Vrije Universiteit, Amsterdam

Faculty of Sciences,
Computer Science Department

Andrei Costea, student no. 2118548

Adrian Ionescu, student no. 2116731

Query Optimization and Execution in Vectorwise MPP

Master's Thesis in Computer Science
Specialization: High Performance and Distributed Computing

Supervisor:
Marcin Żukowski, Actian Corp.

First reader:
Peter Boncz, Vrije Universiteit, Centrum Wiskunde & Informatica

Second reader:
Jacopo Urbani, Vrije Universiteit

Amsterdam, August 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Most organizations nowadays demand from their online analytical processing (OLAP) capable DBMS a short response time to queries on volumes of data that increase at a fast pace. In order to comply with these requirements, many commercial database vendors have developed massively parallel processing (MPP) solutions. The Vectorwise DBMS is based on a vectorized query execution engine that takes advantage of the capacities of modern CPUs [Zuk09]. In this thesis, we explore the development challenges encountered in the initial steps towards an MPP solution for the Vectorwise DBMS. Our work will focus on the two most critical layers: query optimization and query execution.

Note: For clarification purposes, henceforth we will refer to the new solution as *distributed*, while the original, multi-core version of Vectorwise will be referred to as *non-distributed.*

Research Questions

- Is there a non-intrusive, yet scalable way to implement an MPP solution for the Vectorwise engine based on the current multi-core parallelism model?
- Do the characteristics of modern hardware make a shared-disk DBMS architecture scalable, hence an attractive alternative to shared-nothing solutions?
- What are the challenges of job scheduling and query optimization in a distributed DBMS?

1.2 Goals

Throughout the planning and development stages of our work, we have been aiming to create a prototype that:

1. maintains the high performance exhibited by the Vectorwise DBMS engine in OLAP queries
2. requires easy and non-intrusive modifications to the current implementation
3. requires no additional effort in order to configure/use it
4. maintains the cost of hardware to performance ratio

1.3 Non-goals

We acknowledge the importance (or even indispensability) of the following features in any production-ready DBMS. That is why it is one of our commitments to make sure none of our fundamental design decisions will compromise the possibility for these topics to be addressed in future work.

- updates, DDL operations, transaction management are out of scope; we only focus on read-only access
- no explicit data partitioning
- we do not address reliability and high-availability concerns

1.4 Basic Ideas

The outline below is meant to give the reader a flavor of the directions according to which the research was carried in this project.

- implement the Xchg operator of the Volcano model for a distributed setting
- use a cluster of commodity Linux nodes connected by Infiniband network fabric
- use a distributed file-system and follow a "virtual shared disk" approach
- use MPI for inter-node communication

1.5 Vectorwise

Vectorwise[1] is a state-of-the-art relational database management system that is primarily designed for online analytical processing, business intelligence, data mining and decision support systems. The typical workload of these applications consists of many complex (computationally intensive) queries against large volumes of data that need to be answered in a timely fashion. What makes the Vectorwise DBMS unique and successful is its vectorized, in-cache execution model that exploits the potential of modern hardware (e.g. CPU cache, multi-core parallelism, SIMD instructions, out-of-order execution) in conjunction with with a scan-optimized I/O buffer manager, fast transactional updates and a compressed NSM/DSM storage model [Zuk09].

The Volcano Model The query execution engine of Vectorwise is designed according to the widely-used Volcano iterator model [Gra94]. Query execution plans are represented as trees of "operators", hence often referred to as a query execution trees (QET). An operator is an implementation construct responsible for performing a specific step of the overall query processing required.

All operators implement a common interface, providing three methods: `open()`, `next()` and `close()`. Upon being called on some operator, these methods recursively call their corresponding methods on the children operators in the QET. `open()` and `close()` are one-off, complementary methods that perform initialization work and resource freeing, respectively. Each `next()` call produces a new tuple (or a block of tuples, see Vectorized Query Execution below). Due to the recursive nature of the `next()` call, query execution follows a "pull" model in which tuples traverse the operator tree from the leaf operators upwards, with the root operator returning the final results.

[1]Vectorwise website: www.actian.com/products/vectorwise

Multi-core Parallelism Vectorwise is currently capable of multi-core parallelism in two forms: *inter-query parallelism* and *intra-query parallelism*. The latter is achieved thanks to a special operator of the Volcano model, called the *Xchange operator* [Gra90], which will be described in more detail in later sections.

Vectorized Query Execution It has been observed that the traditional tuple-at-a-time model leads to low instruction and data cache hit rates, wasted opportunities for compiler optimizations, considerable overall interpretation overhead and, consequently, poor CPU utilization. The column-at-a-time model, as implemented in *MonetDB* [BK99] successfully eliminates interpretation overhead, but at the expense of increased complexity and limited scalability, due to its full-column materialization policy.

The execution model of Vectorwise combines the best of these two worlds by having the QET operators process and return not one, but a fixed number of tuples (100 - 10000) whenever their *next()* method is called. This batch of tuples is referred to as a *vector* and the number of tuples is chosen such that the vector can fit in the L1-cache memory of the processor. The processing of vectors is done in specialized functions called *primitives*.

This reduces the interpretation overhead of the Volcano model, because the interpretation instructions are amortized across multiple tuples and the actual computations are performed in tight loops, thus benefiting from performance improving features of modern CPUs (e.g. superscalar CPU architecture, instruction pipelining, SIMD instructions) [Zuk09], as well as compiler optimizations (e.g. loop unrolling).

1.5.1 Ingres/Vectorwise Architecture

Vectorwise is integrated within the Ingres[1] open-source DBMS. A high-level diagram of the whole system's architecture, depicting the main components involved, is given in Figure 1.1.

Ingres is responsible for the top layers of the database management system stack. The Ingres server interacts with users running the Ingres client application and parses the SQL queries they issue. Also, Ingres keeps track of all database schemas, metadata and statistics (e.g. cardinality, distribution) and uses this information for computing and providing Vectorwise with an optimal query execution plan.

Vectorwise can be viewed as the "database engine", taking care of query execution, buffer management and storage. Its main components are described below.

Rewriter The optimized query execution plans that Ingres passes to Vectorwise are expressed as logical operator trees in a simple algebraic language, referred to as the *Vectorwise Algebra*, which will be presented in Section 3.2.

The Vectorwise *Rewriter* performs specific semantic analysis on the received query plan, annotating it with data types and other helpful metadata. It also performs certain optimization steps that were not done by Ingres, such as introducing parallelism, rewriting conditions to use lazy evaluation and early elimination of unused columns. More information on the Rewriter module can be found in Section 3.3.2

Builder The annotated query plan is then passed to the Builder, which is responsible for interpreting it and producing a tree of "physical" operators that implement the iterator functionality defined by the Volcano model.

[1]Ingres website: www.actian.com/products/ingres

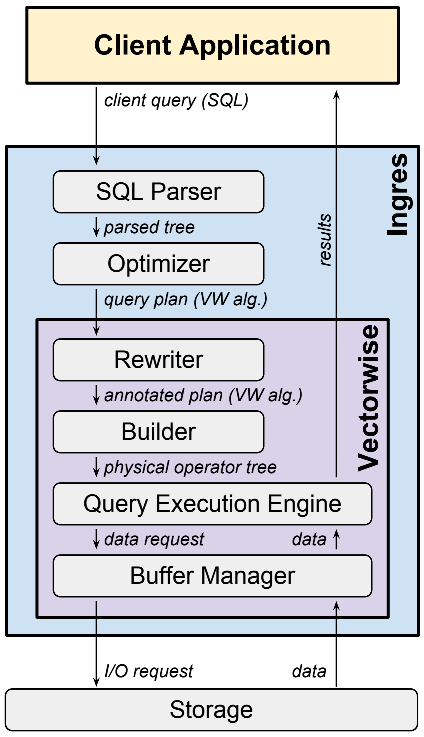


Figure 1.1: Architecture of the Ingres/Vectorwise System

This so-called *builder phase* corresponds to calling the *open()* method on all operators in a depth-first, post-order traversal of the query execution tree. This is when most of the memory allocation and initialization work takes place.

Query Execution Engine Once the builder phase is done and the physical operator tree is constructed, executing the query then simply translates to repeatedly calling *next()* on the root operator of the tree until no more tuples are returned. The paragraphs on the Volcano Model and Vectorized Query Execution should have given a good description of the way the Vectorwise query execution engine works.

Buffer Manager and Storage The Buffer Manager and Storage layer is primarily responsible for storing data on persistent media, accessing it and buffering it in memory. Additionally, it also takes care of handling updates, managing transactions, performing crash recovery, logging, locking and more. However, none of these issues are the focus of this thesis, so we will not elaborate on them.

1.6 Related Work

In this section we present a brief overview of the academic work on parallel and distributed database systems, after which we review some of the most successful commercial parallel / MPP solutions in the field of analytical processing that are currently available.

1.6.1 Academic Research

Extensive work on parallel and distributed database systems was done in the early 1990s. The series of surveys [OV91, Val93, OV96] presents the state-of-the-art of the field at that time, clarifies the related terminology, classifies database architectures and discusses open problems and trends.

Regarding terminology, a *parallel database system* is considered one that takes advantage of a parallel computer, or multiprocessor, that is itself a (tightly coupled) distributed system made of a number of nodes (processors and memories) connected by a fast network within a cabinet. On the other hand, a *distributed database* is a collection of multiple, logically interrelated databases distributed over a computer network, while a *distributed database management system* (distributed DBMS) is defined as a software system that permits the management of the distributed database and makes the distribution transparent to its users.

With respect to parallel database architectures, the consensus is that there are three basic approaches:

- *Shared-Memory* (or *Shared-Everything*): Any processor has access to any memory module or disk unit through a fast interconnect. The main advantages of this approach are simplicity and natural load balancing, but they come at the price of high cost, limited scalability, and low availability. Examples of shared-memory database systems include: *XPRS* [SKPO88], *DBS3* [BCL93] and *Volcano* [Gra90] (notable for its elegant formalism).
- *Shared-Disk*: Any processor has access to the entire data (residing on persistent storage), but makes exclusive use of its main memory. This approach improves on cost, scalability and availability, while still naturally lending itself to load balancing. However, communication overhead among nodes and concurrent access to the shared disk are potential bottlenecks. Considerable less parallel database prototypes can be found in the literature that follow this approach.
- *Shared-Nothing*: Each processor has exclusive access to its main memory and disk unit(s). The difference between shared-nothing parallel databases and distributed databases is to some extent blurry, being mostly a matter of the degree of decoupling between nodes. Such systems rank highest in terms of performance, scalability and availability, but also complexity. Load balancing is potentially compromised. Pioneering shared-nothing prototypes include: *Bubba* [BAC+90], *Gamma* (which demonstrated good scalability) [DGS+90], *EDS* [WT91], or *Prisma* (main-memory system featuring the symmetric HashJoin operator [WA91]) [ABF+92], all of which exploited both inter- and intra-query parallelism. Moreover, most of the commercial solutions presented in the following section have a shared-nothing architecture.

1.6.2 Commercial Solutions

Teradata

Teradata[1] is regarded as the market leader in data warehousing and enterprise analytics for the past two years. It relies on an MPP solution developed over a period of more than 25 years that is based on a shared-nothing architecture taking advantage of a row storage scheme (one of the few row-based RDBMS capable of processing petabytes of data).

Internally, Teradata [BDW04] consists of a *parsing engine* (PE) and multiple *access module processors* (AMPs) connected by a communication channel, namely *BYNET*. PE is responsible for receiving and parsing queries, data localization, query optimization and resource allocation,

[1] http://www.teradata.com/

while AMPs are logical units that correspond to the nodes in a distributed DBMS, with exclusive memories, disks and processing units.

IBM DB2

DB2[1] relies on a shared-nothing architecture composed of several database *partitions* or *nodes*. These partitions operate on their own data, indexes, configuration files and transaction logs [DB210]. Moreover, each of these partitions can act as gateways for user interactions. Depending on the available hardware, the DB2 can achieve both inter-node and intra-node parallelism for a given query.

DB2 is a row-store DBMS in which the administrator can choose between range-partitioning, hash-partitioning or even multidimensional clustering.

Exasol

Similar to DB2, Exasol[2] consists of multiple "peer" nodes with identical software and no shared hardware. As its main focus is data warehousing, it uses a column storage scheme in which tuples are partitioned according to a hash key. Furthermore, to accelerate response times, it is optimized for in-memory processing and is aware of the clustered environment in the query optimizer module.

Vertica

Vertica[3] is a built from scratch analytic database specialized for OLAP queries and data-warehousing [Ver10]. Like, DB2 and Exasol, it relies on a shared-nothing architecture where every node can receive queries expressed in SQL. Moreover, as most DBMSs specialized on analytical processing, it uses a column storage scheme where aggressive compression of data reduces the amount of storage space by up to 90%. This permits Vertica to employ a strategy for providing fault-tolerance and availability that consists of a mix of data replication and partitioning (called *projections*). Projections store every column attribute at multiple sites and, consequently, improves query performance in most cases (e.g. when the data to be processed is local).

Oracle's Real Application Cluster (RAC)

What distinguishes Oracle's RAC[4] from the rest of the commercial DBMS presented in this section is its shared-disk approach [LM10]. Although it requires a high-speed dedicated interconnect between the server instances and the shared storage, this approach eliminates any need for a partitioning scheme. This, in turn, erases any additional administration costs that would come with the shared-nothing approach.

Moreover, RAC relies on a cluster file-system to facilitate its deployment, to assure high-availability and fault tolerance, and to provide an easy mechanism for load balancing when nodes are addded/removed from the system.

[1] http://www-01.ibm.com/software/data/db2/
[2] http://www.exasol.com/en
[3] http://www.vertica.com/
[4] http://www.oracle.com/technetwork/products/clustering/overview/index.html

Other MPP solutions

Of course, there are several other vendors that propose MPP solutions for business analytics and data warehousing. Of these, we would like to mention **Greenplum**[1] and **Paracell**[2].

Greenplum has a more scalable architecture than the rest of the systems discussed in this section as it tries to parallelize all the phases of query processing, even the query optimization phase [Gre10].

Finally, Paraccel tries to take advantage of modern CPU technology by compiling queries and reusing compiled segments [Par10].

[1] http://www.greenplum.com/
[2] http://www.paraccel.com/

Chapter 2

Approach

This chapter presents the high-level architecture of the proposed distributed version of Vectorwise, explaining how the various architectural choices were made in accordance with the primary goals of this project. Ideally, the rationale behind the "Basic Ideas" presented in Section 1.4 should become apparent by the end of this chapter.

To begin with, for a *cost-effective*, *scalable*, yet efficient solution, we focused our attention towards a Linux computer cluster, interconnected by a low-latency, high bandwidth network fabric. Section 2.1 elaborates on the choice for the interconnect family, as well as the communication model. In Section 2.2 we argue that a Shared Disk approach is preferable for ensuring *non-intrusiveness*, *ease of use* and *maintainability*. We present several alternatives for providing the cluster nodes with a globally accessible disk volume and identify distributed parallel file systems as a promising solution for this. In Section 2.3 we describe the hardware and software configuration of the small, in-house cluster we used for developing and testing our prototype and, finally, in Section 2.4 we present a high-level overview of the major components of the distributed version of Vectorwise that we propose.

2.1 Network Communication

From the hardware point of view, the current most dominant interconnect families used in the high-performance computing (HPC) world are Ethernet and Infiniband, each being used in ca. 200 of the "Top 500 Supercomputers" listed by [Top]. While both technologies provide bandwidths of above 100 Gb/s, we decided on the former for practical considerations such as being better established in the market.

Network communication is what ties together the separate database server instances on the different cluster nodes[1] into a single "distributed system"; it is their only means of coordination (e.g. Master-Worker interactions) and cooperation (i.e. distributed query execution). As such, choosing an appropriate network communication paradigm is of great importance to our project.

The most distinctive models for communication that we considered for implementing our parallel DBMS engine are listed below, according to the level of abstraction they provide to the programmer (from highest to lowest):

1. distributed shared memory (DSM)
2. remote procedure calls (RPC)

[1]On a given node it usually makes sense to run a single database server instance, so throughout the text we use these two terms interchangeably.

3. message passing interface (MPI)
4. raw socket programming

Distributed shared memory (DSM) is memory which, although distributed over a network of autonomous computers, gives the appearance of being centralized [SG95]. Specialized middleware or libraries allow both local and remote memory to be accessed through virtual addresses and thus processes become able to communicate by reading and modifying data which is directly addressable. Allowing programmers to use the concept of shared memory, DSM makes programming parallel/distributed applications considerably easier. However, this comes at the expense of performance and scalability limitations and, since for our project the importance of high performance clearly outweighs the benefits of distribution transparency for the programmer, DSM was easily ruled out.

At the other extreme, programming against the raw socket API could potentially give the best results in terms of performance, but only if considerable development effort is invested. This was not an option for our project, given the strict time constraints we had. Moreover, such an approach would not have been portable, since the implementation would be network fabric specific (e.g. socket API vs Infiniband Verbs).

While RPC would be feasible performance-wise, its synchronous and centralized (i.e. client-server) nature makes it incompatible with our our communication pattern, in which any node (be it Master or Worker) needs to be able to initiate communication with any other node during the distributed query execution stage and this communication must be asynchronous in order to overlap.

Therefore, we decided to rely on MPI (Message Passing Interface), which is considered to be the "de facto" standard for communication among processes that model a parallel program running on a distributed memory system. MPI is extensively used in the high performance computing community and offers users a rich and well defined API for collective and point-to-point communication.

2.2 "Virtual" Shared Disk

Shared Disk versus Shared Nothing

As explained before, our goals include creating a pararallel database that is easy to use and maintain, without making radical changes to the existing Vectorwise architecture. We argue that these goals are not incompatible with the primary requirement of obtaining improved OLAP query processing performance and we aim to explain below how using a "Virtual Shared Disk" approach can provide the solution to all this.

[Hog09] presents a concise and rather informal, but extremely insightful comparison of the Shared Disk (SD) and Shared Nothing (SN) architectures, elaborating on the advantages and disadvantages of each with respect to several functional and non-functional requirements.

The crucial difference between the two approaches is whether or not *explicit data partitioning* is required. SD does not require it, as every node of the cluster has full access to read and write against the entirety of the data. On the other hand, the latter statement does not hold in the case of a SN architecture and additional effort is required for designing a partitioning scheme that minimizes inter-node messaging, for splitting data among the cluster nodes and for implementing a mechanism to route requests to the appropriate database servers.

Moreover, the additional effort involved with data partitioning is not only required once, during the initial set-up of the system, but rather periodically, as a part of a maintenance routine. This is because as data gets added or removed from the database, the initial partitioning scheme can become sub-optimal. This is especially the case if the query pattern changes over time. Also, a SD database system is naturally suited for dynamic load-balancing, while a SN system can only allow limited load balancing, based upon the chosen partitioning scheme.

However, in terms of performance, the scalability of a shared-disk approach is limited by the inter-node messaging required for cooperative query execution and general coordination (e.g. resource management, locking, heart-beat, etc.). Shared-nothing systems typically require less or no such coordination overhead. However, their performance is highly impacted by the necessity of function- and data-shipping in cases when (e.g. Join) requests span multiple nodes.

The general idea is that, with good partitioning schemes, function- and data-shipping can be avoided for most query patterns, leading to optimal results in terms of performance. The big challenge is, however, devising such a good partitioning scheme and to maintaining automatically, without user intervention, by regularly re-partitioning and re-balancing data across the cluster nodes. While this is indeed one of the major goals in the quest towards an MPP version of Vectorwise, we deemed it too complex a task to tackle in our pioneering project.

Cluster File System

Considering all the above arguments, we decided to follow the Shared Disk approach. There are several choices available for offering nodes of a cluster access to the same (shared) file system (sometimes referred to as a "cluster file system"). [Heg06] elaborates on the overlapping and often confusing terminology surrounding this subject, giving real-world examples to illustrate the various concepts. In short, from a hardware point of view, three major approaches for building a cluster file system can be distinguished:

- network-attached storage (NAS)
- storage area networks (SAN)
- direct-attached storage (DAS) + distributed parallel file systems (DPFS)

While NAS devices are generally considered to be easy to use and maintain, they either do not scale (e.g. NFS servers), or require costly, specialized hardware (e.g. dedicated NAS appliances). In any case, their performance is inherently limited by the use of file-level protocols. While the second alternative, namely SAN, can provide high performance in terms of I/O throughput (due to block-level protocols) and does scale well, it also requires specialized hardware (e.g. Fiber Channel adapters and fabrics) that is costly and difficult to set-up and maintain. Therefore, we considered NAS and SAN to be incompatible with our stated direction towards a simple, cost-effective, yet efficient solution and focused our attention on distributed parallel file systems.

The advantage of DPFS is that they provide easily extensible, scalable and relatively efficient cluster file system solutions using commodity hardware, eliminating the need for additional infrastructure. Moreover, high availability comes at a cheap price. The "distributed" term in DPFS refers to the property that clients have knowledge about and directly communicate with multiple file servers, while the "parallel" term indicates that a single file, or even a single I/O request, can be striped across multiple servers for improved performance. In a simple DPFS configuration, cluster nodes typically act as both data clients and servers, contributing with their local disk drives to a globally accessible logical volume.

Several DPFS solutions are available free of charge, under GPL license: *Lustre* [Sch03], *PVFS* [CLRT00], *Gfarm* [THS10], *Ceph* [WBM$^+$06], or *GlusterFS* [Glu]. Of all these, Lustre strikes as the most mature, efficient and scalable solution, being often used as a performance yardstick of its field and powering many of the top supercomputers listed by [Top]. Unfortunately, it remains notoriously difficult to deploy and maintain, the main reasons for this being that it runs completely in kernel space (requiring a patched Linux kernel) and has a complex architecture, involving separate management, meta-data and storage servers. The authors of [MKAB04] report investing several days' effort into getting Lustre installed.

Of the remaining DPFS solutions identified above, we focused on GlusterFS, wich has a good reputation in terms of manageability, scalability, as well as performance. It has often been used in production systems, or as the foundation for various pieces of academic research.

GlusterFS is a highly adaptable, feature rich, POSIX-compliant DPFS that runs completely in user-space and can work with either Ethernet or Infiniband fabrics. It provides data striping, replication and self-healing capabilities. Moreover, we found it to be particularly easy to set-up and manage. As such, we proceeded with relying on GlusterFS to fulfill the "Shared-Disk" assumption.

2.3 Experimentation platform

Following the envisioned "approach" and direction, our small Linux cluster is set up in accordance with all the above-mentioned considerations and it has served as an experimentation and test platform throughout the development of our prototype. A brief description of its relevant hardware and software components is given below.

Hardware

Nodes: 2×2 hot-pluggable systems (nodes) in a 2U form-factor (4 nodes in total).

CPUs: 2 Intel Xeon E5645 processors (6 cores, 2.40 GHz, 12 MB cache) per node; Hyper-Threading disabled

Memory: 48 GB DDR3 per node

Storage: 4 solid-state drives (128 GB) per node

Network: 40 Gbit/s InfiniBand Interconnection

- 4X quad data rate (QDR) Host Channel Adapters (HCAs)
- 8-port Mellanox Infiniscale-IV switch

Software and Middleware

The four cluster nodes run a minimal installation of the *CentOS* (release 6.2, kernel version 2.6.32-220) Linux distribution, which was chosen based on it being widely popular[1] and available free of charge.

Of the available MPI implementations we narrowed our choice down to MVAPICH2[2] and Intel MPI[3], which both rank high when evaluated according to the following criteria (listed in the order of importance for our project):

1. full multi-threading support (*MPI_THREAD_MULTIPLE*)
2. compliance with the MPI-2 standard
3. achieving good performance on Infiniband fabrics

[1]In July 2010 CentOS was the most popular Linux distribution on web servers, according to http://w3techs.com/blog/entry/highlights_of_web_technology_surveys_july_2010

[2]MVAPICH2: http://mvapich.cse.ohio-state.edu/overview/mvapich2/

[3]Intel MPI: http://software.intel.com/en-us/articles/intel-mpi-library/

4. extensive user base and successful use in industry applications

We eventually decided on the MPI implementation provided by Intel (*Intel MPI Library 4.0 Update 3*), for the fact that we found it to be easier to use and configure, especially in combination with advanced debugging and profiling tools such as the ones provided by the *Intel Cluster Studio* suite[1].

A software RAID array of level 0 (i.e. block-level striping without parity or mirroring) was assembled out of the four solid state drives (SSDs) on each node, in order to achieve maximum I/O throughput. The resulting arrays were then used as building components of the distributed file system. More precisely, using *GlusterFS (ver. 3.2.6)* we assembled a "striped volume" (with no replication), for the same reason of maximizing I/O throughput. Unfortunately, however, the benchmarks in the next section show that the results obtained with this configuration under typical Vectorwise workloads are significantly worse than expected.

2.3.1 Micro-benchmarks

Network Performance

To measure the performance of the available network fabric we used the InfiniBand Diagnostic Tools package (IBD) delivered as part of the Mellanox OFED software stack (the Infiniband firmware). In addition, we used the Intel MPI Benchmark Suite (IMB) to asses the performance of the chosen MPI implementation. IMB offers a concise set of elementary MPI benchmarks designed for different communication patterns. For a detailed description, the reader is referred to [IMB].

In short, *PingPong* is the classical pattern used for measuring the throughput of (uni-directional) communication between two processes. *PingPing* measures asynchronous, uni-directional communication throughput in the case when outgoing messages are obstructed by incoming ones. *SendRecv* measures the bi-directional throughput that can be obtained with the MPI_SendRecv primitive, while *Exchange* is an asynchronous, bi-directional communication pattern that closely resembles the one required by the Distributed Xchg Hash/Range Split operators which will be described in the following chapters.

Figure 2.1 presents the uni- and bi-directional network **throughput** values reported by the IMB and IBD tools as functions of the message size. The results show that, for sufficiently large messages (1MB or above), Intel MPI reaches close to 100% bandwidth utilization.

Another network performance metric that is relevant in our case is message **latency**. From this point of view, Infiniband is known to be among the most efficient network fabrics available. On our hardware, both IBD and IMB tools report end-to-end message latencies of ca. 1.4 microseconds.

Disk and GlusterFS I/O Throughput

To test the performance of the "virtual" shared disk, we used the traditional "dd test" (using the *dd*[2] Linux command line utility), as well as an in-house tool (called *xib*) for measuring I/O read throughput by writing and reading a large file with a typical Vectorwise access pattern.

With the *dd* tool we measured the single-threaded sequential read access speed, by executing the following command with values for *block_size* ranging from 64KB to 16MB:

```
dd if=dd_file.dat of=/dev/null iflag=direct bs=<block_size> count=<1GB/block_size>
```

[1]Intel Cluster Studio XE: http://software.intel.com/en-us/articles/intel-cluster-studio-xe/
[2]dd: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html

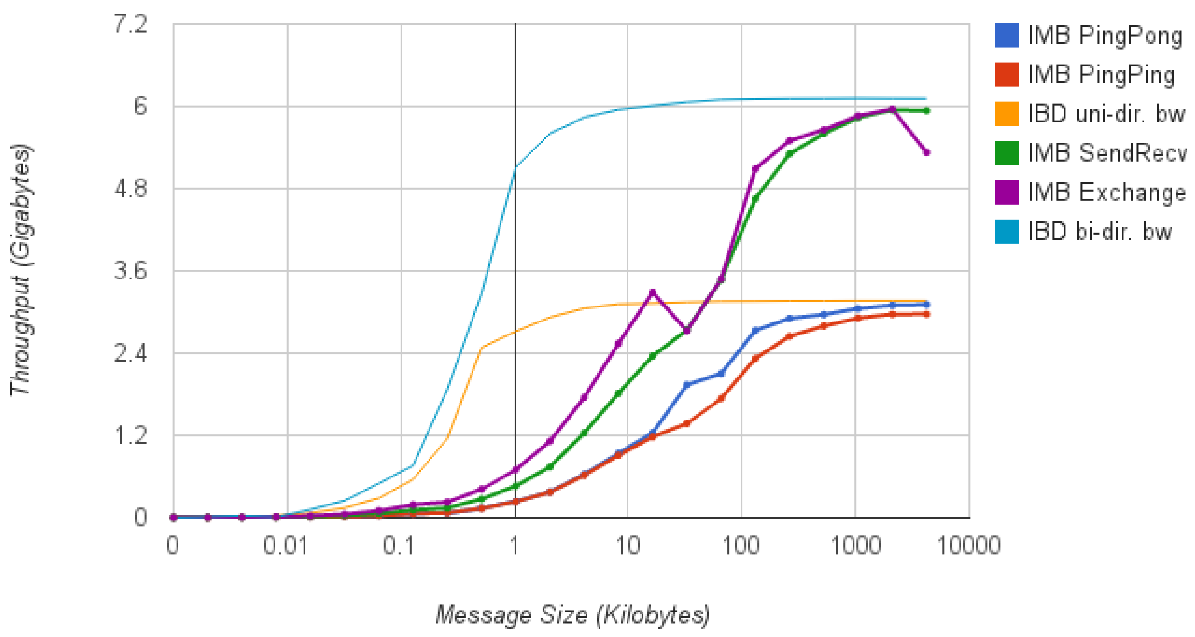


Figure 2.1: Achievable Network Throughput Reported by IMB

By contrast, with *xib* we performed a more advanced test, measuring multi-threaded, random read access speed for the same block size values. We compared the throughput reported by the two tools on one node's RAID 0 array with those obtained on the GlusterFS volume.

The results shown in Figure 2.2 indicate that the throughput of the GlusterFS volume does not exceed 110 MB/s, regardless of the block size used, even though its individual building blocks (i.e. RAID 0 arrays) are capable of delivering up to ca. 730 MB/s read throughput (as reported by the *xib* tool) and the volume is striped, hence supposed to perform even faster than that. We did not manage to overcome this performance limitation, despite all our efforts to troubleshoot the GlusterFS installation and tune its parameters.

Due to the limited time we had at our disposal, we eventually decided to continue with using GlusterFS for providing the shared-disk illusion, but prevented it from having any impact on the performance benchmarks we will present throughout this thesis. As explained in Chapter 6, we ensured this by always measuring "hot runs" only, that are non-initial runs of a query for which all the required data is available in the memory-resident I/O cache, such that the underlying file system does not need to be involved at all.

Moreover, to simulate the effects of a good partitioning scheme, all the tests were run against data stored by *clustered* databases, which in Vectorwise terminology translates into partitioning of tuples within files according to one or more attributes. This way, we benefited from all the effects data locality would have on our generated parallel/distributed plans.

In hindsight, more time should have been invested into a more careful analysis of the various DPFS offerings before going forward with a particular one. In fact, all choices made in the second part of Section 2.2 should be revisited. For example, if no suitable parallel file-system can deliver the performance sought, explicit partitioning can be employed in Vectorwise together

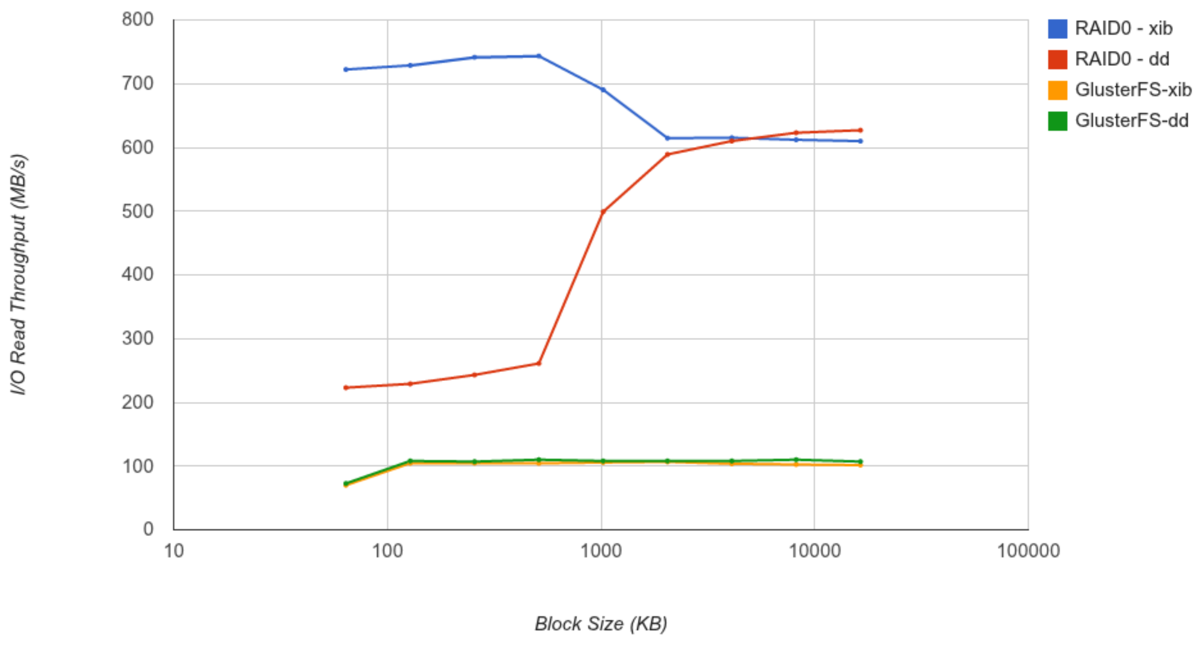


Figure 2.2: I/O Read Throughput

with an abstraction layer in the Buffer Manager, that is responsible for granting (possibly remote) data requests. In any case, however, all the work presented henceforth is completely independent on the way the "Shared-Disk" assumption is fulfilled and therefore remains valid.

2.4 Distributed Vectorwise Architecture

Figure 2.3 presents the high-level architecture of our proposed version of distributed Vectorwise. For simplicity, we followed the *Master-Worker* pattern, with the Master node being the single access point for client applications. The main advantage of this approach is that it hides from clients the fact that the DBMS they connect to actually runs on a computer cluster.

