to directly generate CasaIR will

greatly improve the overall performance of the system.

The design of the new IR is another aspect that will impact the performance of the system. The current hand-written IR code, uses a variety of dialects already present as part of the MLIR ecosystem. The design of a new IR, with a new type system and operator definitions, will help in speeding up the code-generation and compilation process. Compilation overhead seen in the generated C++ code, can be greatly overcome using the new IR. Incorporating Ahead-of-Time compilation and an adaptive model of query evaluation [25] are other areas that can be further developed to improve the system.

Bibliography

[1] George Almasi, *PGAS (Partitioned Global Address Space) Languages*, pp. 1539–1545, 01 2011.

[2] OpenMP API, *https://www.openmp.org/*, 2022, Last accessed: 21 june 2022.

[3] Apache Arrow, *https://arrow.apache.org/*, 2022, Last accessed: 22 June 2022.

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire, *Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources*, Proceedings of the 2018 International Conference on Management of Data (New York, NY, USA), SIGMOD '18, Association for Computing Machinery, 2018, p. 221–230.

[5] Peter Boncz, Thomas Neumann, and Orri Erling, *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*, Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391 (Berlin, Heidelberg), Springer-Verlag, 2013, p. 61–76.

[6] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten, *Database Architecture Optimized for the New Bottleneck: Memory Access*, Proceedings of the 25th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '99, Morgan Kaufmann Publishers Inc., 1999, p. 54–65.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes, *MonetDB/X100: Hyper-Pipelining Query Execution*, CIDR, 2005.

[8] Microsoft C#, *C# docs*, 2022, Last accessed: 21 june 2022.

[9] CCache, *CCache - Compiler Cache, https://ccache.dev/*, 2022, Last accessed: 02 June 2022.

[10] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren, *HorseIR: Bringing Array Programming Languages Together with Database Query Processing*, SIGPLAN Not. **53** (2018), no. 8, 37–49.

[11] MapD Core DB, *https://github.com/wesm/mapd-core*, 2022, Last accessed: 02 June 2022.

[12] Dotnetfoundation, *LLVM based compiler for .net*, 2022, Last accessed: 02 July 2022 https://dotnetfoundation.org/blog/2015/04/14/announcing-llilc-llvm-for-dotnet.

[13] Apache Druid, *https://druid.apache.org/*, 2022, Last accessed: 02 June 2022.

[14] Hubspot Jinjava Template engine for java, *https://github.com/hubspot/jinjava*, 2022, Last accessed: 21 june 2022.

[15] Apache Flink, *https://flink.apache.org/*, 2022, Last accessed: 02 June 2022.

[16] Clang Frontend for LLVM, *https://clang.llvm.org/*, 2022, Last accessed: 21 june 2022.

[17] Henning Funke, Jan Mühlig, and Jens Teubner, *Efficient Generation of Machine Code for Query Compilers*, Proceedings of the 16th International Workshop on Data Management on New Hardware (New York, NY, USA), DaMoN '20, Association for Computing Machinery, 2020.

[18] Goetz Graefe and William J. McKenna, *The Volcano Optimizer Generator: Extensibility and Efficient Search.*, ICDE, IEEE Computer Society, 1993.

[19] The Khronos Group, *Spir overview https://www.khronos.org/spir/*, 2022, Last accessed: 02 July 2022.

[20] Apache Hive, *https://hive.apache.org/*, 2022, Last accessed: 02 June 2022.

[21] intel.com, *Intel C,C++ compilers adopt LLVM*, 2022, Last accessed: 02 July 2022.

[22] Jinja, *Jinja Templating System*, 2022, Last accessed: 21 june 2022.

[23] Alfons Kemper and Thomas Neumann, *HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots*, Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (USA), ICDE '11, IEEE Computer Society, 2011, p. 195–206.

[24] Alfons Kemper and Thomas Neumann, *HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots*, Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany (Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, eds.), IEEE Computer Society, 2011, pp. 195–206.

[25] André Kohn, Viktor Leis, and Thomas Neumann, *Adaptive Execution of Compiled Queries*, 2018 IEEE 34th International Conference on Data Engineering (ICDE) (2018), 197–208.

[26] Konstantinos Krikellas, Stratis Viglas, and Marcelo H. Cintra, *Generating code for holistic query evaluation*, 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010) (2010), 613–624.

