Towards Compilation of SQL Queries into Efficient Execution Plans for Distributed In-Memory Query Processing

by

Shubh Sharma

Bachelor of Technology, Information Technology
National Institute of Technology Durgapur, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT 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:	Gerhard Dueck, Ph.D., Computer Science, Chair
	Arash Habibi Lashkari, Ph.D., Computer Science
	Bo Cao, Ph.D., Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2021

Abstract

A query processing engine is the core component of any modern database system. There are several types of query processing engines that employ different query processing techniques. The speed of data-driven decision-making and analytics is crucial to firms and organizations that build software and system applications. An intuitive way to speed up database querying is to improve the performance of these engines. Conventionally, databases use a disk-oriented, pull-based or tuple-at-a-time interpreted query evaluation model. In this thesis, a compilation-based, in-memory query compiler is introduced that ingests an SQL query and generates a distributed C++ (UPC++)-based physical query plans. As part of this work, different models and components of query processing are explored, efficient "Partitioned Global Address Space"-based parallel programs corresponding to SQL queries are designed and developed, which are emitted by a code generator that uses a data-centric compilation strategy. The approach proposed in this thesis combines high-performance parallel programs with database query processing to take advantage of the advances in hardware available and offers a 2× speedup in query performance over the best existing approach.

Dedication

To my loving family. Thank you for all the support.

Acknowledgements

This thesis would not have materialized from a research idea into a working system without help and support from a few people. I will use this section to talk about all such people. First and foremost, I would like to thank my supervisor **Dr. Suprio Ray** for his expert academic and research guidance and generous administrative support. He has always motivated me to expand my boundaries and adopt a systematic approach to work. Next, I would like to thank **Dr. Kenneth Kent** for providing this amazing opportunity to work on an industry-led project at IBM CAS—Atlantic and accommodating my needs for remote work from outside Canada during the Coronavirus pandemic. He is a great leader. I would also like to thank **Dr. Mark Stoodley** from IBM Runtime Laboratories for providing his expert insights and feedback at different stages in the project.

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the colleagues and facilities of CAS–Atlantic in supporting this research. I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

I would also like to thank **Nithin Ivan** for his crucial contribution to this project. The several years of technical expertise that he carries has definitely helped this

project take the current shape. I would also like to thank **Dr. Brandon Myers**, Lecturer at University of Iowa, for his invaluable inputs from time to time on what it takes to build a distributed system. I would like to thank **Stephen MacKay** for his constant attention to detail and **DeVerne Jones** for keeping the systems up and running and providing the right infrastructure support whenever needed. Last but not the least, a big thank you to my **family** in India for believing in me and providing all the support that I ever needed.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

$PGAS$	Partitioned Global Address Space
$RACO$	Relational Algebra Compiler
UPC	Unified Parallel C
$TPC-H$	Transaction Processing Council Ad-hoc/decision support benchmark
$UPC++$	Unified Parallel C++

List of Algorithms

Chapter 1

Introduction

In recent times, data-driven decision-making has taken a center stage in both academia as well as industry. In order to extract meaningful information and insights from large amounts of data, fast, efficient and large-scale systems are needed that can operate on the data efficiently. The traditional database systems that exist today have undergone various newer developments to fulfil the aforementioned demands for data analytics. These systems have given more importance to data processing than the conventional database management systems do, as their focus is more on data management. The system presented in this thesis combines high-performance computing with parallel and distributed data processing algorithms for data-intensive applications running on modern computer architectures.

1.1 Motivation

For disk-based database management systems, the biggest performance bottleneck is disk I/O and hence their focus has always been on minimizing disk I/O [8]. However, the advances in modern computer architectures have called for changes in the design principles of current database systems. Since memory is available at much cheaper rates now, relatively large amounts of data can fit in main memory these days than it

used to earlier and so less data has to be kept on disks [21]. Unlike the conventional disk-based database management systems, the new generation of big data systems focuses on achieving high performance by optimizing not just memory usage but also adopting CPU-efficient techniques [30, 19]. It is important to note here that high-performance computing also deals with solving the same problem that we just saw–optimizing CPU and memory usage. In recent years, query compilation has attracted considerable attention from the research community. Query compilation is different from traditional interpretation based query processing, as it allows for the generation of query-specific and data-centric code. However, the focus of the query compilation research has primarily been based on a single node. Therefore, in this thesis an attempt has been made to use modern query compilation algorithms to generate high-level physical query plans that are based on the existing high-performance computing technology stacks for parallel and distributed systems.

1.2 Thesis Statement

When applied to database processing, high-performance distributed shared memory systems and advanced database query compilation techniques support enhanced query performance, developer productivity and efficient scalability of query evaluation systems.

1.3 Research Questions

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

- **RQ1:** What are the various query processing models and which one of them is best suited for data-intensive query processing?
- **RQ2:** What are the various high-performance computing frameworks and lan-

guages and which one of them is best suited to implement query execution plans to run on a cluster of computers in a parallel fashion?

- **RQ3:** How can we generate high-performance query execution plans using the query processing technique ascertained while pursuing RQ1?

1.4 Thesis Contributions

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

- Carrying out a literature survey of existing query processing models including query compilation and state-of-the-art in high-performance parallel programming framework called Partitioned Global Address Space (PGAS) [9],
- Designing and developing PGAS parallel query execution plans and templatizing them
- Customizing and implementing a data partitioning algorithm to suit the needs of this system
- Integrating the PGAS backend with a query compiler for a single node to generate PGAS-based execution for a cluster of computers.

RQ1 was investigated by researching various query processing models including pull-based and push-based models and single-node and distributed query processing models. **RQ2** was pursued by exploring a high-performance programming framework called Partitioned Global Address Space (PGAS) and the programming language libraries that implement PGAS — UPC and UPC++. These literature findings have been documented in **Chapter 2** as Background and Related Work. The design and implementation of CasaDB, the query compiler used in this work has also been discussed in **Chapter 2**. **RQ3** was pursued by designing and developing a framework/backend to build/generate PGAS-based query execution plans and integrating

this backend with CasaDB. CasaDB Distributed, the final product of this integration exercise, ingests an SQL query and emits the corresponding PGAS code (execution/-physical plan) which can then be run in a PGAS runtime environment to yield query results. **Chapter 3** serves as the design and implementation document for CasaDB Distributed. CasaDB Distributed is benchmarked using the TPC-H suite of queries and the results have been recorded and discussed in **Chapter 4**. **Chapter 5**, summarizes the contributions of this work and also discusses its limitations and scope for future work.

Chapter 2

Background and Related Work

This chapter sets the stage for the rest of this thesis by introducing and explaining the important concepts, libraries, frameworks and benchmarks used throughout this research work. Previous work related to our research and how it is different from what we have done in this thesis is also discussed.

2.1 Background

This section discusses the concepts of query processing and its stages, models of query execution, distributed and in-memory query progressing, partitioned global address space (PGAS), PGAS languages and the TPC-H Benchmark.

2.1.1 Query Processing

Query processing [11, 25, 31] refers to the activity of extracting data from a database based on a declarative query language such as the structured query language (SQL). Query processing can be broken down into three stages:

1. parsing and translation

2. planning and optimization

3. query execution

The following sub-sections talk about each of these stages.

2.1.1.1 Parsing and Translation

In this stage of query processing, the parser breaks down the SQL query into tokens and creates a parse tree of the query, based on the grammar of the SQL flavour used. After this, semantic analysis is carried out to to test the semantic correctness of the SQL query. This is done to make sure that table/column mentioned in the query exist and have the correct data type. The validated parse tree is passed on to the next stage of query processing.

2.1.1.2 Planning and Optimization

The input from the planning stage is fed into the query planning components. The query planner generates a logical plan tree based on relational algebra. A query rewrite/optimize module iterates over the possible query plans to generate the most optimized version of the query plan. This logical plan is passed on to a physical query planner that generates an evaluation plan by appending physical operators to the existing plan. The resultant query plan is ready to be executed in a runtime.

2.1.1.3 Execution

The evaluation plan generated in the step above is given as input to a query execution engine. This query execution engine can either be interpretation-based [20] or compilation-based [41]. The engine scans the data from data sources, applies predicates, aggregates and filters on the data as per the evaluation plan and output the result tuples.

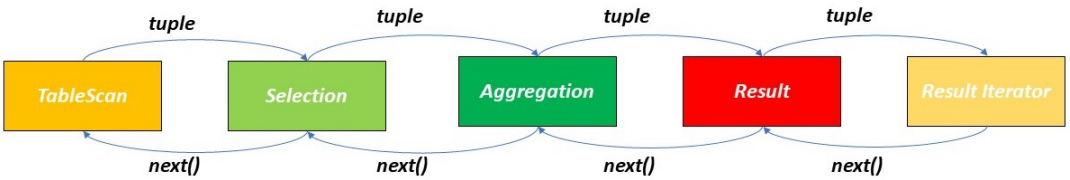


Figure 2.1: Volcano Iterator Model of Query Execution for a Typical SQL Query

2.1.2 Models of Query Execution

Query execution engines primarily use two models for the query execution process: (a) pull-based query execution [20], also known as volcano or iterator model of query execution, and (b) push-based query execution [40]. In a pull-based system, the operators sit idle until another operator asks them for a tuple. This means the process of getting results out of the system involves an operator asking the system for a row and it giving the row to that operator. However, this also means that the behaviour of the system is very tightly coupled to its consumers; the system works if only asked to and not otherwise. In the push-based system, the system sits idle until an operator tells it about a tuple. Thus, the work the system is doing and its consumption are decoupled. The following subsections expand on these concepts briefly:

2.1.2.1 Pull-Based Model

A lot of query execution engines employ the iterator or volcano model due to its simple and intuitive way to filter data. Each operator is implemented as a subclass of the iterator class, using interface functions such as

- open()
- next()
- close()

An iterator can be used as input to any other iterator, thereby enabling universal handling of iterators or iterator combinations by the system, regardless of the particular function they implement. However, the overhead of processing data in a tuple-at-a-time, iterator-based fashion leads to inefficient execution of queries when running on modern hardware and deep memory hierarchies. When several rows are combined together and they move from one operator to the other as a batch or vector, the performance of the iterator model improves significantly. This technique is called vectorization. [41].

2.1.2.2 Push-Based Model

In the push-based model of query execution [35], the direction of data flow is reversed and the processing is data centric instead of being operator centric. The incoming tuples are pushed towards the consumer operators by the producer operators instead of being pulled by them. This leads to better data locality. The tuples are continued to be pushed until a point of materialization is reached. A point of materialization is an operator which causes a tuple to be taken out of the CPU registers. They are also referred to as pipeline breakers as they break the pipeline the data flows in by taking it out of CPU registers. With the push-based model, data is always pushed from one materializing operator to another. Pipeline breakers are explained in Figure 3.4. Operations performed between two such operators act upon the data in CPU registers and are therefore offer cheap computation.

2.1.3 Distributed Query Processing

When dealing with large amounts of data, query processing can be made faster by taking advantage of the advances in modern computing hardware [25]. Distributing the computation task in query processing to a cluster of nodes can result in better performance due to parallel execution of queries. Several commercial and open-

source database systems, like Cloudera Impala [24], support parallel and distributed execution of queries. However, none of them use high-performance computing (HPC) language frameworks to implement this parallelism. That is something this thesis has aimed to accomplish by using Partitioned Global Address Space (PGAS) [9], an HPC programming framework, to augment C++ with distributed runtime capabilities.

2.1.4 In-Memory Columnar Query Processing

In-memory database systems [21, 32] tend to outperform disk-based database systems in terms of speed of query execution on modern hardware as they do not have to deal with disk I/O operations. They load all the data in memory and perform operations on the data sitting in CPU registers/memory. It has also been noticed that in-memory columnar storage tends to yield better results in comparison to row-oriented in-memory systems [17]. For analytics workloads, column-oriented execution has been shown to be faster for a variety of reasons: no wastage of I/O bandwidth for projected columns, compression, late-materialization, and block iteration [7, 17]. In-memory columnar storage libraries like Apache Arrow [1] provide extensive support to implement such functionalities in modern database systems. Arrow C++ has been used as the data reading and loading library in this project too.

2.1.5 Partitioned Global Address Space

Partitioned global address space (PGAS) [9] is a parallel programming model for the development of high-performance applications on computer clusters. It provides a global address space partitioned among the cluster nodes, and is supported in programming languages like C, C++, and Java by means of APIs. An important feature of PGAS is the support for one-sided communication: a computer can directly read and write the memory located at a remote computer, without deliberate synchronization with the processes running on the remote side. One-sided com-

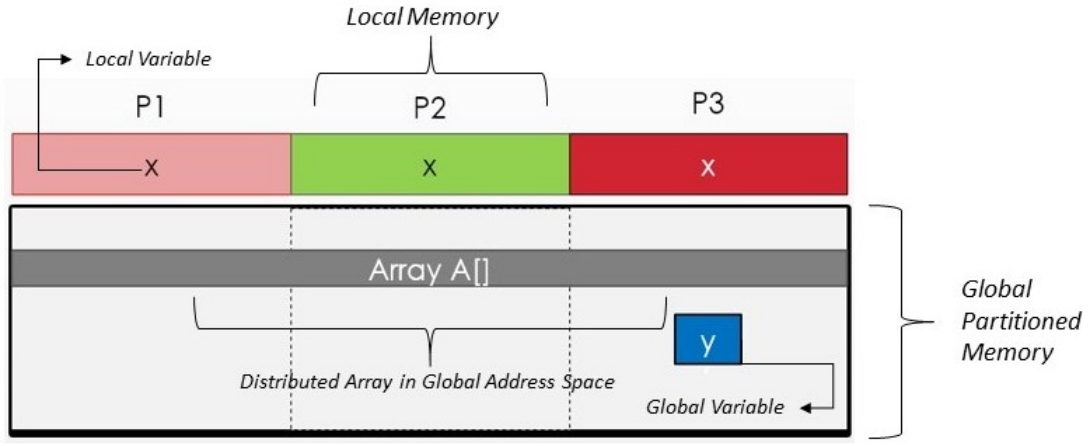


Figure 2.2: Private, Shared and Distributed Data Structures in a PGAS Model

munication improves performance by decoupling process synchronization from data transfer, but requires the developer to think about appropriate synchronizations that will be needed between reads and writes.