It is hence inevitable that the Master node must become involved in all queries issued to the system, in order to gather results and return them to clients, at a minimum. However, as can be seen in the architecture diagram, we assigned it more work than this, making it the only node responsible for:

- parsing SQL queries and optimizing query plans
- keeping track of the system's resource utilization, as it has knowledge of all queries that are issued to the system
- taking care of scheduling / dispatching queries, based on the above information
- producing the final, annotated, parallel and possibly distributed query plans

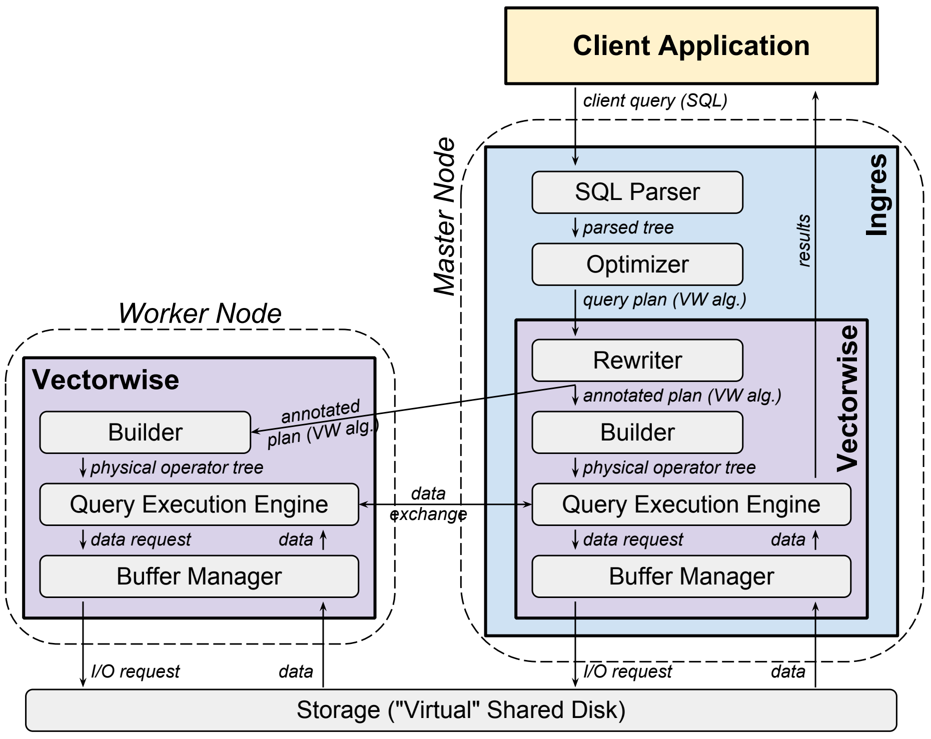


Figure 2.3: Architecture of the Ingres/Vectorwise System

- broadcasting them to all worker nodes that are involved

Moreover, depending on the system load and scheduling policy, the Master may also get involved in query execution to a larger extent than simply gathering results.

The rationale behind this decision is twofold. First, as previously explained, our system is intended for highly complex queries across large volumes of data. In this situation, the vast majority of the total query processing time is spent in the query execution stage. Parallelizing the latter is enough for achieving good speedups and performing the above processing steps sequentially is acceptable.

Second, this allows for a non-intrusive implementation, with only the following modules of the single-node version of Vectorwise needing to be modified:

- *Rewriter*: A new logical operator (called *Distributed Exchange*) is introduced in query plans, that completely encapsulates parallelism and distribution. Also, transformations of the parallelism rule and operator costs are adapted. (See Chapter 3)
- *Builder*: Every node is able to build its corresponding parts of a query execution tree and physically instantiate the new operator. (See Chapter 4)
- *Execution Engine*: The new operator is implemented in such a way to allow the Master and various Worker nodes to efficiently cooperate (e.g. exchange data, synchronize) when executing a given query. No other changes were required at this layer. (See Chapter 4)

Chapter 3

Introducing Parallelism into Query Plans

3.1 Preliminaries

In every relational DBMS, the query optimizer module is responsible for generating an efficient *execution plan* for a given query [Cha98]. The *execution plan* is usually a tree of operators that are defined on the physical level, e.g. they define if a Join is done by hashing or merging or if the Scan is indexed or not. Given that the number of equivalent execution plans is usually quite large (this number is influenced by the number of different implementations, the join order, etc.), the query optimization problem becomes a *search problem* where:

- the *search space* consists of all the functionally equivalent execution plans
- the *cost model* assigns an estimate cost to a plan
- the *search strategy* is the enumeration algorithm which explores the search space and chooses the execution plan with the best cost

The process of query optimization in the Vectorwise/Ingres DBMS is run in two distinct phases [Ani10]:

1. query optimization on the Ingres side: this is the *standard* relational optimization part where, for example, the order of the Joins is established or where the SQL operators are mapped to a physical implementation
2. *local* optimization in the Vectorwise Rewriter module

This chapter will focus only on the parallelism rule of the Vectorwise Rewriter and the modifications it requires to produce distributed query execution plans.

3.2 Vectorwise Algebra

The Vectorwise engine accepts queries expressed in the Vectorwise algebra. After parsing and optimizing a SQL query, Ingres creates a query plan composed of algebraic operators defined at the physical level. Figures 3.1 and 3.2 illustrate how Query 14 of the TPC-H set is translated from SQL into the relational Vectorwise algebra.

Figure 3.1 TPC-H query 14 expressed in SQL

```
select 100.00 * sum(case when p_type like 'PROMO%'
                    then l_extendedprice * (1 - l_discount)
                    else 0
                    end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from lineitem, part
where l_partkey = p_partkey and l_shipdate >= date '1995-09-01'
        and l_shipdate < date '1995-09-01' + interval '1' month;
```

In Figure 3.2, tuples are first scanned from the *part* table by the **MScan** operator. Then, they are joined with tuples scanned from the *lineitem* table with *l_shipdate* in September, 1995. Note that all arithmetic operations in the Vectorwise Algebra are expressed using the prefix notation. Finally, two **Project** and an **Aggr** operator calculate the revenue generated by the promotional items sold and rename its percentage of the total revenue into *promo_revenue*.

The **HashJoin01** and **MScan** are two examples of operators defined in the Vectorwise algebra whose name carries information about their implementation. This section will focus more on the syntax of the Xchg operators, as they are responsible for introducing parallelism in the Vectorwise engine. Nevertheless, an overview of the syntax of most of the Vectorwise algebraic operators is given in Table 3.1.

Figure 3.2 TPC-H query 14 expressed in Vectorwise algebra

```
Project(
 Aggr(
  Project(
   HashJoin01(
    Select(
     Select(
      MScan(
              lineitem,['l_partkey','l_shipdate','l_discount','l_extendedprice']
              ), >=(l_shipdate, date('1995-09-01'))
            ), <(l_shipdate, date('1995-10-01'))
           ), [l_partkey],
    MScan(
            part, ['p_partkey','p_type']
            ), [p_partkey]
                  ), [a=ifthenelse(like(p_type, str('PROMO%')),
                     *(-(decimal('1'),l_discount),l_extendedprice),
                     decimal('0.0000')),
                     b=*(-(decimal('1'),l_discount),l_extendedprice)]
            ), [] , [c = sum(b), d = sum(a)]
      ), [promo_revenue = /(*(d, decimal('100.00')),c]
           )
```

3.2.1 Xchg Operators

The key to parallelism in relational DBMSs is to partition data into *streams* that can be processed concurrently and independently. In the Volcano model, *exchange* operators (abbreviated **Xchg**) control the flow and number of streams in parallel query execution trees [Gra90]. They do not modify data, but are merely responsible for redistributing it. The syntax of the Xchg operators in the Vectorwise algebra is presented in Table 3.2, while a detailed description of

Name	Attributes				
MScan	*table*	*columns*			
Array	*dimensions*				
Select	*relation*	*condition*			
Project	*relation*	*columns*			
NullOp	*relation*				
As	*relation*	*name*			
Aggr	*relation*	*grpby list*	*aggr list*		
OrdAggr	*relation*	*grpby list*	*aggr list*		
Sort	*relation*	*columns*			
TopN	*relation*	*columns*	*N*		
CartProd	*relation1*	*relation2*			
MergeJoin	*relation1*	*keys1*	*relation2*	*keys2*	
HashJoin	*relation1*	*keys1*	*relation2*	*keys2*	*condition*

where:

relation	the data-flow
table	the table to scan
columns	the columns to scan/project/sort after
dimensions	boundaries in every direction when generating tuples e.g. an **Array** with dimensions $(5, 4)$ only generates integer tuples (a, b) with $0 \leq a < 5$ and $0 \leq b < 4$
condition	the condition used in the Select/Join
name	the alias name
keys	the columns that participate in the Join
grpby list	the columns to group by
aggr list	the list of columns used in the aggregation

Table 3.1: Some of the operators defined in the Vectorwise algebra

their implementation will be given in Section 4.1. We will proceed with an informal description of each:

- Xchg(N:M) this operator consumes tuples from N **Producer** streams and outputs them to M **Consumer** streams. It is mainly used for load balancing.
- XchgUnion(N:1) is an Xchg operator with a single consumer (M=1). It is the top-most Xchg operator introduced in every parallel plan.
- XchgDynamicSplit(1:M) is an Xchg operator with M consumer threads and a single producer (N=1). The tuples produced by this operator are distributed to consumers in the order of their requests (thus, in a *dynamic* fashion) and one tuple is consumed by exactly one consumer (thus, the tuples are *split* among the consumers).
- XchgHashSplit(N:M) is an Xchg operator that redistributes the data according to a hash function that is applied to one or more tuple attributes. This operator is often used to feed operators that have certain partitioning requirements, like HashJoins or Aggregations. More on this subject in Section 3.3.1.
- XchgDynamicHashSplit(1:M) it is a HashSplit with exactly one producer (N=1). Although the name of the operator might suggest a *dynamic* distribution, all tuples are assigned to a specific consumer and therefore they are deterministically split.
- XchgBroadcast(N:M) inputs tuples from N producer streams and outputs them to every of its M consumer streams.

- XchgRangeSplit(N:M) consumes tuples from N producer streams and splits them to M consumer streams, according to the range the value of the *key* attribute belongs to. The ranges are defined by the *ranges* parameter.

Name	Attributes			
Xchg	*relation*	*num_prod*		
XchgUnion	*relation*	*num_prod*		
XchgDynamicSplit	*relation*			
XchgHashSplit	*relation*	*keys*	*num_prod*	
XchgDynamicHashSplit	*relation*	*keys*		
XchgBroadcast	*relation*	*num_prod*		
XchgRangeSplit	*relation*	*key*	*ranges*	*num_prod*

where:

num_prod	number of producer streams
keys	the partitioning columns used by the HashSplit operator
key	the attribute used in the RangeSplit operator
ranges	a list of ranges that cover the domain of the *key* attribute

Table 3.2: The Xchg operators defined in the Vectorwise algebra

The reader might have noticed that in the definition of the Xchg operators in the Vectorwise algebra (Table 3.2) the number of consumer streams is omitted. The reason behind this is that this information can be obtained from one of its ancestors (i.e. the number of producer streams of the closest Xchg ancestor or 1 if there is no such ancestor). Figure 3.3 illustrates a parallel query plan for query 14 that incorporates the Xchg operators. With the exception of the family of Xchg operators, all operators in the Vectorwise algebra are oblivious to parallelism, i.e. they process tuples in the same way they would in a sequential plan.

3.2.2 Extending the Vectorwise Algebra

The family of Xchg operators of the Vectorwise algebra permits the construction of parallel execution trees that run locally, on a single machine. In order to create distributed plans, the algebra needs to be augmented with distributed Xchg operators (from now on referred to as **DXchg**) that specify on which machines and on how many streams the data is to be processed.

The main design choice that arises when adding this new operator is whether it will be a complement to the existing Xchg operator or it will replace it and provide (distributed) parallelism by itself. Figure 3.4 illustrates the two approaches to introducing a DXchg operator into query execution trees.

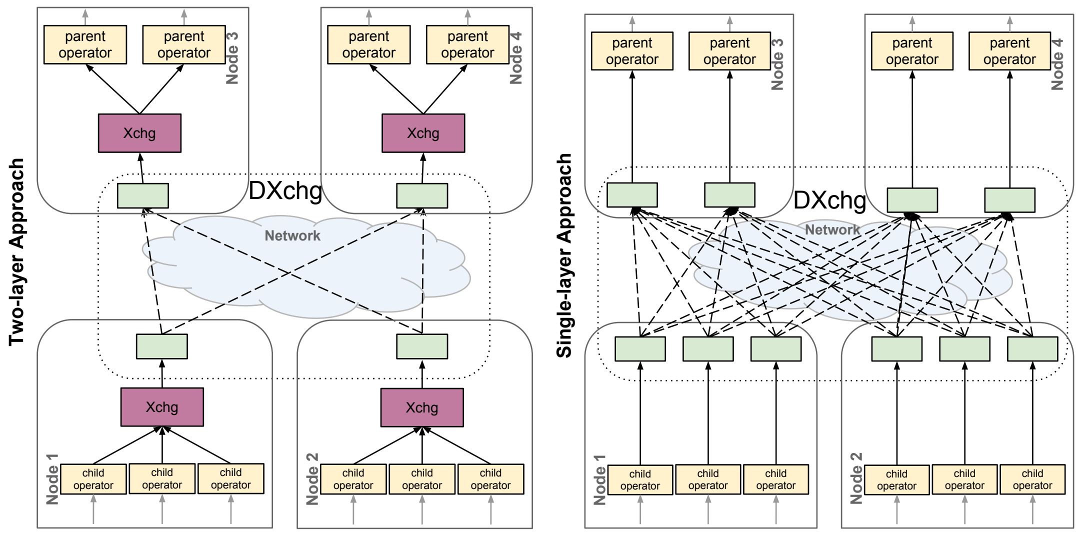
The first approach consists of two layers of distribution: a first one done across nodes, and a second one done locally using the old Xchg operator. The second approach is based on pairwise connections between every stream on every machine. Considering implementation details too, we will proceed with a comparison of the benefits and drawbacks of each approach.

Two-layer approach - Advantage The resources needed to implement this approach are considerably fewer. This is because there are two synchronization phases: one in the Xchg operator and the other in the DXchg operator. Thus, the memory complexity becomes $O(n+t)$ [1], where n is the number of nodes and t is the maximum number of threads running on one machine. This permits this approach to scale significantly more than its alternative in terms of memory usage.

[1]The synchronization between n producer streams and p consumers streams has a memory complexity in the class of $O(np)$ assuming private buffers

Figure 3.3 Multi-core parallel query plan produced by the Rewriter for TPC-H query 14

```
Project(
 Aggr(
  XchgUnion(
   Aggr(
    Project(
     HashJoin01(
      XchgHashSplit(
       Select(
        Select(
         MScan(
                  lineitem,['l_partkey','l_shipdate','l_discount','l_extendedprice']
                  ), >=(l_shipdate, date('1995-09-01'))
                ), <(l_shipdate, date('1995-10-01'))
               ), [l_partkey], 10
                         ), [l_partkey],
     XchgHashSplit(
      MScan(
               part, ['p_partkey','p_type']
               ), [p_partkey], 10
                        ) , [p_partkey]
                   ), [a=ifthenelse(like(p_type, str('PROMO%')),
                       *(-(decimal('1'),l_discount),l_extendedprice),
                       decimal('0.0000')),
                       b=*(-(decimal('1'),l_discount),l_extendedprice)]
              ), [] , [c_p = sum(b), d_p = sum(a)]
         ), 12
               ), [], [c = sum(c_p), d = sum(d_p)]
       ), [promo_revenue = /(*(d, decimal('100.00')),c]
        )
```



(a) (b)

Figure 3.4: Two approaches to distributed Xchg operators

Two-layer approach - Disadvantages

- a new rule needs to be implemented to introduce the new DXchg operator into query plans. Since all *transformations*(Section 3.3.2) that are used to introduce the Xchg operator can be used to introduce the Dxchg operator too, more than 5000 lines of possibly duplicated code would be needed in this implementation[1].
- query plans written in the Vectorwise algebra assume that all operators are identical, regardless of the stream of data they process. What this means is that two Xchg operators that belong to the second layer of distribution will always have the same number of producers (or consumers, depending whether the Xchg operator is used for merging streams into the Dxchg or splitting the stream that flows out of the Dxchg). While this is not an issue when a query is run in solitary on a homogeneous cluster, when multiple queries are running concurrently and the nodes are unbalanced in terms of load, producing such plans would only degrade the performance.

Single-layer approach - Advantages

- it preserves the simplicity and consistency of the current model, i.e. a single operator is responsible for introducing parallelism
- the modifications made to the Rewriter are straight-forward as the only difference between the DXchg and the Xchg is that they differ in signature, i.e. the producer attribute of the DXchg is a list of integers that specifies the number of producer streams on each machine

Single-layer approach - Disadvantage The most important drawback of this approach is that, when implemented, it has a memory complexity of $O(nt^2)$ (more details in Section 4.3.4).

A possible additional disadvantage of the single-layer approach would be that creating and managing such a high number of connections would add considerable overhead to the implementation. However, our choice of communication library, namely MPI, would effectively handle this problem.

Some improvements could be made to both approaches:

- the Vectorwise algebra can be extended to allow query plans with Xchg operators that have different numbers of producers on different machines, in order to alleviate the second disadvantage of the two-layer approach.
- the memory complexity of the second approach can be reduced to as low as $O(nt)$ at the expense of some synchronization overhead. Suppose, for example, that $k < t$ buffers are allocated for a particular consumer. When $p > k$ producers on one machine need to write data destined for the same consumer, $p - k$ producers will have to wait at least once to get a free buffer.

Given the short amount of time allocated to the project, these improvements do not make the scope of this thesis. Nevertheless, they were considered when a decision was made and are briefly presented in Section 6.4.

Overall, considering that the intrusiveness of the first approach would violate one of the goals and that it would also restrict the search space, we have decided to take the second approach. Implementing the second optimization to reduce the memory complexity is on our future work agenda.

[1]The current rule that introduces parallelism in the Vectorwise Rewriter is implemented in ≈ 5000 lines of Tom code (http://tom.loria.fr)

3.2.3 DXchg Operators

The novel DXchg operators have the same syntax as the original Xchg operators, with the exception that instead of a single integer, the producers are expressed as a list of pairs (n_i, p_i) denoting that the DXchg operator has p_i producers on node n_i. Table 3.3 contains a summary of the syntax of all DXchg operators implemented in the system.

Name	Attributes					
DXchg	*relation*	*num_prod*	*id_prod*	*tds_prod*		
DXchgUnion	*relation*	*num_prod*	*id_prod*	*tds_prod*		
DXchgDynamicSplit	*relation*	*prod*				
DXchgHashSplit	*relation*	*keys*	*num_prod*	*id_prod*	*tds_prod*	
DXchgDynamicHashSplit	*relation*	*keys*	*prod*			
DXchgBroadcast	*relation*	*num_prod*	*id_prod*	*tds_prod*		
DXchgRangeSplit	*relation*	*key*	*ranges*	*num_prod*	*id_prod*	*tds_prod*

where:

num_prod	number of producer nodes
id_prod	the identifiers of the producer nodes, e.g. MPI ranks
tds_prod	the number of threads (streams) running on the producer nodes
prod	a single identifier for the producer of the **Dynamic** operator

Table 3.3: The DXchg operators defined in the Vectorwise algebra

We give an generic representation that will be used in the following chapters, that also contains the number of consumers for clarification purposes:

$$< DXchg\_op > [(n_1, p_1), (n_2, p_2), ..., (n_P, p_P) : (m_1, c_1), (m_2, c_2), ..., (m_C, c_C)],$$

where a pair (n_i, p_i) means that p_i streams produce data on node n_i and a pair (m_i, c_i) means that c_i streams consume data on node m_i.

3.3 Distributed Rewriter

The Vectorwise Rewriter is responsible for annotating the query plan with data types or information that will be used in the Builder, for eliminating dead code or for optimizing the query plan by introducing parallelism. All these actions are done by using a set of rules that modify the query tree. This section will focus on the rule that introduces parallelism.

The existing *parallelism rule* in the Vectorwise Rewriter consists of three phases:

1. determining the cost for the sequential plan and getting the maximum level of parallelism *mpl* [1]

2. determining the best plan using at most *mpl* cores

3. updating the system state

Section 3.2 showed how a DXchg operator can partition data such that it is suited for distributed processing. Incorporating these operators into sequential query plans is done in the Vectorwise Rewriter using *transformations*. They provide equivalent alternatives to existing query plans (both sequential and parallel) by introducing DXchg operators at different levels of the query

[1]It is roughly the number of cores available. A more detailed description of how the maximum level of parallelism is calculated will follow in Chapter 5

trees. Section 3.3.1 will focus on how the "equivalence" is preserved, while section 3.3.2 will discuss in detail some examples of transformations.

3.3.1 Structural Properties

In order to parallelize operators like Aggregation or Join, some structural properties have to hold when partitioning the data. For example, a Join that operates on partitioned relations will produce the correct result only if both of its children are partitioned on the key columns. Following the work in [ZaLC10], a two-layer classification of such structural properties will be made:

1. **Partitioning** applies to the whole relation and is therefore a *global* property. Formally, a relation is partitioned on the set of columns P if:

   $$(\forall i, j)\Pi_P(a_i) = \Pi_P(a_j) \Rightarrow Part(a_i) = Part(a_j)$$

   where $\Pi_P(a)$ is the projection of row a on columns P and $Part(a)$ denotes the partition to which row a has been assigned. The attributes in P are called the *partition keys*. In the Vectorwise Rewriter, each partition is associated to a *parallel stream*. Consequently, the order of tuples within a partition is preserved, but two tuples belonging to different streams can be processed in any order.

2. **Grouping** (or **Clustering**) is a *local* property as it defines how data is arranged within a partition. More specifically, a relation is grouped on columns C if:

   $$(\forall i \leq j \leq k)\Pi_C(a_i) = \Pi_C(a_k) \Rightarrow \Pi_C(a_i) = \Pi_C(a_j) = \Pi_C(a_k).$$

   The columns in C are called the *grouping keys*. In [Ani10], this property is presented as global when $P = \emptyset$ and the streams in which it holds are called *clustered* streams. However, when $P \neq \emptyset$ grouping becomes a property of tuples within that partition. Therefore, it makes sense to consider grouping as characteristic of the layout of a partition (even when there is a single partition).

3. **Sorting** is another *local* property that characterizes the order of rows within a partition. Formally, a relation is sorted on columns S if:

   $$(\forall a_i, a_j) i \leq j \Rightarrow \Pi_S(a_i) \leq \Pi_S(a_j) \text{ or}$$

   $$(\forall a_i, a_j) i \leq j \Rightarrow \Pi_S(a_i) \geq \Pi_S(a_j)$$

   where $a \leq b$ states that a is lexicographically smaller or equal than b. The columns in S are called the *sorting keys*. Note that the order of columns in S matters in this case.

3.3.2 Vectorwise Rewriter

This section will get the reader acquainted with the elements of query optimization implemented in the parallelism rule.

A search state is a tuple (X, Str, Res, P, C, S). Table 3.4 provides a detailed description of the tuple values.

Transformations are used in the parallelism rule to enumerate equivalent plans of a given query. To guarantee that a tested plan produces correct results, a transformation is applied to a subtree of a query plan only if:

- the output of the transformed plan has the same structural properties as the original plan

Attribute	Description
X	subtree of the query plan
Str	how many streams the subtree has to produce per machine
Res	resources (e.g. cores) still available per machine
P	a set of keys after which the output streams have to be partitioned. $\emptyset$ denotes non-partitioned streams.
C	a set of columns after which the output partitions have to be grouped. $\emptyset$ denotes non-clustered streams.
S	a list of columns after which the output partitions have to be sorted. Am empty list denotes non-sorted streams.

Table 3.4: A search state (X, Str, Res, P, C, S) in the Vectorwise Rewriter

- the input requirements of the root operator of the subtree can be enforced on its children output streams

Table 3.5 contains the transformations implemented in the Vectorwise Rewriter. The rest of this section will provide only some insight into the most interesting transformations. A comprehensive description of them is given in [Ani10].

There are two additional **MScan** transformations that handle the cases in which output streams have to be clustered. These transformations do not modify the query plan, but only annotate the **MScan** node with some information about the ranges to scan in the clustered database and therefore were not included in Table 3.5.

Consider the 18th transformation. This subtree has to produce $Str > 1$ streams and the transformation modifies the plan by introducing a DXchgBroadcast on the right side of the Hashjoin. When executing this plan, there will be a HashJoin that operates on each stream and will join a different part of the $r1$ relation with the whole $r2$ relation. In the HashJoin operator, the right relation (or inner relation) is only used to build the hash table. Therefore, the structural properties of the relation produced by the HashJoin operator are not influenced by $r2$, so we can relax them.

Let us take another example, the 12th transformation. In the transformed plan, the aggregation is forced to produce multiple streams at first, then an **DXchgUnion** "gathers" all the results and, finally, these results are aggregated once more in order to ensure correctness. Depending on the type of aggregation performed, the parameters of the second **Aggr** operator can be set, e.g. a count aggregation would impose that the second **Aggr** sums the partial results.

Search space As the Rewriter only has to introduce DXchg operators into already optimized plans, there is little concern about the explosion of the search space. This is because:

- transformations are only applied to subtrees with strict structural requirements for their output relation
- there is a one-to-one mapping between a transformation and the enforced child requirements
- dynamic programming is used: only the generated query sub-plans with an optimal cost are stored, everything else is pruned [Ani10]

No.	Output requirements	Transformed plan	Child requirements
1	$P = \emptyset, C = \emptyset$	**Project**(T(r))	none
2	$P \neq \emptyset, C = \emptyset, S = \emptyset$	**DXHS**(**Project**(T(r)))	$P = \emptyset$
3	P, C	**Project**(T(r))	P', C' [1]
4	none	**Select**(T(r))	none
5	$P \neq \emptyset, S = \emptyset$	**DXHS**(**Select**(T(r)))	$P = \emptyset$
6	$Str = 1$ [2] $, S = \emptyset$	**DXU**(**Select**(r)))	$Str > 1$
7	none	**TopN**(T(r))	none
8	$Str = 1$	**TopN**(**DXU**(T(r)))	$Str > 1$
9	$Str = 1, S \subseteq sortlist$	**Sort**(T(r))	$S = \emptyset$
10	$P \neq \emptyset, C \neq \emptyset, S \subseteq sortlist$	**Sort**(T(r))	$P = C = \emptyset, S = \emptyset$
11	$Str = 1$	**Aggr**(T(r))	none
12	$Str = 1, S = \emptyset$	**Aggr**(**DXU** (**Aggr**(T(r))))	$Str > 1$
13	$Str = 1, S = \emptyset$	**DXU**(**Aggr**(T(r)))	$P = grpby, Str > 1$
14	$Str > 1, P = grpby$	**Aggr**(T(r))	none
15	$P \neq \emptyset$	**DXHS**(**Aggr**(T(r)))	$P = grpby$
16	$Str = 1, P, S$	**HashJoin**(T(r1), T(r2))	$P_{r2} = S_{r2} = \emptyset$
17	$Str = 1, S = \emptyset$	**DXU**(**HashJoin**(T(r1), T(r2)))	$Str > 1$
18	$Str > 1$	**HashJoin**(T(r1), **DXBC**(T(r2)))	$P_{r2} = C_{r2} = S_{r2} = \emptyset$
19	$Str > 1, P = \emptyset$	**HashJoin**(T(r1), T(r2))	$P_{r1} = K_1, P_{r2} = K_2$
20	$Str > 1, P \neq \emptyset, C = \emptyset$	**DXHS**(T(**HashJoin**(r1, r2)))	$P = \emptyset$
21	$Str = 1$	**MergeJoin**(T(r1), T(r2))	$S_{r1} = K_1, S_{r2} = K_2$
22	$Str = 1, S = \emptyset$	**DXU**(**MergeJoin**(T(r1), T(r2)))	$Str > 1$
23	$Str > 1, P = \emptyset, C = \emptyset$	**MergeJoin**(T(r1), T(r2))	$P_i = C_i = S_i = K_i$, i=1,2
24	$Str > 1, (P, C \subseteq K_1 \vee P, C \subseteq K_2)$	**MergeJoin**(T(r1), T(r2))	$P_i = C_i = S_i = K_i$, i=1,2
25	$Str > 1, P \neq \emptyset, C = \emptyset$	**DXHS**(T(**MergeJoin**(r1, r2)))	$P = \emptyset$
26	$Str = 1 \vee (Str > 1, P = \emptyset, C = \emptyset)$	**MScan**	not applicable
27	$P \neq \emptyset, C = \emptyset$	**DXHS**(**MScan**)	not applicable
28	$Str = 1, P = \emptyset$	**CartProd**(T(r1),T(r2))	none
29	$Str > 1, P \neq \emptyset, C = \emptyset$	**DXDHS**(**CartProd**(T(r1),T(r2)))	$P = \emptyset, Str_1 = Str_2 = 1$
30	$Str > 1, P = \emptyset, C = \emptyset$	**CartProd**(T(r1),**DXBC**(T(r2)))	none
31	$Str = 1$	**OrdAggr**(T(r))	$C = grpby$
32	$Str = 1, S = \emptyset$	**DXU**(**OrdAggr**(T(r)))	$P = C = S = grbpy$
33	$Str > 1, P = \emptyset$	**OrdAggr**(T(r))	$P = C = S = grpby$

	r	is the child relation of the root operator
	T(r)	is an equivalent plan of r
	K_i	are the keys used in the Join
	grpby	is the set of *group-by* columns of the aggregation operator
where:	*sortlist*	is the list of columns by which the sorting is done
	DXHS	abbreviation for DXchgHashSplit
	DXU	abbreviation for DXchgUnion
	DXDHS	abbreviation for DXchgDynamicHashSplit
	DXBC	abbreviation for DXchgBroadcast

If an attribute of the search state is not specified in the output requirements, then that transformation applies to that search state regardless of the value of the attribute. The child requirements only contain those attributes that differ from the output requirements.

Table 3.5: The *transformations* implemented in the Vectorwise Rewriter

[1] P' and C' are the columns from the child relation that are used in the projection list, see [Ani10]

[2] Although *Str* is a list of integers, we use this notation to express that the operator has to produce only 1 stream on a particular machine

Cost model With the exception of the family of DXchg operators, all of the operators defined in the Vectorwise Algebra are oblivious of distribution. Therefore, the cost formulas for these operators remain unchanged in a distributed setting and the reader is referred to [Ani10] for a more detailed description of them. This section will present the cost formulas of the novel DXchg operators by making use of knowledge of the implementation details that are discussed in the following chapters.

In the Vectorwise DBMS, the cost of an operator is calculated as the sum of:

- the *build* cost, which is an estimate of the time it takes to allocate the resources needed by all the instances of the operator. This allocation is done in the Builder module (see Section 4.2).
- the *execution* cost, which is an estimate of the maximum total time spent in the *next()* method over all the instances of the operator

In the case of a DXchg operator, the amount of memory needed depends on the size of the buffers, the number of nodes and the number of receiver threads on each node. Moreover, the build phase on two different machines is done in parallel in the distributed Vectorwise DBMS (see Section 4.4). Therefore the build cost was calculated according to the formula:

$$BuildCost_{DXchg} = k * max(ts_j|\forall j) * sum(tr_i|\forall i) * buffer\_size + c,$$

where ts_j represents the number of sender threads on node j, tr_i represents the number of receiver threads on node i and k and c are experimentally determined parameters. Note that the overall time of building the DXchg instances on all nodes is dominated by the time the node responsible for building the most instances of the DXchg operator requires (i.e. $max(ts_j|\forall ts_j)$ from the formula).

Since the DXchg operator performs all the operations that an ordinary Xchg operator does (see Section 4.3.2), the execution cost for a DXchg operator should include the cost of a classical Xchg operator and the cost of communication. However, in our implementation we make use of the non-blocking communication primitives provided by MPI and therefore we overlap the communication and computation to a large extent. Therefore the formula for the DXchg execution cost should include the network transfer time only as a lower bound:

$$ExecutionCost_{DXchg} = \begin{cases} spread\_cost + copy\_cost, & \text{if } \frac{data}{bandwidth} < T_{child} \\ \frac{data}{bandwidth} - T_{child}, & \text{otherwise} \end{cases}$$

where *spread_cost* is the cost of computing each tuple's destination, *copy_cost* is the cost of copying the tuples into the buffers, *data* is the amount of data transferred by the DXchg operator and T_{child} is the total estimated execution time of the pipeline segment in the sub-tree rooted at the child of the DXchg operator (e.g. excluding the build-phase of HashJoins or the time spent below materializing operators). Note that the *spread_cost* and *copy_cost* form the cost of an ordinary Xchg operator.

Example: Query 14 In the beginning of this chapter, a representation in the Vectorwise algebra of a sequential query plan that executes query 14 of the TPC-H query set was given. Figure 3.5 illustrates a distributed query plan for it, produced by Rewriter.

There are four nodes in the cluster, each having 12 cores. The Rewriter has decided that the plan with the optimal cost, given the state of the system, uses 18 cores on machines $n0$ and $n1$. To introduce the DXchg operators, the Rewriter has applied:

1. the 19th transformation. It requires both of its children to produce partitioned streams. Data scanned from the *part* table is partitioned using a **DXchgHashSplit** operator on

Figure 3.5 Distributed query plan produced by the Rewriter for TPC-H query 14

```
Project(
 Aggr(
  DXchgUnion(
   Aggr(
    Project(
     HashJoin01(
      DXchgHashSplit(
       Select(
        Select(
         MScan(
                  lineitem,['l_partkey','l_shipdate','l_discount','l_extendedprice']
                  ), >=(l_shipdate, date('1995-09-01'))
                ), <(l_shipdate, date('1995-10-01'))
               ), [l_partkey], 2, ['n0', 'n1'], [12, 6]
                          ), [l_partkey],
     DXchgHashSplit(
      MScan(
               part, ['p_partkey','p_type']
               ), [p_partkey], 2, ['n0', 'n1'], [12, 6]
                      ) , [p_partkey]
                   ), [a=ifthenelse(like(p_type, str('PROMO%')),
                       *(-(decimal('1'),l_discount),l_extendedprice),
                       decimal('0.0000')),
                       b=*(-(decimal('1'),l_discount),l_extendedprice)]
              ), [] , [c_p = sum(b), d_p = sum(a)]
         ), 2, ['n0', 'n1'], [12, 6]
               ), [], [c = sum(c_p), d = sum(d_p)]
       ), [promo_revenue = /(*(d, decimal('100.00')),c]
        )
```

the *p_partkey* attribute, while data scanned and selected from the *lineitem* table is partitioned using another **DXchgHashSplit** on the *l_partkey* attribute.