[27] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (USA), CGO '04, IEEE Computer Society, 2004, p. 75.

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko, *MLIR: A Compiler Infrastructure for the End of Moore's Law*, 2020.

[29] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] llvm.org, *LLVM Developer Policy, https://llvm.org/docs/DeveloperPolicy.html*, 2022, Last accessed: 02 july 2022.

[31] Volker Markl, *Query Processing (in Relational Databases)*, Encyclopedia of Database Systems, Second Edition (Ling Liu and M. Tamer Özsu, eds.), Springer, 2018.

[32] MLIR, *Dialects, https://mlir.llvm.org/docs/dialects*, 2022, Last accessed: 12 June 2022.

[33] Brandon Myers, Mark Oskin, and Bill Howe, *Compiling queries for high-performance computing*, (2016).

[34] Thomas Neumann, *Efficiently Compiling Efficient Query Plans for Modern Hardware*, Proc. VLDB Endow. **4** (2011), no. 9, 539–550.

[35] Thomas. Neumann, *Evolution of a Compiling Query Engine*, Proc. VLDB Endow. **14** (2021), no. 12, 3207–3210.

[36] Sudsanguan Ngamsuriyaroj and Rangsan Pornpattana, *Performance Evaluation of TPC-H Queries on MySQL Cluster*, Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (USA), WAINA '10, IEEE Computer Society, 2010, p. 1035–1040.

[37] Eclipse OMR, *https://github.com/eclipse/omr/tree/master/jitbuilder/release*, 2022, Last accessed: 01 june 2022.

[38] Eclipse OpenJ9, *https://www.eclipse.org/openj9/*, 2022, Last accessed: 21 june 2022.

[39] OpenJDK, *https://www.ibm.com/cloud/support-for-runtimes*, 2022, Last accessed: 02 June 2022.

[40] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Matei A. Zaharia, and Stanford InfoLab, *Weld : A Common Runtime for High Performance Data Analytics*, 2016.

[41] PostgreSQL, *Documentation https://www.postgresql.org/docs/*, 2022, Last accessed: 21 june 2022.

[42] PostgreSQL:, *PostgreSQL, https://www.postgresql.org/*, 2022, Last accessed: 22 June 2022.

[43] PostgreSQL., *Source https://github.com/postgres/postgres/*, 2022, Last accessed: 21 june 2022.

[44] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy M. Lohman, *Compiled Query Execution Engine using JVM*, 22nd International Conference on Data Engineering (ICDE'06) (2006), 23–23.

[45] Mark Stoodley, *Building OMR JitBuilder with CMake https://mstoodle.github.io/BuildingJitBuilderWithCMake/*, 2022, Last accessed: 01 june 2022.

[46] Apache Storm, *https://storm.apache.org/*, 2022, Last accessed: 02 June 2022.

[47] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, *A Dynamic Optimization Framework for a Java Just-in-Time Compiler*, Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (New York, NY, USA), OOPSLA '01, Association for Computing Machinery, 2001, p. 180–195.

[48] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf, *How to Architect a Query Compiler, Revisited*, Proceedings of the 2018 International Conference on Management of Data (New York, NY, USA), SIGMOD '18, Association for Computing Machinery, 2018, p. 307–322.

[49] TPC-H, *https://www.tpc.org/tpch/*, 2022, Last accessed: 01 June 2022.

[50] UPC++, *https://upcxx.lbl.gov/docs/html/guide.html*, 2022, Last accessed: 21 june 2022.

[51] Wikipedia, *Third Normal From*, 2022, Last accessed: 21 june 2022 https://en.wikipedia.org/wiki/Third_normal_form.

[52] Marcin Zukowski, Mark van de Wiel, and Peter Boncz, *Vectorwise: A Vectorized Analytical DBMS*, Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, IEEE Computer Society.