PGAS programs are single program, multiple data (SPMD) [16] programs running on a cluster. At run time, a PGAS program consists of multiple processes executing the same code on different nodes. Each process has a rank, which is the identifier of the node it runs on. The processes can access a global address space partitioned into local address spaces for each process. Local addresses can be accessed directly. Remote addresses belonging to different processes are accessed using API calls, also known as remote procedure calls (RPC). Figure 2.2 illustrates the memory of a PGAS system. The pink, green and red memory areas are local to each of the processes $P1$, $P2$ and $P3$ respectively. The grey area of memory is shared among all the three nodes. Each node/ranks has its own copy of the variable x. P1 can only access its own x. It can't access the green and the red x. However, it can access the variable y, as y is present in the global address space of this cluster. The array $A[]$ is a shared distributed data structure. Each process can access it. GASNet [12], a PGAS network protocol, provides both synchronous and asynchronous versions of

reads and writes. A GASNet command returns a handle that can be waited upon, and a return from a wait implies full completion of the operation. The order in which asynchronous operations complete is optimized internally.

2.1.5.1 UPC

Unified Parallel C (UPC) [5, 18, 14] is a PGAS (Partitioned Global Address Space) language that offers an alternative to MPI [43] and OpenMP [15] and is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming paradigm for shared and distributed memory-based hardware. The developer is presented with a single shared, partitioned address space, where variables can be directly read and written by any processor, but each variable is physically associated with a single processor.

In order to achieve parallelism, UPC extends ISO C 99 with the following parallelism and language constructs:

- **Explicitly Parallel Execution Model:** The UPC execution model is similar to that used by the message passing style of programming (MPI or PVM). This model is called Single Program Multiple Data (SPMD) and it is an explicitly parallel model. In UPC's terms, the execution unit for a program is called a thread. The language defines a private variable - MYTHREAD - that can be used to differentiate between the threads of a UPC program. A UPC program is run entirely independently on each of the threads. The language does not define any correspondence between a UPC thread and its Operating System level counterparts, nor does it define any mapping to physical CPUs. Due of this, UPC threads can be implemented either as full-fledged operating system processes or as user or kernel-level threads. On a parallel system, a UPC

program running with shared data will contain at least one UPC thread per physical processor available. To represent the amount of parallelism available to a program (number of UPC threads), the UPC language introduces a new variable - THREADS. The value of this variable can be set in two ways: (a) at compile time, for the cases where the amount of parallelism is already known and (b) at run-time.

- **Shared Address Space:** UPC categorizes the data available to a thread into shared data and private data. This categorization is made through the usage of a new C type-qualifier: shared. Data qualified as shared is accessible from within any UPC thread (the same address on each thread refers to the same physical memory location). Data that does not have the shared qualifier is considered thread private data. For all such data, the same address on each thread refers to distinct physical memory locations. At the compiler level, there is no syntactic difference between the accesses to a shared variable and the accesses to a private variable.

  The UPC language also defines the physical association between shared data items and UPC threads. This association, known as affinity in UPC terms, indicates that a particular thread "owns" a particular data item. From the implementation point of view, affinity translates into storing data into the physical memory of the CPU where the UPC thread is running. When defining the affinity, the language differentiates between scalar data and array data. To accommodate data affinity, UPC defines rules for arithmetic on pointers to shared data items and provides language level primitives to inspect the affinity of a shared data item. All scalar data (of any of the primitive C types, pointer type or user defined aggregate type) has affinity with thread 0. For arrays, the language allows for three affinity granularities:

  – **cyclic (per element)** - successive elements of the array have affinity

with successive threads

  - **blocked-cyclic (user-defined)** - the array is divided into user-defined size blocks and the blocks are cyclically distributed among threads
  - **blocked (run-time)** - each thread has affinity to one contiguous part of the array. The size of the contiguous part is determined in such a way that the array is evenly distributed among threads.

- **Synchronization Primitives:** UPC makes no implicit assumptions about the interaction between threads. All thread interactions are explicitly managed by the developer through the primitives provided by the UPC language: barriers, memory fences and blocks.

- **Memory Consistency Model:** In order to define the interaction between memory accesses to shared data, UPC provides two user controlled consistency models. Each memory reference in a program, can be annotated to be either strict or relaxed. In the strict model, the program executes in a sequential consistency model. It means that it appears to all threads that the strict references within the same thread appear in the program order, relative to all other accesses. In the relaxed model, it appears to the issuing thread that all shared references within the thread appear in the program order. The UPC language allows the user to specify the access consistency on a per variable basis or, at an even coarser granularity, on a per statement basis.

- **Parallel Utility Libraries:** The UPC language provides a number of utility libraries that encapsulate functionality commonly required for writing parallel applications and provide standardized interfaces to capabilities frequently found in modern HPC hardware. Some of these include:

  - Parallel File System I/O

- High-Performance Wall-Clock Timers
- Collective Operations (for example, reductions, broadcast, exchange, scatter, gather, scans)
- Blocking and Non-Blocking Data Copy
- Atomic Memory Operations

2.1.5.2 UPC++

UPC++ [45, 6] is a C++11 library that supports Partitioned Global Address Space (PGAS) programming, and is designed to inter-operate smoothly and efficiently with MPI [43], OpenMP [15], CUDA [29] and other HPC frameworks. It leverages GASNet-EX to deliver low-overhead, fine-grained communication, including Remote Memory Access (RMA) and Remote Procedure Call (RPC). UPC++ exposes a PGAS memory model, including one-sided communication (RMA and RPC). However, there are departures from the approaches taken by some predecessors such as UPC. These changes reflect a design philosophy that encourages the UPC++ developer to directly express what can be implemented efficiently.

UPC++ is designed for writing efficient, scalable parallel programs on distributed-memory parallel computers. The PGAS model is single program, multiple-data (SPMD) in which each separate thread of execution (referred to as a process) has access to private memory as well as a global address space. This global address space is accessible to all processes and is allocated in shared segments that are distributed over the processes. UPC++ provides various efficient methods for accessing and using the global memory. In UPC++, all accesses made to remote memory are explicit, through a special set of methods and there is no implicit communication. This encourages developers to be aware of the cost of data movement, which may lead to expensive communication. Moreover, all remote-memory access operations are asynchronous by default. Together, these two constraints are intended to en-

able developers to develop code that performs well at scale. The value points about UPC++ can be summarized as follows:

- All communication is explicit - there is no implicit data motion.
- Most operations are non-blocking, and the powerful synchronization mechanisms encourage applications to design for aggressive asynchrony.
- UPC++ encourages the use of scalable data-structures and avoids non-scalable library features.

Features of UPC++ are discussed next:

- **RMA:** UPC++ provides asynchronous one-sided communication (Remote Memory Access, such as `Put` and `Get`). for movement of data among processes.
- **RPC:** UPC++ provides asynchronous Remote Procedure Call for running code (including C++ lambdas) on other processes.
- **Futures, promises and continuations:** Futures are crucial in order to handle asynchronous operation of RMA and RPC. UPC++ uses a continuation-based model to express task dependencies.
- **Global pointers and memory kinds:** UPC++ provides uniform interfaces for RMA transfers among host and device memories, including acceleration of GPU memory transfers via RDMA offload on compatible hardware.
- **Remote atomics** use an abstraction that enables efficient offload where hardware support is available.
- **Distributed objects:** UPC++ enables construction of a scalable distributed object from any C++ object type, with one instance on each rank of a team. RPC can be used to access remote instances.

- **Serialization:** UPC++ has introduced several complementary mechanisms for efficiently passing large and complicated data arguments to RPCs.

- **Non-contiguous RMA:** UPC++ provides functions for non-contiguous RMA data transfers to and from arrays in shared memory to efficiently copy or transpose sections of N-dimension dense arrays.

- **Teams:** Teams in UPC++ represent ordered sets of processes and play a role in collective communication. At present, UPC++ supports barrier, broadcast and reductions, including abstractions to enable offload of reductions supported in hardware.

- **Progress guarantees:** Since UPC++ has no internal service threads, the library makes progress only when a core enters an active UPC++ call. However, the concept of personas makes writing progress threads simple.

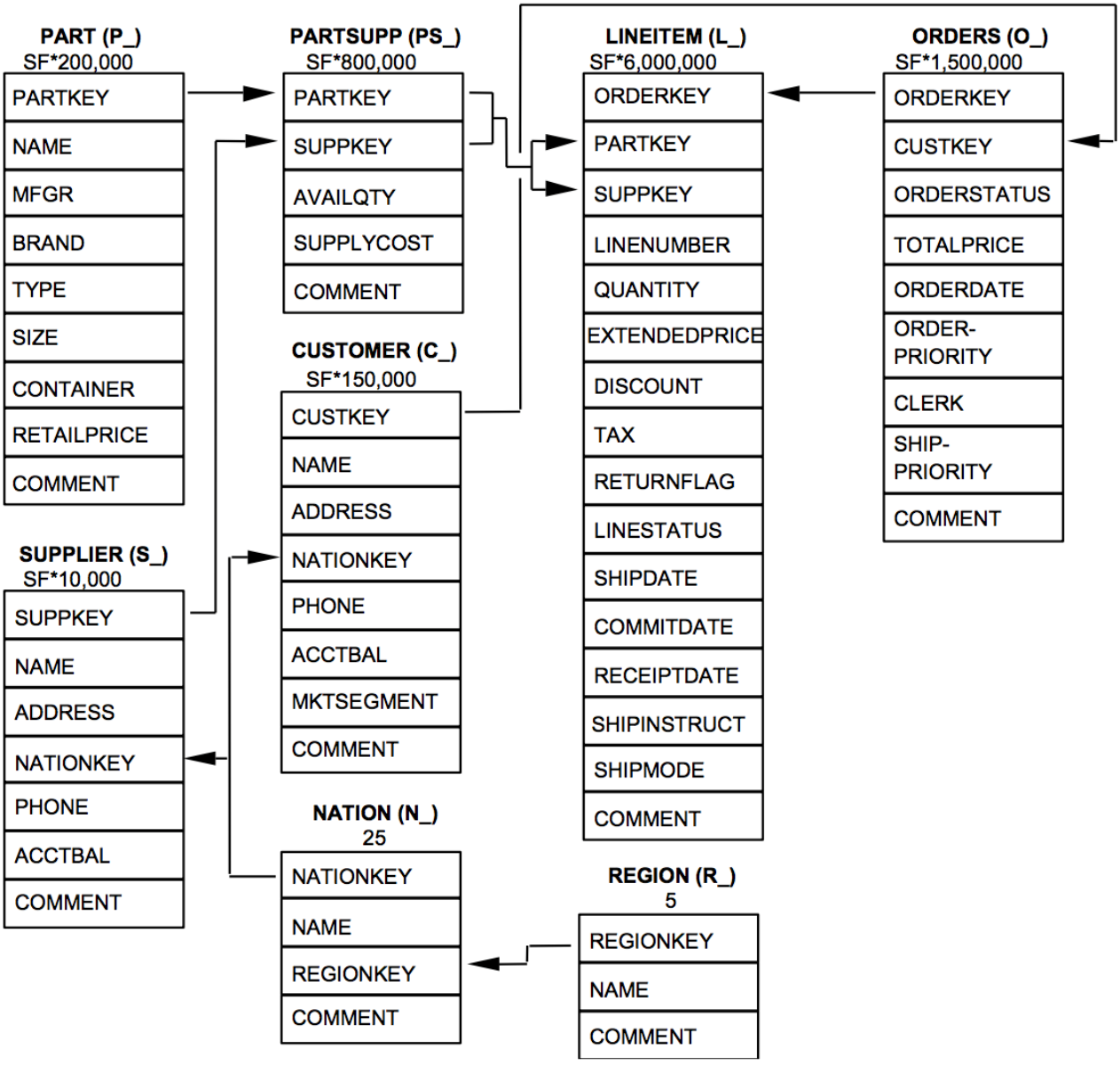
2.1.5.3 Comparison between UPC and UPC++

As part of this thesis, a second backend has been developed for CasaDB (discussed in Section 3.1). Since CasaDB was already generating C++ code, UPC++ seemed to be a natural choice for a PGAS framework for better integration with the existing C++ backend. As mentioned earlier, UPC and UPC++ are both implementations of the PGAS programming model. UPC stands for Unified Parallel C and UPC++ stands for Unified Parallel C++. In both the scenarios, the data is either shared (and distributed) or private. Shared data is accessible by all the PGAS ranks and private data is only accessible by its owner rank/process. UPC is based on the idea of communication abstraction and simplification. In UPC, there is no way for the developer to distinguish between intra-node memory operations (between processes running on the same hardware node) and the inter-node processes. This ensures an efficient and simple way of creating programs in UPC at the possible expense

of performance. By default, UPC++ also follows the same design pattern. With the introduction of teams of threads in UPC++, it offers a native way to distinguish between intra- and inter-node communication. Conceptually, the main difference between UPC and UPC++ is that UPC++ implements a specialized version of PGAS, which is called APGAS, Asynchronous Partitioned Global Address Space [39, 42]. The APGAS model extends the PGAS with ideas from task-based asynchronous execution model, which describes the semantics of an application as a hierarchy of tasks that are dynamically created and scheduled during the execution time. Furthermore, unlike UPC, UPC++ is distributed as a library rather than a language, which leads to better software maintenance and scalability.