2. The 12th transformation is then applied to merge the 18 streams

After a distributed query plan has been produced by the Rewriter, it is then forwarded to the Builder module where physical instances of operators are built (forming a physical query tree) and, finally, these instances are started during the execution phase. An example of physical query tree for the plan depicted in Figure 3.5 can be found in chapter 6 (Figure 6.9b).

Chapter 4

Operator Implementation

4.1 Xchg Operators

The various flavours of the *exchange* operator (discussed in Section 3.2.3) are implemented in a uniform manner, following the producer-consumer pattern. The three logical entities (*producers*, *consumers* and *buffers*) are mirrored by physical (implementation) constructs and the unit of work is a vector of data. Depending on the operator type, there are one or more producers and one or more consumers, all sharing a common "state" structure which hosts the buffer pool.

Each producer is started in a new thread and since the only way to introduce parallelism within a query execution tree (QET) is by adding Xchg operators, it follows that each consumer also operates in a separate thread, as it is either part of the initial stream, or it is the (indirect) child of a producer of a different Xchg operator above. Therefore, the Xchg operator can be viewed as a synchronization point between multiple threads.

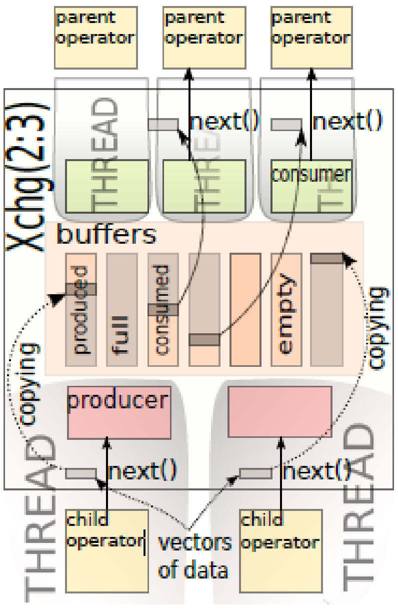


Figure 4.1: Xchg(2:3) Operator Implementation

Figure 4.1 above gives a concise, but accurate illustration of the internal structure of an Xchg operator. What follows is a brief description of the depicted components. For a more thorough explanation of the implementation details, including pseudo-code fragments and micro-benchmarks, the interested reader is referred to [Ani10].

Buffers The buffer pool allows a certain degree of decoupling between producers and consumers, thus enabling them to work in parallel. Buffers are a temporary storage space for data coming from multiple vectors and, since they preserve the same layout, they can basically be viewed as larger sized vectors.

More precisely, suppose that the tuples flowing through an Xchg operator consist of the following attributes (this example will be used on several occasions throughout this chapter):

[*ID* (decimal), *Name* (string), *Birth Date* (date), *Income* (integer)]

Assume that on a 64bit architecture the widths of the internal data types are: 8 bytes for decimals, string pointers and dates and 4 bytes for integers. As shown in Figure 4.2 below, there would be one separate memory area per expression. For variable length data types such as strings, apart from the associated pointer column, a so-called "virtual heap" (or *VHeap*) structure is used to store the actual values, which consists of multiple variable-size memory areas that are dynamically allocated whenever additional space is required.

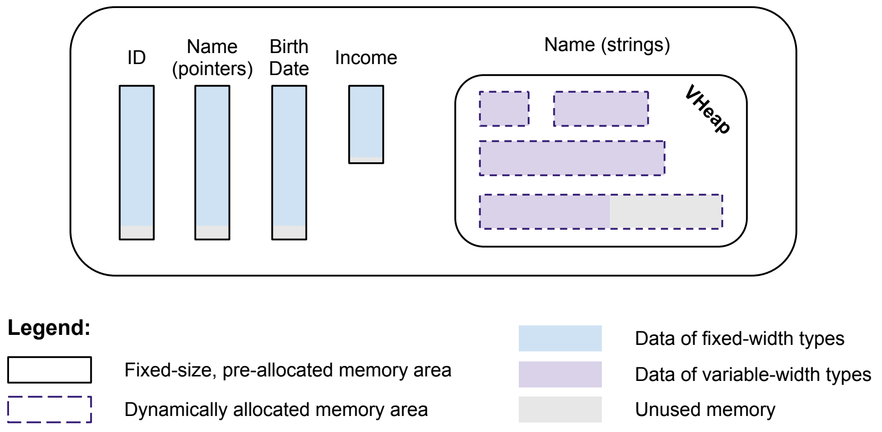

ID
Name (pointers)
Birth Date
Income
Name (strings)
VHeap
Legend:
Fixed-size, pre-allocated memory area
Dynamically allocated memory area
Data of fixed-width types
Data of variable-width types
Unused memory

Figure 4.2: Xchg Buffer Layout

During the execution of a query, there are four different states buffers cycle through:

- *EMPTY*: The buffer does not hold any data and it is not being held by any producer or consumer. Initially, all buffers are in this state.
- *PRODUCED*: The buffer is exclusively held by a producer, which is copying data from its child operator into the buffer, as long as there is enough free space.
- *FULL*: The buffer was filled and released by a producer (when it could not fit an additional vector of data), but no consumer has claimed it yet.
- *CONSUMED*: The buffer is being held by one or more consumers, which are passing data from it to their parents.

Since the buffer pool is shared, any buffer state change needs to happen as an atomic operation, hence locking is required. To reduce lock contention, it is best to combine multiple buffer acquisitions and releases whenever possible.

The size and number of buffers have an important impact on the actual degree of decoupling between producers and consumers. The more slack allowed, the more threads can be active at the same time and do work in parallel. This way, processing speed variations are better balanced (resulting in less skew), but, on the other hand, memory consumption increases, as more data needs to be materialized. Finding a good trade-off is an interesting optimization problem that is tackled in [Ani10]. As the author explains, the number of buffers should typically be proportional to the number of producers times the number of consumers and buffer size must also increase with the number of threads used.

Producers The basic task of a producer is to deliver data from its child operator to the appropriate consumer(s). Which of the consumers is appropriate for a given tuple depends on the exchange operator type and directly relates to the number of buffers that a producer must acquire. For partitioning operators (e.g. XchgHashSplit, XchgRangeSplit), a producer needs to hold as many buffers as there are consumers (because any tuple may reach any consumer). In this case, when released, buffers are tagged with their addressee's identifier. For the other exchange operator types, producers need only one buffer, with no explicit destination - it can be picked up by any consumer(s).

Producers drive the data flow in their corresponding stream by performing a loop comprising the following steps:

1. Call *next()* on their child operator.
2. If 0 tuples are returned, then break.
3. Compute the destination of every tuple in the returned vector. This either involves a range or hash computation, or is a no-op.
4. Release any buffers in which destined data would not fit (marking them as *FULL* and notifying consumers); acquire *EMPTY* ones instead. The latter operation is possibly blocking, as empty buffers might not be readily available.
5. Copy tuples into their appropriate buffers, that are now known to have enough free space.

Consumers The main steps of the consumer's *next()* routine are:

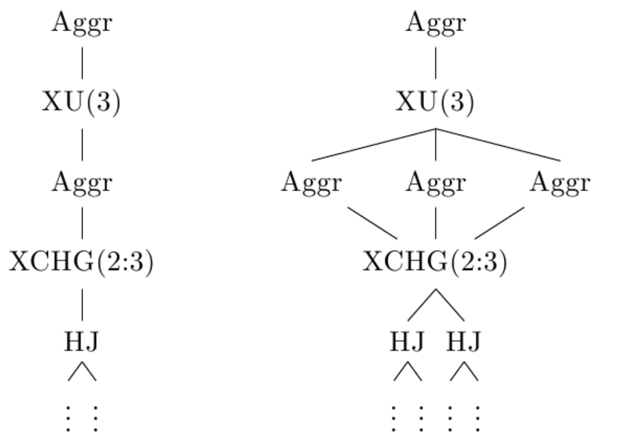
1. If there are still tuples left in the currently held buffer, return the first $max(tuples\_left, vectorsize)$ of them (as a vector) to the parent operator.
2. Otherwise, mark the current buffer as EMPTY and release it, notifying the producers.
3. Search for a new buffer marked FULL to acquire.
4. If no such buffer is found and there are no working producers left, then return 0 tuples.
5. If no such buffer is found, but there are still some producers active, then wait for a notification from them and then go to step 2.
6. Now that a new buffer was found, mark it as CONSUMED and go to step 1.

When their *next()* method is called, consumers need to deliver data to their parent operators. For this, they must either be holding a non-empty buffer (in state *CONSUMED*), or else release the one they hold (if any) and acquire a new one (steps 3-5). Again and similarly to the producers' case, acquiring a buffer is a possibly blocking operation (step 5). Once this is accomplished, the actual data delivery does not involve memory copying but only passing pointers - an O(1) operation (step 1).

Regardless of the Xchg operator type, consumers never hold more than one buffer. However, their access to it is not always exclusive. In the case of XchgBroadcast all the consumers need to read the same data. Allowing this to be done concurrently introduces the following modifications to the buffer acquisition and release subroutines in step 2: Not only *FULL* buffers may be acquired, but also those marked as *CONSUMED* The latter type of buffers takes precedence over the former, in order to release resources as soon as possible. The first consumer to acquire a *FULL* buffer changes its state to *CONSUMED*. Also, only the last consumer to finish reading the data from a buffer may mark it as *EMPTY* and release it, notifying the producers.

4.2 Building a Parallel Query Plan

As mentioned in Chapter 3, the purpose of the Rewriter module in the non-distributed Vectorwise context is to produce as output an efficient parallel plan for a given query. This *logical* query plan representation, consisting of Vectorwise algebra operators, then needs to be translated into a *physical* query execution tree, composed of object instantiations of the operators involved. As such, the parts of the query tree that need to be executed in parallel by multiple threads need to be instantiated multiple times.



(a) QET representation used. It is more concise, but implicit.

(b) Alternative notation with operators and streams shown explicitly. In this representation it is clear to see that each Aggregation and HashJoin operator are separate objects.

Figure 4.3: Different query plan representations. Courtesy of Kamil Anikiej.

Figure 4.3 shows the two different graphic representations for an example parallel query plan that are given in [Ani10]. In fact, the condensed form (a) depicts the logical, Vectorwise algebra plan, while the second representation (b) corresponds to the physical query execution tree. The Builder is responsible for interpreting the former and producing the latter. Building an Xchg operator basically means allocating and initializing the constructs depicted in Figure 4.1.

4.3 DXchg Operators

As described in the previous chapter, the DXchg operators are designed to achieve the same goal of splitting a data stream into multiple ones, but they must also allow these streams to be located on different machines. We will first present a generic implementation that always assumes the latter situation and in which, as a consequence, data exchange between producers and consumers is always done over the network. Section 4.3.5 below will present solutions for avoiding this whenever possible.

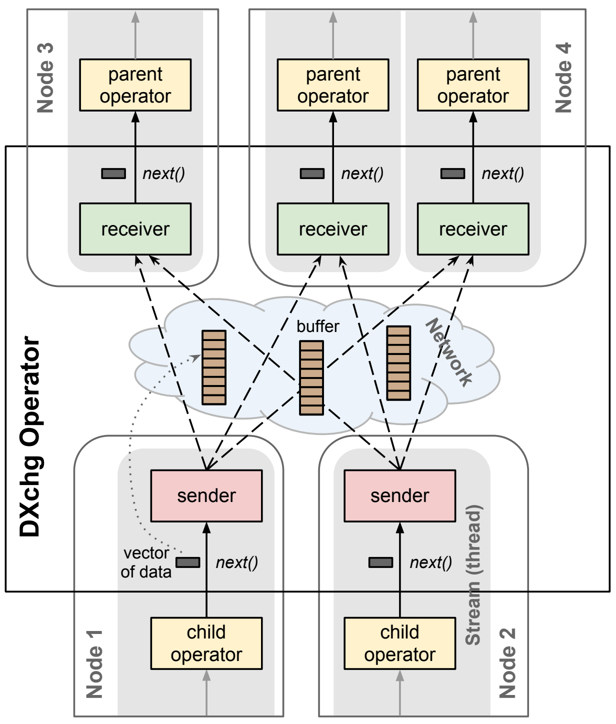


Figure 4.4: DXchg [(*n1*, 1), (*n2*, 1) : (*n3*, 1), (*n4*, 2)] Operator Implementation

To preserve the design of the Xchg operator and simplify the implementation, we chose to allow direct communication between any pair of producers and consumers (see Figure 4.4). Note that for being able to call MPI primitives from different threads concurrently, one needs to specifically ask for such support from the MPI library[1]. This decision was the determining factor in choosing which MPI implementation to use, as not all of them support this feature and, of those which do, some have been shown to exhibit performance degradation for higher levels of threading support.

For reasons of clarity, throughout the rest of this section we will refer to the producers and consumers of DXchg operators as *senders* and *receivers*, respectively. Despite the fact that, from a lower level implementation point of view, both senders and receivers are somewhat similar to regular (e.g. Select) operator instances, they cannot be regarded as separate operators of the Vectorwise algebra as they are dependent on each other and meaningless by themselves in a query execution plan.

4.3.1 Buffers

Since producers and consumers are assumed to be located on different machines, there is no such thing as shared memory that could allow them to share the same buffer pool. Instead, each of them needs to manage their own set of local buffers for sending or receiving data.

[1]by initializing its execution environment with the specialized function *MPI_Init_thread()* and the *MPI_THREAD_MULTIPLE* parameter

Sending Buffers over the Network

Conceptually, buffers are the granularity at which data exchange is done. However, as implemented in Vectorwise, the layout of the Xchg buffers (Figure 4.2) does not allow them to be sent in one go, because data is not stored in a single contiguous memory area. Therefore, separate messages would be required to send each of the buffer's expressions and VHeap vectors.

To efficiently utilize the available network bandwidth, the buffer's expressions' size would then need to be about 128KB each at least, according to the results of the network benchmarks in Section 2.3.1.

Returning to the example given in Section 4.1, the total width of such a tuple (excluding the *Name* string itself) would be $8 + 8 + 8 + 4 = 24(bytes)$. Vectorwise has a configuration parameter for specifying the maximum Xchg buffer size (not including the VHeap) which is set by default to 512KB as this is the experimentally determined optimal value for the TPC-H query set[1]. It follows from this that the contiguous memory area for expression 4 (Income) would be $\frac{4}{28} \times 512KB \approx 73KB$, which is less than the 128KB target we earlier set. Such a scenario is not uncommon, especially in the case of OLAP queries over tables that tend to be flat (storing tens of attributes).

One could address this problem by simply increasing the buffer size, but high memory consumption is a recurrent problem of our approach, as will be seen throughout the rest of the text. Also, larger buffers would impact the benefits of the pipelined execution model [BZN05].

Another important aspect is that having to send multiple messages for the same piece of work further increases the complexity of the already intricate message tagging scheme (see Section 4.5) required in such a highly-concurrent environment.

For these two reasons, we decided it is desirable to send buffers in a single message and thus match the conceptual granularity of data exchange.

Serializing Buffers

A simple, but ineffective, way of serializing a buffer would be to simply "glue" all the memory areas for expressions and the VHeap vectors together once the buffer is filled. However, this naive approach involves expensive memory copying that turns out to be unnecessary. In fact, this explicit serialization step can be avoided altogether by directly writing data into a contiguous memory zone as it gets produced, with the only observation that *in order to minimize de-serialization overhead while preserving the vectorized processing capability, interleaving expression data from different vectors needs to be avoided.*

More specifically, what the above observation states is that it is not acceptable to simply write data from vectors in the order they are produced ($\cdot$ is the concatenation symbol):

$$[data\_from\_vector_1 \cdot data\_from\_vector_2 \cdot ... \cdot data\_from\_vector_n]$$

This is because one useful side-effect of the Xchg operator would thus be lost, that is producing "full vectors" (i.e. those in which $vectorsize$ tuples are used). Imagine $vectorsize = 1024$ and a DXchgHashSplit with 20 destinations (receivers) in total. On average, $data\_from\_vector_i$ would consist of $1024/20 = 52$ tuples and this would then be the size of the vectors returned by receivers to their parent operators. As explained in [Zuk09], small vectors cause the benefits of vectorized execution to fade away and, therefore, negatively impact performance. The same argument holds for the case of a DXchgUnion above a Select operator with a small (e.g. 5%) selectivity rate.

[1]obtained on a similar architecture as the one described in Section 2.3

A solution that works well in the absence of variable length data (such as strings) is to logically partition the memory area into zones of sizes proportional to the (known) expression widths and use some pointer arithmetic to maintain per-expression cursors for writing. However, extending this solution to also cope with strings becomes problematic.

One possibility is to send a subsequent message of variable size, with the associated disadvantages mentioned above, plus the overhead of frequent dynamic memory allocations (Figure 4.5 (a)). Alternatively, one could extensively over-allocate buffers such that there is always room for all strings to be serialized after the fixed-size expression data (Figure 4.5 (b)). This latter approach is not always feasible and requires a known bound on the strings' length, but, given the conclusion of the previous sub-section, we initially decided to choose it over the former.

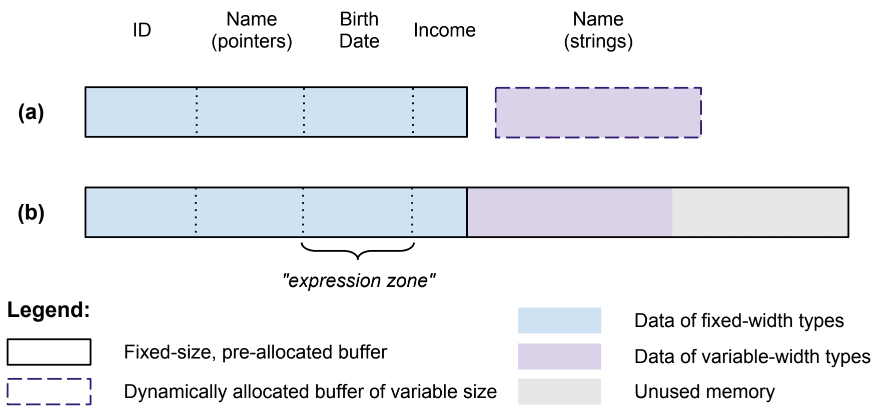

ID
Name (pointers)
Birth Date
Income
Name (strings)
(a)
(b)
"expression zone"
Legend:
Fixed-size, pre-allocated buffer
Dynamically allocated buffer of variable size
Data of fixed-width types
Data of variable-width types
Unused memory

Figure 4.5: Buffer serialization approaches: (a) two messages; (b) over-allocation

A third and preferable solution is given in the *Optimizations* section below (Figure 4.7).

4.3.2 Senders

Senders serve the same purpose as the Xchg producers and so the two are similar to a large extent. Most importantly, they are both drivers of the data flow in their own streams and route tuples to their appropriate destinations. They achieve this by operating in a loop, performing the same basic steps (Algorithm 4.1):

1. Call *next()* on child operator (line 12).
2. If 0 tuples are returned, send all non-empty buffers, broadcast termination messages to inform all receivers that this sender is done and will not send any further messages, then exit (lines 43-56).
3. Compute each tuple's destination (receiver) (line 16).
4. Release any full buffers and acquire empty ones instead.
5. Copy data into the appropriate buffers, which are now known to have enough empty space (lines 39-40).

The first two steps of the loop are trivial. For step 3, the only thing to be noted is that, when computing tuples' destinations, the total number of receiver threads (across all receiving nodes) needs to be taken into account (i.e. *dxchg_op.num_receivers*). The outcome of this step is a set of *selection vectors* that can be used to determine which receiver a certain tuple will go to. More precisely, receiver i will get the following tuples from the *data* vector returned by the *child* operator:

Algorithm 4.1 Sender Routine

```
Require: $dxchg\_op$, $num\_send\_buf > 0$, $child$

if $dxchg\_op.type = DXchgBroadcast$ then
    $num\_buckets \leftarrow 1$; $num\_handles \leftarrow dxchg\_op.num\_receivers$
else
    $num\_buckets \leftarrow dxchg\_op.num\_receivers$; $num\_handles \leftarrow 1$
end if
for $i \leftarrow 1$ to $num\_buckets$ do
    $active\_buffer[i] \leftarrow 1$
    reset_buffer($buffer[i][1]$)
end for
loop
    $n$, $data \leftarrow child$.next()
    if $n = 0$ then
        break loop
    end if
    $vector\_count[]$, $selection\_vector[][] \leftarrow$ compute_tuples_dest($n$, $data$, $num\_buckets$)
    for $i \leftarrow 1$ to $num\_buckets$ do
        $j \leftarrow active\_buffer[i]$
        if $buffer[i][j].num\_tuples + vector\_count[i] > BUFFER\_CAPACITY$ then
                                                // release (send) current buffer
            for $k \leftarrow 1$ to $num\_handles$ do
                if $dxchg\_op.type = DXchgBroadcast$ then
                    $destination \leftarrow dxchg\_op.receiver[k]$
                else
                    $destination \leftarrow dxchg\_op.receiver[i]$
                end if
                $buffer[i][j].send\_handle[k] \leftarrow$ start_sending($buffer[i][j]$, $destination$)
            end for
            $j \leftarrow ((j + 1) \bmod num\_send\_buf) + 1$ // acquire next buffer (round robin)
            $active\_buffer[i] \leftarrow j$
            for $k \leftarrow 1$ to $num\_handles$ do
                wait_for_transmission($buffer[i][j].send\_handle[k]$)
            end for
            reset_buffer($buffer[i][j]$)
        end if
    end for
    for $i \leftarrow 1$ to $num\_buckets$ do
        $j \leftarrow active\_buffer[i]$
        copy_data_into_buffer($buffer[i][j], selection\_vector[i], vector\_count[i], child$)
        $buffer[i][j].num\_tuples \leftarrow buffer[i][j].num\_tuples + vector\_count[i]$
    end for
end loop
for $i \leftarrow 1$ to $num\_buckets$ do
    $j \leftarrow active\_buffer[i]$
    for $k \leftarrow 1$ to $num\_handles$ do
        if $dxchg\_op.type = DXchgBroadcast$ then
            $destination \leftarrow dxchg\_op.receiver[k]$
        else
            $destination \leftarrow dxchg\_op.receiver[i]$
        end if
        if $buffer[i][j].num\_tuples > 0$ then
            $buffer[i][j].send\_handle[k] \leftarrow$ start_sending($buffer[i][j]$, $destination$)
        end if
        send_termination_msg($destination$)
    end for
end for
```

$\{selection\_vector[i][0], selection\_vector[i][1], \ldots, selection\_vector[i][vector\_count[i]-1]\}$

In step 4 of the loop, by full buffers we mean those in which the fixed-width data of the selected tuples from the current vector does not fit entirely. Releasing buffers translates to sending them over the network. This message announces the receiver that there are more tuples available. The opposite notification is implicit, in the sense that if the receiver is too busy with processing, it will not be ready to receive additional messages and so any sends for this destination would remain blocked. This way, a simple blocking call to *MPI_Send()* is enough to replace the whole explicit synchronization mechanism implemented in the Xchg operators.

Non-Blocking Communication To allow network traffic associated with buffers being sent to overlap with the execution of the operators below the sender in its corresponding stream (thread), and thus reduce the time spent waiting for messages to be delivered, non-blocking MPI primitives can be used. Releasing a buffer thus becomes instantaneous[1] and waiting for its delivery to complete is moved to the buffer acquisition phase.

A possible approach to implement this is to establish in the initialization phase a fixed number of buffers per destination and associate with each of them one (or more, as explained below) MPI request handle(s). Step 4 above becomes:

4' For each destination, if the active buffer is full, call *MPI_Isend()* on it (line 27), set as active the next buffer for the current destination (in a round-robin fashion) (lines 29-30) and call *MPI_ Wait()* (line 32).

Number of Buffers Let the senders' number of buffers per destination be a configuration parameter $num\_send\_buf > 1$. Figure 4.6 (a) below depicts the collection of a sender's buffers in the general case. DXchgBroadcast is again the exception, as it is desirable that the senders not write the same data multiple times, into different buffers. Consequently, a single logical destination is used, but all the num_send_buf buffers will have one request handle for each physical destination[2] (see Figure 4.6 (b)).

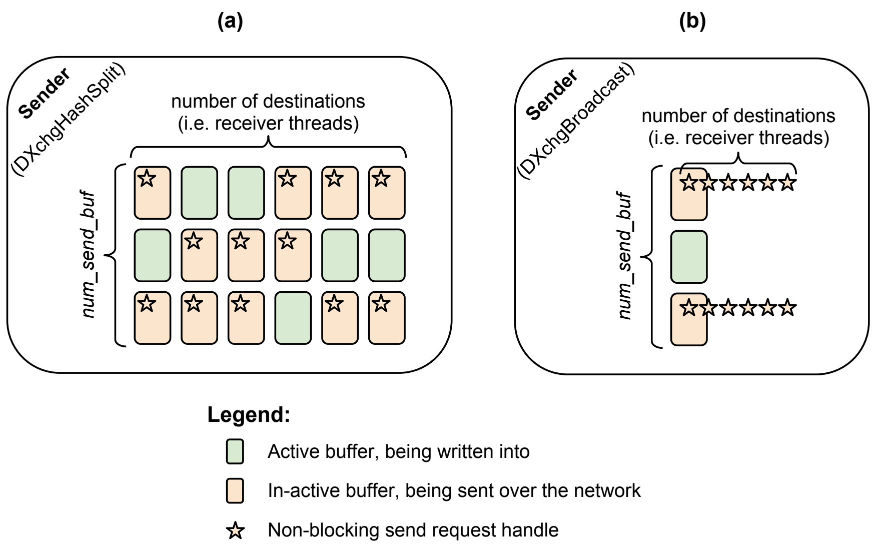

(a)
(b)
Sender (DXchgHashSplit)
Sender (DXchgBroadcast)
number of destinations (i.e. receiver threads)
num_send_buf
Legend:
Active buffer, being written into
In-active buffer, being sent over the network
Non-blocking send request handle

Figure 4.6: Senders' Buffers for (a) DXchgHashSplit and (b) DXchgBroadcast operators

[1]assuming MPI can buffer messages (i.e. non-synchronous mode)

[2]The MPI-2 standard does not require implementations to offer support for non-blocking collective operations such as *MPI_ Ibcast()*, so we implemented this as multiple point-to-point communications instead.

Writing data into buffers Finally, writing data into buffers is done in accordance to the chosen approach for serializing buffers that was described in the previous section. Despite the fact that the new buffer layout is significantly different from that of the Xchg buffers, not many changes are required for the last step of the loop. More precisely, for fixed-width data types, the same data copying (or condensing) primitive can be used, with slightly adjusted parameters (see line 3 of Algorithm 4.2). However, variable-length data needs to be copied outside this primitive into the designated buffer zone. While doing so, offsets are stored instead of the pointer column in Figure 4.2 (lines 10-11). They will allow the receiver to recreate the latter efficiently, as will be explained below.

Algorithm 4.2 copy_data_into_buffer(buf, sel_vec[], vec_cnt, $child$)

```
Require: buf: buffer with enough empty space to store vec_cnt tuples; num_expr: number
    of expressions
Ensure: tuples specified by sel_vec[] are copied from child into buf
 1: for $i \leftarrow 1$ to $num\_expr$ do
 2:     if $buf.expr[i].type$ is of fixed width then
 3:         condense_prim($child.expr[i]$, $buf.expr\_zone\_ptr[i]$, $sel\_vec[i]$, $vec\_cnt$)
 4:         $buf.expr\_zone\_ptr[i] \leftarrow buf.expr\_zone\_ptr[i] + vec\_cnt * child.expr[i].width$
 5:     else
 6:         for $j \leftarrow 1$ to $vec\_cnt$ do
 7:             $src\_ptr \leftarrow child.expr[i].ptr + sel\_vec[j] *$ width_of(pointer)
 8:             $len \leftarrow$ get_len($src\_ptr$, $child.expr[i].type$)
 9:             copy $len$ bytes from $src\_ptr$ to $buf.vldata\_zone\_ptr$
10:             $offset \leftarrow buf.vldata\_zone\_ptr - buf.vldata\_zone\_start$
11:             write at $buf.expr\_zone\_ptr[i]$ value $offset$
12:             $buf.vldata\_zone\_ptr \leftarrow buf.vldata\_zone\_ptr + len$
13:             $buf.expr\_zone\_ptr[i] \leftarrow buf.expr\_zone\_ptr[i] +$ width_of($offset$)
14:         end for
15:     end if
16: end for
```

Algorithm 4.3 reset_buffer($buffer$)

```
Require: buffer: a buffer whose data can safely be overwritten; num_expr the number of
    expressions
Ensure: buffer.num_tuples will be reset to 0 and expression cursors will point to the begin-
    ning of the (pre-computed) expression zones
1: for $i \leftarrow 1$ to $num\_expr$ do
2:     $buffer.expr\_zone\_ptr[i] \leftarrow buffer.expr\_zone\_start[i]$
3: end for
4: $buffer.vldata\_zone\_ptr \leftarrow buffer.vldata\_zone\_start$
5: $buffer.num\_tuples \leftarrow 0$
```

4.3.3 Receivers

Receivers are homologous to Xchg-consumers, having the role of delivering data from senders to the operators above them whenever their *next()* method is called. However, unlike Xchg consumers, they need to manage their own buffer space for storing incoming messages.

The main steps of a receiver's *next()* routine (Algorithm 4.4) are:

1. If there are still buffered tuples not yet consumed, return the first $max(tuples\_left, vectorsize)$ of them (as a vector) to the parent operator (lines 22-25).

2. Otherwise, if there are no active senders left, return 0 tuples (lines 7-8).
3. Wait for a message from any sender.
4. If a termination message is received, then mark its sender as inactive and go to step 2 (lines 13-14).
5. Deserialize the received message and go to step 1 (lines 16-18).

Algorithm 4.4 Receiver - DXchg operator's *next*() routine

```
Require: $buffer[]$, $active\_buffer$, $rec\_handles\_list$
Ensure: Returns a vector of tuples to caller (or 0 if no data left).
 1:
 2: $buf \leftarrow buffer[active\_buffer]$
 3: if $buf.read\_tuples = buf.num\_tuples$ then
 4:     loop
 5:         $rec\_handle \leftarrow$ start_receiving($buf$, $ANY\_SOURCE$)
 6:         $rec\_handles\_list$.add($rec\_handle$)
 7:         if detect_termination() then
 8:             return 0 tuples
 9:         end if
10:         $active\_buffer$, $rec\_handle \leftarrow$ wait_for_any_transmission($rec\_handles\_list$)
11:         $rec\_handles\_list$.remove($rec\_handle$)
12:         $buf \leftarrow buffer[active\_buffer]$
13:         if termination_msg_received($buf$) then
14:             mark_sender_finished($rec\_handle.source$)
15:         else
16:             deserialize($buf$)
17:             $buf.read\_tuples \leftarrow 0$
18:             break loop
19:         end if
20:     end loop
21: end if
22: $returned\_tuples \leftarrow$ min ($VECTOR\_SIZE$, $buf.num\_tuples - buf.read\_tuples$)
23: set_pointers_for_returned_data($buf$, $returned\_tuples$)
24: $buf.read\_tuples \leftarrow buf.read\_tuples + returned\_tuples$
25: return $returned\_tuples$
```

Non-blocking Communication Like in the sender case, the use of non-blocking MPI communication primitives (e.g. *MPI_Irecv()*) in conjunction with multiple buffers allows network traffic to take place in the background, while the thread may continue with executing the upstream operators on the available tuples. The following changes are required: Suppose a receiver has $num\_rec\_buf > 1$ buffers (configuration parameter). Initially, one buffer is chosen to be active and the above-mentioned MPI primitive is called on each of the other ones. Step 3 above is replaced with:

3' Mark the active buffer as inactive and call *MPI_IRecv()* on it (lines 5-6). The new active buffer becomes the one returned by a blocking call to *MPI_ WaitAny()* (lines 10-12).

Note that using the latter MPI primitive (which is somewhat similar to the *select()* system call in Unix) could have also been used for senders, but without any advantage over the round-robin approach, since the message have the same source and destination thread and so the MPI library guarantees to deliver them in FIFO order. For receivers, however, this is not the case, as when two subsequent non-blocking receives are matched by two messages coming from different threads or processes, then the second one might in fact be the first to complete[1].

This approach minimizes the time wasted on waiting for messages to be delivered. If, however, considerable time is still spent in this blocking primitive, then this is a strong signal that either the sender threads are not able to match the receiver's processing speed, or that network bandwidth is the bottleneck.

Deserialization What the deserialization subroutine in step 5 (line 16) involves is setting reading cursors to the beginning of expression zones (which can easily be done in $O(num\_expr)$ time, using the message preamble information) and obtaining pointer columns for each variable-width data type of the records. Having received per-value offsets from the sender, expensive scanning through the entire heap data, looking for separation symbols, is not required. Instead, the receiver can simply perform $O(num\_var\_len\_expr \times num\_tuples)$ pointer arithmetic operations, which can be done efficiently using an existing constant-addition primitive (where $num\_var\_len\_expr$ is the number of variable-length data type expressions).

4.3.4 Memory Consumption

The major problem limiting the scalability of the chosen design is the high memory consumption caused by the large number of buffers required for a thread-to-thread communication pattern.