Appendix

Graphs with data labels

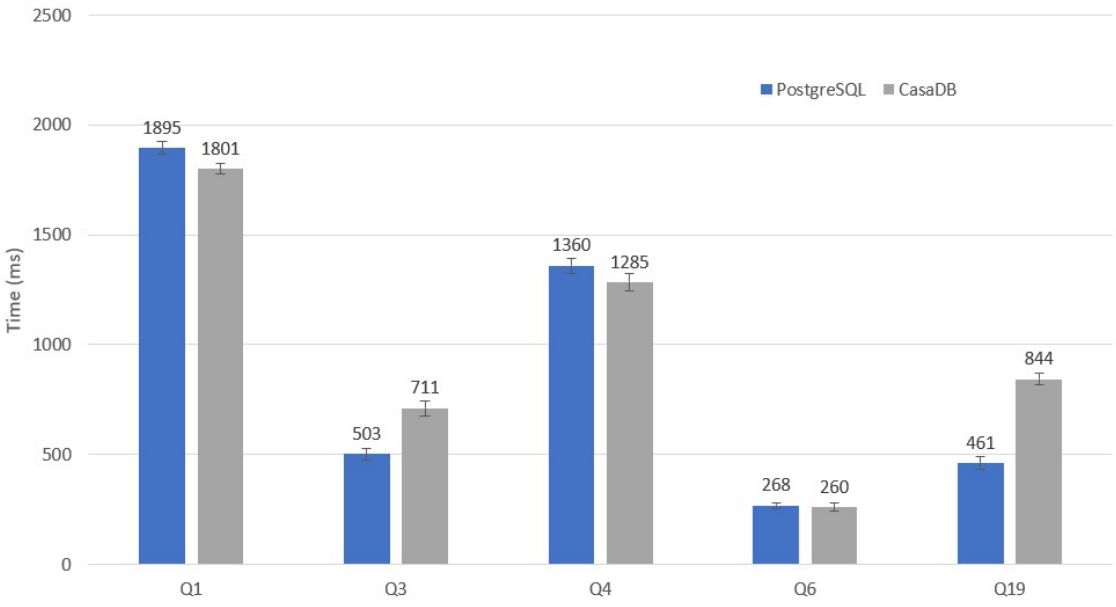

Figure 5.1: Execution Times SF1 with data label

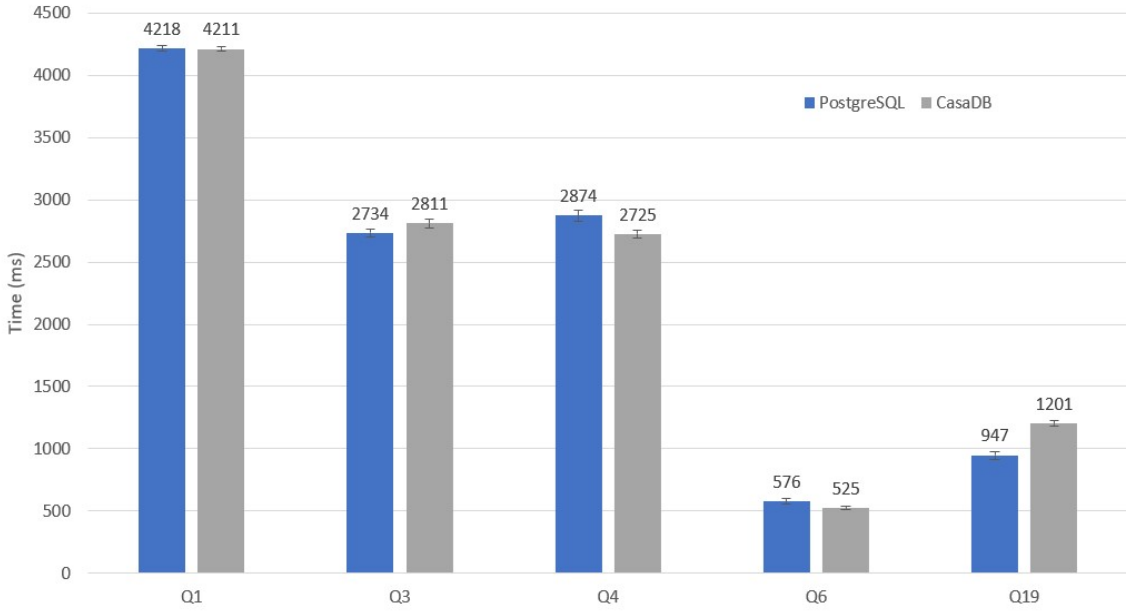

Figure 5.2: Execution Times SF2 with data label

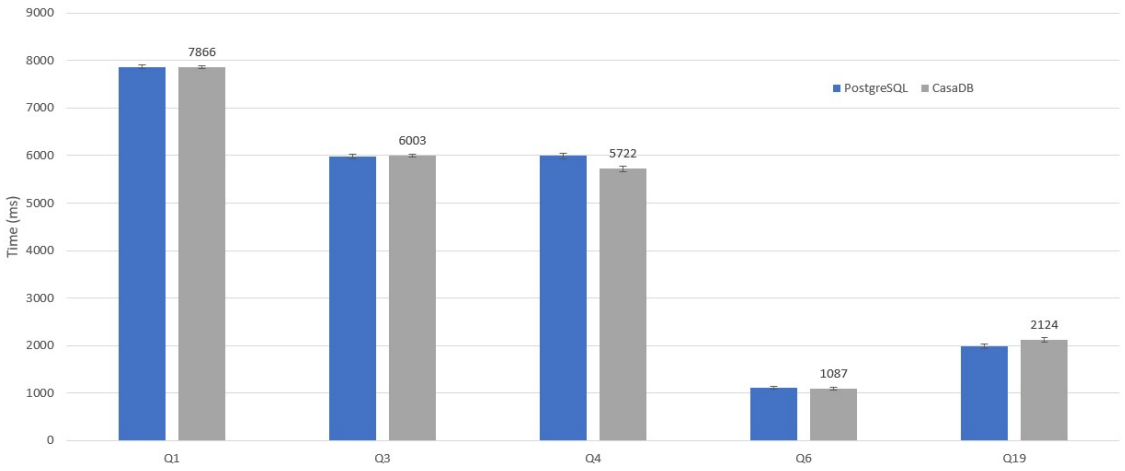

Figure 5.3: Execution Times SF4 with data label

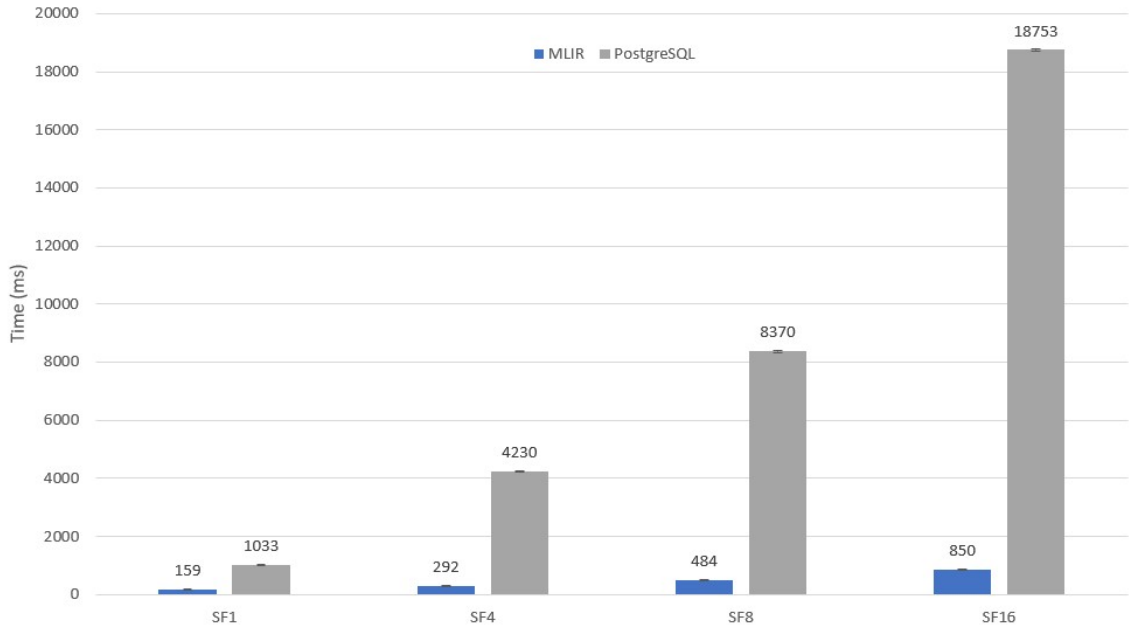

Figure 5.4: Execution Times MLIR Q6

Vita

Candidate's full name: Nithin Ivan

University attended:

- Bachelor of Technology in Information Technology, Mahatma Gandhi University, India, 2005
- Master of Science in Software systems, BITS-Pilani, India, 2014
- Master of Computer Science, University of New Brunswick, Canada, 2022