2.2 TPC-H Benchmark Queries

The TPC-H is a decision support benchmark [4, 13, 36]. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size), and reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power when queries are submitted by a single stream, and the query throughput when queries are submitted by multiple concurrent users.



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate

Figure 2.3: TPC-H Schema [4]

2.2.1 Schema and Scale Factors

The components of TPC-H consist of eight separate and individual tables (the Base Tables). The relationships between columns in these tables are illustrated in Figure 2.3, taken from the TPC-H website [4]. TPC-H comes with various data set sizes to test different scaling factors.

2.2.2 Queries

The TPC-H queries used in thesis are listed and described next.

- **Shipping Priority Query (Q3):** 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 non-shipped orders exist, only the 10 orders with the largest revenue are listed. However, since our query planner does not support "order-by" and "limit" operators, we have omitted these two from our version of Q3 as shown in Figure 3.7.

- **Order Priority Checking Query (Q4):** The Order Priority Checking Query, shown in Figure 2.5 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. We have not sorted our result set as we do not support "order-by" yet.

- **Local Supplier Volume Query (Q5:)** This query lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. This query has not been used in our benchmarking workload but

```
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
```

Figure 2.4: TPC-H Query 3

```
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
```

Figure 2.5: TPC-H Query 4

serves as a good example for schema graph generation example in Section 3.2.3 due to the large number of joins that it has as shown in Figure 2.6.

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_orderdate >= date '1994-01-01'
        and o_orderdate < date '1995-01-01'
group by
        n_name
```

Figure 2.6: TPC-H Query 5

- **Forecasting Revenue Change Query (Q6):** This query, shown in, Figure 2.7, 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. Asking this type of "what if" query can be used to look for ways to increase revenues. The Forecasting Revenue Change Query 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. Note that the potential revenue increase is equal to the sum of [*l_extendedprice*$*$*l_discount*] for all lineitems with discounts and quantities in the qualifying range.

- **Large Volume Customer Query (Q18):** The Large Volume Customer

```
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
```

Figure 2.7: TPC-H Query 6

Query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level. The query is depicted in Figure 2.8

- **Discounted Revenue Query (Q19):** The Discounted Revenue Query 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. The query is shown in Figure 2.9.

2.3 Related Work

As discussed in the previous sections, the pull-based iterator model of query processing is not very efficient when dealing with very large amounts of data, due to the large number of virtual function calls that are made by it. A better approach for query processing is to eliminate the overheads of the iterator model by compiling SQL queries into specialized programs. This approach is used in HIQUE [26] and HyPer [3, 21]. LegoBase [22] uses an incremental query compilation framework and provides significant performance improvement over previous systems. However, these systems do not run on a distributed cluster and are targeted for standalone query evaluation. Cloudera Impala [24] generates code to be run on a distributed cluster.

```
select
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice,
        sum(l_quantity)
from
        customer,
        orders,
        lineitem
where
        o_orderkey in (
                select
                        l_orderkey
                from
                        lineitem
                group by
                        l_orderkey having
                                sum(l_quantity) > 300
        )
        and c_custkey = o_custkey
        and o_orderkey = l_orderkey
group by
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice
```

Figure 2.8: TPC-H Query 18

```
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'
        )
```

Figure 2.9: TPC-H Query 19

However, Impala uses an adaptive query evaluation model implementing both interpretation and compilation and therefore it is difficult to measure the performance of a purely compilation-based query evaluation model.

The work that is most similar to ours is Radish [33], developed at the University of Washington. Radish is an end-to-end implementation of a query processing engine, which integrates a query optimizer (RACO), shared distributed data structures and a PGAS C++ library called Grappa [34]. However, Radish does not have a data partitioning module and the data is distributed across the cores in a random fashion. Also, Grappa does not support asynchronous operations and therefore it is difficult to customize the behavior of PGAS processes. In this thesis, we have developed a

distributed CasaDB backend to overcome these limitations of Radish.

2.4 Summary

In this chapter **RQ1** and **RQ2** have been pursued by surveying the several query processing models like demand-driven iterator model and push-based model. The PGAS framework was also explored and two popular PGAS implementations, UPC and UPC++, were discussed. Previous similar systems like HIQUE and Radish have also been referenced in this chapter.

Chapter 3

Design and Implementation

This chapter discusses the design and implementation of the complete system developed in this thesis. A detailed description of code generation algorithm and partitioning module is presented. The architecture of CasaDB and its distributed backend is also documented. An end-to-end description of the query evaluation process is depicted through appropriate block diagrams.

3.1 CasaDB

The system developed as part of this research work leverages another component being developed in parallel—**CasaDB**. CasaDB is being developed as a database system and is named after our research centre - **Centre for Advanced Studies–Atlantic**, Canada (CASA). At present, CasaDB uses RACO (**R**elational **A**lgebra **CO**mpiler) as the DB engine front-end to parse, validate and plan SQL queries. The query plan emitted by the query planner is fed into a Java-based code generator developed on the principles of the compilation strategy proposed by Thomas Neumann [35]. Work is underway to partly replace the front-end with our own components.

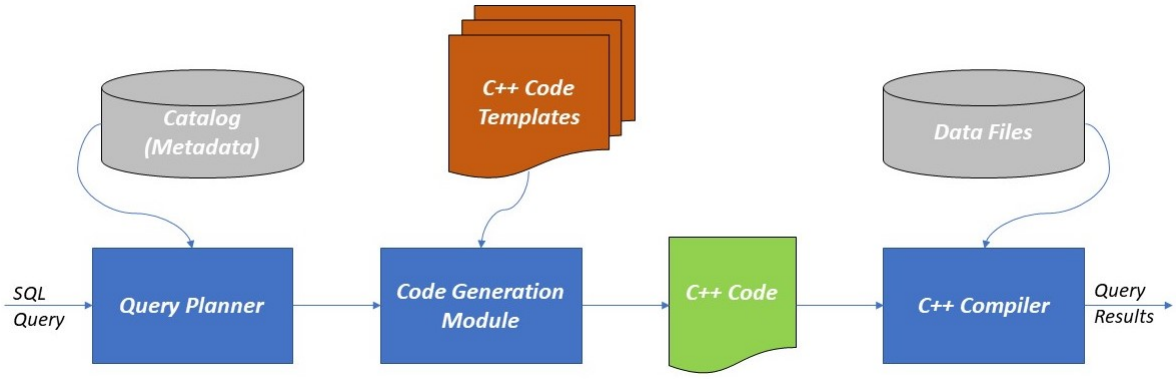


Figure 3.1: CasaDB - C++: Design

3.1.1 Components

An overall architecture of CasaDB is shown in Figure 3.1. CasaDB is made up of the following components:

- **Query Planner:** The query planner used in this thesis is a part of Myria, the big data management stack developed by the database group at the University of Washington in collaboration with UW eScience Institute [37]. The Relational Algebra Compiler (RACO) is Myria's query optimizer and federated query executor. RACO has adopted an extended relational algebra as the core model of query processing and computation, but also supports compilation of relational algebra expressions into computations over certain array, graph, and key value engines. The reasoning behind such a design is that while performance characteristics vary widely across database systems, relational algebra is more than enough to capture the semantics of their query interfaces. RACO rewrites rules to transform relational algebra expressions into the specific API calls, operators, or query primitives supported by the selected backend system, and use rule-based optimization to generate a federated query plan that takes advantage of the specialized features of multiple backend systems. This is the reason why we chose RACO for our initial experiments so that we can first use it for a C++-based backend followed by PGAS and Eclipse OMR

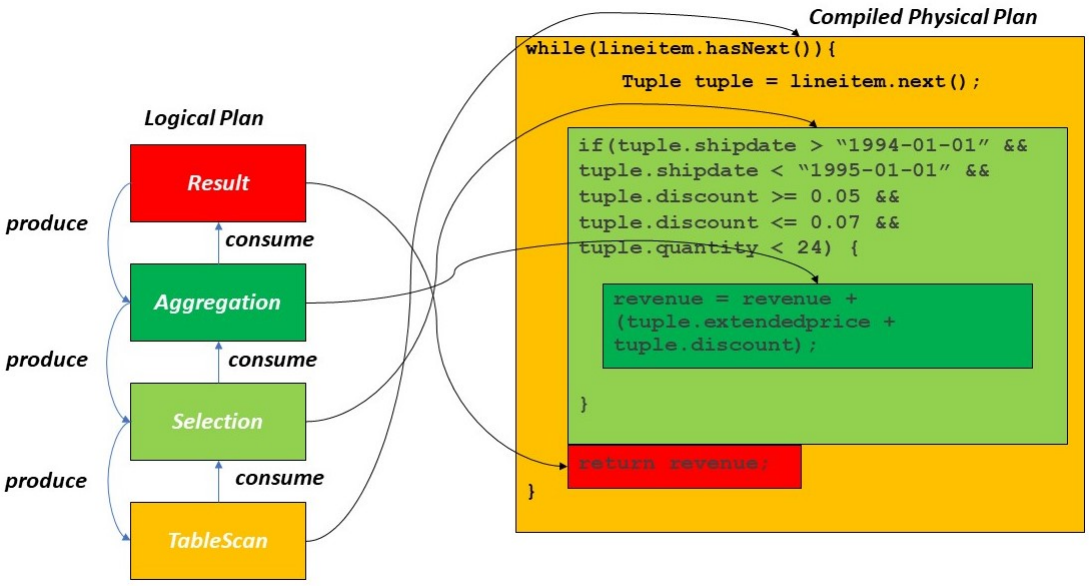


Figure 3.2: Code Generation Algorithm for TPC-H Query 6

JitBuiler-based backends [2].

- **Templates Library:** An important part of the C++ code generation algorithm is to use C++ templates to support the implementation of various database operators and functionalities. C++ code templates also helped us to extend the code generation algorithm in CasaDB to generate distributed evaluation plans that are UPC++-based.

- **Code Generator:** CasaDB's code generation module uses the compilation strategy proposed by Thomas Neumann [35]. In this strategy, the developers tend to build programs that can generate specialized code corresponding to each query. However this is complicated as the complex operator logic directly affects the code generation. In the iterator model, there is a simple interface for query modelling and execution. However, too many virtual function calls and frequent memory accesses make the iterator model inefficient and slow. Using this compilation strategy, close-to-optimal C++ code, which is specific to a particular query, can be generated and all the relevant data can be kept in

CPU registers. In this model, all operators offer a uniform interface of *produce()* and *consume()* methods, but they exist only in the query compiler itself and not in the actual generated code. From query compilation's perspective, the operators offer an interface that is as simple and intuitive as the iterator model. On a high level, each operator has two functions:

- **produce()**
- **consume(attributes, source)**

The *produce()* method asks the operator to produce its result rows. The result rows are then pushed towards the consumer operators by calling their respective *consume()* functions. In the example depicted in Figure 3.2 for TPC-H Query 6, the code generation main module calls the produce function for the "Resul" operator. Since, "Result" does not have any rows in the beginning, it in turn calls the produce() method on "Aggregation" followed by "Selection" and "TableScan". Since, "TableScan" is a terminal operator and can actually consume tuples from the storage layer, it ingests the tuples and calls consume() on "Selection". This process again continues until the final code generation module that had asked for tuples in the beginning receives them.

However, it is important to understand that this produce/consume interface is only a conceptual model. These functions do not exist explicitly and are only used by the code generation algorithm. When compiling an SQL query, the query is first processed as usual. The query is parsed, translated into algebra, and the algebraic expression is optimized. Only after this step, the compilation strategy deviates from the standard methodology. The final logical plan is not translated into a physical algebra plan that can be executed, but instead compiled/transpiled into a C++ program. This compilation step is where the

produce/consume interface is used internally to produce the required C++ code as depicted in Figure 3.2.

3.1.2 Design

CasaDB takes a SQL query as input and emits the corresponding C++ program. It uses a catalog containing the schema metadata and a templates library to aid logical plan generation and C++ code generation respectively. The C++ code generated by the CasaDB backend is then passed through a C++ compiler to produce query results on a single node. The design and code and data flow through CasaDB is illustrated in Figure 3.1. RACO acts as the SQL front-end and ingests an incoming SQL query, parses and validates it and generates a logical plan. It uses the metadata stored in a catalog file to interpret the schema of the data set. The logical plan emitted by RACO is pipelined into the code generation module which uses the producer/consumer compilation strategy to generate a C++ program corresponding to the input SQL query. This C++ program acts as an evaluation plan/high-level intermediate representation (IR) and can be run through a compiler to obtain the query output. The code generation module also uses a set of C++ code templates for various database operators.

3.2 PGAS/UPC++ Backend

In order to augment CasaDB with the ability to run queries on a distributed cluster to take advantage of the advances in modern hardware, many experiments were conducted as part of this research work. The PGAS backend of CasaDB generates UPC++ programs corresponding to SQL queries. Figure 3.3 illustrates the design and architecture of CasaDB with UPC++ backend.

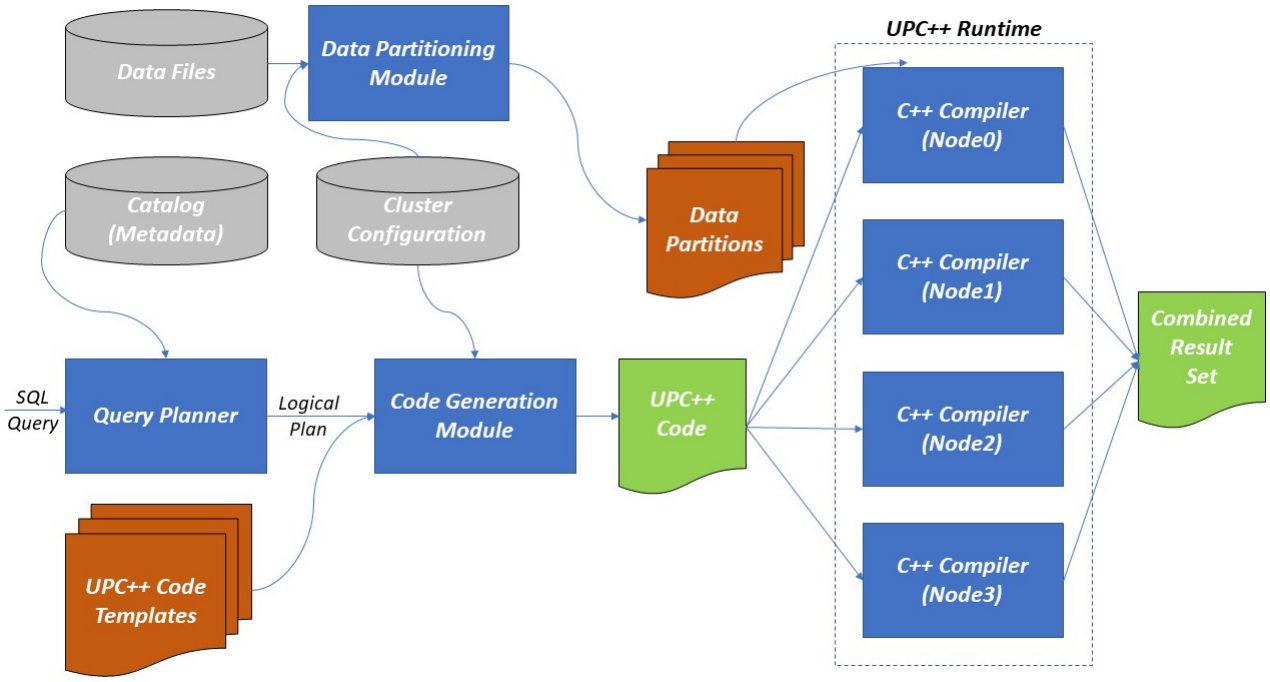


Figure 3.3: CasaDB - UPC++: Design

3.2.1 Query Plan Generation

RACO is used as the query engine front-end. The functionality of RACO is same as seen in CasaDB's C++ backend. Even though RACO generates query plans for single nodes, it can be used to generate distributed code due to the SPMD nature of the PGAS framework. This is one of the several advantages offered by PGAS-based distributed systems. Not many changes are requires in terms of infrastructure needed when up-scaling a single-node query compiler to a multi-node query execution engine. RACO uses the catalog file to generate a logical plan corresponding to the input SQL query.

3.2.2 Code Generation

Algorithm 1: Pseudo-Code Algorithm for Code Generation
Input: Query Plan
Output: UPC++ Code for SQL Query