More precisely, for the generic representation of the distributed exchange operator:

$$< DXchg\_op > [(n_1, p_1), (n_2, p_2), ..., (n_P, p_P) : (m_1, c_1), (m_2, c_2), ..., (m_C, c_C)],$$

the total number of buffers required by it on a given node x can be expressed with the following formula:

$$total\_num\_buf(n) = (p \times num\_send\_buf \times \textstyle\sum_{k=1}^{C} c_k) + (c \times num\_rec\_buf),$$

where

$$p = \begin{cases} p_k, & \text{if } \exists k \in \{1..P\} \text{ s.t. } x = n_k \\ 0, & \text{otherwise} \end{cases}$$

and

$$c = \begin{cases} c_k, & \text{if } \exists k \in \{1..C\} \text{ s.t. } x = m_k \\ 0, & \text{otherwise} \end{cases}$$

Simplifying the above formula, we get that the number of buffers required is $O(num\_nodes \times num\_cores^2)$ (assuming a homogeneous cluster). This means the memory requirement of a DXchg operator increases linearly with the number of nodes, but quadratically with the number of CPU cores per node. The latter correlation is inherited from the regular Xchg operator, which also suffers from this issue.

Table 4.1 shows the estimated memory consumption (in MB) of a DXchgHashSplit operator from all nodes to all nodes in a cluster, for $buffer\_size = 512KB$, $num\_send\_buf = num\_rec\_buv = 2$, and fixed-width data types only.

On a positive note, memory consumption for the other distributed exchange operators is actually much lower, as they may either have a single receiver thread on a single machine (e.g. DXchgUnion), a single sender thread (e.g. DXchgDynamic[Hash]Split, Table 4.2) or a single logical destination (bucket) (e.g. DXchgBroadcast). In these cases, the required space only increases linearly with the number of cores.

[1]according to http://www.mpi-forum.org/docs/mpi22-report/node54.htm

cores \ nodes	2	4	8	12	16	24	32
2	10.24	18.43	34.82	51.20	67.58	100.35	133.12
4	36.86	69.63	135.17	200.70	266.24	397.31	528.38
8	139.26	270.34	532.48	794.62	1,056.77	1,581.06	2,105.34
12	307.20	602.11	1,191.94	1,781.76	2,371.58	3,551.23	4,730.88
16	540.67	1,064.96	2,113.54	3,162.11	4,210.69	6,307.84	8,404.99

Table 4.1: Estimated memory consumption (in MB) of a DXchgHashSplit operator on one node

cores \ nodes	2	4	8	12	16	24	32
2	6.14	10.24	18.43	26.62	34.82	51.20	67.58
4	12.29	20.48	36.86	53.25	69.63	102.40	135.17
8	24.58	40.96	73.73	106.50	139.26	204.80	270.34
12	36.86	61.44	110.59	159.74	208.90	307.20	405.50
16	49.15	81.92	147.46	212.99	278.53	409.60	540.67

Table 4.2: Estimated memory consumption (in MB) of a DXchgDynamic[Hash]Split operator on one node

Even though feasible for a limited number of nodes, it is clear that the current solution is not scalable. However, in Section 6.4 we present a possible way to reduce the memory consumption of the DXchgHashSplit operator to $O(num\_nodes \times num\_cores)$, meaning similar values to those shown in Table 4.2.

4.3.5 Optimizations

We present below certain optimizations that we successfully implemented into our prototype. However, this is only a small subset of all the potential opportunities for improvement that we identified. The rest are briefly described in Section 6.4 on Future Improvements, along with a first analysis of the expected benefits of these proposed optimizations.

Use (non-distributed) Xchg operators whenever possible

As explained in Section 3.2, we embraced a uniform approach in which we completely eliminated Xchg symbols from the Vectorwise Algebra by always replacing them with DXchg operators, even when all the producer and consumer threads reside on the same node.

In cases like these, however, we can completely eliminate network traffic by simply replacing the DXchg operators' implementation with that of the corresponding (regular) Xchg operators. This way we can leverage the existing code that is known to perform well. The only downside is that this comes at the expense of increasing the parameter space for the application, since the optimal number and size of the DXchg buffers may differ from those of the regular Xchg buffers.

Only send pointers locally

When the above optimization is not applicable because the set of threads is distributed across two or more machines, we can still reduce network traffic and the burden on the MPI library by only communicating pointers between those producers and consumers that do reside on the same node.

This approach, however, reintroduces the need for explicit synchronization mechanisms (locking, waiting and notifying) when acquiring and releasing buffers. The reason for this is that while for buffers destined to remote nodes the completion of the call to *MPI_ Wait()* means that the buffer was successfully sent and can be safely overwritten, this does not hold for those buffers sent locally, since the latter event only implies that the pointer was successfully delivered, but says nothing about whether or not the consumer finished reading data from the buffer. Therefore, before proceeding to writing new data into it, the producer must wait for a notification from the consumer, which the latter must post when releasing the buffer.

PAX buffer layout and MPI Persistent Communication

Considering the observations made in Section 4.3.4, any means of limiting memory consumption rank high on the priority list of optimizations.

In Section 4.3.1 we raised the problem of finding an efficient way to serialize buffers for communication over the network when variable length data types are involved. The only solution given that would not involve more than one message per buffer was to pessimistically allocate enough memory to fit all the strings in a worst-case scenario. While this approach has the advantage of maximizing network bandwidth efficiency (since the exact number of useful bytes can thus be sent), it significantly aggravates the above-mentioned problem of high memory consumption.

Fortunately, using a similar layout to what is known in the relational database storage literature as the Partition Attributes Across (PAX) model [ADH02], it becomes possible to directly store both fixed and variable-length data types into a single, fixed size, contiguous memory area, in a both time- and space-efficient manner. In this case, the memory consumption estimates shown in Table 4.1 hold true for variable-length data types as well.

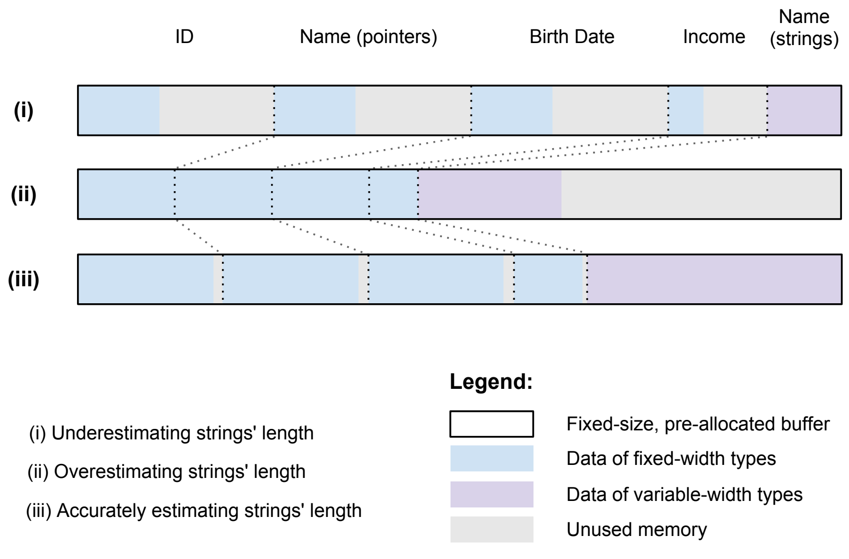

ID
Name (pointers)
Birth Date
Income
Name (strings)
(i)
(ii)
(iii)
Legend:
(i) Underestimating strings' length
(ii) Overestimating strings' length
(iii) Accurately estimating strings' length
Fixed-size, pre-allocated buffer
Data of fixed-width types
Data of variable-width types
Unused memory

Figure 4.7: PAX Buffer Layout

One downside of this approach that is inherent in the PAX layout is that network bandwidth utilization is negatively impacted. This is because fractions of the buffer at the end of its expression zones may often go unutilized, but will nevertheless be sent over the network. However, this percent of "trash" information may be reduced by obtaining good estimates on the strings' length, such that they can be treated as normal (fixed-width) data types and dividing the buffer accordingly. Moreover, this partitioning need not be fixed, but may be dynamically adjusted throughout query execution by maintaining a running average of strings' size and communicating expression offsets to receivers as a message preamble. Figure 4.7 illustrates how the expression zones can be adjusted to minimize the percent of wasted buffer space.

Finally, having fixed size buffers was the only missing ingredient required for taking advantage of the MPI performance-enhancing feature that is "Persistent Communication"[1]. The use of persistent requests and the associated MPI primitives reduces communication overhead associated with redundant message setup in programs which repeatedly call the same point-to-point message passing routines with the same arguments. This scenario description maps perfectly to our situation, as Send operators have distinct buffers for every different destination, be it a specific Receive thread (i.e. for DXchgHashSplit, DXchgRangeSplit), or a generic tag on which multiple Receivers listen (i.e. for DXchgUnion, DXchgBroadcast).

Depending on the buffer size, database size and query pattern, the number of similar messages (same source, destination, tag, buffer and size) may amount up to tens of thousands, so eliminating message setup overhead may seem like a promising improvement. However, in practice, this optimization only renders a 5-15% improvement for messages that are shorter than 8KB [2]. This means only local communication would benefit from having persistent requests, when only pointers to buffers are exchanged.

4.3.6 Microbenchmarks

To validate our implementation of the DXchg operators, we have conducted three experiments on the available cluster nodes described in Section 2.3. In all cases num_send_buf and num_recv_buf were both set to 2 (experimentally detected as optimum). Finally, the execution times shown or involved were computed as the average of three successive runs.

Experiment 1

The first set of experiments is meant to show what percentage of the network throughput reported by the Intel MPI Benchmark (IMB) is achieved with our implementation. For this, we used a custom query with the following plan:

$$\text{Test Query 1: } Array \rightarrow DXchgUnion \rightarrow NullOp$$

The *Array* and *NullOp* operators are implemented for debugging purposes only. *Array* is used to generate synthetic tuples at a fast rate instead of reading them from a table. Its parameters determine the number and width of the tuples returned. We chose them such that 1.2 billion tuples of size 24 bytes (three 8-byte decimal attributes) are produced on one node and sent over the network to a different node by the DXchgUnion operator. *NullOp* simply discards any tuples received and is meant to add as little computational overhead as possible such that the execution time of this query is completely network-bound.

Figure 4.8 shows the results obtained for different numbers of sender threads and increasing buffer sizes. The throughput values are computed by dividing the (average) execution time of *Test Query 1* by the total number of bytes processed (which is $24B \times 1.2 \cdot 10^9 = 28.8GB$) and their accuracy was validated by real-time observations performed with the *collectl*[3] Infiniband traffic monitoring tool.

[1] http://www.mpi-forum.org/docs/mpi-11-html/node51.html
[2] https://computing.llnl.gov/tutorials/mpi_performance/#Persistent
[3] http://collectl.sourceforge.net/

The conclusion is that for sufficiently large buffers (of at least 512KB), the end-to-end useful data throughput achieved is ca. 83% of that reported by the Intel MPI benchmark. This is a satisfactory result, considering the fact that the actual number of bytes being transferred is slightly larger (due to message preamble information and unused buffer space) and that the benchmark uses the *MPI_ THREAD_ SINGLE* environment, as opposed to *MPI_ THREAD_- MULTIPLE*, with its associated overhead for ensuring primitive thread-safeness.

Interestingly, it also follows from this experiment that the non-distributed system's default buffer size for the Xchg operators (i.e. 512KB) also works best from the network communication point of view.

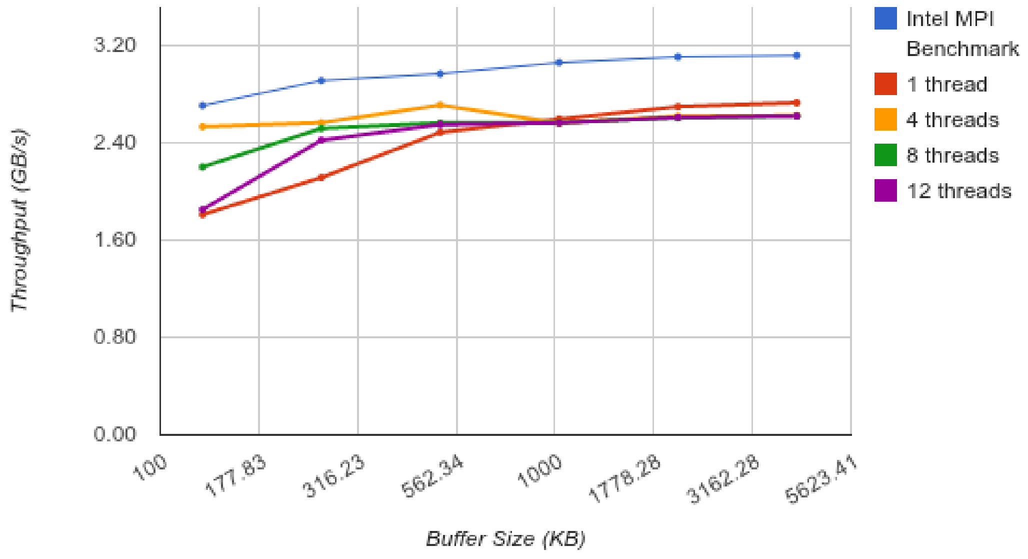


Figure 4.8: End-to-End Network Throughput

Experiment 2

The second set of experiments aims to test the single-node parallelization capability of the DXchg set of operators, by comparison with the regular Xchg ones. The fact that on a single node, the former can simply be replaced with the latter (see Section 4.3.5) does not render this experiment useless, because it gives a strong indication of how much additional overhead we introduce for the cases when multiple nodes are involved. Along the way, we also assess the usefulness of other above-mentioned optimizations, except for the PAX buffer layout, which is built-in and thus always enabled.

For these experiments we created another debugging-purpose operator, namely *SlowOp*, which simply slows down the data flow of its stream by adding a configurable delay in its *next()* call[1]. This is useful for creating artificial queries that are perfectly parallelizable (close to 100% efficiency) by reducing the data rate of producer streams to values that are lower than

[1]The delay is obtained by a call to the *usleep()* function and the SlowOp parameter is interpreted as the number of microseconds to sleep.

the maximum achieved network throughput (see the first experiment). Using *SlowOp*[1], we constructed the following two queries:

$$\text{Test Query 2: } Array \rightarrow SlowOp \rightarrow DXchgUnion \rightarrow NullOp$$

$$\text{Test Query 3: } Array \rightarrow SlowOp \rightarrow DXchgHashSplit \rightarrow NullOp \rightarrow DXchgUnion$$

Test Query 2 is a modified, single-node version of *Test Query 1*, with slower producer streams. *Test Query 3* features the DXchgHashSplit operator. A DXchgUnion is still required to create multiple receiver streams for DXchgHashSplit, but the NullOp in between ensures that DXchgUnion does not influence the overall time (since it does not process any data).

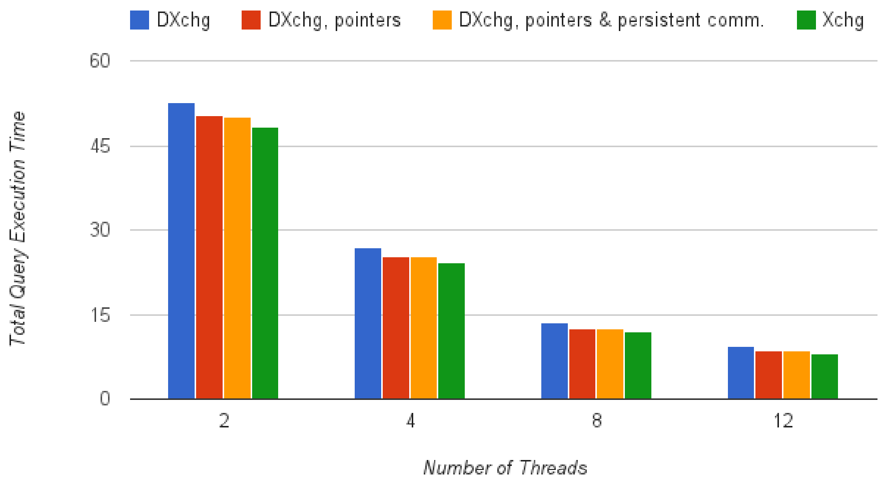


Figure 4.9: Test Query 2 Execution Time

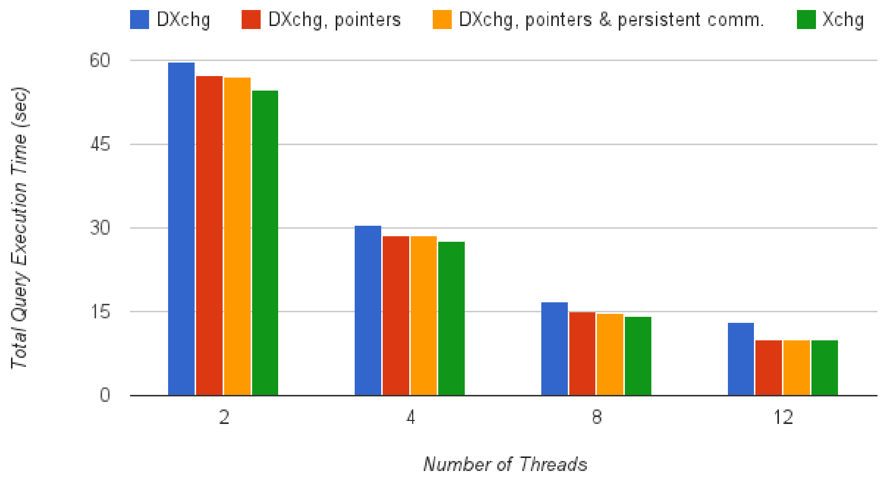


Figure 4.10: Test Query 3 Execution Time

The results shown in Figure 4.9 and Figure 4.10 are consistent and their main message is that our implementation of the DXchg operator closely matches that of the regular Xchg operator in

[1]parameter value: 100

terms of efficiency. As expected, the latter is always faster, but only by 4% at most (compared to our "pointers & persistent communication" version). The optimization of only sending pointers to local buffers yields a 4 - 11% improvement[1]. Finally, the use of MPI persistent requests hardly makes any difference (at most 1% improvement).

Experiment 3

We used *Test Query 3* again for the last experiment to present the efficiency of the DXchgHashSplit operator as a function of the combined throughput of the producer streams on a given node (henceforth referred to as X). The different values of X were obtained by varying the amount of delay introduced by the *SlowOp* operator on each stream. This experiment is meant to help the reader understand when to expect the DXchgHashSplit operator to achieve good speedup and when not to.

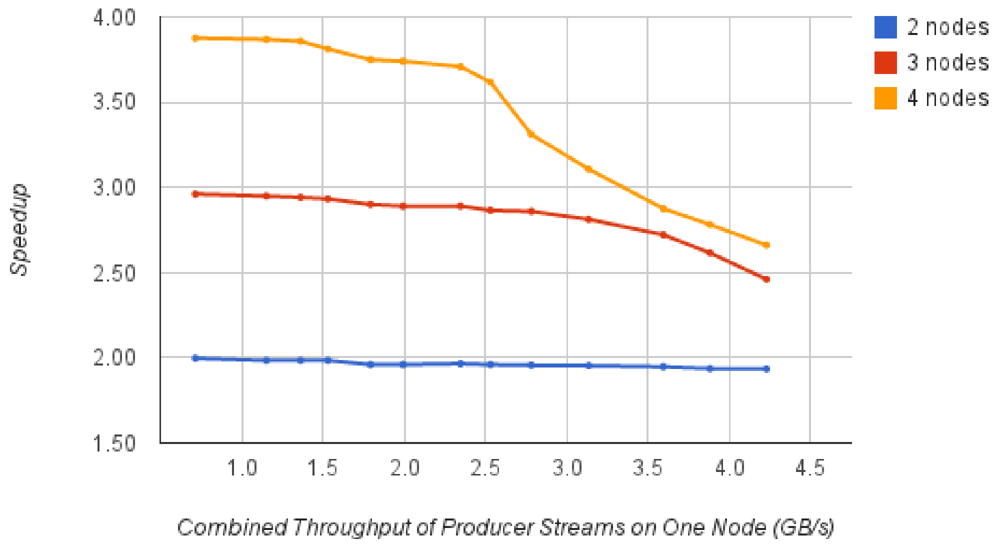


Figure 4.11: Experiment 3: DXchgHashSplit performance vs. the rate at which tuples are produced

The fact that the speedup on 2 nodes (the blue line in Figure 4.11) is still close to perfect even for X values much higher than the maximum achievable network throughput found in our first experiment (i.e. ca. 2.4 GB/s) may seem confusing. The explanation is that, in fact, only half of the data is actually sent over the network, while the other half is sent locally, as pointers, thus having a much smaller impact on network utilization. What this means is that the network throughput of the DXchgHashSplit operator on a given node (call it Y) can be estimated with the following formula:

$$Y = X \cdot \frac{num\_nodes - 1}{num\_nodes}$$

In our case, for Y = 2.4 GB/s we get that the X values on 2, 3 and 4 nodes for which the network limitation should be hit are 4.8, 3.6 and 3.2 GB/s respectively. It now becomes clear

[1]We observed that on network-bound queries the benefit of the "pointers" optimization is even more significant.

why the speedup on 2 nodes does not drop in our graph. On 4 nodes, however, we observe that the network becomes the bottleneck sooner (around 2.5 GB/s) than the predicted X value of 3.2 GB/s. This is no surprise when considering the three observations below.

First, the above formula for Y ignores the impact of the local exchange of messages carrying pointers to buffers on the network throughput achievable by the DXchg operator and therefore only gives an under-approximation of the latter. Second, we also completely ignored the inbound network traffic (which is theoretically equal to the outbound one) because the network devices are capable of bidirectional communication, but the results in Section 2.3.1 show that bi-directional throughput is not exactly twice as high as the uni-directional one. Finally, we assumed no data skew and a uniform hash function, such that data gets evenly distributed among the receiver nodes. However, we observed that this does not happen in practice, even though the *Array* operator generates sequential, hence uniform data.

4.4 Building a Distributed Query Plan

In the distributed context, the Rewriter on the Master node produces as output a possibly distributed query plan, depending on the chosen policy and the load of the system. Once such a plan is found, any nodes involved in the execution of the query need to build their relevant part(s) of the QET. For this, they need to be informed about the new query's arrival and they must at least receive the part of the query plan that was assigned to them.

Since the query plans produced by the Rewriter are of a relatively small size (few kilobytes), there is no need to burden the Master node with breaking the plan into multiple parts and sending them to the corresponding worker nodes one by one. Instead, the entire plan can simply be broadcast to all the nodes involved and they can then independently identify the parts of it they need to build. This information can be derived from the parameters of the DXchg operators encountered while traversing the tree, according to the recursive algorithm 4.5 that is briefly explained below. Consequently, the builder phase is done in a distributed fashion.

In lines 1-7 of the algorithm, the number of threads for the subtrees below the current operator is computed. Unless the latter is a DXchg operator, the number of threads remains unchanged. Otherwise, it either becomes 0 (if the node is not involved in this DXchg operator), or the specified number of producer threads on this node. The Builder function is then called recursively on the children of the current operator (if any), with the number of threads found above (lines 11-13). When encountering a DXchg symbol for which the optimization described in Section 4.3.5 is applicable, the corresponding Xchg operator is built instead (lines 16-18). Otherwise, the corresponding senders and/or receivers are instantiated (lines 20-25).

The Master node takes part in the execution of all the queries that arrive to the system. This is inevitable, as it must at least gather the results and deliver them to the upper layers (Ingres). Since it must always build the operators of the top-most stream, it will call the *Build_Operator_Tree* procedure setting *alg_op* to the root operator symbol of the query plan and *num_threads* to 1. Any other participating nodes will use the same value for *alg_op*, but will set *num_threads* to 0 initially.

Figure 4.12 gives an example of a distributed query plan and the corresponding output of the builder on all the participating nodes.

Starting the sender threads

To start the execution of the query, the *next()* method of the root operator on the master node is called. According to the Volcano model, the root operator will call the *next()* method on

Algorithm 4.5 Build_Operator_Tree($alg\_op, num\_threads, node\_id$)

```
Require: $alg\_op$: the Vectorwise Algebra operator that is the root of the tree to be built;
    $num\_threads$: the number of threads the tree is to be built for; $node\_id$: the rank of the
    process (node) executing this function.
Ensure: The part(s) of the QET that are to be executed by node $node\_id$.
 1: if $alg\_op$.type ≈ $DXchg*$ then
 2:     if $node\_id$ is among the producers of $alg\_op$ then
 3:         $num\_threads\_child \leftarrow$ number of producer threads on $node\_id$ for $alg\_op$
 4:     else
 5:         $num\_threads\_child \leftarrow 0$
 6:     end if
 7: else
 8:     $num\_threads\_child \leftarrow num\_threads$
 9: end if
10:
11: for all $child \in alg\_op$.children do
12:     Build_Operator_Tree($child, num\_threads\_child, node\_id$)
13: end for
14:
15: if $alg\_op$.type ≈ $DXchg*$ then
16:     if $node\_id$ is the only producer and the only consumer node for $alg\_op$ then
17:         $xchg\_op \leftarrow$ corresponding Xchg operator for $alg\_op$
18:         instantiate_operator($xchg\_op$)
19:     else
20:         for $i \leftarrow 1$ to $num\_threads$ do
21:             instantiate_receiver($alg\_op$)
22:         end for
23:         for $i \leftarrow 1$ to $num\_threads\_child$ do
24:             instantiate_sender($alg\_op$)
25:         end for
26:     end if
27: else
28:     for $i \leftarrow 1$ to $num\_threads$ do
29:         instantiate_operator($alg\_op$)
30:     end for
31: end if
```

its children and so forth. When the *next()* method of a Receiver is called for the first time, it starts a new thread for each local Sender that belongs to the same logical DXchg operator (if any). The threads thus spawned will run the Sender routine outlined in Algorithm 4.1.

All "orphan" Senders (i.e. those Senders of a DXchg operator on a given node, for which there are no Receivers on the same node) will have their threads started at once, just before the execution phase begins.

4.5 Message Addressing Scheme

While it is easy to understand that the dashed/dotted arrows in Figure 3.4 (b), Figure 4.4 and Figure 4.12 denote logical point-to-point connections between the corresponding senders and receivers, we still owe to the reader a better explanation for how these connections can actually be implemented with the chosen network communication library.

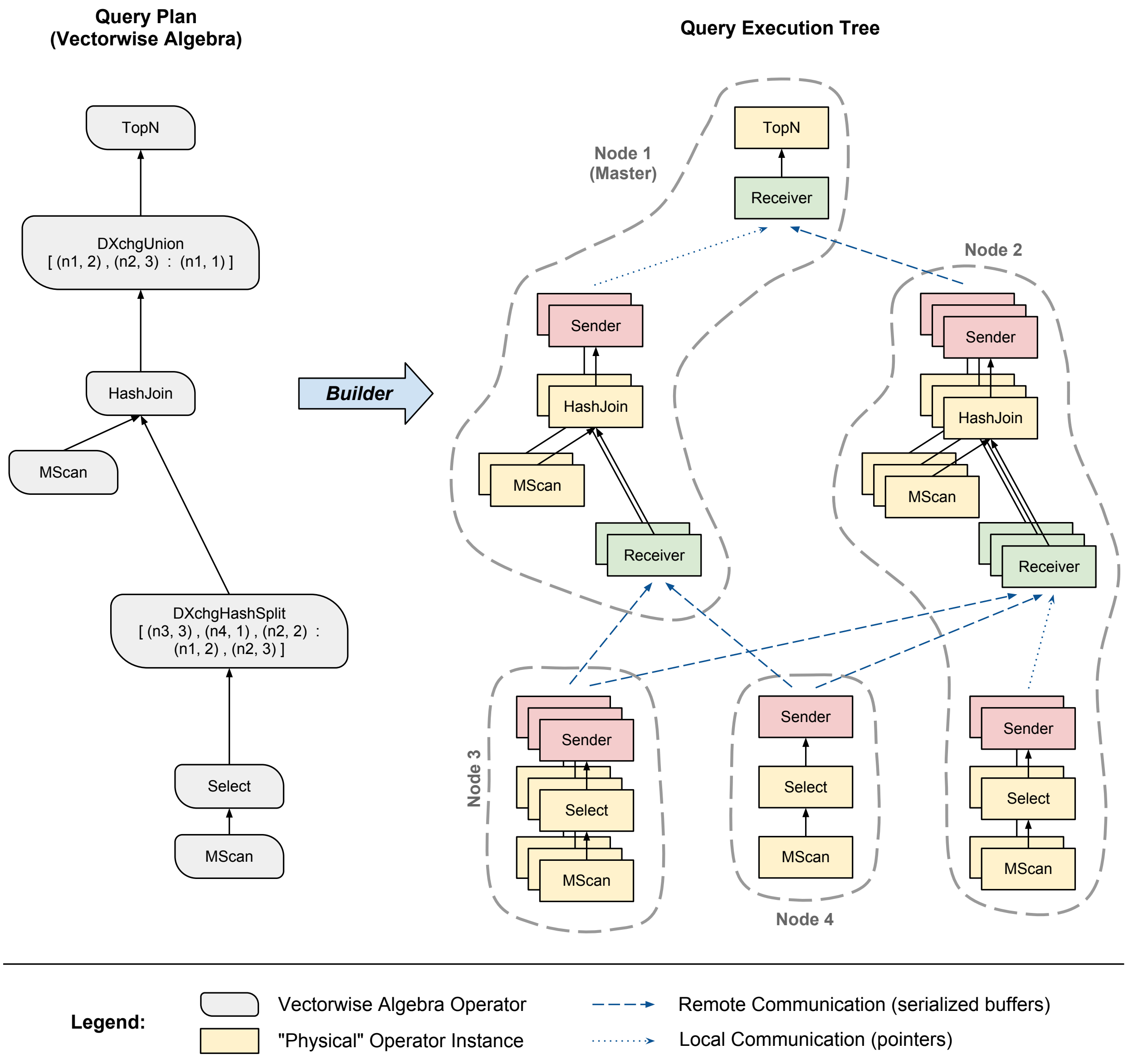


Figure 4.12: Building distributed query plans.

The above concept of a connection does not have a direct mapping in MPI terminology. Instead, a combination of the following message labeling information can be used to compose a *logical address*: (*destination_node, message_tag, MPI_communicator*). The problem then translates to assigning receivers globally unique logical addresses that are known to the relevant senders.

Any given node might be running multiple concurrent queries and, for each, it might be involved in multiple DXchg operators with one or more receiver threads. A globally unique address for a receiver needs to encode all this information. Since on each node, the Builder has knowledge of the entire query plan, it is capable of computing these addresses and passing them to senders and receivers while instantiating them, provided that it is given a unique query identifier.

Our solution is to have the Master node compute a serial number for each query - call it *query_id* - and attach it to the query plan before broadcasting it to all nodes involved. On each machine, the Builder can assign every DXchg operator within that query a unique id - call it *dxchg_id*. Then, when sending a buffer to a destination node *n*, *message_tag* can be computed as a function (e.g. concatenation) of both *query_id* and *dxchg_id*, as well as the receiver thread number on node *n*, assuming that the ranges of these parameters are known. The problem with this approach is that MPI message tags are defined as 4-byte positive integer values and care must be taken to ensure overflows do not cause messages to reach the wrong destinations.

A more elegant, scalable and less error-prone solution is to create separate MPI communicators for each query and then, within a query, separate communicators for each distributed exchange operator. This way, the message tag would only be used to distinguish between the different receivers on a destination node. We leave this as a possibility for future work.

Chapter 5

Query Scheduling

Load balancing in multiprogrammed parallel processing systems has been extensively studied in the literature [Sev94, PS95]. Depending on the characteristics of the system (on-line vs. off-line, closed vs. open, etc.) and the nature of the jobs/tasks (preemptive vs. non-preemptive, service times with a known vs. unknown distribution, prioritized vs equal jobs, etc.), several static and dynamic scheduling algorithms have been proposed [Sev94].

In parallel databases systems, however, the consensus is that dynamic load balancing is mandatory for effective resource utilization [Rah95, MRS02]. Various algorithms have been proposed for these systems depending on the different levels of parallelism: inter-transaction, inter-query, inter-operator and intra-operator parallelism [Rah93]. This chapter will discuss the intra-operator load balancing techniques currently implemented in the non-distributed Vectorwise DBMS, as well as two novel scheduling policies for the distributed version of Vectorwise.

5.1 Metrics

The *objectives* of a job scheduling policy in a parallel database system can be divided in two categories [FR98]:

- **maximize** the *throughput* (expressed as the number of queries per time unit) and/or the *resource utilization*
- **minimize** the *response time* (i.e. the time elapsed between the moment a client issues a query and the moment he receives the results) and/or the *makespan* (the total execution time of running a predetermined set of queries)

5.2 Elements of job scheduling in the Vectorwise DBMS

The unit of work (job) in the Vectorwise DBMS is a *query.* Queries processed by the execution engine are independent, their service time distribution is unknown and differences in workloads are significant (e.g. queries 06 and 13 of the TPC-H query set have execution times that differ by a factor of 100). Moreover, queries have different degrees of resource utilization and of parallelism at different levels of the query plan.

Given the above-mentioned characteristics of the jobs in the Vectorwise DBMS (an open on-line system), *equipartition* has been chosen as the scheduling strategy. Equipartition divides the resources equally between concurrent queries and has been proven to be efficient for a large class of workloads and service time distributions [PS95]. Moreover, equipartition relies also on the operating system to schedule queries when the number of threads is larger than the number of processors and therefore it better utilizes the system's resources when queries exhibit varying levels of parallelism during their execution.

Nevertheless, the reader should keep in mind that resource management is an open research subject and all the solutions implemented for the scope of this project are limited, but, at the same time, sufficient for our purposes.

5.2.1 Job scheduling in the non-distributed DBMS

When a query is received in the non-distributed version of the Vectorwise DBMS, an equal share of resources is allocated in the beginning to it. This amount is called the *maximum parallelism level* (*mpl*) and it is calculated as the maximum number of cores available (possibly multiplied by an *over-allocation factor*) divided by the number of queries running in the system:

$$mpl = \begin{cases} nC, & \text{if } nQ = 0 \\ \frac{nC * OAF}{nQ}, & \text{otherwise} \end{cases}$$

where nC is the number of cores available, nQ is the number of queries already running in the system and OAF is the over allocation factor and it is by default set to 1.25. Notice that *mpl* is equal to nC when a single user is in the system, therefore making full use of the system's resources.

Then, when transformations are applied, the *mpl* is used as an upper bound to the number of parallel streams in generated query plans. Finally, once the plan with the best cost is chosen, the state of the system is updated.

Information about the number of queries running in the system and the number of cores used by each query is stored in this state. Note that this information is a mere approximation of the CPU utilization. This is because, depending on the stage of execution and/or the number and size of the Xchg buffers, a query can use less cores than the number specified, but it can also use more cores for short periods of time.

5.2.2 No-distribution policy

The *no-distribution* policy tries to keep the network traffic at a minimum and, at the same time, process linearly more queries than the non-distributed DBMS. It does so by processing each query on a single node and then communicating the results through the master node, to the client.

The idea behind the no-distribution policy is to choose a single least loaded node in the cluster, calculate the *mpl* relative to the node, apply the transformations to generate a parallel query plan that runs entirely on that machine and, finally, add a DXchg(1:1) operator on top of it, if necessary. This operator sends the results to the master node so that they can be communicated to the client. Algorithm 5.1 contains a pseudo-code of the policy.

The challenging part of this algorithm is how to determine the "least loaded" node. In order to determine the load of a particular node, a state of the system has to be maintained, just like in the non-distributed version of the Vectorwise DBMS. As all of the queries are executed through the master node, it makes sense to keep this information there. To determine on which node to run a received query, the master node assigns a *load factor* to every node in the cluster:

Algorithm 5.1 no-distribution(*query*, *state*)

```
Require: query: a sequential query plan; state: the current state of the system;
Ensure: An equivalent parallel plan
1: $ll\_node \leftarrow get\_least\_loaded(state)$
2: $mpl \leftarrow get\_mpl(ll\_node, state)$
3: $resources \leftarrow array(0)$
4: $resources[ll\_node] \leftarrow mpl$
5: $add\_state\_load(state, resources)$
6: $par\_plan \leftarrow generate\_best\_plan(query, resources)$
7: $actual\_used \leftarrow get\_resources(par\_plan)$
8: if $actual\_used < resources$ then
9:     $remove\_state\_load(state, resources - actual\_used)$
10: end if
11: if $ll\_node \neq master\_node$ then
12:     $par\_plan = DXchange(par\_plan, 1, [ll\_node], [1])$
13: end if
14: return $par\_plan$
```

$$L_i = w_t * T_i + w_q * Q_i,$$

where:

- T_i is the *thread load* on node i. It is defined as the estimated total number of threads running on i divided by the number of available cores on i.
- Q_i is the *query load* on node i and it is defined as the number of running queries on i divided by the total number of running queries in the system. This value and the thread load are the only available information about the state of the system. Since the thread load is only an approximation of the current utilization of a particular node, the query load becomes a complement of it and acts as a tiebreaker in some cases (e.g., when the thread load is equal, the node with less queries running is chosen).
- w_t and w_q are associated weights, with $w_t + w_q = 1$.

As the state of the system influences where a query will be executed, it is crucial that this information is up-to-date. For example, if concurrent requests arrive at step 1, while the current query is optimized in step 6, they will be assigned the same node. Therefore, in Algorithm 5.1, step 5 performs an eager update of the system state (adding *mpl* threads to the chosen node immediately) to solve this problem. Then, to keep the state accurate, after an optimized plan has been computed that does not use all the resources assigned to the request, step 9 performs a second update. Finally, to ensure consistency of reading and writing to the state, locking mechanisms are needed when executing steps 1,2,5 and 9 of Algorithm 5.1.

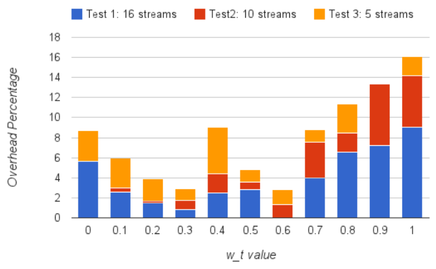
Experiments using different weights To determine the optimal values for the w_t and w_q parameters, three tests were run on 2 and 4 nodes, for 11 w_t [1] values:

Test 1. This test evaluates the weights by running the same amount of streams on 2 and 4 nodes. 16 TPC-H streams [2] were run both on 2 and 4 nodes.

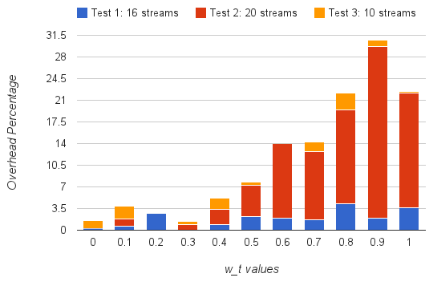
Test 2. This test evaluates the weights by running linearly more streams: 10 streams on 2 nodes and 20 streams on 4 nodes.

[1] The weight associated to the query load can be determined from the formula $w_q = 1 - w_t$

[2] These streams are permutations of the TPC-H query set. More on this subject in chapter 6



(a) 2 nodes



(b) 4 nodes

Figure 5.1: Cumulative overhead percentages

Test 3. The number of streams in this test increases linearly too, but the number of nodes does not divide the number of streams, i.e. 5 and 10 streams on 2 and 4 nodes, respectively.

For all the combinations (test, w_t value), the average time of 7 successive runs was taken (let us denote this with $A_i(w_t)$ where i is the number of the test). Then, to each w_t value we associated an *overhead percentage* ($O_i(w_t)$), calculated as:

$$O_i(w_t) = \frac{A_i(w_t) - min(A_i(w)|\forall w)}{min(A_i(w)|\forall w)}, \text{ where w is one of the 11 tested values}$$

The overhead percentages for all three tests are illustrated in figure 5.1.

We deliberately omitted the execution times and speedups of this policy as they will be the subject of Section 6.5. We notice that both on 2 and 4 nodes, w_t values close to 1 exhibit the highest overhead percentage. When these values are used, unbalanced scenarios can occur. For example, suppose we have 10 streams and the state of a system with 2 nodes is (9 queries, 10 threads) on node 1 and (1 query, 10 threads) on node 2. There is clearly more resource contention on node 1 and if the query on node 2 finishes, there would be a short period of time when none of the CPUs on node 2 would be utilized. The no-distribution policy configured with larger w_t values tends to create more such scenarios.

On the other extreme, a w_t value of 0 assigns an equal share of queries to the master node which also has to execute, for every query, the Rewrite stage and communicate with the client. On 2 nodes this overhead is visible as more queries are assigned to the master (and therefore more queries are slowed down).

Finally, w_t values ranging from 0.2 to 0.5 have overhead percentages that differ by at most 5% (i.e. about 1.7% per test). As these are throughput tests and the decisions made are based on the system state that changes frequently, such small differences can only suggest that any value chosen in this range will exhibit a close to optimal performance.

Overall, from the values presented in figure 5.1, 0.3 seems a good choice for w_t (and consequently 0.7 for w_q). Section 6.5 contains the throughput results of the no-distribution policy configured with these weight values.

We acknowledge that a more accurate estimation of the utilization of a particular node (e.g. by employing a state update protocol at run-time) would improve the performance of the No-distribution policy as the "least loaded" node would be more accurately identified. This would also apply for the Distribution policy described in the following section and we consider it a subject for future research.

5.2.3 Distribution Policy

This policy aims to produce distributed query plans that give a good throughput performance in a multi-user setting. It is an extension of the policy used in the non-distributed DBMS and it was inspired by our approach: a single DXchg operator that works at the granularity of threads. This policy unifies the resources available on the cluster and shares them equally between concurrent queries.

Algorithm 5.2 contains the pseudo-code of this policy.

Algorithm 5.2 distribution(*query*, *state*)

```
Require: $query$: a sequential query plan; $state$: the current state of the system;
Ensure: An equivalent parallel plan
 1: $mpl \leftarrow get\_global\_mpl(state)$
 2: $resources \leftarrow array(0)$
 3: $marked \leftarrow array(FALSE)$
 4: $hostnode \leftarrow NULL$
 5: while $mpl > 0$ do
 6:     $ll\_node \leftarrow get\_least\_loaded(state, marked)$
 7:     $resources[ll\_node] \leftarrow MIN(cores[ll\_node], mpl)$
 8:     $marked[ll\_node] \leftarrow TRUE$
 9:     $mpl \leftarrow mpl - resources[ll\_node]$
10:     if $hostnode = NULL \vee ll\_node = master\_node$ then
11:         $hostnode \leftarrow ll\_node$
12:     end if
13: end while
14: $add\_state\_load(state, resources)$
15: $par\_plan \leftarrow generate\_best\_plan(query, resources)$
16: $actual\_used \leftarrow get\_resources(par\_plan)$
17: if $actual\_used < resources$ then
18:     $remove\_state\_load(state, resources - actual\_used)$
19:     if $actual\_used[master\_node] = 0$ then
20:         $hostnode \leftarrow node|(\forall other\_node)actual\_used[node] \geq actual\_used[other\_node]$
21:     end if
22: end if
23: if $hostnode \neq master\_node$ then
24:     $par\_plan \leftarrow DXchange(par\_plan, 1, [host\_node], [1])$
25: end if
26: return $par\_plan$
```

First, the *mpl* is calculated in step 1 using the formula:

$$mpl = \begin{cases} \frac{SC*OAF}{TQ}, & \text{if } TQ > 0 \\ SC, & \text{otherwise} \end{cases}$$

where SC is the sum of the number of cores on each node, OAF is the over-allocation factor and is typically set to 1.25 and TQ is the total number of queries already running in the system.

Then, in steps 5-13 the smallest set of least loaded nodes that can provide this parallelism level is calculated. For every node in this set, the query is allocated all the cores on that node, with possibly one exception (when the remaining parallelism level required is in fact smaller than the number of cores on that node). Step 11 determines the node on which the actual computation will be started.

Similar to the no-distribution policy, step 14 performs an eager update, while step 15 generates the optimal distributed query plan given the *mpl*. Steps 17-22 of the algorithm update the state information and the host node if necessary.

Finally, Steps 23-26 add an DXchg(1:1) on top of the query tree, if necessary, to communicate the results to the master node.

The throughput performance of this policy can be found in Section 6.5.

Chapter 6

Results

6.1 Experimental Setup

All the experiments we present in this section were carried out on the available cluster of computers that was described in Section 2.3. Table 6.1 shows the exact values we used for the relevant configuration parameters of the distributed Vectorwise engine.

Config. Parameter	**Description**	**Value**
vector_size	Processing unit size: the (maximum) number of tuples returned by an operator's *next()* method.	1024
num_cores	Number of available cores.	10
bufferpool_size	The amount of memory the I/O buffers can use to cache data from disk.	40GB
num_recv_buf	Number of buffers a DXchg Receiver uses to store incoming messages.	2
num_send_buf	Number of buffers per destination used by a DXchg Sender.	2
max_xchg_buf_size	The maximum size of the Xchg buffers, also used as the fixed size of DXchg buffers (MPI messages).	256KB
thread_weight	The weight of the thread load (w_t, see Section 5.2.2)	0.3
enable_profiling	Flag specifying whether detailed per-query profiling information should be collected.	TRUE

Table 6.1: Parameter Values

We specified the number of cores to be used to 10 (of 12 available), to account for the various background threads running in the system at all times and thus reduce the effects of thread context switching on overall performance. Moreover, we also accounted for the fact that the MPI implementation for InfiniBand uses polling (implemented with busy waiting) to guarantee low-latency for incoming messages.

All tests were performed on so-called "hot" I/O buffers, meaning that MScan operators always read data from memory and there was no disk access whatsoever. This is because the impact of reading data from disk is so big in our case (see Section 2.3.1), that measuring database performance out of cache would essentially translate to measuring the performance of the underlying distributed file-system provided by GlusterFS, as the time spent waiting on disk access would take up the vast majority of the overall query processing time. As such, before running any tests, we performed prior "cold" runs to ensure all the required data would then be available in the I/O buffers. For this, we allowed the latter to store up to 40GB of data.

One of the limitations of the query plans presented in the next sections is the absence of the **Reuse** operator. The Reuse operator materializes the results of a query sub-tree such that they can be reused in a different part of the query plan, without executing the same sub-tree twice. Although it should not influence the speed-up results discussed henceforth, we acknowledge the importance of this optimization and it is included in our future work agenda.

Moreover, in all the experiments in this section, only queries expressed in the Vectorwise algebra were used and, therefore, only the Vectorwise engine was tested. Nevertheless, since our solution does not modify the way Ingres interacts with Vectorwise, there should be no issues with their integration.

6.2 The TPC-H Benchmark

For assessing the performance of our solution, we used the same benchmark according to which the non-distributed, commercial version of Vectorwise is usually evaluated internally, namely the TPC-H benchmark [TH], which simulates a real-world workload for large scale decision-support applications that is designed to have broad industry-wide relevance. This benchmark is designed and maintained by the Transaction Processing Performance Council and comes with a data generator and a suite of business oriented ad-hoc queries and concurrent data modifications.

The size of the data warehouse to be generated can be controlled by a parameter called *Scale Factor* (SF). For our tests we used SF 100 and SF 500, which produce approx. 60GB and 440GB of data respectively. We disabled the concurrent data modifications, as they are not yet supported in our system, but executed all the 22 read-only SQL queries provided (for which the reader is referred to the TPC-H website).

The TPC-H benchmark consists of two different tests for measuring the performance of a system for which we will present separate results below:

- The *Power Test* measures the query execution power of the system when connected with a single user. Queries are run in a sequential manner and their total elapsed time is measured.
- The *Throughput Test* measures the ability of the system to process the most queries in the least amount of time in a multi-user environment. Each simulated user runs its own version of the TPC-H Power Test simultaneously.

Disclaimer: all the TPC-H results presented in this section are for pure educational purposes and no official statements can be made about the performance of our solution based on these.

6.3 Power Test Results

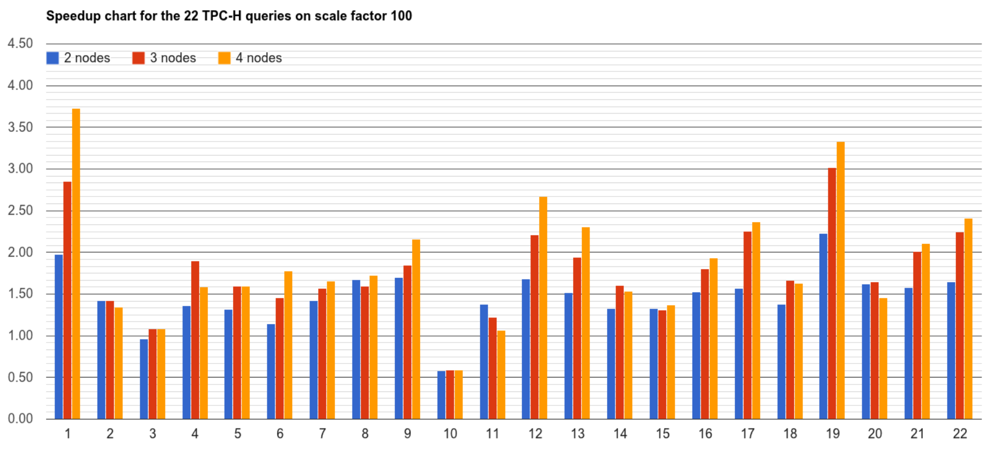
The query times shown were computed as the average of 3 "hot" runs, preceded by a cold run meant to ensure all required data is loaded into I/O buffers and hence available from memory.

Table 6.2 presents the numerical query processing times (in seconds) that we obtained for the 22 TPC-H queries on SF 100 and SF 500 databases, along with the corresponding per-query speedups, which are then graphically depicted in Figure 6.1.

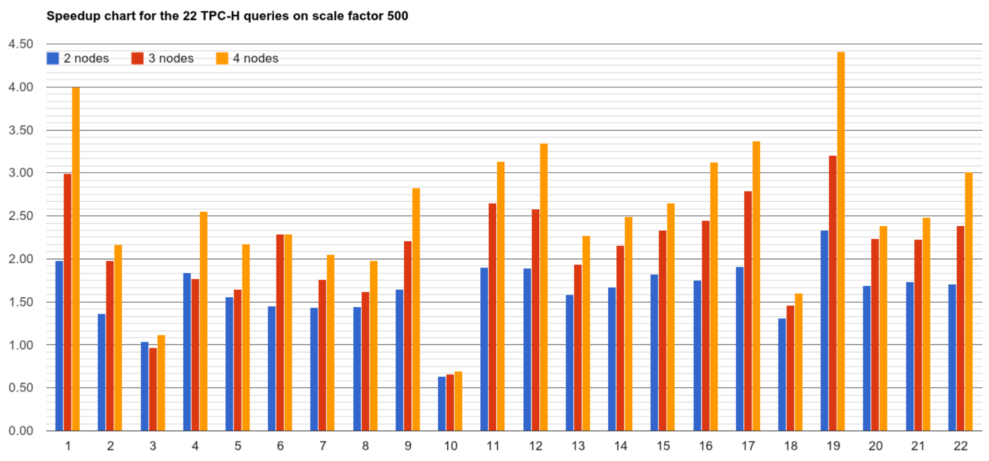
Figure 6.2 shows the percentage of time query execution accounts for of the total query time, where the latter is measured as the difference between the time the last result was returned to

	SF 100				SF 500			
Query	1 node	2 nodes	3 nodes	4 nodes	1 node	2 nodes	3 nodes	4 nodes
1	**2.57s**	**1.3s**	**0.9s**	**0.69s**	**9.19s**	**4.64s**	**3.07s**	**2.3s**
speedup		(1.98)	(2.86)	(3.72)		(1.98)	(2.99)	(4.00)
2	**0.47s**	**0.33s**	**0.33s**	**0.35s**	**1.82s**	**1.33s**	**0.92s**	**0.84s**
		(1.42)	(1.42)	(1.34)		(1.37)	(1.98)	(2.17)
3	**0.27s**	**0.28s**	**0.25s**	**0.25s**	**1.04s**	**1.00s**	**1.07s**	**0.93s**
		(0.96)	(1.08)	(1.08)		(1.04)	(0.97)	(1.12)
4	**0.19s**	**0.14s**	**0.1s**	**0.12s**	**0.46s**	**0.25s**	**0.26s**	**0.18s**
		(1.36)	(1.90)	(1.58)		(1.84)	(1.77)	(2.56)
5	**0.67s**	**0.51s**	**0.42s**	**1.51s**	**3.29s**	**2.11s**	**2.00s**	**1.51s**
		(1.31)	(1.60)	(1.60)		(1.56)	(1.65)	(2.18)
6	**0.16s**	**0.14s**	**0.11s**	**0.09s**	**0.48s**	**0.33s**	**0.21s**	**0.21s**
		(1.14)	(1.45)	(1.78)		(1.45)	(2.29)	(2.29)
7	**0.91s**	**0.64s**	**0.58s**	**0.55s**	**3.18s**	**2.22s**	**1.81s**	**1.55s**
		(1.42)	(1.57)	(1.65)		(1.43)	(1.76)	(2.05)
8	**1.02s**	**0.61s**	**0.64s**	**0.59s**	**3.52s**	**2.43s**	**2.17s**	**1.78s**
		(1.67)	(1.59)	(1.73)		(1.45)	(1.62)	(1.98)
9	**5.6s**	**3.3s**	**3.03s**	**2.59s**	**25.39s**	**15.39s**	**11.48s**	**8.99s**
		(1.70)	(1.85)	(2.16)		(1.65)	(2.21)	(2.82)
10	**1.91s**	**3.31s**	**3.24s**	**3.24s**	**8.62**	**13.53s**	**13.09s**	**12.42s**
		(0.58)	(0.59)	(0.59)		(0.64)	(0.66)	(0.69)
11	**0.33s**	**0.24s**	**0.27s**	**0.31s**	**1.88s**	**0.99s**	**0.71s**	**0.6s**
		(1.38)	(1.22)	(1.06)		(1.90)	(2.65)	(3.13)
12	**0.64s**	**0.38s**	**0.29s**	**0.24s**	**2.01s**	**1.06s**	**0.78s**	**0.6s**
		(1.68)	(2.21)	(3.13)		(1.90)	(2.58)	(3.35)
13	**11.46s**	**7.57s**	**5.89s**	**4.97s**	**43.67s**	**27.58s**	**22.54s**	**19.24s**
		(1.51)	(1.95)	(2.31)		(1.58)	(1.94)	(2.27)
14	**0.69s**	**0.52s**	**0.43s**	**0.45s**	**2.94s**	**1.76s**	**1.36s**	**1.18s**
		(1.33)	(1.60)	(1.53)		(1.67)	(2.16)	(2.49)
15	**0.81s**	**0.61s**	**0.62s**	**0.59s**	**2.68s**	**1.47s**	**1.15s**	**1.01s**
		(1.33)	(1.31)	(1.37)		(1.82)	(2.33)	(2.65)
16	**1.39s**	**0.91s**	**0.77s**	**0.72s**	**6.38s**	**3.64s**	**2.61s**	**2.04s**
		(1.53)	(1.81)	(1.93)		(1.75)	(2.44)	(3.13)
17	**1.35s**	**0.86s**	**0.6s**	**0.57s**	**6.01s**	**3.14s**	**2.15s**	**1.78s**
		(1.57)	(2.25)	(2.37)		(1.91)	(2.80)	(3.38)
18	**2.67s**	**1.94s**	**1.6s**	**1.64s**	**10.7s**	**8.17s**	**7.3s**	**6.68s**
		(1.38)	(1.67)	(1.63)		(1.31)	(1.47)	(1.60)
19	**2.23s**	**1s**	**0.74s**	**0.67s**	**7.63s**	**3.27s**	**2.38s**	**1.73s**
		(2.23)	(3.01)	(3.33)		(2.33)	(3.21)	(4.41)
20	**1.12s**	**0.69s**	**0.68s**	**0.77s**	**3.55**	**2.1s**	**1.59s**	**1.49s**
		(1.62)	(1.65)	(1.45)		(1.96)	(2.23)	(2.38)
21	**4.26**	**2.7s**	**2.12s**	**5.99s**	**14.85s**	**8.55s**	**6.67s**	**5.99s**
		(1.58)	(2.01)	(2.11)		(1.74)	(2.23)	(2.48)
22	**1.3s**	**0.79s**	**0.58s**	**0.54s**	**5.24**	**3.18s**	**2.27s**	**1.8s**
		(1.65)	(2.24)	(2.41)		(1.70)	(2.39)	(3.01)
Total	**42.02s**	**28.77s**	**24.19s**	**22.38s**	**164.71s**	**108.14s**	**87.59s**	**74.85s**
Average Speedup		(1.47)	(1.77)	(1.88)		(1.62)	(2.10)	(2.55)

Table 6.2: Power test on scale factors 100 and 500 using 10 cores per machine



(a) SF 100



(b) SF 500

Figure 6.1: Power test results: per-query speedup

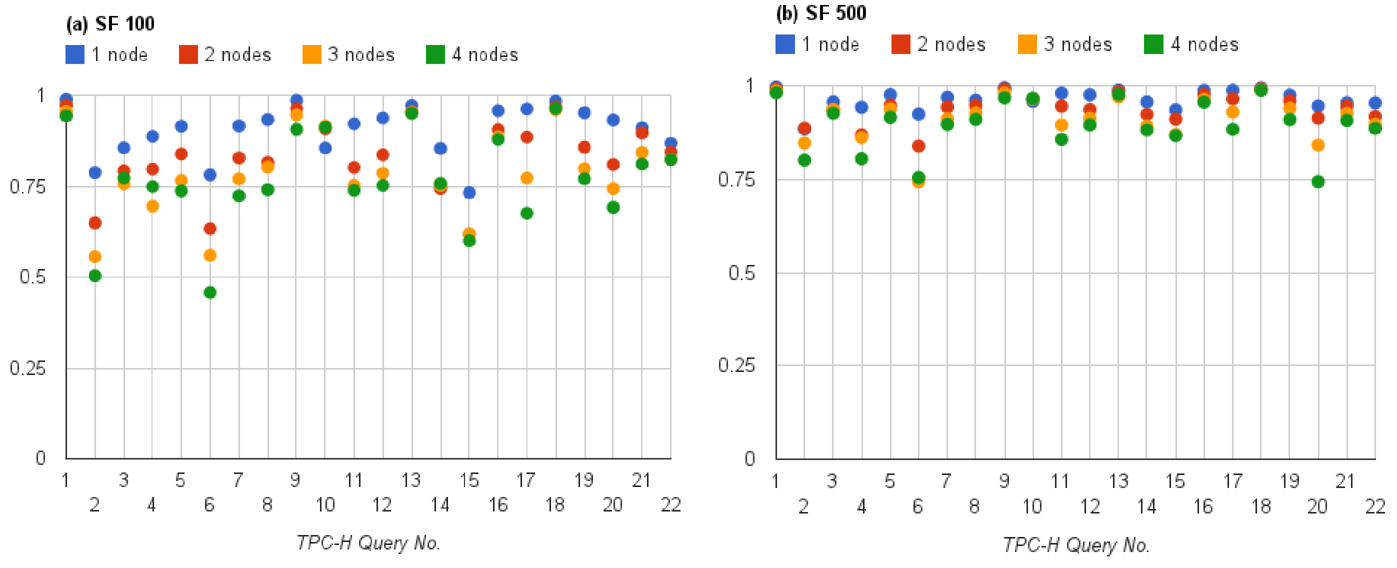


Figure 6.2: The percentage of the total query run time spent in the execution phase

the client and the time the database engine received the query. As expected, this percentage is higher for the bigger scale factor, because more data is processed, while the overhead related to the query interpretation, rewriter and profiling phases remains roughly the same. Since only the Builder and query execution phases are actually distributed, this figure becomes the explanation why we consistently obtained better speed-ups on the larger scale factor (1.62, 2.10 and 2.55 vs. 1.47, 1.62 and 2.55 respectively) and suggests that they will keep improving as the number of records in the database increases (according to Amdahl's famous law). Therefore, all the Power Test related discussions will be made based on the SF 500 results.

6.3.1 Per-Query Discussion

This subsection is dedicated to a more detailed per-query analysis of the SF 500, 4 -node results that aims to identify the reasons why some queries obtained good, or even close to linear speedup, while others did not benefit from distributed execution.

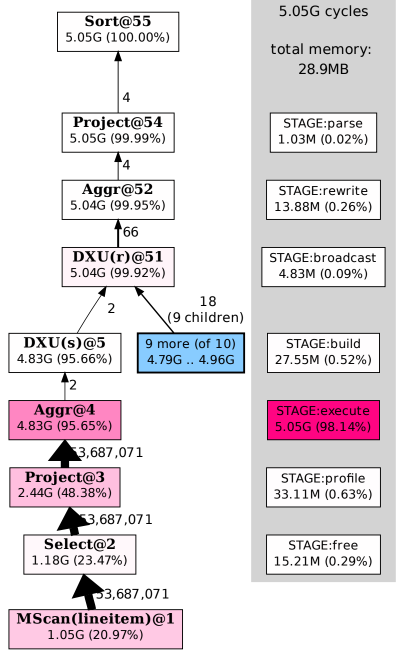
For each of the 22 TPC-H queries, Figures 6.3-6.14 present the associated operator trees, enriched with profiling information. The different degrees of coloring are meant to highlight the particularly time-consuming operators and reflect the percentage of the total execution time that was spent inside the operator's *next()* method. The numbers after the '@' symbol are operator labels, those inside operator boxes show the cumulative processing time (expressed in CPU cycles) of the corresponding sub-tree, while those on the edges refer to the total number of tuples returned by one operator to the other. Finally, for each query, the time spent in each of the various processing stages is shown alongside the operator tree.

The reader should keep in mind that, for simplicity, the graphs only depict one producer stream for every DXchg operator, while the remaining ones are combined in a blue box. Therefore, the numbers on the graph correspond to the operator *instances* belonging to the stream that is shown. Moreover, these graphs only present the perspective of the Master node. This is why, apart from the one explicit producer stream and those combined in the blue box, a given DXchg Receiver may also receive data from similar producer streams on other nodes, but of which there is no trace in these graphs. To exemplify this, consider the DXchgUnion receiver in Q01 (Figure 6.3a), which returns 66 tuples even though it only receives 20 tuples from its local senders. The difference of 46 tuples are received from senders on remote nodes. In the future, profiling information from all nodes can be unified in order to produce complete graphs. The same observation applies to logging information.

Query 1 [4.00 speedup on 4 nodes] Query 1 exhibits a linear speedup. Threads communicate their partial Aggregation results only at the end of their computation and there are only 66 partial results in total transferred over the network. Therefore, more than 98% of the computation is done independently and concurrently in the sub-trees rooted below the DXchgUnion operator.

Query 2 [2.17] The query plans generated by the distributed and non-distributed versions of Vectorwise differ slightly for this query. The sub-tree rooted at the Select@27 operator is the same in both plans and it is almost linearly parallelized. The only fractions of this sub-tree that cannot be parallelized are the build phase of the HashJoin01@12 [1] and HashJoin01@19 operators as these phases are executed sequentially. This is because the 18th transformation was applied in both cases and, in the current version of the Vectorwise DBMS, HashJoin operators on different threads share a hash table constructed on a single thread. Although this fragment

[1]The HashJoin01 operator joins tuples from the outer relation that have 0 or 1 matching keys in the inner relation. When there is exactly 1 matching key, the operator is called HashJoin1 and in any other case it is named HashJoinN.


(a) Q01

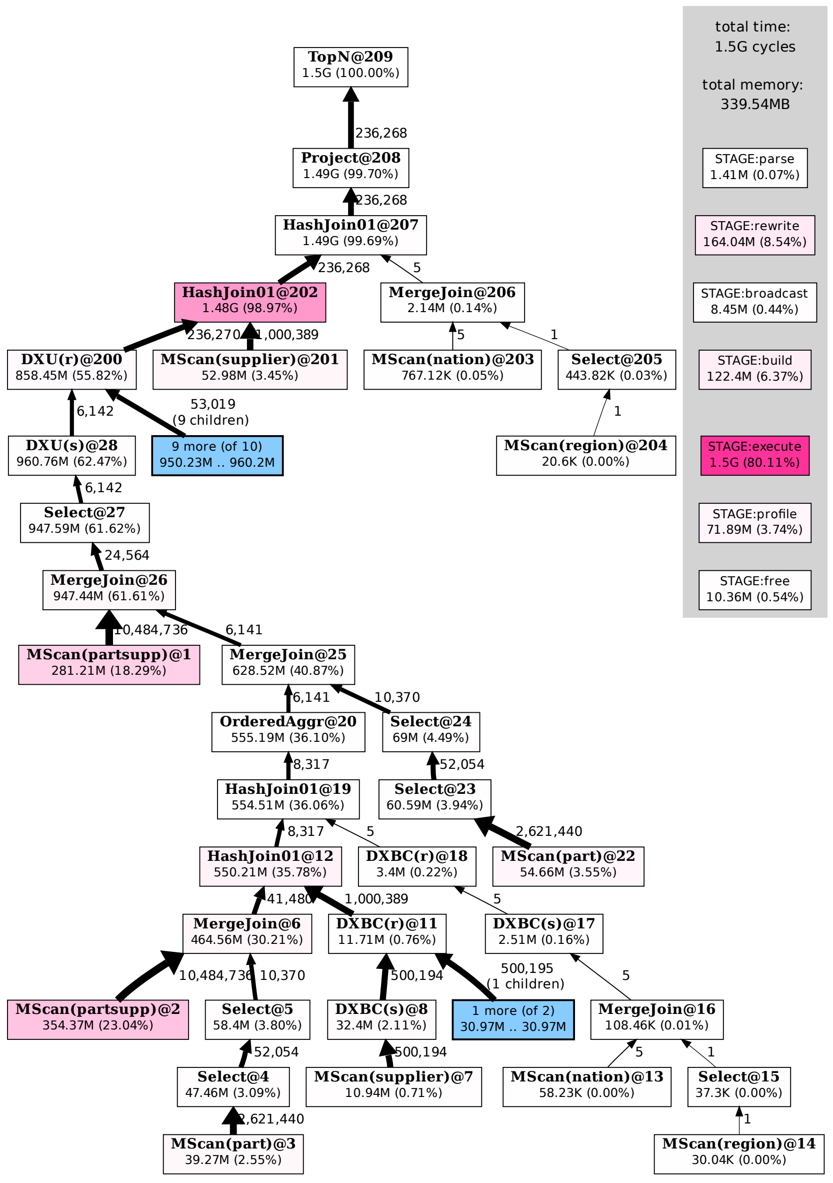

(b) Q02

Figure 6.3: Profile graph for TPC-H queries 1 and 2

does not amount to a significant percentage of the total execution time in this plan, this issue reoccurs in more than 5 TPC-H queries and, therefore, parallelizing the construction of the HashJoin's hash table (PHT, see Section 6.4) when the 18th rule is applied will significantly improve the average speedup.

On the other hand, 40% of the total execution time of Query 2 is spent in the HashJoin01@202 operator, which is executed on a single thread. This comes in contrast with the query plan generated by the non-distributed Rewriter which moves the DXchgUnion above the HashJoin01@202 and adds an DXchgBroadcast (applying the 18th transformation) above Mscan(supplier)@201. In both cases, 85% or more of the execution time of the HashJoin01@202 is spent in the build phase and therefore, a parallel build phase would also speed-up this part of the tree.

Query 3 [1.12] 68% of the execution time of Query 3 is spent in the HashJoin01@36 operator, and of this fragment 60% is spent in the build phase. PHT is needed here to reduce the execution time.

Moreover, tuples scanned and selected from the customer table are then broadcast over the network. We encounter here the first case where the network becomes a bottleneck. This happens because the network bandwidth is lower than the speed at which the tuples are generated by the Select operator. In general, in the case of perfect overlap of computation with communication, the lower bound for the active time (the time elapsed between the moments it produces its first and last tuples) of a Receive operator is:

$$act\_time(R) \geq min(cum\_time(S), \frac{data\_transferred}{network\_bandwidth})$$

where S is any Sender operator corresponding to the same logical DXchg operator. Let us apply the formula to verify that indeed Query 3 is network bound. The DXBC(r)@35 operator receives $\approx$ 15 million tuples that have 2 attributes, a 4-byte integer and string consisting of 10 characters on average. Considering that we have to send offsets for every string, we have a total tuple width of 22 bytes. Therefore $data\_transferred \approx$ 314.71MB and with a bandwidth of $2.4GB/s$ (see Figure 4.8), the lower bound for the active time of the DXBC(r)@35 operator is around 314.71 million cycles. Considering that in Figure 6.4a, only the cumulative time is depicted and this time is only a part of the active time, we can conclude that the right child of the HashJoin01@36 operator is network bound. In the following discussions we will omit these calculations for the sake of brevity.