```
1  initialize CodeGen module;
2  while traversal of Query Plan is not complete do
3      call produce() on the current operator;
4      if current operator has incoming tuples then
5          generate code to read the incoming tuples;
6          generate code to process and filter the incoming tuples;
7          call consume() on the previous operator to push the tuples above;
8          set previous operator as the current operator;
9      else
10         call produce() on the next operator;
11         set next operator as the current operator;
12     end
13 end
```

The code generation module accepts the logical plan generated by the query planner as the input and iterates over it to generate UPC++ code corresponding to it. The code generation algorithm is summarized in Algorithm 1. The complete code generation process from generating a logical plan to the emitted UPC++ code for Query-6 is depicted in Figure 3.4. The code generation module uses the information about the PGAS cluster on which the query has to be run. This is important in cases where there are distributed remote joins taking place and the results from different nodes have to be combined together to produce one consolidated output, as would be expected for a standalone/single-node query compilation system. Since UPC++ uses the SPMD model of execution, the same program runs in parallel on all the nodes and there can be instances when results or rows from different nodes need to

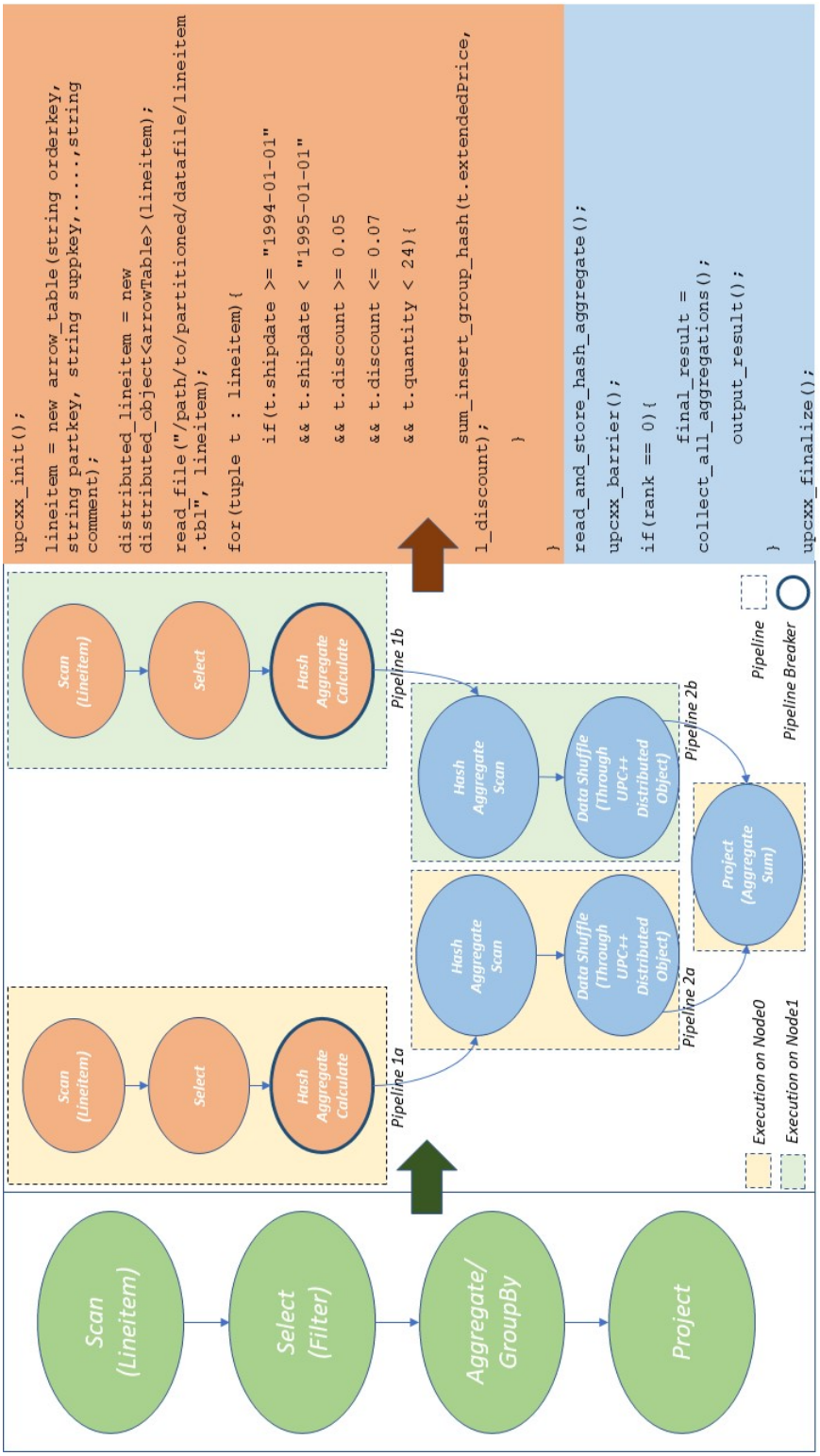


Figure 3.4: Translation from Query Plan to UPC++ Program for Query-6

be shuffled in order to carry out the query execution process in an accurate fashion. This thesis tries to address the problem of shuffling to some extent by developing a partitioning module to partition data according to the PREF partitioning scheme [44]. Section 3.2.3 talks about the partitioning module in greater detail.

As described in section 3.1.1, each physical operator in the code generator has a two-faceted iterator interface (implemented by *produce()* and *consume()* methods). These operators generate code that pushes rows of data through the generated program. Code generated for an operator that consumes tuples is nested inside the code generated for a producer operator. The process of code generation begins at the operator that acts as a sink for all the incoming data. In Figure 3.2, *Result* is such an operator. The entire query plan is traversed from top to bottom by calling *produce()* methods corresponding to each successive operator. Once the actual source of data (*TableScan* operator in Figure 3.2) is reached, the code generation begins by calling *consume()* on each of the preceding operators in the hierarchy. This process is illustrated in Figure 3.2.

As shown in Figure 3.4, the physical plan is broken into several pipelines owing to the nature of the *produce()* and *consume()* methods. The operators can be broadly categorized into two groups:

- Pipelined Operators
- Pipeline Breakers

For pipeline breakers (**HashAggregateCalculate** in Figure 3.4), that cause the data to be moved from registers into memory, *produce()* generates code to iterate over entire tables. However, for pipelined operators (**Scan** etc.), *consume()* generates code for processing and sending the incoming row to the next operator in pipeline. Therefore, this code generation model is modular in nature and allows generation of simple code to consume and produce incoming tuples.

3.2.3 Data Partitioning

Algorithm 2: Pseudo-Code Algorithm for Data Partitioning
Input: Data Tables, Queries, Information about Joins, Join Keys, No. of Nodes in Cluster
Output: Partitioned Data

```
1  construct schema graphs for queries in workload;
2  combine schema graphs for all queries into one connected component;
3  generate a maximum spanning tree for the connected component above;
4  set base_table = lineitem;
5  set current_table = base_table;
6  while no table is left to be partitioned do
7      if current table is base_table then
8          range_partition(base_table) on order_key;
9      else
10         partition current table on its join key with its reference table;
11     end
12     set partitioned table as reference table for all tables connected to it;
13     set the next connected table as current table;
14 end
```

As part of this research work, the data is partitioned in a way that data shuffling is minimized while query execution takes place to make sure that network costs are reduced. The partitioning algorithm is implemented here for a given workload and a particular database schema, which is based on the TPC-H benchmark. The workload comprises of queries 3, 4, 6, 18 and 19 for scale factor 1 and 2 from the TPC-H benchmark suite. TPC-H queries have been discussed in greater detail in Section 2.2.2 from Chapter 2. Algorithm 2 summarizes the data partitioning algorithm, which is described in greater details in the following text:

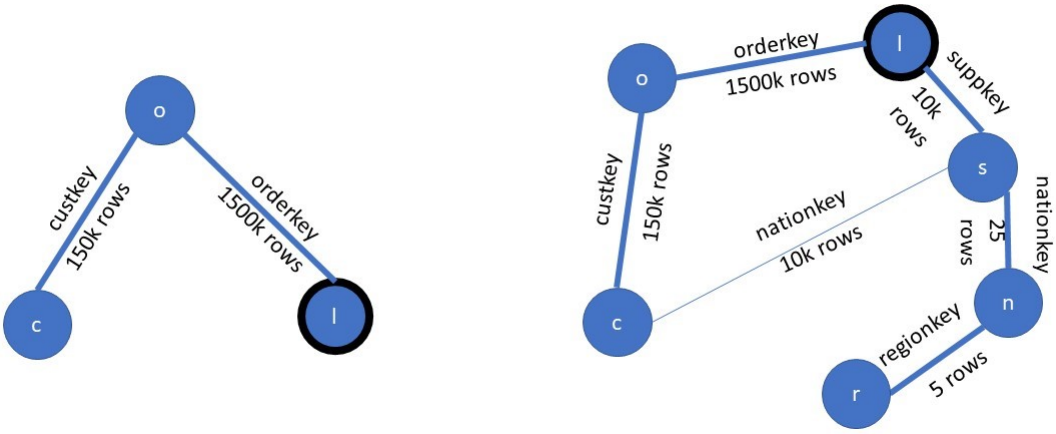


Figure 3.5: Maximum Spanning Trees for TPC-H Query 3 and 5

1. **Schema Graph Generation:** A graph $G = (V, E)$ is constructed out of each query that is a part of the workload being considered. V is the set of all the vertices (tables) present in the workload and E represents a join between two tables $v_m \in$ V and $v_n \in$ V. This graph is constructed such that the tables used in the query represent the vertices of the graph and the join keys between each pair of tables (vertices) represent an edge between those two tables. The maximum (worst) number of rows that would be exchanged over the network if there were remote join between these two tables, would represent the cost of each edge in the graph. Two such graphs for queries 3 and 5 have been illustrated in Figure 3.7 and Figure 3.6 respectively.

2. **Maximum Spanning Tree Generation:** Once all the graphs have been constructed, a maximum spanning tree is constructed. This is to make sure that edges with the maximum weights (or remote joins with maximum data shuffling) are all included in the resultant spanning tree. This maximum spanning tree will represent all the connected tables that can be PREF (predicate + reference) partitioned [39] (partitioning on the basis of their join keys, as discussed in the point 4). Maximum spanning trees corresponding to query 3 and 5 are illustrated in Figure 3.5. The darkened edges represent the edges

```
...
from
        customer as c,
        orders as o,
        lineitem as l,
        supplier as s,
        nation as n,
        region as r
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        ...
```

G = (V, E)
V = {c, o, l, s, n, r}
E = {c-o, l-o, l-s, c-s, s-n, n-r}

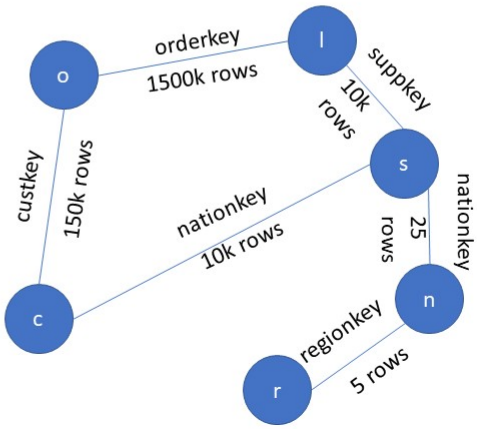


Figure 3.6: Graph for TPC-H Query 5

```
...
from
        customer as c,
        orders as o,
        lineitem as l
where
        ...
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        ...
```

G = (V, E)
V = {c, o, l}
E = {c-o, l-o}

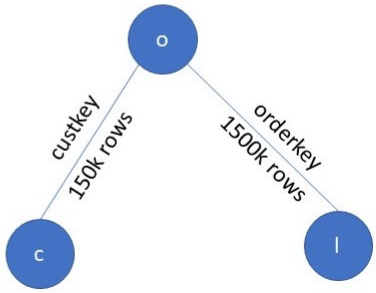


Figure 3.7: Graph for TPC-H Query 3

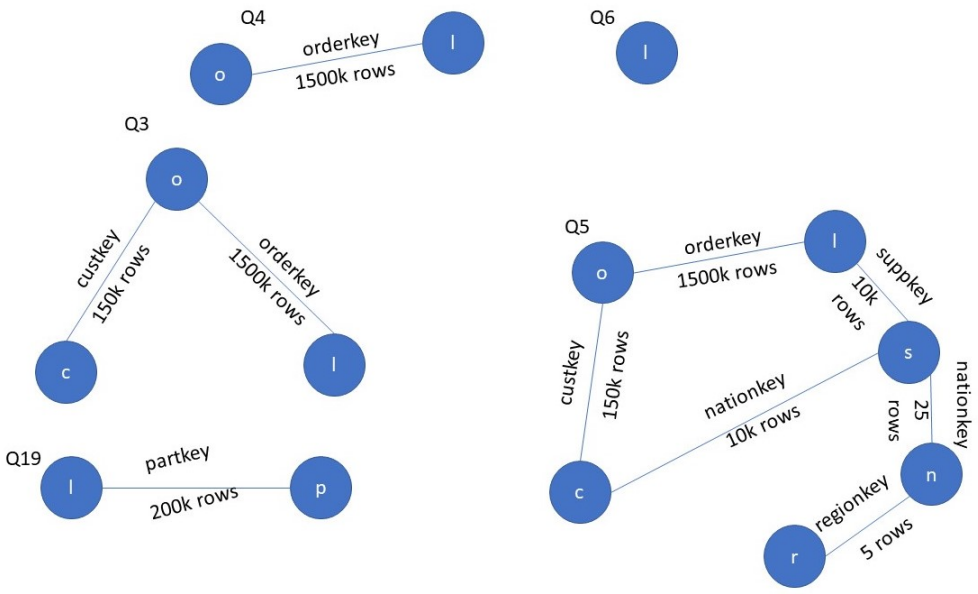


Figure 3.8: Query Schema Graphs

of the spanning tree, while the lighter edges are the ones which have not been included in the tree. The circles with black outlines represent the table used as reference for partitioning (lineitem, in this case).

3. Combining Graphs: Graphs representing several different queries can be merged together into a single connected component to ultimately construct a maximum spanning tree corresponding to all the queries in a specific workload. This is illustrated in a step-wise fashion in figures 3.8, 3.9, 3.10 and 3.11. Figure 3.9 shows the final maximum spanning tree for a workload consisting of queries 3, 4, 5, 6 and 19 from the TPC-H benchmark suite (though this partitioning configuration can fit even a more queries, like query 1 and 18 etc.).

4. **PREF Partitioning:** One of the tables in schema is chosen as the *reference* table and is range/hash-partitioned (range-partitioned here). The reference table is usually the largest table to make sure that less number of rows move in the later stages of partitioning (lineitem in our case). Once the reference table has been range-partitioned, other tables that share a join key with the reference

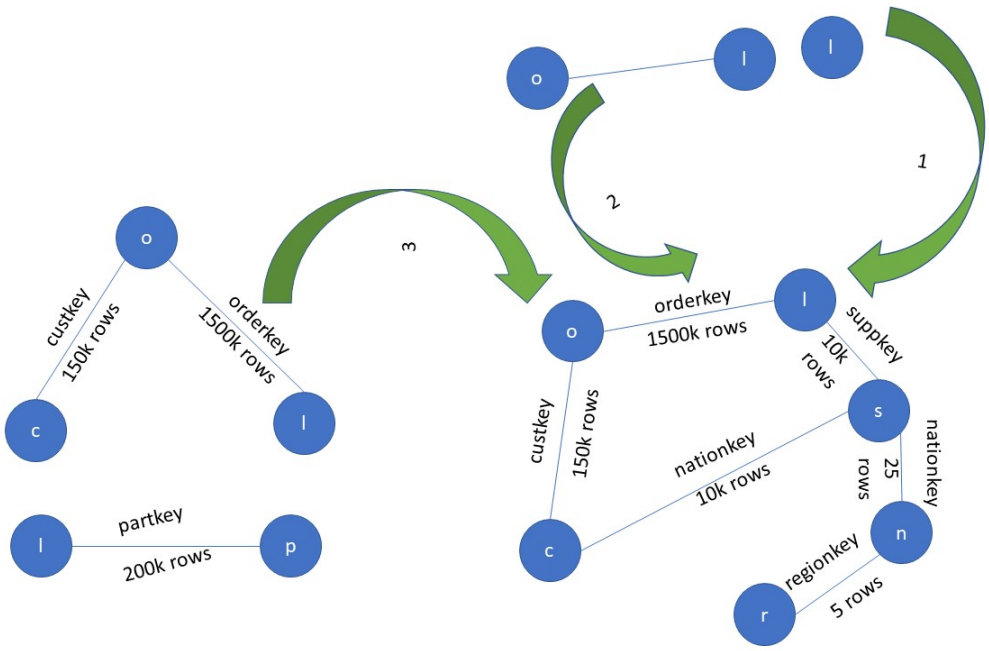


Figure 3.9: Merging Several Graphs to Create 1 Connected Component

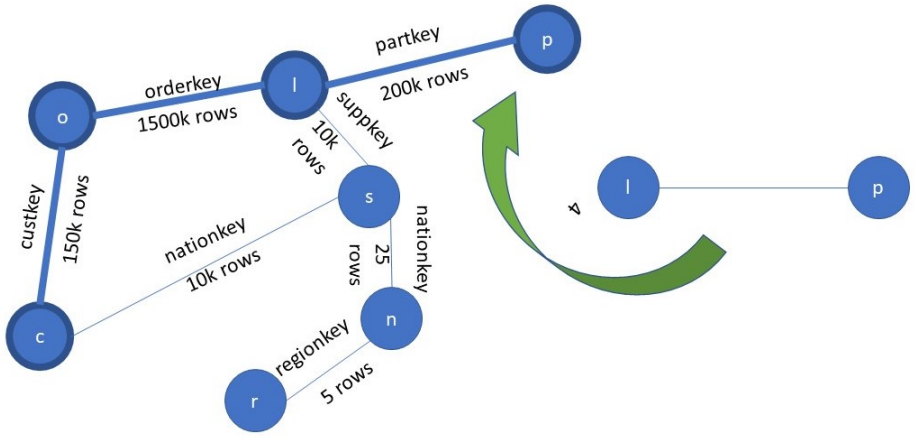


Figure 3.10: Merging Several Graphs to Create 1 Connected Component

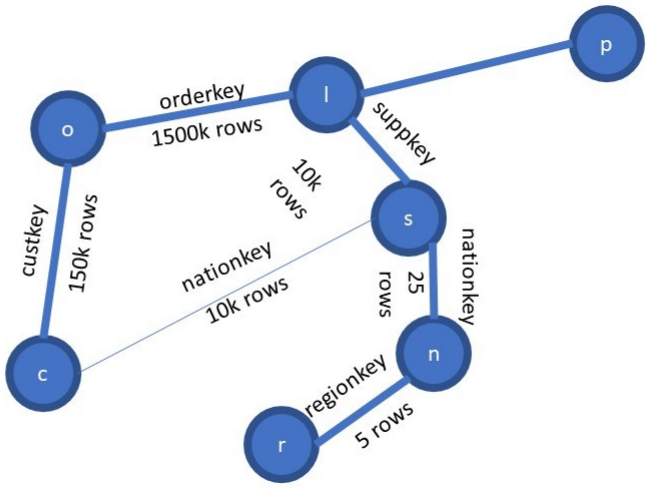


Figure 3.11: Final Maximum Spanning Tree for the Given Workload

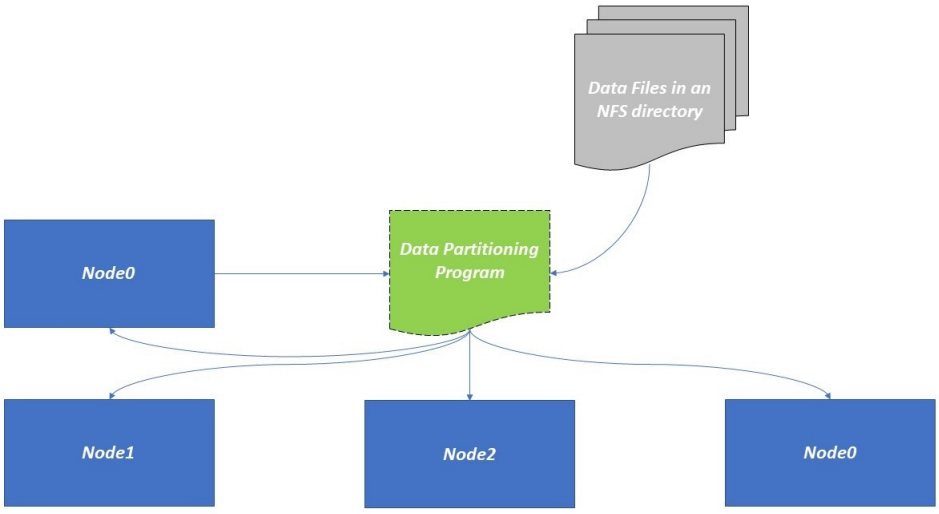


Figure 3.12: Data Partitioning across Cluster Nodes

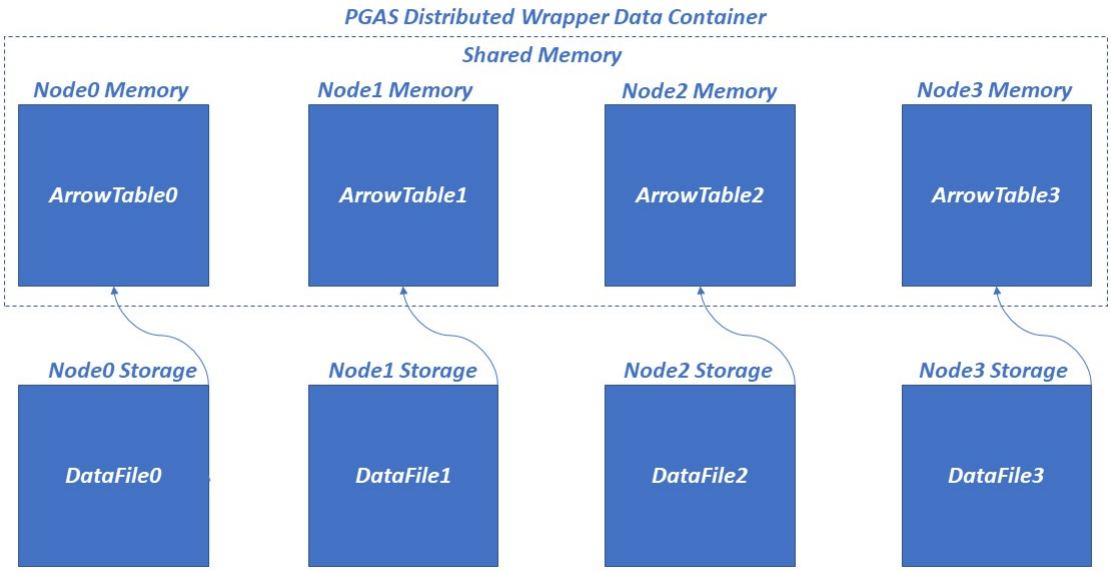


Figure 3.13: Loading Partitioned Data into Arrow Tables on Each of the Nodes in Cluster

table can be partitioned based on that key. So for example, in Figure 3.11, lineitem is chosen as the reference table since it has a large number of rows (6M at scale factor 1) and is partitioned among the nodes of the distributed cluster being used. Once lineitem has been partitioned, the dependent tables orders, supplier and part can be PREF partitioned on the basis of lineitem. This in turn will lead to PREF partitioning of customer (on the basis of orders) and nation (on the basis of supplier). A third wave of PREF partitioning will cause region to be partitioned on the basis of nation. An important point to note here is that since the edge c–s is not a part of the maximum spanning tree, these two tables cannot be partitioned using their join key, so a join between these two tables will be a remote join and will incur network costs. A block diagram illustrating the partitioning process over a cluster of 4 nodes is shown in Figure 3.12.

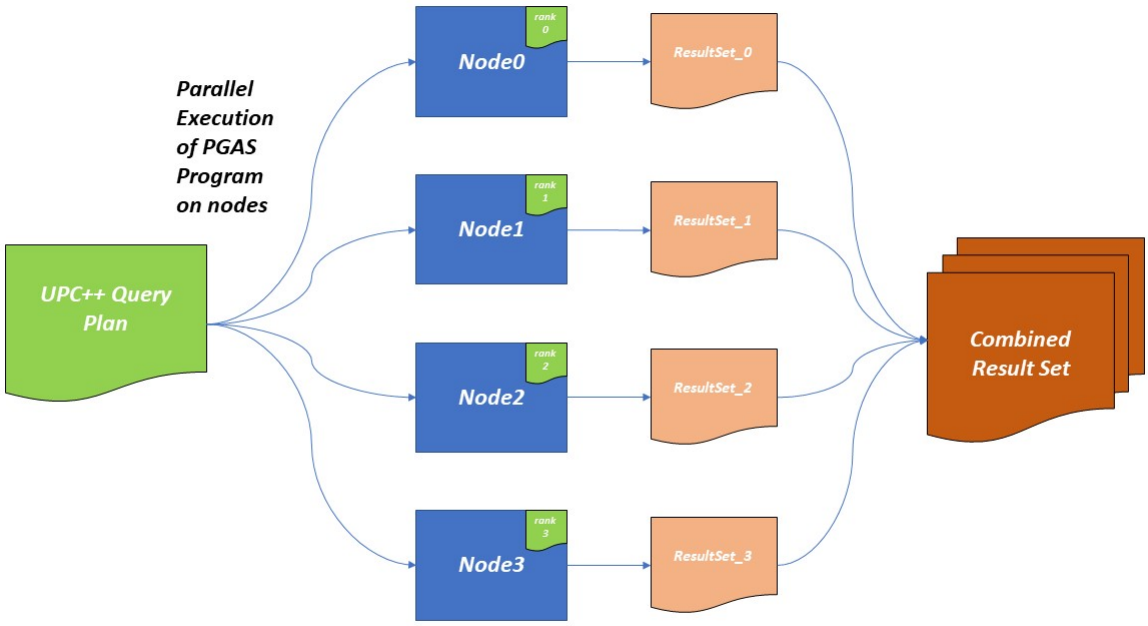


Figure 3.14: Distributed Query Execution

3.2.4 Query Execution:

The C++/UPC++ program/evaluation plan generated by the code generation module is compiled into an executable and the resultant executable is run on each of the nodes of the UPC++ cluster. During its execution, the UPC++ program on each node reads data files from their partitions sitting on that particular node, processes the data according to the logic of the generated code and emits its part of the results into a distributed object in the partitioned global address space. One of the nodes collect results from all the other nodes and emits a consolidated result set. This is illustrated in figure 3.14. For the purpose of data loading, the Apache Arrow C++ library has been used in this project. Apache Arrow is an in-memory columnar data storage library that is used by several modern database and big data systems. Each node loads its portion of data into a local Arrow table, which is wrapped inside a global distributed object shared across all the nodes due to a global address space. A global address space ensures that remote joins can take place in a seamless manner

without calculating the location of a remote tuple on a different node. Figure 3.13 shows how each node loads data from its partitions into Arrow tables for query execution.

3.3 Summary

In this chapter, **RQ3** is pursued and the design and implementation of the PGAS/UPC++-based backend is discussed. As a prerequisite, the architecture and compilation strategy used in CasaDB is also discussed. There is a discussion around the code generation and partitioning module too supported with pseudo-codes and block diagrams.

Chapter 4

Evaluation

The performance of the system that has been designed and implemented in this thesis is benchmarked using the TPC-H benchmark suite of queries and its results and analysis have been recorded in this chapter. The experiments in this chapter are designed to evaluate several aspects of our query compilation engine, ranging from the impact of scale factor, cluster size and data partitioning on the query execution times to the overall analysis of results.

4.1 Experimental Setup

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

4.1.1 Benchmark

The benchmark queries used in the experiments in this thesis are derived from the TPC-H benchmark suite with slight modifications. These queries and their respective modifications are listed below:

- **Query-3:** As depicted in Figure 2.4 from Chapter 2, the operations *Order By*

and *Limit* have been omitted from query-3, as CasaDB does not support these operators yet. This is in-line with the baseline system used for performance comparison (Radish), as Radish also does not support these operators in its implementation. This query has an aggregation, two joins, four predicates and three group by's.

- **Query-4:** *Order By* has also been omitted from query-4 and the query depicted in Figure 2.5 is used in our experiments. This query has an aggregate, three predicates (one of which has a correlated sub-query nested inside it) and one group by.

- **Query-6:** Query-6 has been kept intact and is used as it is. The SQL code for query-6 is shown in Figure 2.7. This query has an aggregate and four predicates. It does not have any joins.

- **Query-18:** Similar to query-3 above, the operators *Order By* and *Limit* query-18 have been omitted for the purpose of the experiments performed in this thesis. As shown in Figure 2.8, this query has an aggregate, three joins (one of which is due to the correlated sub-query in the *where* clause) and five group by's.

- **Query-19:** Query-19 is used in its original form and has a single join and 21 predicates.

The aforementioned specifications of the query operation group that each query represents is listed in Table 4.1. We have only used these queries as they are the best candidate queries to represent all the operations supported by our system presently. The queries that have not been used in this thesis either have a substantial operator overlap with these five queries or have operators that are not supported by our system.

Aggregates	3, 4, 6, 18, 19
Predicates	3, 4, 6, 18, 19
Joins	3, 4, 18, 19
Group By	3, 4, 18
Correlated Sub-Queries	4, 18

Table 4.1: Query Operations and the corresponding TPC-H Queries

Node CPU	Four-Core AMD Opetron
Node Memory	16 GB
UPC++	version 2021.3.0
g++ Compiler	version 7.5.0
JDK	Oracle JDK 8

Table 4.2: Hardware and Software Specifications

4.1.2 Hardware and Software