To reduce the network traffic in this case, we can eliminate the string attribute as it is only used to filter tuples scanned from the *customer* table. The DCE optimization (see Section 6.4) will alleviate some of the network issues.

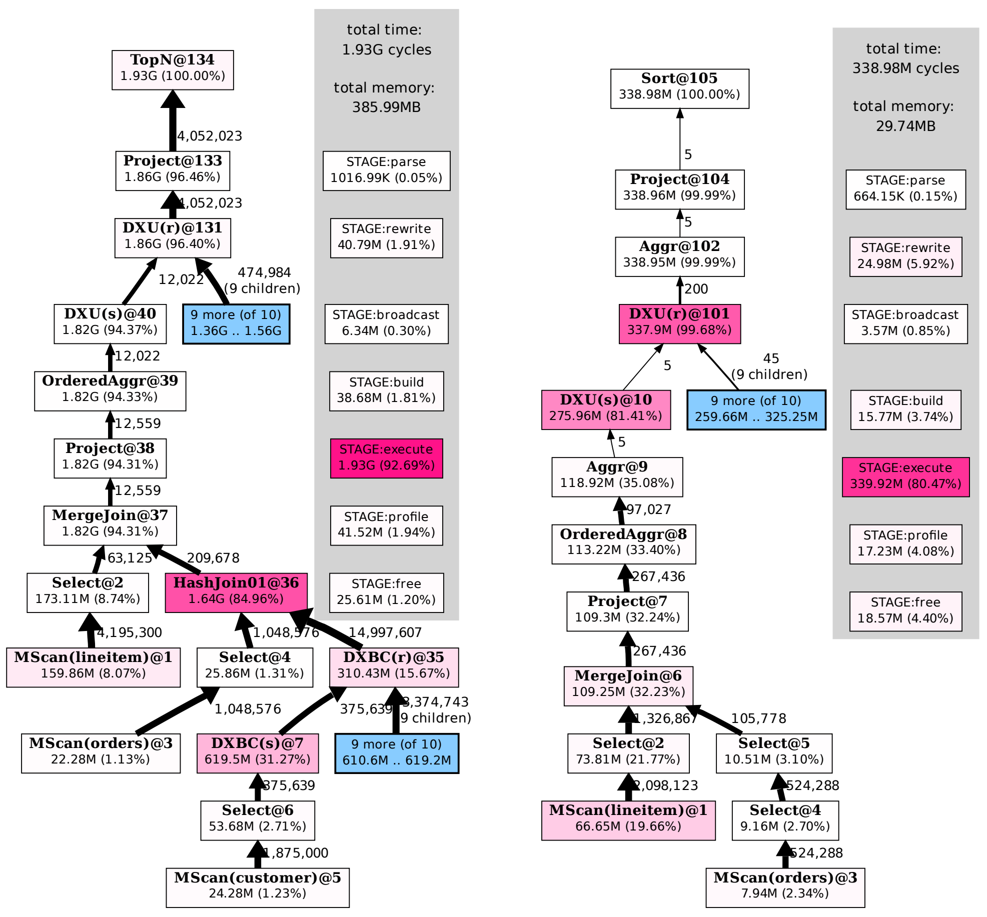
Query 4 [2.56] This is one of the fastest of the TPC-H set: it is processed in less than half a second on a single node. Because of this, measuring the performance and speedup of this query is particularly sensitive to noise.

On 4 nodes, the execution stage only amounts to 80% of the total query run time, with most of the remaining time being spent on the Master node. However, the major problem limiting speed-up in this case is data skew: while the depicted instance of the MergeJoin operator outputs 267 thousand tuples, other instances of it produce up to 1.3 million tuples. As such, times in the Aggr operator below DXchgUnion also vary significantly.

Query 5 [2.18] The problem of the HJ-DXchgBroadcast transformation is exhibited here again: 54% of the total execution time is spent in the HashJoin01@35 operator, with half of this time being spent in its (sequential) build phase. This is another query that would benefit from the PHT optimization.

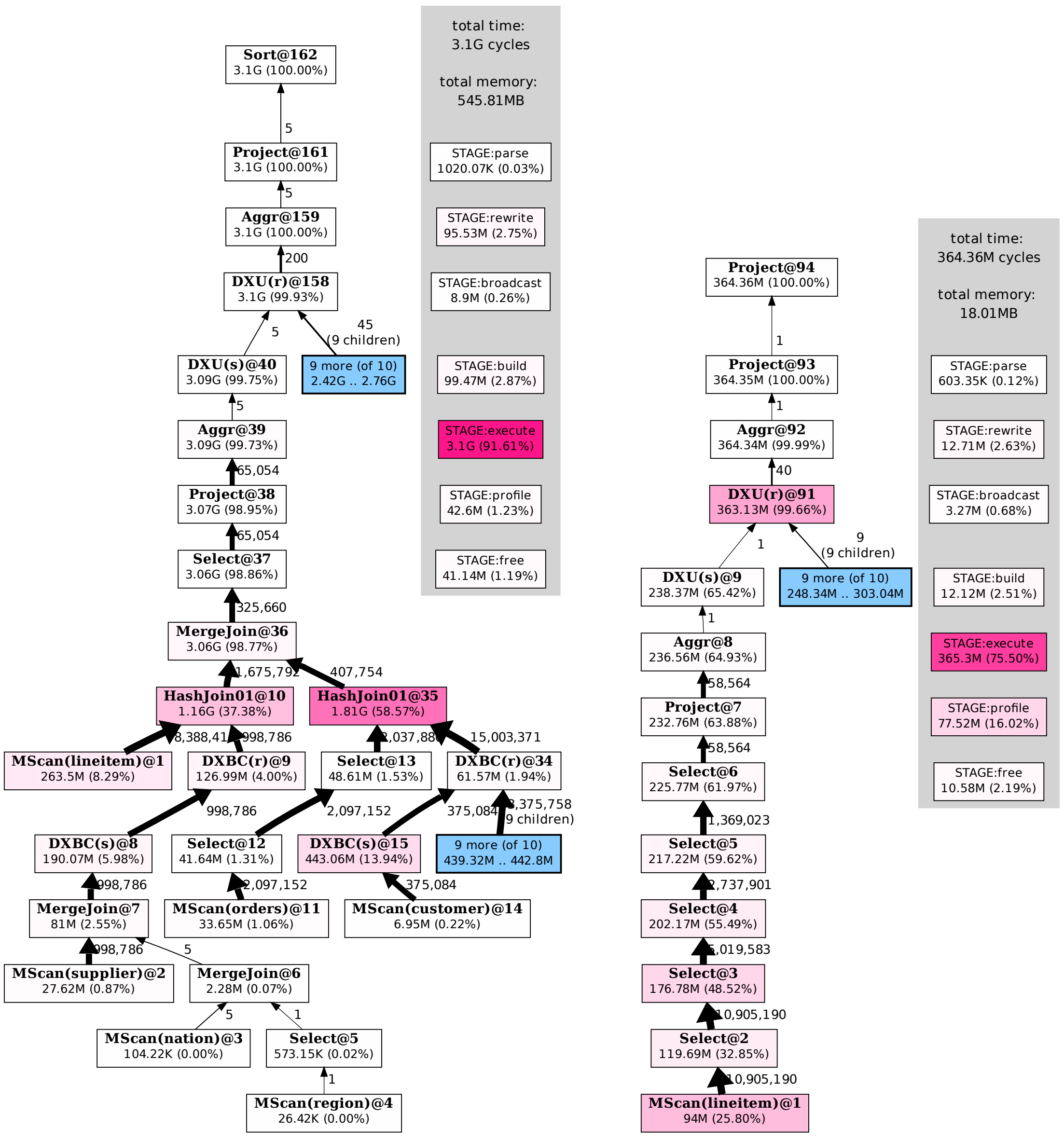
Also, there is a small data-skew in the outer relation of the above-mentioned HashJoin operator (in the probe phase, instances of the same logical operator process either 1.5 or 2 million tuples).

Query 6 [2.29] As Q04, this query takes very little time on a single node (under 0.5s), with only 75% of its total processing time being spent in the execution phase. Moreover, there is also considerable data skew, as the Project@7 operator processes 58 thousand tuples in some cases, while in others up to 750 thousand. For these two reasons, despite having a very similar plan to Query 1 (with no Join operators and only a few tuples exchanged), the speedup obtained on this query is not as good as that of the first one.


total time: 1.93G cycles
total memory: 385.99MB
STAGE:parse 1016.99K (0.05%)
STAGE:rewrite 40.79M (1.91%)
STAGE:broadcast 6.34M (0.30%)
STAGE:build 38.68M (1.81%)
STAGE:execute 1.93G (92.69%)
STAGE:profile 41.52M (1.94%)
STAGE:free 25.61M (1.20%)
TopN@134 1.93G (100.00%)
4,052,023
Project@133 1.86G (96.46%)
4,052,023
DXU(r)@131 1.86G (96.40%)
12,022
474,984 (9 children)
DXU(s)@40 1.82G (94.37%)
9 more (of 10) 1.36G .. 1.56G
12,022
OrderedAggr@39 1.82G (94.33%)
12,559
Project@38 1.82G (94.31%)
12,559
MergeJoin@37 1.82G (94.31%)
63,125
209,678
Select@2 173.11M (8.74%)
HashJoin01@36 1.64G (84.96%)
4,195,300
1,048,576
14,997,607
MScan(lineitem)@1 159.86M (8.07%)
Select@4 25.86M (1.31%)
DXBC(r)@35 310.43M (15.67%)
1,048,576
375,639
3,374,743 (9 children)
MScan(orders)@3 22.28M (1.13%)
DXBC(s)@7 619.5M (31.27%)
9 more (of 10) 610.6M .. 619.2M
375,639
Select@6 53.68M (2.71%)
1,875,000
MScan(customer)@5 24.28M (1.23%)
total time: 338.98M cycles
total memory: 29.74MB
STAGE:parse 664.15K (0.15%)
STAGE:rewrite 24.98M (5.92%)
STAGE:broadcast 3.57M (0.85%)
STAGE:build 15.77M (3.74%)
STAGE:execute 339.92M (80.47%)
STAGE:profile 17.23M (4.08%)
STAGE:free 18.57M (4.40%)
Sort@105 338.98M (100.00%)
5
Project@104 338.96M (99.99%)
5
Aggr@102 338.95M (99.99%)
200
DXU(r)@101 337.9M (99.68%)
5
45 (9 children)
DXU(s)@10 275.96M (81.41%)
9 more (of 10) 259.66M .. 325.25M
5
Aggr@9 118.92M (35.08%)
97,027
OrderedAggr@8 113.22M (33.40%)
267,436
Project@7 109.3M (32.24%)
267,436
MergeJoin@6 109.25M (32.23%)
1,326,867
105,778
Select@2 73.81M (21.77%)
Select@5 10.51M (3.10%)
2,098,123
524,288
MScan(lineitem)@1 66.65M (19.66%)
Select@4 9.16M (2.70%)
524,288
MScan(orders)@3 7.94M (2.34%)

(a) Q03 (b) Q04

Figure 6.4: Profile graph for TPC-H queries 3 and 4


total time:
3.1G cycles
total memory:
545.81MB
STAGE:parse 1020.07K (0.03%)
STAGE:rewrite 95.53M (2.75%)
STAGE:broadcast 8.9M (0.26%)
STAGE:build 99.47M (2.87%)
STAGE:execute 3.1G (91.61%)
STAGE:profile 42.6M (1.23%)
STAGE:free 41.14M (1.19%)
Sort@162 3.1G (100.00%)
5
Project@161 3.1G (100.00%)
5
Aggr@159 3.1G (100.00%)
200
DXU(r)@158 3.1G (99.93%)
5
45 (9 children)
DXU(s)@40 3.09G (99.75%)
9 more (of 10) 2.42G .. 2.76G
5
Aggr@39 3.09G (99.73%)
65,054
Project@38 3.07G (98.95%)
65,054
Select@37 3.06G (98.86%)
325,660
MergeJoin@36 3.06G (98.77%)
1,675,792
407,754
HashJoin01@10 1.16G (37.38%)
HashJoin01@35 1.81G (58.57%)
8,388,41
998,786
2,037,88
15,003,371
MScan(lineitem)@1 263.5M (8.29%)
DXBC(r)@9 126.99M (4.00%)
Select@13 48.61M (1.53%)
DXBC(r)@34 61.57M (1.94%)
998,786
2,097,152
375,084
3,375,758 (9 children)
DXBC(s)@8 190.07M (5.98%)
Select@12 41.64M (1.31%)
DXBC(s)@15 443.06M (13.94%)
9 more (of 10) 439.32M .. 442.8M
998,786
2,097,152
375,084
MergeJoin@7 81M (2.55%)
MScan(orders)@11 33.65M (1.06%)
MScan(customer)@14 6.95M (0.22%)
998,786
5
MScan(supplier)@2 27.62M (0.87%)
MergeJoin@6 2.28M (0.07%)
5
1
MScan(nation)@3 104.22K (0.00%)
Select@5 573.15K (0.02%)
1
MScan(region)@4 26.42K (0.00%)
total time:
364.36M cycles
total memory:
18.01MB
STAGE:parse 603.35K (0.12%)
STAGE:rewrite 12.71M (2.63%)
STAGE:broadcast 3.27M (0.68%)
STAGE:build 12.12M (2.51%)
STAGE:execute 365.3M (75.50%)
STAGE:profile 77.52M (16.02%)
STAGE:free 10.58M (2.19%)
Project@94 364.36M (100.00%)
1
Project@93 364.35M (100.00%)
1
Aggr@92 364.34M (99.99%)
40
DXU(r)@91 363.13M (99.66%)
1
9 (9 children)
DXU(s)@9 238.37M (65.42%)
9 more (of 10) 248.34M .. 303.04M
1
Aggr@8 236.56M (64.93%)
58,564
Project@7 232.76M (63.88%)
58,564
Select@6 225.77M (61.97%)
1,369,023
Select@5 217.22M (59.62%)
2,737,901
Select@4 202.17M (55.49%)
5,019,583
Select@3 176.78M (48.52%)
10,905,190
Select@2 119.69M (32.85%)
10,905,190
MScan(lineitem)@1 94M (25.80%)

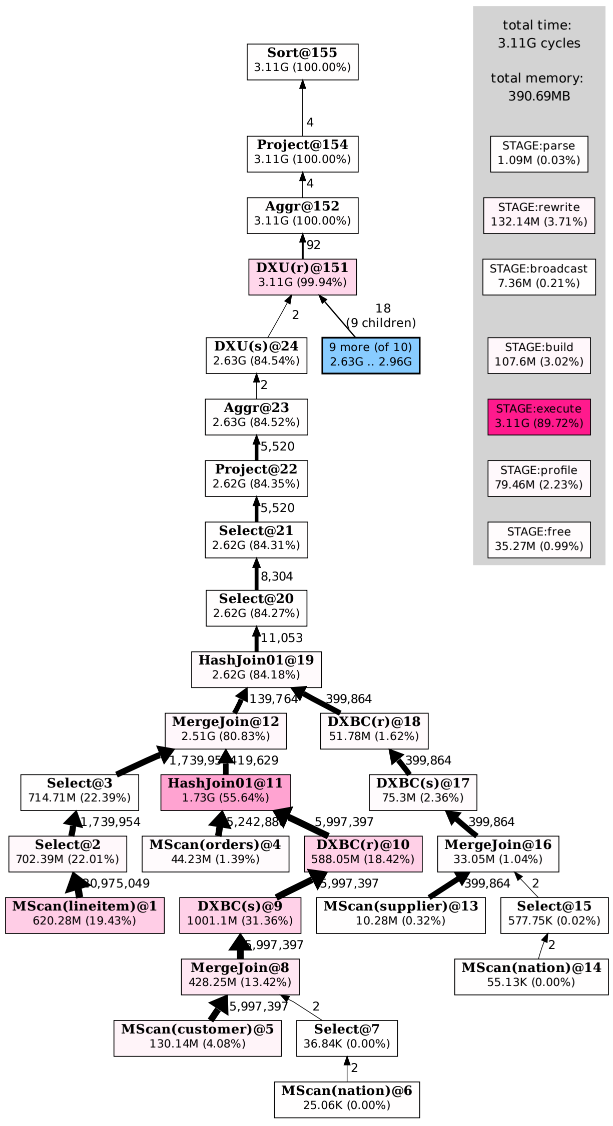
(a) Q05 (b) Q06

Figure 6.5: Profile graph for TPC-H queries 5 and 6

Query 7 [2.05] 55% of the execution time is spent in the subtree rooted at HashJoin01@11. This operator is another example of computation done in the build phase that cannot be parallelized. Also, the right side of the HashJoin is network bound.

Query 8 [1.98] The 18th transformation was applied 3 times in this query, rendering 1.2 billion cycles (30%) of sequential computation. PHT or applying a different transformation for the HashJoins might help in this case, e.g. transformation 19.

Query 9 [2.82] Query 9 spends 16% of its total execution time in building hash tables for the HashJoin01@77 and HashJoin01@11. PHT would make this query scale linearly.


(a) Q07

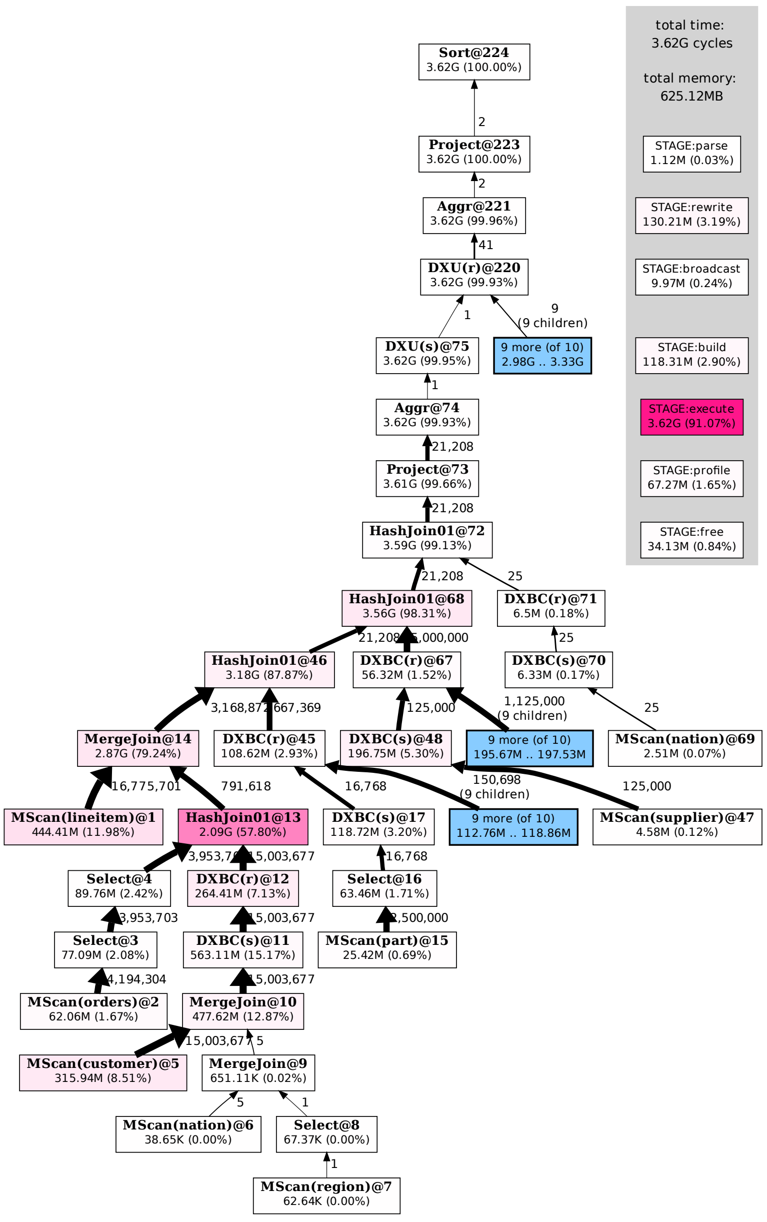
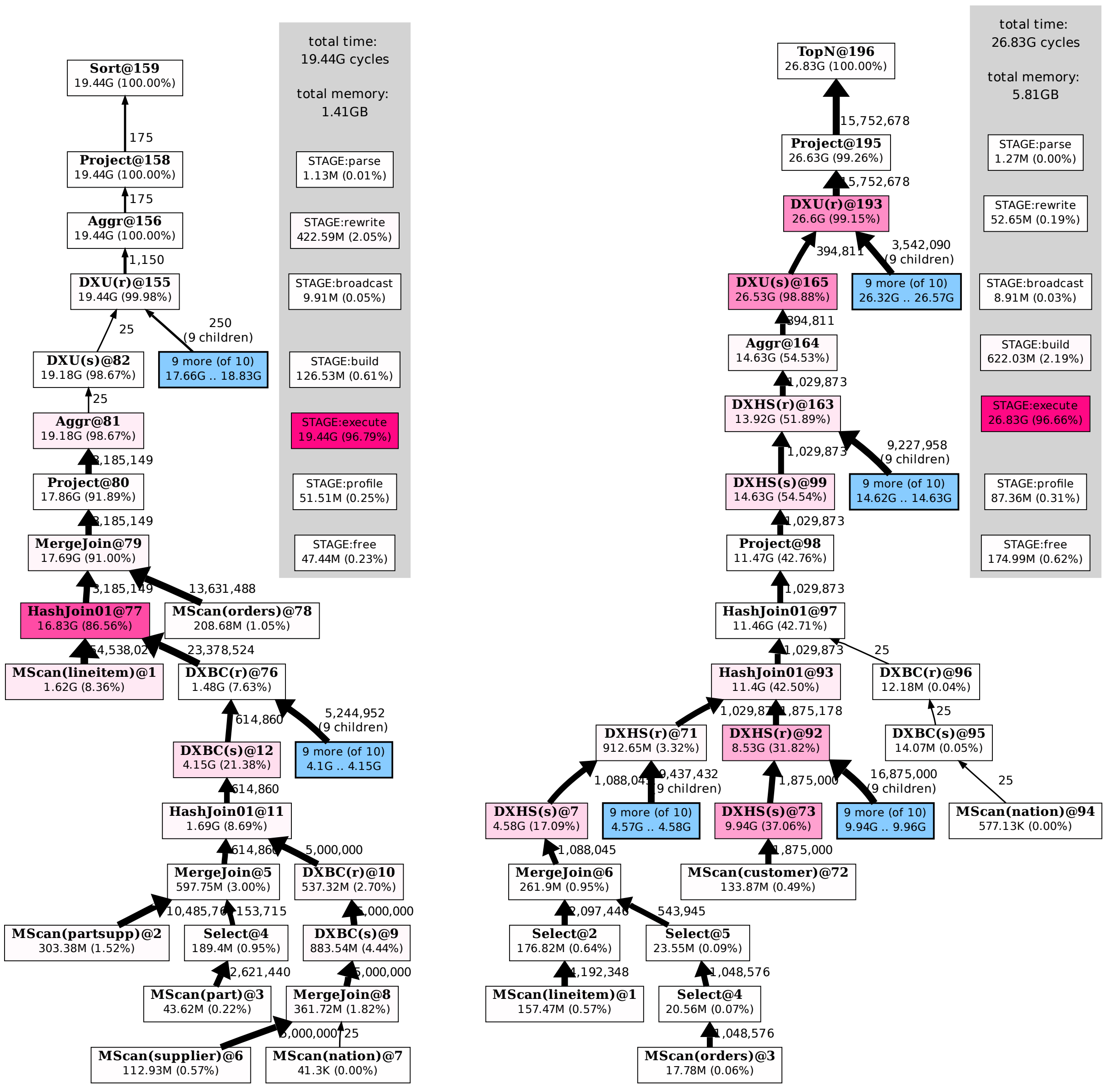

(b) Q08

Figure 6.6: Profile graph for TPC-H queries 07 and 08

Query 10 [0.69] Query 10 is the only query of the 22 that exhibits a speedup of below 1 on 4 nodes. The reason behind this is that tuples scanned from *customer* table contain attributes that are relevant only when presenting the results to the client. The non-distributed version uses the persistence optimization (see Section 6.4) to efficiently handle the string values read. In the distributed version, however, 15 million tuples are being transferred over the network, along with their various string attributes, even though, in the end, only 20 of them are returned to the client, while the others are simply discarded. LP (see Section 6.4) would drastically reduce the execution time of this query in both the distributed and non-distributed versions of the DBMS.



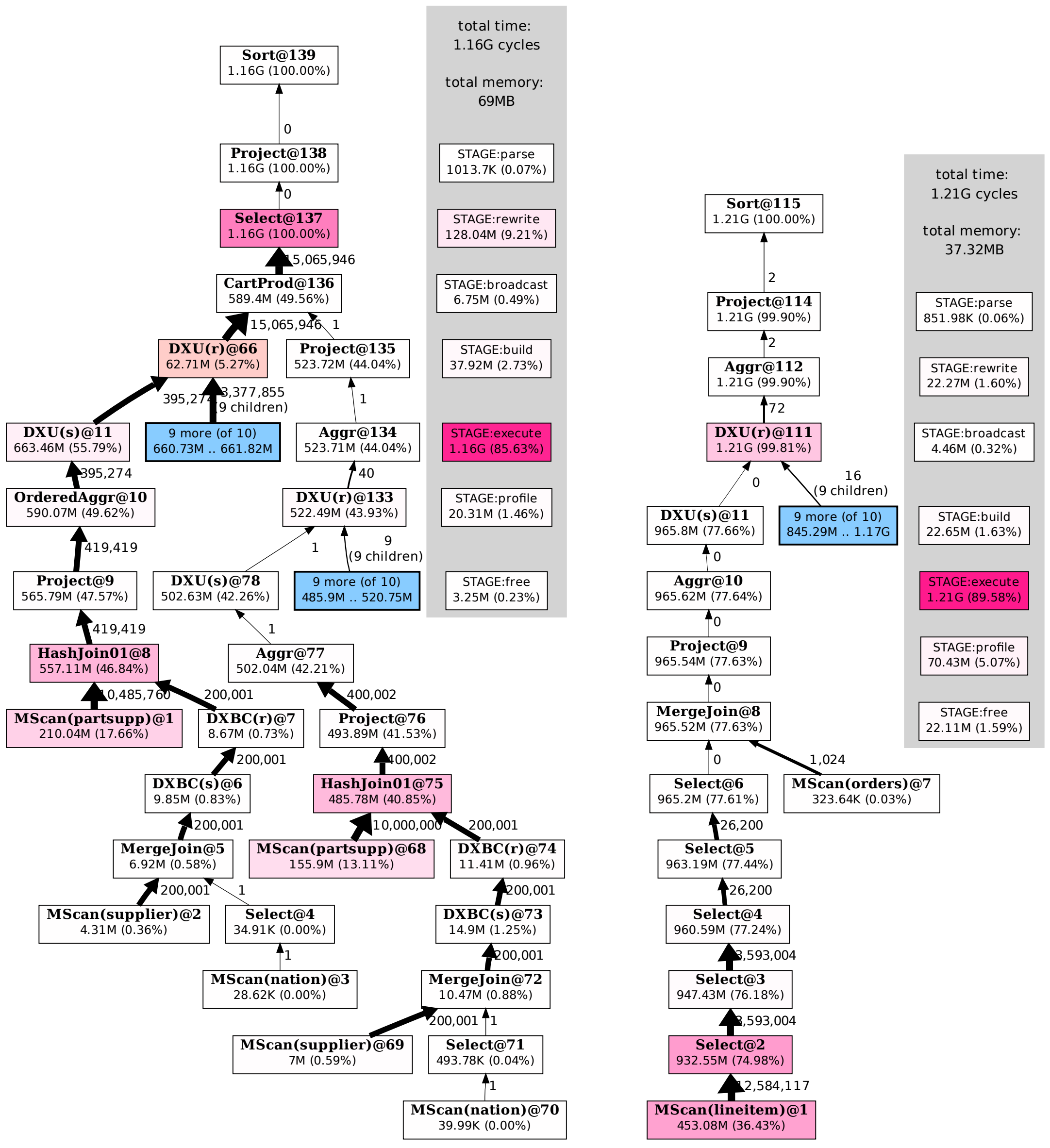
total time: 19.44G cycles
total memory: 1.41GB
STAGE:parse 1.13M (0.01%)
STAGE:rewrite 422.59M (2.05%)
STAGE:broadcast 9.91M (0.05%)
STAGE:build 126.53M (0.61%)
STAGE:execute 19.44G (96.79%)
STAGE:profile 51.51M (0.25%)
STAGE:free 47.44M (0.23%)
Sort@159 19.44G (100.00%)
Project@158 19.44G (100.00%)
Aggr@156 19.44G (100.00%)
DXU(r)@155 19.44G (99.98%)
DXU(s)@82 19.18G (98.67%)
9 more (of 10) 17.66G .. 18.83G
Aggr@81 19.18G (98.67%)
Project@80 17.86G (91.89%)
MergeJoin@79 17.69G (91.00%)
HashJoin01@77 16.83G (86.56%)
MScan(orders)@78 208.68M (1.05%)
MScan(lineitem)@1 1.62G (8.36%)
DXBC(r)@76 1.48G (7.63%)
DXBC(s)@12 4.15G (21.38%)
9 more (of 10) 4.1G .. 4.15G
HashJoin01@11 1.69G (8.69%)
MergeJoin@5 597.75M (3.00%)
DXBC(r)@10 537.32M (2.70%)
MScan(partsupp)@2 303.38M (1.52%)
Select@4 189.4M (0.95%)
DXBC(s)@9 883.54M (4.44%)
MScan(part)@3 43.62M (0.22%)
MergeJoin@8 361.72M (1.82%)
MScan(supplier)@6 112.93M (0.57%)
MScan(nation)@7 41.3K (0.00%)
total time: 26.83G cycles
total memory: 5.81GB
STAGE:parse 1.27M (0.00%)
STAGE:rewrite 52.65M (0.19%)
STAGE:broadcast 8.91M (0.03%)
STAGE:build 622.03M (2.19%)
STAGE:execute 26.83G (96.66%)
STAGE:profile 87.36M (0.31%)
STAGE:free 174.99M (0.62%)
TopN@196 26.83G (100.00%)
Project@195 26.63G (99.26%)
DXU(r)@193 26.6G (99.15%)
DXU(s)@165 26.53G (98.88%)
9 more (of 10) 26.32G .. 26.57G
Aggr@164 14.63G (54.53%)
DXHS(r)@163 13.92G (51.89%)
DXHS(s)@99 14.63G (54.54%)
9 more (of 10) 14.62G .. 14.63G
Project@98 11.47G (42.76%)
HashJoin01@97 11.46G (42.71%)
HashJoin01@93 11.4G (42.50%)
DXBC(r)@96 12.18M (0.04%)
DXHS(r)@71 912.65M (3.32%)
DXHS(r)@92 8.53G (31.82%)
DXBC(s)@95 14.07M (0.05%)
DXHS(s)@7 4.58G (17.09%)
9 more (of 10) 4.57G .. 4.58G
DXHS(s)@73 9.94G (37.06%)
9 more (of 10) 9.94G .. 9.96G
MScan(nation)@94 577.13K (0.00%)
MergeJoin@6 261.9M (0.95%)
MScan(customer)@72 133.87M (0.49%)
Select@2 176.82M (0.64%)
Select@5 23.55M (0.09%)
MScan(lineitem)@1 157.47M (0.57%)
Select@4 20.56M (0.07%)
MScan(orders)@3 17.78M (0.06%)

(a) Q09 (b) Q10

Figure 6.7: Profile graph for TPC-H queries 09 and 10

Query 11 [3.13] Half of the computation in Query 11 is done in the Select@137 operator, on a single thread. The Rewriter in this query might not have chosen the best plan. A better plan would be to apply the 6th transformation on the Select operator (introducing an DXchgUnion above it) and the 30th transformation to the CartProd operator below.