The evaluation experiments were conducted on a cluster of four nodes, each with a four-core AMD Opetron CPU, with an aggregate memory of 16 GB RAM. UPC++ version was 2021.3.0 and the UPC++ programs were compiled using the g++ compiler (version 7.5.0). The initial experiments were conducted on Ubuntu 16.04 virtual machines (not included) but the results included in thesis are from the experiments conducted on the aforementioned machines. These hardware and software specifications are summarized in Table 4.2.

4.2 Results

Each query was run 20 times to warm up the system and the average execution time of the last 10 query runs has been reported in this thesis. The experiments were run for the scale factors 1 and 2 for the TPC-H benchmark suite. The query execution times for UPC++ code and Radish-generated code is illustrated in Figures 4.1 and 4.2 for a 2-Node and 4-Node cluster respectively.

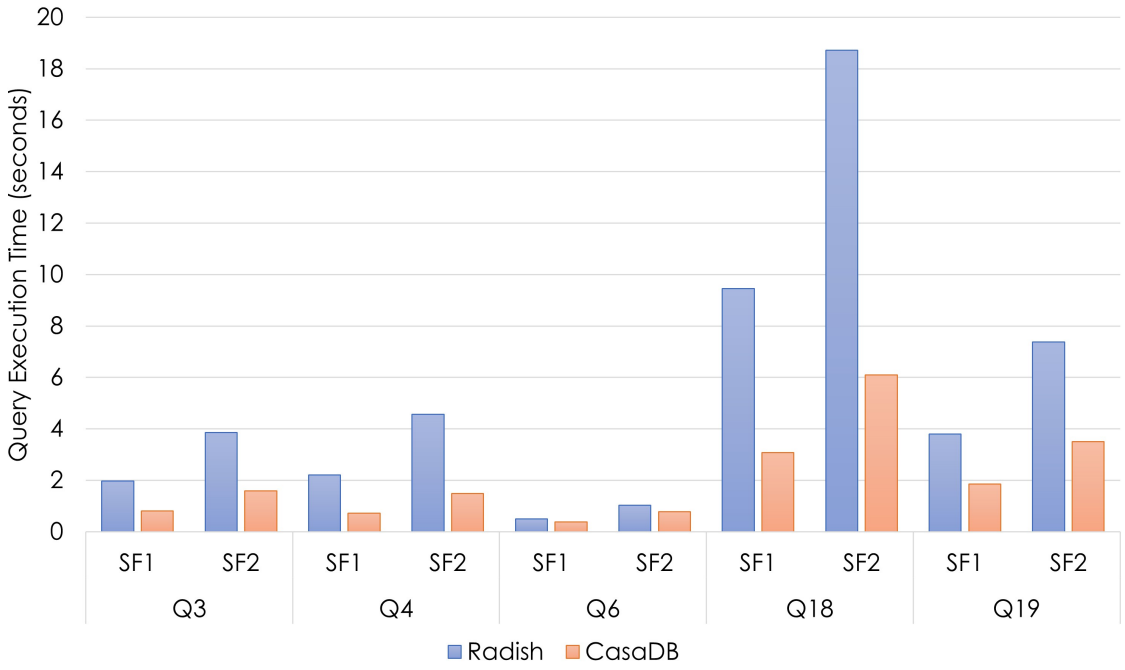


Figure 4.1: Performance Results: Radish vs CasaDB on a 2-Node Cluster

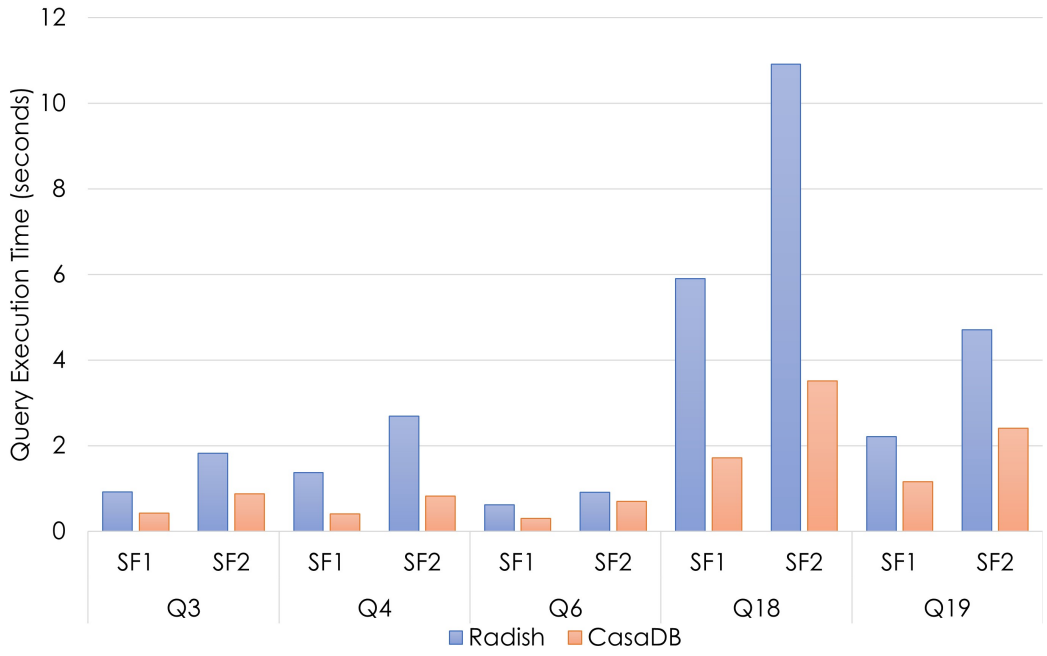


Figure 4.2: Performance Results: Radish vs CasaDB on a 4-Node Cluster

4.2.1 Impact of Scale Factors

The experiments are performed with scale factors 1 and 2. The scale factor of a data set denotes its size. For example, in TPC-H, scale factor 1 implies a data set of size one gigabyte. The execution times of queries may increase in the same proportion as do the scale factors themselves. In Figure 4.1, the query execution time on the 2-node cluster for all the five queries increases approximately by a factor of 2 on going from scale factor 1 to 2. A similar relationship between scale factors and query execution times is observed in Figure 4.2 for a cluster comprising of 4 nodes.

4.2.2 Impact of Cluster Size

The experiments are performed on clusters of variable sizes to understand the impact of horizontal scaling on the system. On comparing the figures corresponding to execution times in Figure 4.1 and Figure 4.2, it can be observed that the execution of queries on the 4-node cluster becomes twice as fast as the execution of queries on the 2-node cluster. This is due to the fact that with an increase in the number of compute nodes, the computation task is divided among a higher number of nodes working in parallel, leading to reduced overall execution times. However, it is important to note here that the relationship between the speedup achieved and the increased number of nodes will not always follow a linear speedup due to the significant amount of data shuffling that happens over communication networks as a result of the increase in the number of nodes in the distributed cluster. So the actual query execution speedup achieved on horizontal scaling is the net sum of positive impact of the division of computation tasks and negative impact of data shuffling over the communication network. In our experiments, the overall speedup achieved on increasing the size of the cluster is positive because of the PREF-partitioning algorithm employed to minimize data shuffling. The only instance of data shuffling in execution of all the five queries takes place when the aggregation results from each node are combined

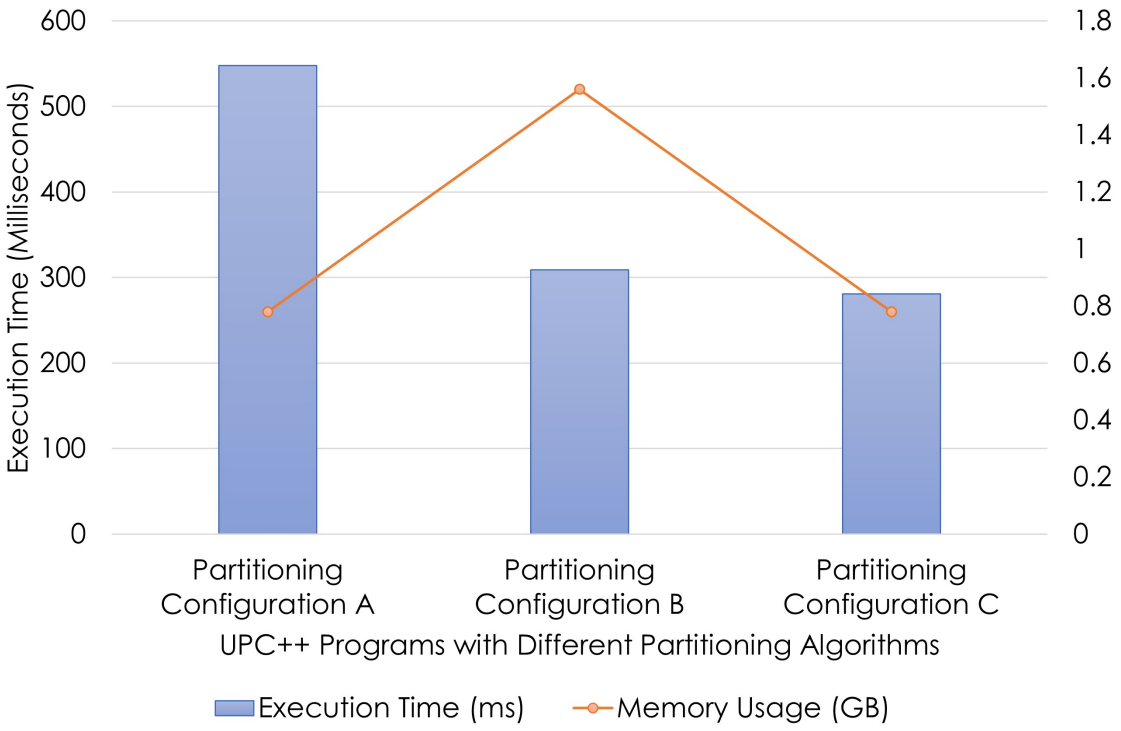


Figure 4.3: Relationship between Data Partitioning Algorithms, Query Execution Time and Memory Usage

to emit the final result. There is no intermittent transfer of rows. Therefore, in our case the impact of horizontal scaling follows an approximately linear relationship.

4.2.3 Impact of Data Partitioning

A small experiment is performed to compare the performance of a UPC++ implementation of TPC-H query 19 with and without PREF-partitioned data. The impact of data partitioning can be evaluated on the basis of two metrics:

- Data Locality
- Data Redundancy

When partitioning data, a good balance has to be maintained between these two factors to make sure that data shuffling is minimized, while replicating as less number of rows as possible. This experiment is performed with different data partitioning

configurations A, B and C which offer different degrees of data locality and data redundancy. This experiment illustrates the effectiveness of our PREF-partitioning implementation (represented by Partitioning Configuration C). The experimental setup is described as follows:

- **Partitioning Configuration A:** UPC++ programs with no defined partitioning protocol (the files are randomly cut into smaller slices that are scattered on different nodes). There will be no redundant data in this case, as none of the rows have more than one instance present in the cluster. However, since no particular partitioning algorithm is used, our UPC++ program will not be able to take advantage of data locality resulting in communication overhead due to data shuffling. This is depicted in Figure 4.3. For partitioning configuration A, even though the memory usage, and hence data redundancy, is minimal as data is loaded only once, the execution time is quite high when compared to the other two partitioning configurations. This is due to the fact that the query execution using partitioning configuration A does not have any data partitioning algorithm that can take advantage of data locality on the nodes of the cluster.

- **Partitioning Configuration B:** UPC++ programs with all the data files replicated across all PGAS nodes. This is a simple partitioning algorithm providing significant speedup over the previous approach. In this scenario, data locality can be taken advantage of. However, replication of each table on all nodes would lead to high data redundancy. This does not adversely impact the query execution times as much as it does efficient utilization of resources (memory). As shown in Figure 4.3, data redundancy increases significantly when execution moves from partitioning configuration A to partitioning configuration B. This is due to the fact that simple replication of data is employed in partitioning configuration B, which leads to a wastage of memory resources.

- **Partitioning Configuration C:** This partitioning configuration executes UPC++ programs with PREF-partitioned data files. This partitioning configuration maintains a good balance between data locality and data redundancy. As shown in Figure 4.3, partitioning configuration C does well in terms of both execution time as well as memory usage by maintaining a balance between data locality and data redundancy.

This small experiment illustrates how data partitioning has led to better overall execution times for UPC++ programs corresponding to SQL queries. However, it is also important to note here that data partitioning is an exercise whose primary goal is not to speed up query execution but to scale the query compilation algorithm to a cluster of nodes from a standalone instance for very large data sets. The speedup observed due to partitioning done in the experiments recorded in this thesis is only a side-effect of the effort needed to scale the query compilation infrastructure and minimize data shuffling.

4.2.4 Compilation Overhead

The UPC++ programs generated by our code generation module have to be compiled first before they can be executed. Figure 4.4 shows stacked bars corresponding to the times taken for plan generation, code generation, code compilation and query execution (on a 2-Node cluster with a scale factor 1) for each of the five queries used in this experiment. The compilation task takes most of the time during the processing of the SQL queries. Therefore, it can be observed that the compilation of the UPC++ programs is an added overhead for the first query run. However, once a program corresponding to an SQL query has been generated and compiled, this additional overhead can be avoided and the program flow can jump to the execution phase directly. However, changing certain query parameters would require a re-generation of code followed by re-compilation. This problem can be addressed

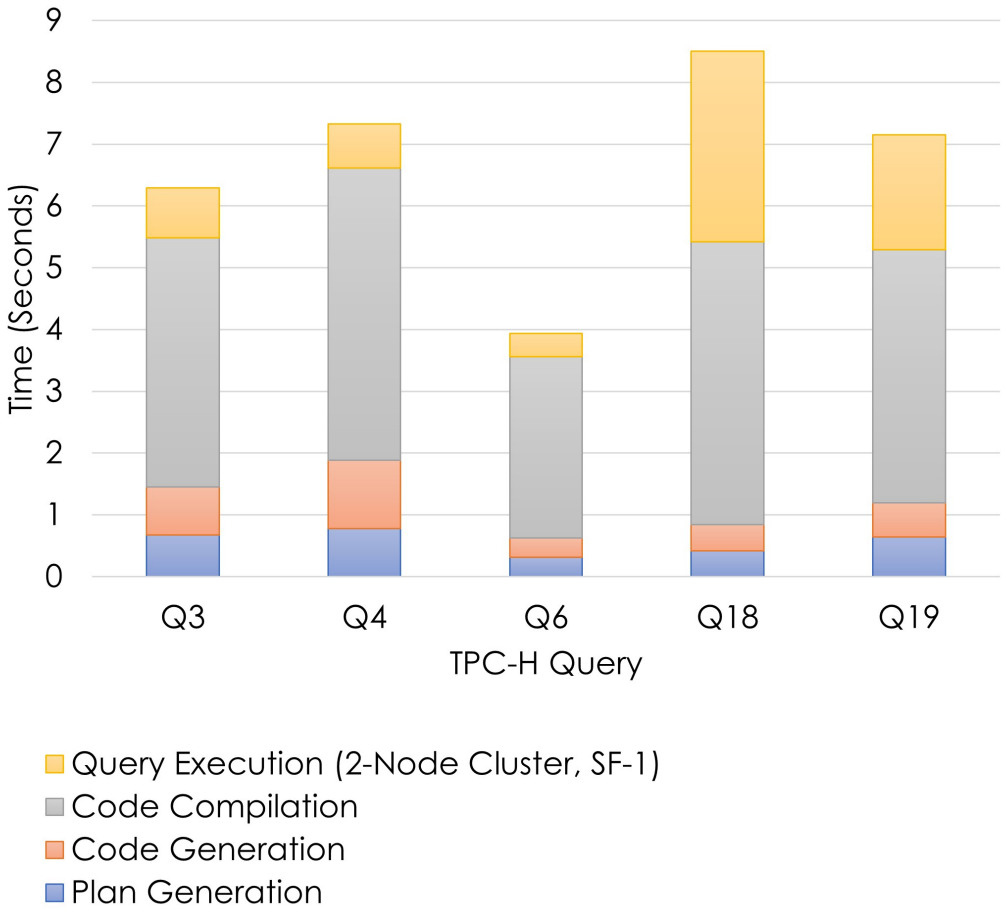


Figure 4.4: End-to-End Query Processing Time

by making query parameters pluggable and employing techniques like ahead-of-time compilation [27] for code that is less likely to change for a particular query (query initialization, data ingestion, view materialization etc.) and just-in-time compilation [10] for query parameters that can be changed during the multiple runs of a query as per the user/business requirement. The compilation overhead can further be reduced by pruning away the extra optimization stages during compilation that do not offer any significant execution speedup. Employing the aforementioned techniques for reducing compilation overhead is an active area of research currently, and more experiments are needed to ascertain the extent of benefit provided by them.

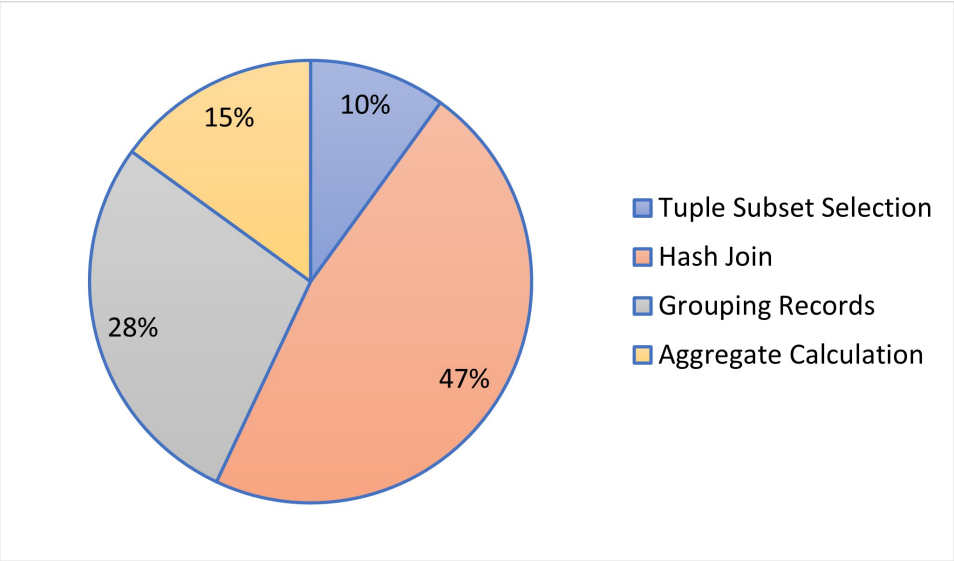


Figure 4.5: Performance Profiling for CasaDB Program Run for Query-3

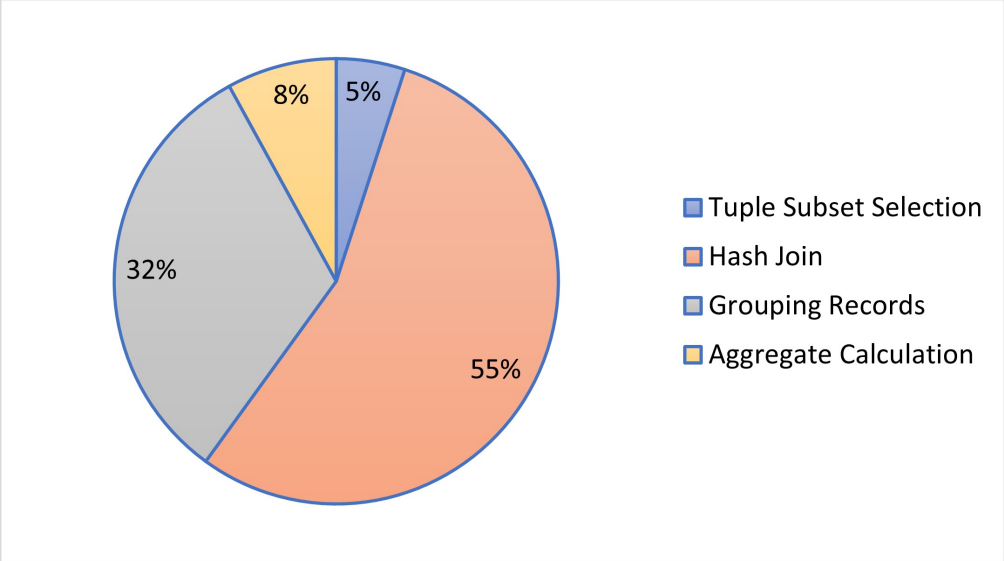


Figure 4.6: Performance Profiling for Radish Program Run for Query-3

4.2.5 Profiling and Analysis of Results

The query run for query-3 is profiled and the results are illustrated in Figure 4.5 for UPC++ programs and Figure 4.6 for Radish programs. It can be observed that UPC++ programs spend around 47% and 28% of their execution time in performing hash joins and grouping tuples respectively. The remaining 25% of execution time is spent in tuple subset selection (filtering tuples based on the predicates) and cal-

culating aggregate functions. On the other hand, in case of Radish around 87% of the execution time is spent performing hash joins and grouping rows for aggregation. Radish programs spend higher time during this phase due to the heavy communication involved between the two nodes that are used for this experiment. This is a direct consequence of the absence of a sophisticated partitioning algorithm designed to minimize data shuffling between nodes during query processing. From the charts in Figures 4.1 and 4.2, it can be observed that the UPC++ code is doing slightly better than Radish-generated code, usually around 2 times. This speedup can be attributed to the partitioning algorithm used in our research, which is not present in case of Radish. Another factor that adds up to the performance improvement is the fact that UPC++ supports asynchronous operations whose design can customized by the programmer. This feature is not supported by Grappa, the PGAS library used by Radish.

4.3 Summary

In this chapter, the results corresponding to the TPC-H benchmark suite have been recorded and analyzed. It is observed that the system developed in this thesis performs approximately 2× better than a similar system developed previously. This is mainly due to the fact that along with an efficient code generation algorithm, an effective data partitioning algorithm is also employed. The UPC++ programs, that can take advantage of the library's asynchronous nature, execute using an SPMD model and generate results concurrently on different nodes, which are then combined to produce a consolidated result set. However, there is a compilation overhead associated with the approach proposed in this thesis and can be avoided by using JIT and AOT compilation techniques. Even though our system does well on certain queries and scale factors, it is not in a state to be used as a replacement for full-

fledged data management systems. This is mainly due to the fact that this work is proof-of-concept of how high-performance computing can be used to build systems to query data. There are a lot of other mechanisms that go into the development of an actual working system that can further slow down the performance and has not been accounted for in this thesis.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The research work in thesis is the first of its kind in terms of using an asynchronous PGAS framework to implement a query compilation backend. Earlier, Radish had tried to bring the areas of high-performance computing and query compilation together by generating Grappa-based query evaluation plans. This thesis augments the current state-of-the-art in high-level query evaluation plans by using a more advanced high-performance programming library and running them on a distributed cluster with PREF-partitioned data sets. On an average, our query evaluation programs preformed approximately two times better than Radish-based plans on all five of the TPC-H queries evaluated in this thesis. The results demonstrate that efficient partitioning of data sets and allowing asynchrony in program execution can yield significant performance improvement over randomly distributed data and synchronous model of program execution. This model of query execution with UPC++ also allows better scalability due to its simple SPMD program execution model.

5.2 Future Work

In its current state, the UPC++ backend does not support some SQL operators, such as order by, limit and like. This is because the query planner does not support these operators. The full set of SQL operators can be supported if the current planner is replaced with a more sophisticated relational algebra compiler, like Apache Calcite [11]. At present, CasaDB does not offer any improvement over Radish in terms of code generation and query compilation time, although it performs significantly better than Radish in terms of query execution time. This can be changed by implementing code caching techniques in the code generator module.

Moreover, the data set used for the experiments conducted in this thesis is not skewed. However, a lot of data generated these days can have a considerable amount of skew that has not been accounted for while designing and developing the partitioning module developed for this thesis. The partitioning algorithm can be augmented to support skewed benchmarking data sets like TPC-DS.

An advanced query compiler design technology, called Lighweight Modular Staging (LMS) [38], uses an incremental query compilation technique so that at each stage the query generation process is optimized, resulting in faster execution plans. LegoBase [22] uses this compilation technique for single-node query processing. The research in this thesis can be extended by adopting an LMS-based compilation strategy. As we saw earlier in Chapter 4, the compilation overhead involved in UPC++ programs is a significant performance bottleneck. In order to avoid this additional overhead, the work in this thesis can also be extended by generating an Eclipse OMR JitBuilder-based IR instead of high-level query evaluation plans as done in HyPer by generating LLVM-based IR [2, 28]. This will open doors for incorporating ahead-of-time compilation and just-in-time compilation strategies in the current system.

Purely compilation-based query evaluation is not always the most efficient approach. Adaptive query evaluation [23] that combines the best of interpretation and compi-

lation based query evaluation strategies can be incorporated in our current system to yield better results.

Bibliography

[1] *Apache Arrow*, `https://arrow.apache.org/docs/`, Accessed: 2020-09-30.

[2] *Eclipse OMR*, `https://github.com/eclipse/omr`, Accessed: 2020-09-30.

[3] *HyPer WebInterface*, `https://hyper-db.de/interface.html#`, Accessed: 2020-09-30.

[4] *TPC-H*, `http://www.tpc.org/tpch/`, Accessed: 2020-09-30.

[5] *UPC Wiki*, `https://upc.lbl.gov/docs/`, Accessed: 2020-09-30.

[6] *UPC++ Wiki*, `https://bitbucket.org/berkeleylab/upcxx/wiki/Home`, Accessed: 2020-09-30.

[7] Daniel J Abadi, Samuel R Madden, and Nabil Hachem, *Column-stores vs. row-stores: how different are they really?*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, pp. 967–980.

[8] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood, *Dbmss on a modern processor: Where does time go?*, VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK, no. CONF, 1999, pp. 266–277.

[9] George Almasi, *Pgas (partitioned global address space) languages.*, 2011.