Query 12 [3.35] The only reason this query does not exhibit a perfect speedup is that data skew affects the load balance. For example the MergeJoin@8 depicted in Figure 6.8b processes 0 tuples on its left side, while other instances of the same logical MergeJoin operator process around 430,000 tuples.


total time:
1.16G cycles
total memory:
69MB
STAGE:parse 1013.7K (0.07%)
STAGE:rewrite 128.04M (9.21%)
STAGE:broadcast 6.75M (0.49%)
STAGE:build 37.92M (2.73%)
STAGE:execute 1.16G (85.63%)
STAGE:profile 20.31M (1.46%)
STAGE:free 3.25M (0.23%)
Sort@139 1.16G (100.00%)
Project@138 1.16G (100.00%)
Select@137 1.16G (100.00%)
CartProd@136 589.4M (49.56%)
DXU(r)@66 62.71M (5.27%)
Project@135 523.72M (44.04%)
DXU(s)@11 663.46M (55.79%)
9 more (of 10) 660.73M .. 661.82M
Aggr@134 523.71M (44.04%)
OrderedAggr@10 590.07M (49.62%)
DXU(r)@133 522.49M (43.93%)
Project@9 565.79M (47.57%)
DXU(s)@78 502.63M (42.26%)
9 more (of 10) 485.9M .. 520.75M
HashJoin01@8 557.11M (46.84%)
Aggr@77 502.04M (42.21%)
MScan(partsupp)@1 210.04M (17.66%)
DXBC(r)@7 8.67M (0.73%)
Project@76 493.89M (41.53%)
DXBC(s)@6 9.85M (0.83%)
HashJoin01@75 485.78M (40.85%)
MergeJoin@5 6.92M (0.58%)
MScan(partsupp)@68 155.9M (13.11%)
DXBC(r)@74 11.41M (0.96%)
MScan(supplier)@2 4.31M (0.36%)
Select@4 34.91K (0.00%)
DXBC(s)@73 14.9M (1.25%)
MScan(nation)@3 28.62K (0.00%)
MergeJoin@72 10.47M (0.88%)
MScan(supplier)@69 7M (0.59%)
Select@71 493.78K (0.04%)
MScan(nation)@70 39.99K (0.00%)
total time:
1.21G cycles
total memory:
37.32MB
STAGE:parse 851.98K (0.06%)
STAGE:rewrite 22.27M (1.60%)
STAGE:broadcast 4.46M (0.32%)
STAGE:build 22.65M (1.63%)
STAGE:execute 1.21G (89.58%)
STAGE:profile 70.43M (5.07%)
STAGE:free 22.11M (1.59%)
Sort@115 1.21G (100.00%)
Project@114 1.21G (99.90%)
Aggr@112 1.21G (99.90%)
DXU(r)@111 1.21G (99.81%)
DXU(s)@11 965.8M (77.66%)
9 more (of 10) 845.29M .. 1.17G
Aggr@10 965.62M (77.64%)
Project@9 965.54M (77.63%)
MergeJoin@8 965.52M (77.63%)
Select@6 965.2M (77.61%)
MScan(orders)@7 323.64K (0.03%)
Select@5 963.19M (77.44%)
Select@4 960.59M (77.24%)
Select@3 947.43M (76.18%)
Select@2 932.55M (74.98%)
MScan(lineitem)@1 453.08M (36.43%)

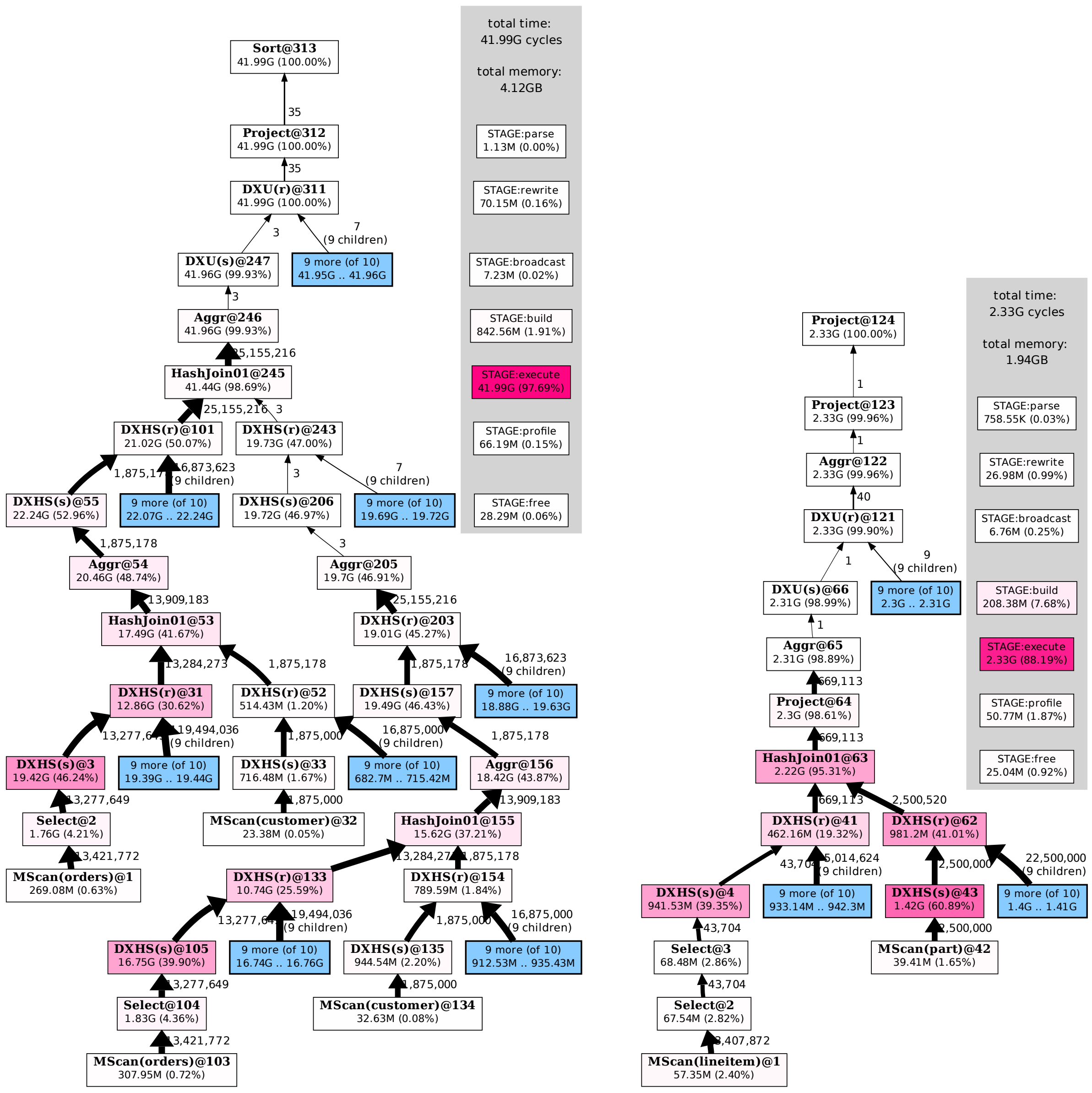
(a) Q11 (b) Q12

Figure 6.8: Profile graph for TPC-H queries 11 and 12

Query 13 [2.27] Query 13 filters some orders (e.g. in the Select@2 or Select@104 operators) by their *o_comment* attribute, which is not used again throughout the query. As the strings stored in *o_comment* have on average 50 characters, over 14.3GB of data is redundantly sent over the network. The original Vectorwise alleviates this problem by taking advantage of string persistence. However, DCE is the only optimization that would solve this problem in the distributed version.

On the other hand, the transmission of the tuples scanned from the *customer* table is bounded by the network bandwidth (e.g. from the DXHS(s)@135 to the DXHS(r)@154 operators).

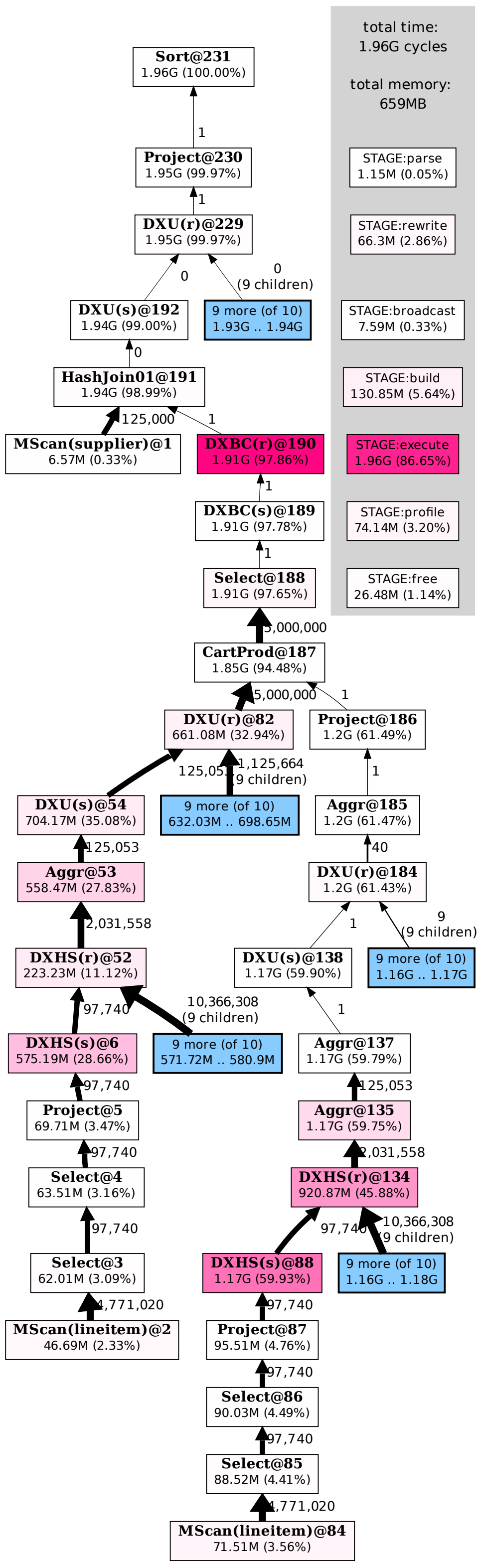
Query 14 [2.49] The right and left DXchgHashSplit operators under the Hashjoin@63 are network bound. To reduce the network traffic on the left side we can attach bloom filters to the DXHS(s)@4 operator (see Section 6.4). Unfortunately, the right DXchgHashSplit will still be network bound. A possible improvement would be to change the plan by flipping the relations of the HashJoin (the outer relation is smaller than the inner relation). However, making such a decision in the Rewriter is difficult as it would have to be robust and accurate in all other cases (even when the cardinality estimates are not precise).


total time:
41.99G cycles
total memory:
4.12GB
STAGE:parse 1.13M (0.00%)
STAGE:rewrite 70.15M (0.16%)
STAGE:broadcast 7.23M (0.02%)
STAGE:build 842.56M (1.91%)
STAGE:execute 41.99G (97.69%)
STAGE:profile 66.19M (0.15%)
STAGE:free 28.29M (0.06%)
Sort@313 41.99G (100.00%)
Project@312 41.99G (100.00%)
DXU(r)@311 41.99G (100.00%)
DXU(s)@247 41.96G (99.93%)
9 more (of 10) 41.95G .. 41.96G
Aggr@246 41.96G (99.93%)
HashJoin01@245 41.44G (98.69%)
DXHS(r)@101 21.02G (50.07%)
DXHS(r)@243 19.73G (47.00%)
DXHS(s)@55 22.24G (52.96%)
9 more (of 10) 22.07G .. 22.24G
DXHS(s)@206 19.72G (46.97%)
9 more (of 10) 19.69G .. 19.72G
Aggr@54 20.46G (48.74%)
Aggr@205 19.7G (46.91%)
HashJoin01@53 17.49G (41.67%)
DXHS(r)@203 19.01G (45.27%)
DXHS(r)@31 12.86G (30.62%)
DXHS(r)@52 514.43M (1.20%)
DXHS(r)@157 19.49G (46.43%)
9 more (of 10) 18.88G .. 19.63G
DXHS(s)@3 19.42G (46.24%)
9 more (of 10) 19.39G .. 19.44G
DXHS(s)@33 716.48M (1.67%)
9 more (of 10) 682.7M .. 715.42M
Aggr@156 18.42G (43.87%)
Select@2 1.76G (4.21%)
MScan(customer)@32 23.38M (0.05%)
HashJoin01@155 15.62G (37.21%)
MScan(orders)@1 269.08M (0.63%)
DXHS(r)@133 10.74G (25.59%)
DXHS(r)@154 789.59M (1.84%)
DXHS(s)@105 16.75G (39.90%)
9 more (of 10) 16.74G .. 16.76G
DXHS(s)@135 944.54M (2.20%)
9 more (of 10) 912.53M .. 935.43M
Select@104 1.83G (4.36%)
MScan(customer)@134 32.63M (0.08%)
MScan(orders)@103 307.95M (0.72%)
total time:
2.33G cycles
total memory:
1.94GB
STAGE:parse 758.55K (0.03%)
STAGE:rewrite 26.98M (0.99%)
STAGE:broadcast 6.76M (0.25%)
STAGE:build 208.38M (7.68%)
STAGE:execute 2.33G (88.19%)
STAGE:profile 50.77M (1.87%)
STAGE:free 25.04M (0.92%)
Project@124 2.33G (100.00%)
Project@123 2.33G (99.96%)
Aggr@122 2.33G (99.96%)
DXU(r)@121 2.33G (99.90%)
DXU(s)@66 2.31G (98.99%)
9 more (of 10) 2.3G .. 2.31G
Aggr@65 2.31G (98.89%)
Project@64 2.3G (98.61%)
HashJoin01@63 2.22G (95.31%)
DXHS(r)@41 462.16M (19.32%)
DXHS(r)@62 981.2M (41.01%)
DXHS(s)@4 941.53M (39.35%)
9 more (of 10) 933.14M .. 942.3M
DXHS(s)@43 1.42G (60.89%)
9 more (of 10) 1.4G .. 1.41G
Select@3 68.48M (2.86%)
MScan(part)@42 39.41M (1.65%)
Select@2 67.54M (2.82%)
MScan(lineitem)@1 57.35M (2.40%)

(a) Q13 (b) Q14

Figure 6.9: Profile graph for TPC-H queries 13 and 14

Query 15 [2.65] In Query 15 we discovered an implementation flaw. At first sight, the DXHS(r)@134 seems network bound. However, since the sub-tree rooted at the DXHS(r)@52


(a) Q15

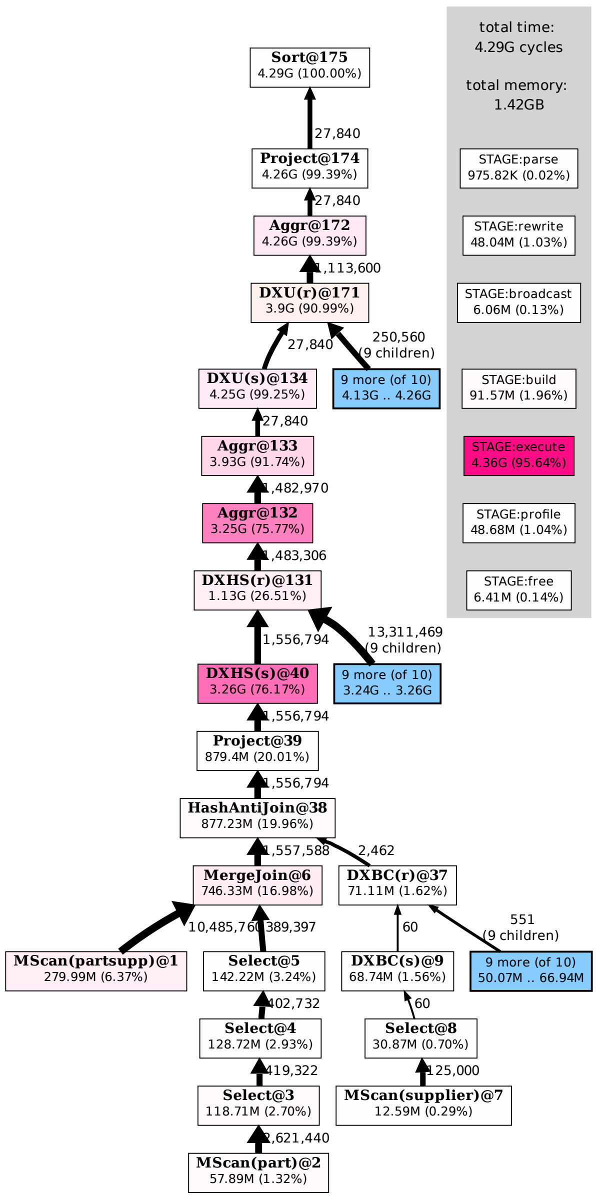

(b) Q16

Figure 6.10: Profile graph for TPC-H queries 15 and 16

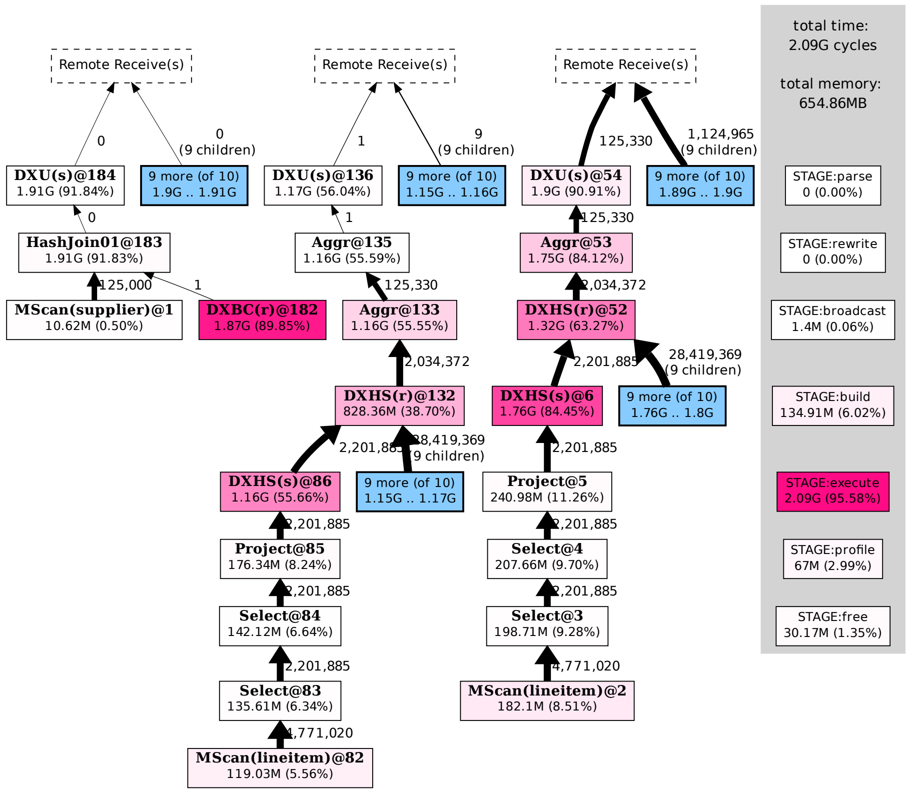


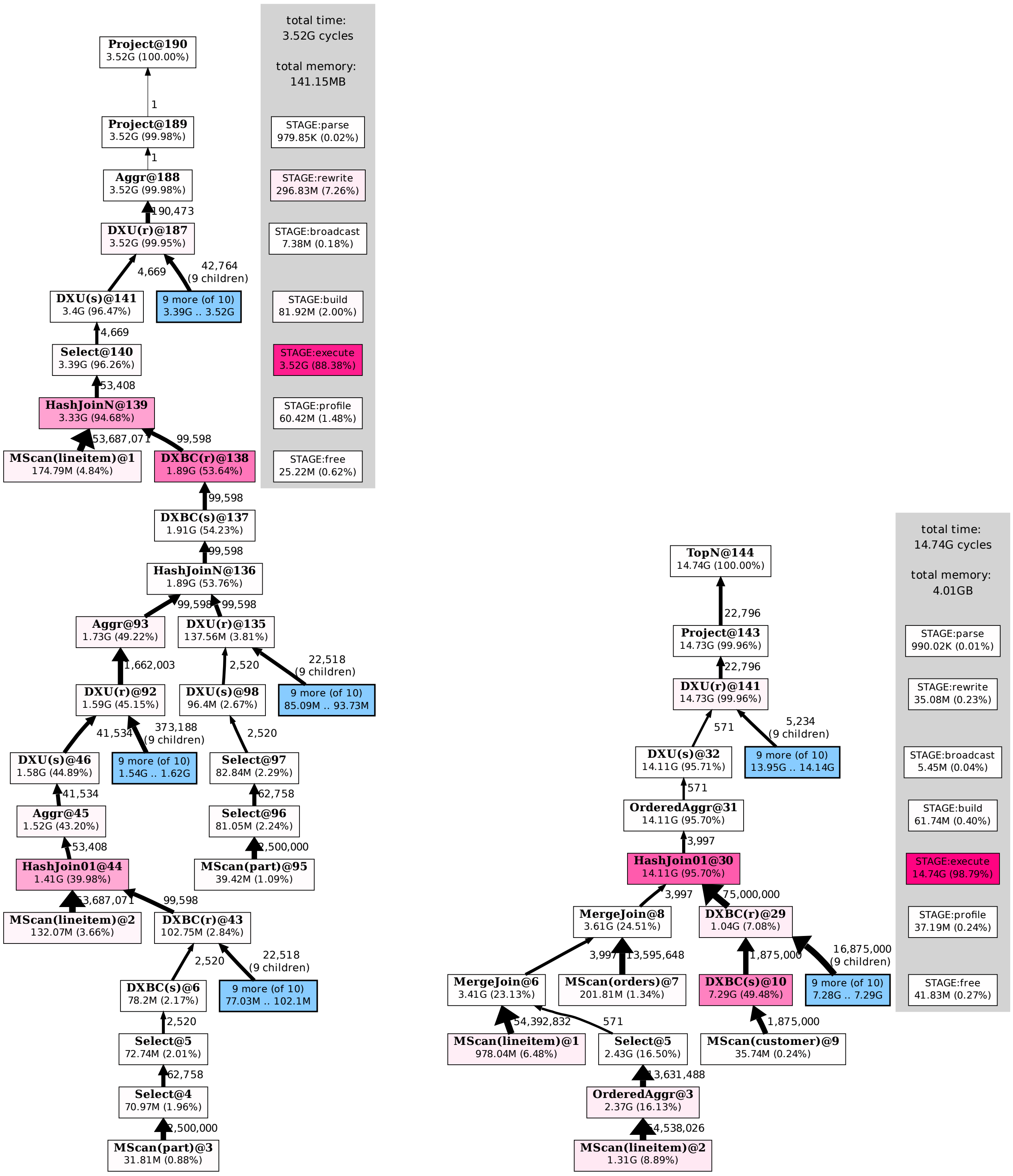
Figure 6.11: Profile graph for TPC-H Query 15 (worker node)

operator is identical to the one rooted at DXHS(r)@134 and has a lower execution time, and the amount of data transferred is ≈ 160MB, this hypothesis no longer holds. Therefore, to fully understand what happens in Query 15, we have to take a look at the operator trees/forests that are executed on other machines (i.e. other than the master node). Figure 6.11 illustrates the forest of operators that are executed on a worker node.

In Section 4.4, we mentioned that the *next()* method of orphan Sender operators is called immediately, at the start of the execution phase. While this is not an issue in most cases, when there is a materializing operator just beneath the Sender operator (e.g. Aggr@135 or Aggr@53 in Figure 6.11) the whole sub-tree is executed before producing any result. Consequently on a worker node during Query 15, there are 20 threads active for ≈ 1.17 G cycles (27.27% of the execution phase). These threads share all the processing resources and the network bandwidth.

To solve this issue, the process of starting the Sender threads should either be coordinated by the master (e.g. by broadcasting a message to start a particular set of sender operators) or it should be the same on all nodes. The latter solution is based on a simulation of the tuple flow on remote nodes, e.g. the family of operators that process tuples from the right side of the CartProd@187 (Figure 6.10a) should be executed completely before any of the operators on the left side are started, even on worker nodes (e.g. in Figure 6.11, DXU(s)@54 should be started only after DXU(s)@136 finishes).

Query 16 [3.13] The sub-tree below Aggr@132 produces tuples at a higher rate than the network can sustain. Enriching the parallelism rule with a transformation similar to the 12th transformation considering this situation (2 aggregations in a row), would eliminate this bottle-


total time: 3.52G cycles
total memory: 141.15MB
STAGE:parse 979.85K (0.02%)
STAGE:rewrite 296.83M (7.26%)
STAGE:broadcast 7.38M (0.18%)
STAGE:build 81.92M (2.00%)
STAGE:execute 3.52G (88.38%)
STAGE:profile 60.42M (1.48%)
STAGE:free 25.22M (0.62%)
Project@190 3.52G (100.00%)
1
Project@189 3.52G (99.98%)
1
Aggr@188 3.52G (99.98%)
190,473
DXU(r)@187 3.52G (99.95%)
4,669
42,764 (9 children)
DXU(s)@141 3.4G (96.47%)
9 more (of 10) 3.39G .. 3.52G
4,669
Select@140 3.39G (96.26%)
53,408
HashJoinN@139 3.33G (94.68%)
53,687,071
99,598
MScan(lineitem)@1 174.79M (4.84%)
DXBC(r)@138 1.89G (53.64%)
99,598
DXBC(s)@137 1.91G (54.23%)
99,598
HashJoinN@136 1.89G (53.76%)
99,598
99,598
Aggr@93 1.73G (49.22%)
DXU(r)@135 137.56M (3.81%)
1,662,003
2,520
22,518 (9 children)
DXU(r)@92 1.59G (45.15%)
DXU(s)@98 96.4M (2.67%)
9 more (of 10) 85.09M .. 93.73M
41,534
373,188 (9 children)
2,520
DXU(s)@46 1.58G (44.89%)
9 more (of 10) 1.54G .. 1.62G
Select@97 82.84M (2.29%)
41,534
62,758
Aggr@45 1.52G (43.20%)
Select@96 81.05M (2.24%)
53,408
2,500,000
HashJoin01@44 1.41G (39.98%)
MScan(part)@95 39.42M (1.09%)
53,687,071
99,598
MScan(lineitem)@2 132.07M (3.66%)
DXBC(r)@43 102.75M (2.84%)
2,520
22,518 (9 children)
DXBC(s)@6 78.2M (2.17%)
9 more (of 10) 77.03M .. 102.1M
2,520
Select@5 72.74M (2.01%)
62,758
Select@4 70.97M (1.96%)
2,500,000
MScan(part)@3 31.81M (0.88%)
total time: 14.74G cycles
total memory: 4.01GB
STAGE:parse 990.02K (0.01%)
STAGE:rewrite 35.08M (0.23%)
STAGE:broadcast 5.45M (0.04%)
STAGE:build 61.74M (0.40%)
STAGE:execute 14.74G (98.79%)
STAGE:profile 37.19M (0.24%)
STAGE:free 41.83M (0.27%)
TopN@144 14.74G (100.00%)
22,796
Project@143 14.73G (99.96%)
22,796
DXU(r)@141 14.73G (99.96%)
571
5,234 (9 children)
DXU(s)@32 14.11G (95.71%)
9 more (of 10) 13.95G .. 14.14G
571
OrderedAggr@31 14.11G (95.70%)
3,997
HashJoin01@30 14.11G (95.70%)
3,997
75,000,000
MergeJoin@8 3.61G (24.51%)
DXBC(r)@29 1.04G (7.08%)
3,997
13,595,648
1,875,000
16,875,000 (9 children)
MergeJoin@6 3.41G (23.13%)
MScan(orders)@7 201.81M (1.34%)
DXBC(s)@10 7.29G (49.48%)
9 more (of 10) 7.28G .. 7.29G
54,392,832
571
1,875,000
MScan(lineitem)@1 978.04M (6.48%)
Select@5 2.43G (16.50%)
MScan(customer)@9 35.74M (0.24%)
13,631,488
OrderedAggr@3 2.37G (16.13%)
54,538,026
MScan(lineitem)@2 1.31G (8.89%)

(a) Q17 (b) Q18

Figure 6.12: Profile graph for TPC-H queries 17 and 18

neck (by eliminating the partitioning requirement for the children of the Aggr@132 operator).

Another slow-down factor is that 10% of this query is run on a single thread (Sort@175, Project@174 and Aggr@172).

Query 17 [3.38] The execution phase is almost linearly parallelized in Query 17. The overhead comes from the time spent in the Rewriter and Builder.

Query 18 [1.60] In Query 18, the HashJoin@30 operator is required to produce clustered streams (enforced by the OrdAggr operator). Therefore, the only HashJoin transformation that can preserve the clustering property when multiple partitioned streams are requested is the 18th. Consequently, this leads to 65% of the computation being done in the build phase of the HashJoin operator. PHT would make this query scale perfectly on 4 nodes.

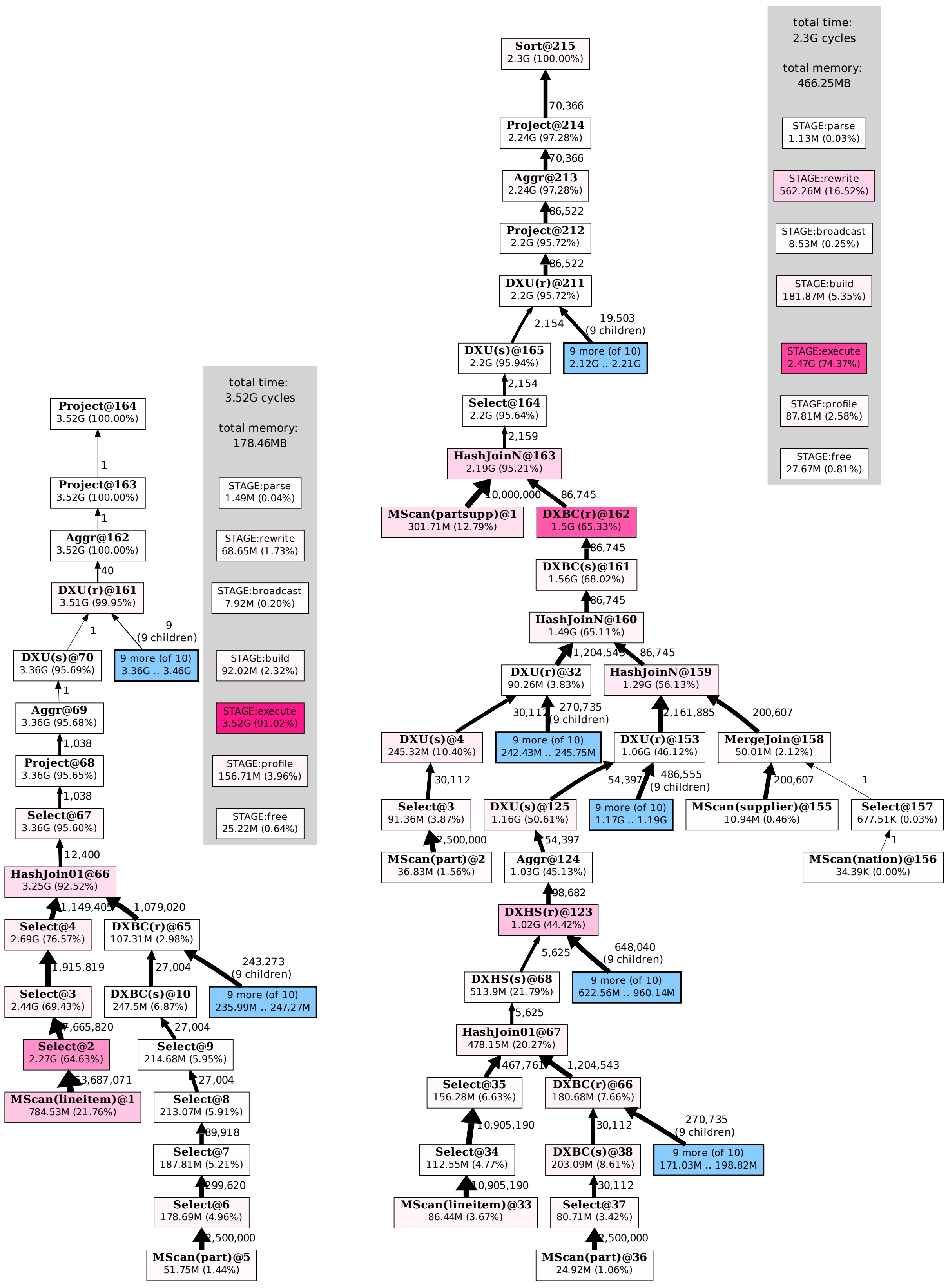
Query 19 [4.41] In the distributed Rewriter, the costs of the HashJoin operators were adjusted to better reflect their execution times. The non-distributed Rewriter applied the 19th transformation to the HashJoin operator, led by the cost model (i.e. the estimate of the build phase of the HashJoin was much higher), and therefore produced a sub-optimal query plan.

Query 20 [2.38] Only 75% of the total running time is spent in the execution phase in Query 20. Adding up to the slow-down are the 4 HashJoin operators that are all building their hash tables on a single thread.

Query 21 [2.48] The processing speed on the left side of the HashRevAntiJoin [1] is bound by the network bandwidth. Moreover, the tuples from its outer relation contain two attributes that are only used in the Select@3 operator, namely the *l_shipdate* and *l_commitdate*. Both DCE and BF can be employed in this case to reduce the network traffic.

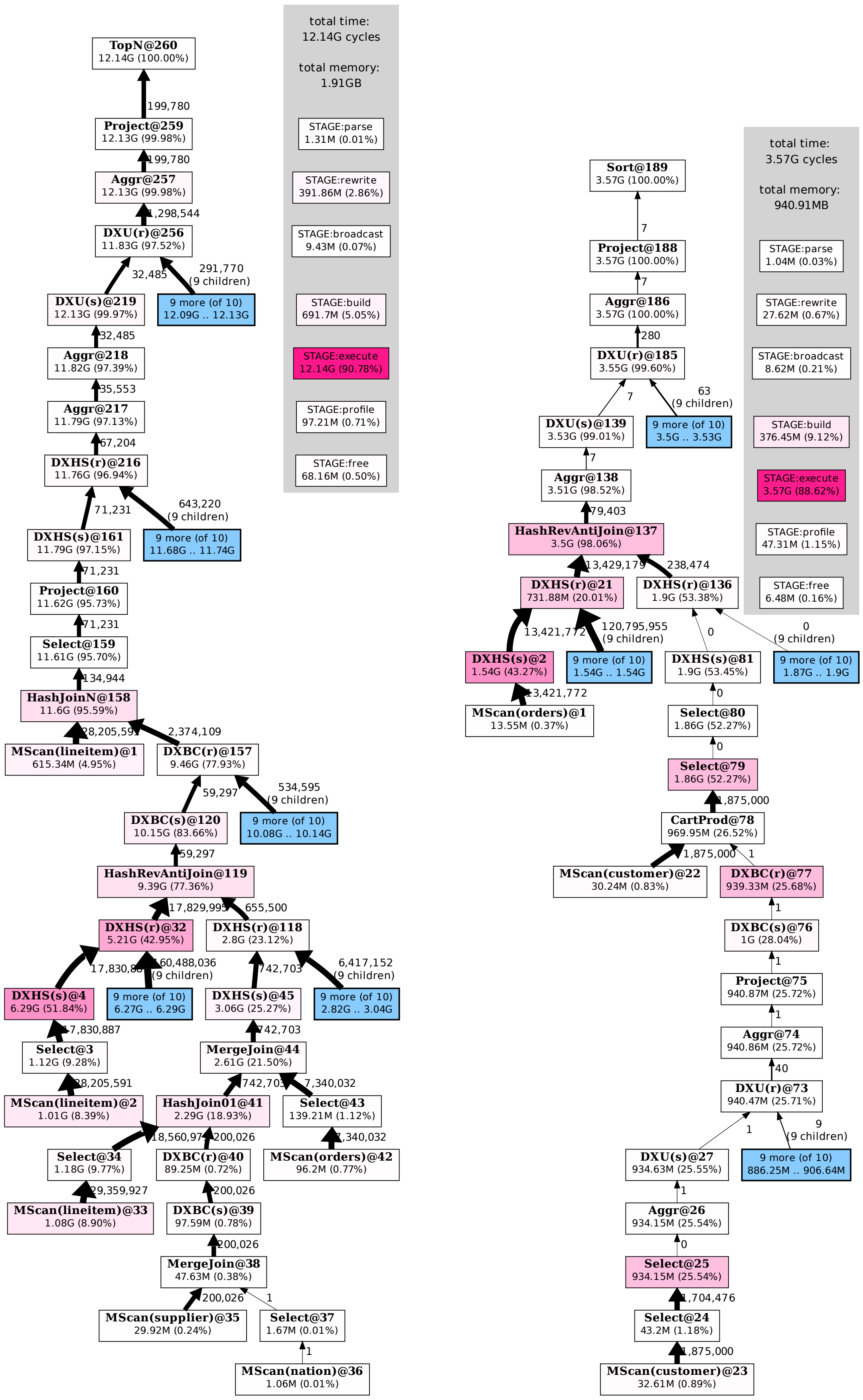
Query 22 [3.01] Query 22 has a better speed-up than Query 21, but it is also slowed-down by the network when scanning and partitioning tuples from the *orders* table. The BF optimization would remove this bottleneck.