[10] John Aycock, *A brief history of just-in-time*, ACM Comput. Surv. **35** (2003), no. 2, 97–113.

[11] 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, 2018, pp. 221–230.

[12] Dan Bonachea and Jaein Jeong, *Gasnet: A portable high-performance communication layer for global address-space languages*, CS258 Parallel Computer Architecture Project, Spring (2002).

[13] Peter Boncz, Thomas Neumann, and Orri Erling, *Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark*, Technology Conference on Performance Evaluation and Benchmarking, Springer, 2013, pp. 61–76.

[14] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren, *Introduction to upc and language specification*, Tech. report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[15] Leonardo Dagum and Ramesh Menon, *Openmp: an industry standard api for shared-memory programming*, IEEE computational science and engineering **5** (1998), no. 1, 46–55.

[16] Frederica Darema, *The spmd model: Past, present and future*, European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2001, pp. 1–1.

[17] Resilient Distributed Datasets, *A fault-tolerant abstraction for in-memory cluster computing*, Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur

Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI **12** (2012).

[18] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick, *Upc: distributed shared memory programming*, vol. 40, John Wiley & Sons, 2005.

[19] Jianqing Fan, Fang Han, and Han Liu, *Challenges of Big Data analysis*, National Science Review **1** (2014), no. 2, 293–314.

[20] Goetz Graefe and William J McKenna, *The volcano optimizer generator: Extensibility and efficient search*, Proceedings of IEEE 9th International Conference on Data Engineering, IEEE, 1993, pp. 209–218.

[21] Alfons Kemper and Thomas Neumann, *Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots*, 2011 IEEE 27th International Conference on Data Engineering, IEEE, 2011, pp. 195–206.

[22] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi, *Building efficient query engines in a high-level language*, Proceedings of the VLDB Endowment **7** (2014), no. 10, 853–864.

[23] André Kohn, Viktor Leis, and Thomas Neumann, *Adaptive execution of compiled queries*, 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 197–208.

[24] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al., *Impala: A modern, open-source sql engine for hadoop.*, Cidr, vol. 1, 2015, p. 9.

[25] Donald Kossmann, *The state of the art in distributed query processing*, ACM Computing Surveys (CSUR) **32** (2000), no. 4, 422–469.

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

[27] Georgiy Krylov, Gerhard W. Dueck, Kenneth B. Kent, Daryl Maier, and Irwin D'Souza, *Ahead-of-time compilation in omr: Overview and first steps*, Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (USA), CASCON '19, IBM Corp., 2019, p. 299–304.

[28] C. Lattner and V. Adve, *Llvm: a compilation framework for lifelong program analysis amp; transformation*, International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86.

[29] David Luebke, *Cuda: Scalable parallel programming for high-performance scientific computing*, 2008 5th IEEE international symposium on biomedical imaging: from nano to macro, IEEE, 2008, pp. 836–838.

[30] Sam Madden, *From databases to big data*, IEEE Internet Computing **16** (2012), no. 3, 4–6.

[31] Volker Markl, *Query processing (in relational databases)*, pp. 2288–2293, Springer US, Boston, MA, 2009.

[32] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar, *The art of efficient in-memory query processing on numa systems: a systematic approach*, 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 781–792.

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

[34] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin, *Grappa: A latency-tolerant runtime for large-scale irregular applications*, International Workshop on Rack-Scale Computing (WRSC w/EuroSys), 2014.

[35] Thomas Neumann, *Efficiently compiling efficient query plans for modern hardware*, Proceedings of the VLDB Endowment **4** (2011), no. 9, 539–550.

[36] Sudsanguan Ngamsuriyaroj and Rangsan Pornpattana, *Performance evaluation of tpc-h queries on mysql cluster*, 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, 2010, pp. 1035–1040.

[37] KL Rajashree, *The myria big data management and analytics system and cloud service.*

[38] Tiark Rompf and Martin Odersky, *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls*, Proceedings of the ninth international conference on Generative programming and component engineering, 2010, pp. 127–136.

[39] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu, *The asynchronous partitioned global address space model*, The First Workshop on Advances in Message Passing, 2010, pp. 1–8.

[40] Amir Shaikhha, Mohammad Dashti, and Christoph Koch, *Push versus pull-based loop fusion in query engines*, Journal of Functional Programming **28** (2018).

[41] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz, *Vectorization vs. compilation in query execution*, Proceedings of the Seventh International Workshop on Data Management on New Hardware, 2011, pp. 33–40.

[42] Olivier Tardieu, *The apgas library: resilient parallel and distributed programming in java 8*, Proceedings of the ACM SIGPLAN Workshop on X10, 2015, pp. 25–26.

[43] David W Walker and Jack J Dongarra, *Mpi: a standard message passing interface*, Supercomputer **12** (1996), 56–68.

[44] Erfan Zamanian, Carsten Binnig, and Abdallah Salama, *Locality-aware partitioning in parallel database systems*, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '15, Association for Computing Machinery, 2015, p. 17–30.

[45] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick, *Upc++: a pgas extension for c++*, 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 1105–1114.

Appendix A

Installation and Build Instructions

Perform these steps on each node of the cluster

A.1 UPC++

1. git clone https://bitbucket.org/berkeleylab/upcxx/src/master/

2. Follow rest of the instructions from the **installation wiki here**.

A.2 Arrow

Please follow the build instructions on **Arrow C++'s Wiki**. Make sure to do a "sudo make install" at the end.

A.3 Radish and Grappa

Build and install instructions are available on the following relevant GitHub repositories:

1. **RACO Repository**

2. **Radish Server repository**

3. **TPC-H Myrial repository**

4. **Grappa repository**

Appendix B

Execution Instructions

B.1 UPC++

The execution instructions and other training materials for UPC++ programs are available **here**.

B.2 Radish

Detailed instructions for benchmarking Radish on TPC-H suite is available **here**.

Glossary

PGAS	Partitioned global address space (PGAS) is a parallel programming model for the development of high-performance applications on clusters. It provides a global address space partitioned among the cluster nodes, and is supported in programming languages like C, C++, and Java by means of APIs.
DSM	Distributed shared memory (DSM) is a form of memory architecture where physically separated memories can be addressed as one logically shared address space.
QueryProcessing	Query processing denotes the compilation and execution of a query specification usually expressed in a declarative database query language such as the structured query language.

Vita

Candidate's full name: Shubh Sharma

University attended :

- Bachelor of Technology in Information Technology, National Institute of Technology Duragpur, India, 2017
- Master of Computer Science, University of New Brunswick, Canada, 2021

Publications:

- Refereed Poster: S. Sharma, S. Ray, M. Stoodley. "Generative Query Compilation for Distributed Query Processing" 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.

Conference Presentations:

- Expo Talk: S. Sharma, S. Ray, M. Stoodley. "Generative Query Compilation for Distributed Query Processing" 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.