[1] Details about this operator and the Bloom filter optimization can be found in Section 6.4



(a) Q19 (b) Q20

Figure 6.13: Profile graph for TPC-H queries 19 and 20



(a) Q21 (b) Q22

Figure 6.14: Profile graph for TPC-H queries 21 and 22

6.4 Future Improvements

In this section we discussed methods that can alleviate/eliminate the scalability issues identified above. Not surprisingly, most of these focus on reducing the amount of inter-node traffic.

Bloom Filters in DXchgHashSplit (BF)

We have seen that the HashJoin operators that process partitioned data from both of their children are usually bandwidth bound. This is because they join very large relations and these relations are usually partitioned and transmitted over the network using a DXchgHashSplit operator.

On the other hand, most of the decision support queries contain joins that are very selective (because these are foreign key joins into a right sub-plan that contains selections). At the time the probe phase is started, the set of all possible key values of the inner relation can be determined. This optimization tries to discard as many tuples as possible from the left (outer) relation before they are sent to the HashJoin operator responsible for processing their partition.

A *bloom filter* is a space-efficient data structure that tests whether a element is part of a set and may produce *false positives*, but never *false negatives* [Blo70]. It consists of an array of m bits that are initially set to 0 and k hash functions. To add an element to the set:

1. the k hash functions are used to map the element to k positions in the array
2. the bits at the marked positions are set to 1

To test if an element is in the set:

1. k positions are obtained by applying the same k hash functions to the element
2. if any of the bits at the marked positions is 0 then the element is definitely not in the set; otherwise the algorithm (possibly wrongly) concludes that the element is in the set

Bloom filters are constructed by each instance of the HashJoin operator during their build phase. Then all processing threads can exchange them such that, before the beginning of the probe phase, each left Sender operator has a filter corresponding to each Receive operator belonging to the same logical DXchgHashSplit operator.

These filters can then be used to discard the tuples of the outer relation that do not have matching tuples in the inner relation. For example, when a tuple is processed by a Sender operator, first a hash value is calculated to determine its possible destination, then a test is made using the associated Bloom filter (i.e., the Bloom filter received from the HashJoin operator that is the father of the Receive operator responsible to process the current tuple) and finally the tuple is either discarded or sent over the network. Both when building the bloom filters and when testing, the hash functions are applied on the join keys.

Consider TPC-H Query 21 (see Figure 6.14a). The fourth transformation was applied to produce a HashRevAntiJoin [1] operator that processes data partitioned by DXchgHashSplit operators. Let us examine how the BF optimization would improve the execution time of the subtree rooted at this operator:

- the cardinalities of the inner and outer relation are approximately $26,220,000$ and $713,275,692$, respectively
- the selectivity of the join is approximately 0.32%
- the width of a tuple from the outer relation is 24 bytes

[1] A HashRevAntiJoin operator returns all the tuples from the right relation that do not have matching keys in the left relation

- the left (logical) DXchgHashSplit operator transfers $\approx 15.94GB$ of data
- for a false positive probability of $p = 5\%$ and $n = 655,500$, the number of elements introduced in the Bloom filter, the optimal values [1] for m and k are $4,084,071$ and 5, respectively. Therefore, $N = 40$ (the number of parallel streams) bloom filters of $\approx 4MB$ would be communicated in the exchange phase
- $\approx 2,396,606$ tuples from the outer relation would pass the bloom filter test, and therefore only $55MB$ of data would be communicated over the network

Since Vectorwise already has implementations of bloom filters, this optimization can be employed with little effort and the effects it can have on network traffic are in some cases significant, e.g. a reduction by a factor of more than 200 for the above-mentioned DXchgHashSplit of Query 21.

Shared Buffers (SB)

The memory complexity of the number of buffers in the current implementation of the DXchg operators becomes an issue in large cluster systems. In order to reduce it, a set of buffer pools destined for every consumer can be used instead (similar to the current implementation of the original Xchg operators). This buffer pool would be shared by all producers on a particular node and the number of buffers it contains could be tuned such that:

1. it limits the memory usage (e.g. the maximum amount of memory a DXchg operator can be allocated)
2. it does not introduce substantial synchronization overhead (i.e. the time producers have to wait before acquiring a buffer of the pool)

Compressed Execution (CE)

Vectorwise already employs different data compression schemes, but only in the data storage layer. However, at execution time, whenever the Scan operators ask for vectors of tuples, the buffer manager retrieves and then decompresses the required data before returning it. Therefore, all the data that is being manipulated throughout query execution is uncompressed (i.e. no *compressed execution*).

The opportunities for employing compressed execution as a performance optimization in (the non-distributed version of) Vectorwise were already explored in [Lus11]. With non-intrusiveness being an important consideration (just as in our case), the authors identified *Run-Length Encoding* (RLE) and *On-the-Fly Dictionaries* to be both efficient and easy to integrate solutions for reducing memory consumption, while improving overall query execution time.

Reducing the amount of data being exchanged over the network would alleviate one of the major scalability issues of our approach, that is the network bandwidth limitation. Therefore, we expect that our project would especially benefit from the results of the work in [Lus11].

Persistent Strings (PS)

String persistence is an optimization of the single-node version of Vectorwise by which an operator that materializes data[2] marks the variable-legth data type expressions as *persistent*, meaning that the values are guaranteed to exist throughout the lifetime of the operator. This

[1] The formulas after which these values were calculated can be found in [Blo70]

[2] A materializing operator is one that maintains (some attributes of the) input data in memory throughout its entire lifetime: Sort, Aggregation, HashJoin

way, other materializing operators upstream can avoid making additional copies of the variable-length data and only use pointers instead.

Consider a query plan in which there is a DXchg above a HashJoin operator and imagine that for every tuple in the *build* relation (the one used for constructing the hash table) there are $n > 1$ matching tuples in the *probe* relation. If one attribute of the build relation is a string (variable-length data type), then the matched strings will be sent over the network n times. If the persistent strings were sent at once to the remote node after the build phase, but before probe, then only pointers (or offsets) would need to be transmitted n times. However, for Join operations with low matching rate, this approach would be detrimental, unless more sophisticated schemes are used, such that only those strings that are actually used (i.e. for which there are some matching tuples) are communicated. For example, per-destination bit-masks can be maintained to record which string values had already been sent. This way, a string value would only be sent upon the first record match, while for any subsequent matches only communicating its identifier would suffice.

Late Projection (LP)

In highly selective queries most tuples fail to contribute to the result, because of being dropped by Select, TopN, and/or Join operators. *Late Projection* is a general optimization technique by which retrieving non-key attribute data from the storage layer is postponed until it is certain (or highly probable) that the corresponding tuples will actually be part of the result. In the context of vectorized processing, this technique would reduce the time spent not only in Scan, but also in Xchg operators, since producers would need to perform less data copying. The benefits are especially notable when variable-length data types are involved.

In our distributed case, this optimization would also reduce network traffic by not transmitting certain attributes until they are actually needed. Let us take for example the worst-performing query of the TPC-H set, that is Q10. Here, tuples that are scanned from the *customer* table have 4 string attributes that are only used when communicating the results to the client: $c\_name, c\_phone, c\_address, c\_comment$. Moreover, these tuples are joined with the *nation* table and an additional string attribute is added, namely n_name. Table 6.3 [1] shows the ratio $\frac{stringsize}{tuplesize}$ of tuples that are transmitted over the network in TPC-H Query 10, together with the actual amount of data sent. The conclusion is that, in this case, Late Projection would reduce network communication by $\approx 85\%$.

Operator	Ratio	Data Transferred
DXUnion	116 : 140(82.85%)	2.05GB
DXHashSplit(below Aggr)	108 : 120(90.00%)	4.58GB
DXHashSplit(right side of HashJoin)	108 : 124(87.09%)	8.66GB
DXHashSplit (left side of HashJoin)	0 : 9(0.00%)	361.36MB
DXBroadCast	8 : 16(50%)	400B

Table 6.3: String/tuple ratio in the DXchange operators of TPC-H Query 10

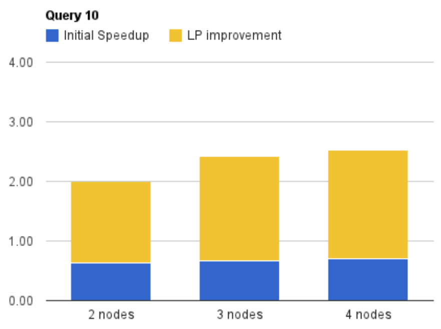
We managed to simulate the effects of Late Projection on Query 10 by manually eliminating the above-mentioned attributes from the MScan's parameters and joining the partial results at the top of the query plan with the *customer* and *nation* table in order to produce the complete results. The speed-up improvement is significant (Figure 6.15).

Although the new plan has reduced the network traffic, the speed-up deteriorates with the

[1] The values determined in this table are only approximations because they were computed using string averages

	SF 500			
Query	1 node	2 nodes	3 nodes	4 nodes
10	**3.46s**	**1.73s**	**1.43s**	**1.37s**
speedup		(2.00)	(2.42)	(2.53)

(a)



(b)

Figure 6.15: The effects of LP on Query 10 (SF 500, 10 cores/machine)

increasing number of threads as both sides of the HashJoin01@93 operator (Figure 6.7b) are still network bound.

Parallel Building of Shared Hash Tables (PHT)

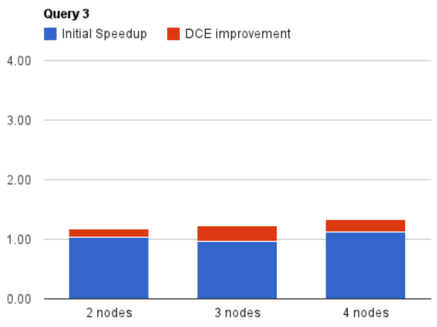
When the 18th transformation is applied to a HashJoin, every instance of that operator needs to have access to a hash table containing all the tuples from the inner relation, at the beginning of the probe phase. Since it does not make sense to build several copies of the same hash table, in the non-distributed DBMS a *master* instance is elected before the build phase starts, then the master builds on a **single** thread the whole hash table and finally pointers/references to the hash tables are sent to the other HashJoin instances.

We have seen that in many queries, building the hash table on a single thread becomes an issue in the distributed version of the Vectorwise DBMS. Therefore, this build phase needs to be parallelized. One easy implementation for this optimization would be to have every instance of the HashJoin operator (belonging to a given node) build separate hash tables in parallel containing a part of the inner relation and then merge them. If necessary, additional improvements can be made to this implementation to reduce the memory complexity (e.g. have several instances work on the same hash table).

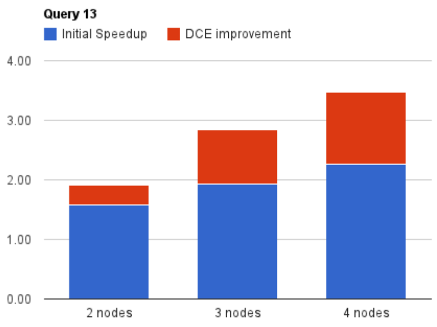
Dead Code Elimination (DCE)

We have seen in Section 6.3 that in some queries there are attributes that are only used to filter a relation. These attributes become "dead" for the rest of the query and should be eliminated. In the Vectorwise Rewriter, the *dead code elimination* rule discards these columns only in materializing operators. In distributed settings, this becomes a bottleneck as large amounts of unnecessary data are sent over the network. Therefore, the DCE rule should consider introducing Project operators as low as possible into query plans to eliminate the dead columns.

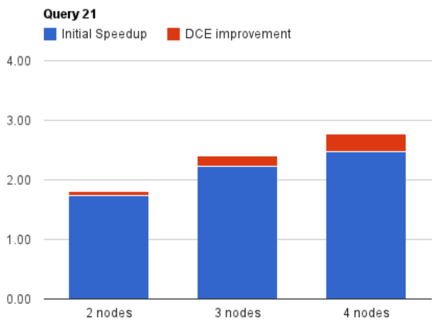
To simulate the effects that DCE has on the TPC-H queries, queries 3, 13 and 21 were modified



(a) Q3



(b) Q13



(c) Q21

Figure 6.16: The effects of DCE on queries 3, 13 and 21 (SF 500, 10 cores/machine)

	SF 500			
Query	1 node	2 nodes	3 nodes	4 nodes
3	**1.05s**	**0.89s**	**0.85s**	**0.79s**
speedup		(1.18)	(1.24)	(1.33)
13	**38.37s**	**20s**	**13.48s**	**11.05s**
		(1.92)	(2.85)	(3.47)
21	**14.41s**	**7.97s**	**5.98s**	**5.19s**
		(1.81)	(2.41)	(2.78)

Table 6.4: The effects of DCE on queries 3, 13 and 21 (SF 500, 10 cores/machine)

by manually introducing a Project operator immediately above the selection to eliminate the "dead" columns. These effects are illustrated in Figure 6.16 and Table 6.4.

The execution times for Query 3 have improved with the DCE optimization, but, as expected, most of the time is still spent in the build phase of the HashJoin01@36 operator (Figure 6.4a).

In Query 13, there are still three DXchgHashSplit operators that are placed on top of Aggr operators that slow down the tuple flow. This happens because Aggr is an operator that materializes all of its data before producing a tuple and therefore it is very hard for the network bandwidth to match the speed of this "burst" of tuples. One way to mask this problem would be to overlap the communication with the computation performed in the part of the query plan that consumes these tuples. However, since in most cases this is not possible, the only way to avoid this issue is to change the query plan.

Finally, in Query 21, although we removed 2 4-byte integer attributes from the left relation of the HashRevAntiJoin operator (Figure 6.14a), the number of the tuples ($\approx$ 180 million) and the size of the remaining attributes (16 byte wide) are still large enough not to completely eliminate the network bottleneck in this query.

Explicit Data Partitioning (EDP)

Vectorwise relies only on the HashSplit operators to partition data, if necessary. This becomes an issue when large relations are processed as it assumes an "all-to-all" redistribution of data that is bound by the network bandwidth.

To eliminate this redistribution of data, a *partitioning scheme* is needed. However, a change in the architectural choice would have to be made to make use of it (currently, we rely on the distributed file-system to transparently manage data storage). Nevertheless, we will analyze the expected effects of such a partitioning scheme on the TPC-H query set, namely the **Bitwise Dimensional co-Clustering** (BDC).

BDC modifies the arrangement of tuples within tables such that the partitioning and clustering requirements are satisfied even at the Scan level [BBS12]. BDC orders tuples by their attribute values on several dimensions to speedup foreign key joins and push-down range-selections (i.e. they are performed in the Scan operator). Consequently, this multidimensional storage scheme eliminates, in most cases, the need for additional re-partitioning inside query plans and therefore eliminates HashSplit operators, altogether.

Per-Query Optimization Effectiveness Matrix

Table 6.4 illustrates in a concise manner the estimated effectiveness of the identified optimizations on the TCP-H query set.

	BF	SB	CE	PS	LP	PHT	DCE	EDP	speedup
1									++
2						++			-
3			+			++	+		- -
4									+-
5				+		++			-
6									-
7			+	+		+			-
8				+		+			- -
9				+		++			+-
10		++	++	++	++			++	- -
11									+
12									+
13		++	++				++	++	-
14		+	++				+	++	-
15		+	+				+	+	+-
16		+	++				+		+
17									+
18			++		+	++			- -
19									++
20		+	+			++		++	-
21	++	++	++				+	++	-
22	++	+	++						+

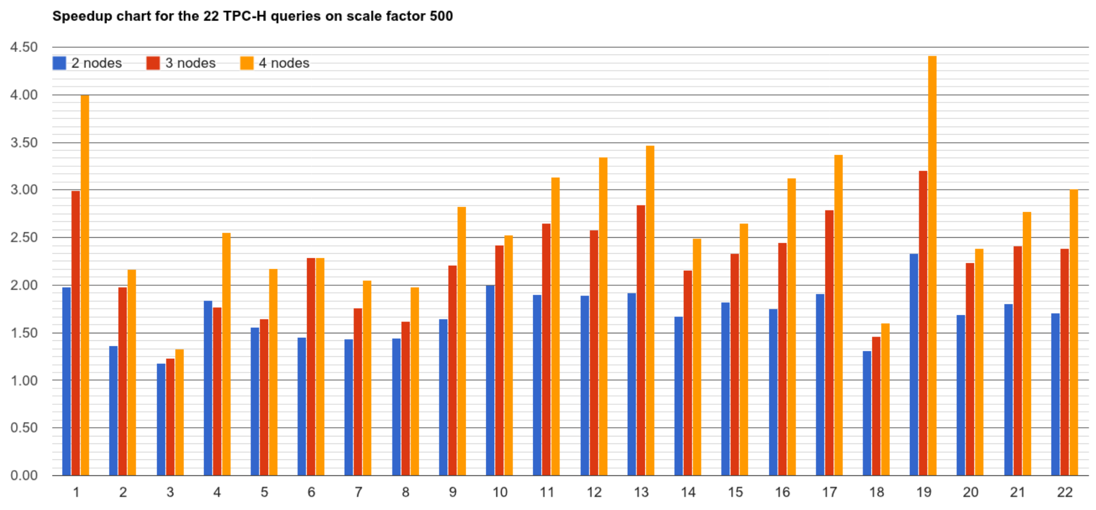
where:

+	improves parallelization
++	essential for a good speedup

and the original speedups (Table 6.2) are classified as:

- -	speedup value < 2
-	speedup $\in [2, 2.5)$
+-	speedup $\in [2.5, 3)$
+	speedup $\in [3, 3.5)$
++	speedup ≥ 3.5

Table 6.5: Per-Query Optimization Effectiveness Matrix



(a) SF 500

Figure 6.17: Power test results on modified TPC-H query set: per-query speedup

6.4.1 Power Test Results on the Modified TPC-H query set

In the previous section, some of query plans of the TPC-H queries were manually modified in order to simulate the effects of Late Projection and Dead Code Elimination (two identified optimizations out of eight). In comparison, the overall average speedups on SF-500 are: **1.71**(1.62), **2.25**(2.10) and **2.71**(2.55) on 2, 3 and 4 nodes, respectively.

Figure 6.17 presents the per-query speedups on this modified query set.

6.5 Throughput Test Results

Another important aspect of a distributed database system is its scalability with respect to the number of concurrent client connections it supports. Ideally, by doubling the number of nodes, the system should be able to service double the amount of concurrent clients within the same time frame, assuming that client workloads are similar.

To test this, we compared the overall time it takes to run 5 concurrent streams (simulating 5 concurrent clients) on 1 node, 10 streams on 2 nodes, 15 streams on 3 nodes and, finally, 20 streams on 4 nodes, using the **No-Distribution Policy** described in Section 5.2.2. In each of these streams, all the 22 queries are ran sequentially, but in a different order (as defined by the TPC-H benchmark). The overall times shown are computed as the absolute difference between the time clients start issuing queries and the time the system finishes processing the last of their queries.

Given that, as explained, it is only relevant for us to perform tests on "hot" I/O buffers and that, unlike in the power test, it was no longer possible to interleave "cold" runs, we could not run the throughput test on the SF500 database due to memory limitations.

The results shown in Table 6.6 for the SF100 database are promising: only 5% overhead is introduced and, apparently, this is independent of the number of nodes. It is arguable that this remark cannot hold true when much larger numbers of machines are involved, since, with our approach, the Master node is involved in all queries issued to the system for scheduling purposes, optimizing plans and passing results back to the Ingres front-end. However, the TPC-H queries generally limit the number of returned results (Q16 and Q20 are the only exceptions), so network bandwidth is unlikely to become a bottleneck. Moreover, the time the Master takes to schedule a query and build the corresponding DXchgUnion operator is negligible, while the percentage of time spent in the rewriter phase decreases as the the database grows in size. Repeating this test on a larger cluster would be the best way to validate/invalidate these speculations, but, to conclude, we conjecture that, from the perspective of query throughput performance, our solution obtains close-to-ideal speedup.

1 node, 5 streams	**2 nodes, 10 streams**	**3 nodes, 15 streams**	**4 nodes, 20 streams**
176.78 sec	184.46 sec	183.22 sec	185.93 sec

Table 6.6: Throughput Results, No-Distribution Policy

The caveat of the above experiment using the No-Distribution policy is that despite the overall throughput performance of the system is shown to scale well with the number of cluster nodes, the performance perceived by an individual user for a given query will never be better than that obtained on the single-node version of the DB engine, because, as its name suggests, the chosen policy completely eliminates distributed query execution (i.e. intra-query parallelism at the level of cluster nodes).

The **Distribution Policy** is meant to address the issue raised above by finding a compromise between achieving good overall query throughput performance and leveraging the benefits of distributed query execution. The following experiment, whose result are presented in Figure 6.18b, aims to reveal the extent to which the Distribution policy attains its goal by presenting the overall speedup achieved on increasing numbers of parallel query streams. We simulated a multi-user environment with two, three and four nodes on SF100, varying the number of streams from 1 to 8.

An apparent anomaly happens when the system is tested by executing 7 streams of queries. The reason behind the super-linear speedup becomes, however, clear if we take a closer look at the actual utilization of the single-node system and of the cluster:

- 1 node: after the few first queries, every plan produced by the Rewriter is granted 1 core (see Section 5.2.1, with an *OAF* of 1.25). Consequently, most of the time [1], 7 threads are running on 10 cores.
- 2 nodes: 3-thread query plans are produced (see Section 5.2.3), 21 threads on 20 cores.
- 3 nodes: 5-thread query plans, 35 threads on 30 cores.
- 4 nodes: 7-thread query plans, 49 threads on 40 cores.

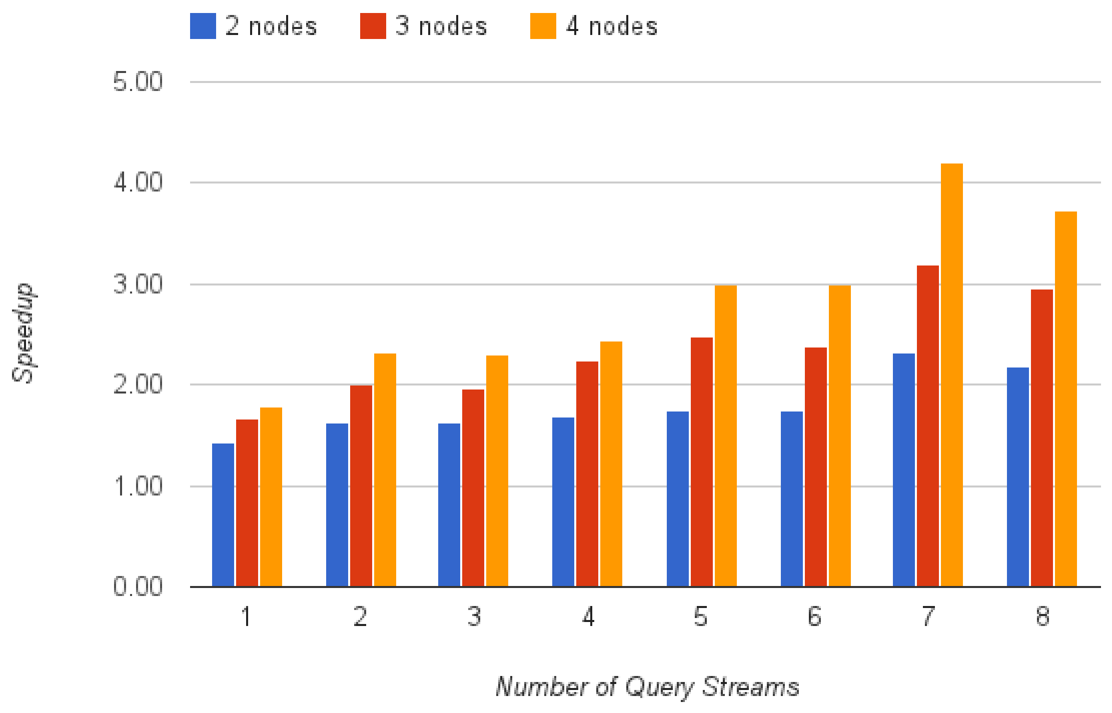
Furthermore, the alert reader may notice that the total times (and, therefore, speedups) obtained with a single query stream are slightly worse than those presented in Table 6.2 for SF100, even though there should have been no difference between these two sets of results. The explanation comes from the fact that "total time" was used with slightly different meanings: for the Power Test, it is the sum of the 22 individual query processing times, whereas for the Throughput Tests total time refers to the duration between the time when streams start issuing queries and the time the last query is processed. In the case of a single stream, the latter measure includes the former, but also accounts for inter-query processing overhead that amounts to ca. 3 seconds, regardless of the number of nodes.

Nevertheless, it can be seen that, as the number of streams increases, the speedups improve, getting closer and closer to those obtained with the No-Distribution policy. Lacking a more precise metric to evaluate it, we consider the Distribution policy to achieve positive results, successfully meeting its goals.

[1] Remember, the level of parallelism may change during the execution of a query

Number of	SF 100			
Streams	1 node	2 nodes	3 nodes	4 nodes
1	**45.01s**	**31.72s**	**27.12s**	**25.34s**
speedup		(1.42)	(1.66)	(1.78)
2	**83.83s**	**51.84s**	**41.76s**	**36.17s**
		(1.62)	(2.01)	(2.32)
3	**107.94s**	**66.57s**	**55.08s**	**46.97s**
		(1.62)	(1.96)	(2.30)
4	**140.76s**	**83.34s**	**62.85s**	**57.76s**
		(1.69)	(2.24)	(2.44)
5	**179.27s**	**103.20s**	**72.01s**	**59.94s**
		(1.74)	(2.49)	(2.99)
6	**200.68s**	**115.51s**	**84.03s**	**67.16s**
		(1.74)	(2.39)	(2.99)
7	**309.72s**	**133.13s**	**96.72s**	**73.79s**
		(2.33)	(3.20)	(4.20)
8	**323.34s**	**147.89s**	**109.50s**	**86.68s**
		(2.19)	(2.95)	(3.73)

(a) Total Execution Times



(b) Speedup Graph

Figure 6.18: Throughput Test Results, Distribution Policy

Chapter 7

Conclusion

7.1 Contributions

In this Master's thesis we undertook the challenge of laying the groundwork towards an MPP solution for the Vectorwise database engine. The milestones we aimed to reach were deciding on the high-level system architecture and creating a prototype that demonstrates improved query execution times, as well as the ability to process larger volumes of data and serve more clients in parallel.

With a cluster of four commodity nodes interconnected by low-latency, high throughput fabric, we obtained an average query processing speed-up of 2.55 on the SF500 TPC-H benchmark. Moreover, our solution was shown to scale linearly in terms of the number of concurrent users.

Coming back to our initial research questions, we argue that with an easy and non-intrusive implementation it is possible to obtain a reasonably scalable distributed version of Vectorwise. Moreover, with a shared-disk approach, good job scheduling policies and close-to-optimal parallel/distributed plans can be devised with little effort.

However, the most severe factor limiting the scalability of our solution is network bandwidth. Although we identified a series of optimizations that would alleviate this problem, it would inevitably reoccur when increasing the number of nodes and/or the volume of data. As such, we believe that explicit data partitioning is essential for a truly scalable MPP solution, despite the fact that it would require substantial additional development effort and compromise the non-intrusiveness and usability characteristics of our prototype.

7.2 Future Work

Apart from the performance-enhancing opportunities we identified and described in Section 6.4, one would also need to address the challenge of providing the following functionality and features in the context of a distributed DBMS.

Support for DDL, DML, distributed transactions

In this project we only focused on one of the many classes of queries that need to be supported by a full-blown DBMS, namely read-only queries (i.e. the *SELECT* command). Future work would also need to address the remaining operations of the Data Manipulation Language (DML) (i.e. *INSERT* / *UPDATE* / *DELETE*), as well as those of the Data Definition Language (DDL):

CREATE / ALTER / DROP. Additionally, support for transactions is crucial for a DBMS that aims to service many users concurrently, while ensuring the integrity of the data it stores.

Having already chosen a centralized solution (with a process acting as "Master") simplifies the implementation of the DDL operations, especially if the "shared-disk" approach is preserved. For example, the DDL command could be broadcast and executed by all the server instances on their in-memory data structures, with only the Master node acting on the data on disk.

Vectorwise uses the concept of "Positional Delta Trees" (PDTs) [HNZB08] to speed up updates on the underlying read-optimized column store. PDTs are hierarchical data structures that manage data modifications resulting from non-*SELECT* DML operations (or, simply, "updates") in such a way that they can be efficiently merged on-the-fly with table-resident data in order to provide readers with the most up-to-date information.

Without going into much detail, there are three levels of PDTs:

- *trans-PDT*: very small (L1 cache sized) PDT that only stores differential updates corresponding to a single transaction. Upon COMMIT, the Write-Ahead Logger (WAL) saves the trans-PDT to persistent storage (for crash-recovery purposes) and propagates its contents to the write-PDT.
- *write-PDT*: L2 cache sized PDT that is replicated for achieving snapshot isolation. As write-PDTs grow, their contents are propagated to the read-PDT.
- *read-PDT*: in-memory PDT (several hundred MB) that is shared by all queries in the system. A periodic *checkpointing routine* replaces old tables with new ones that include the differential updates stored in the read-PDT, such that the latter can be freed.

A simple solution for extending this mechanism to work in the distributed version of Vectorwise is to have the Master be the only process to execute update queries and run the WAL and checkpointing routines. Along with the distributed plans for SELECT queries, it can serialize and attach the corresponding trans-PDT, in order to inform the worker nodes about the latest transaction-specific differential updates. Also, upon COMMIT, after logging the trans-PDT to persistent storage, the Master should make all workers load the committed updates into memory just as they would reconstruct their PDTs after a crash. Finally, upon each Checkpoint operation, the worker nodes need to be informed that they need to switch to the newly-created tables and drop their read-PDTs.

We are aware that the above approach would incur non-negligible synchronization overhead, but devising more advanced solutions (e.g. distributing the execution of update queries) can be an interesting research task.

Availability, Reliability and Maintainability

Yet some other important characteristics of any production-ready DBMS - and especially for a distributed one - that were not addressed in this thesis are *availability*, *reliability* and *maintainability*.

Simply put, availability is the proportion of time a system is in a functioning condition, while reliability expresses the ability of a system or component to perform its required functions under stated conditions for a specified period of time [oEE90]. For ensuring high levels of reliability and availability, a system needs to be able to remain operational in the presence of failures and it must not require extended periods of planned downtime.

In a distributed environment, the occurrence of failures increases with the number of components (machines) involved. Despite being highly efficient and extensively used for HPC applications, the available MPI implementations have the disadvantage that they do not provide any user-transparent mechanism for handling the failure of a process. The message passing standard only defines two behaviors for these situations: either the communication primitives must *attempt to*[1] return an error code whenever a failed process is involved (either as the destination process, or as part of a collective operation), or - the default behavior - all such errors are treated as fatal and the application is terminated. Some implementations, however, go beyond implementing the standard and aim to provide process-level fault tolerance at the MPI API level (see FT-MPI [FD00]), but they mostly remain at the level of research projects. Therefore, in general, the responsibility of ensuring fault tolerance lies entirely on the implementer of MPI applications. Traditional approaches are described in [GL02], with the most notable one being the well-known "checkpointing" technique.

Maintainability refers to "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment" [oEE90]. The ability to add or remove server instances on the fly would not only improve the maintainability of the Vectorwise distributed DB engine, but also its availability, by minimizing the need for planned downtime. This does not comply with the static model of MPI-1, in which the number of tasks is fixed at application launch time, but, fortunately, the "dynamic process management" feature of MPI-2 allows an MPI process to spawn new processes and it could be used, for example, to adjust the number of cluster nodes running Vectorwise server instances depending on user load, the necessity for maintenance work, or other criteria. More information on the topic of dynamic process management support offered by MPI-2 compliant implementations can be found in [GKP09], along with a set of micro-benchmarks for evaluating their performance.

[1]Whether or not this attempt is successful is implementation-dependent.

Authors' Contributions

This is a joint thesis for the entirety of which the two authors have been working closely together. While the majority of the content is the result of their shared work, there are some sections of text that may be attributed to one author or the other:

Andrei Costea focused on Query Optimization and Sections 3 and 5 bear his signature.

Adrian Ionescu concentrated on Query Execution and Sections 2 and 4 can be attributed to him.

Acknowledgements

This thesis was made possible thanks to the Actian corporation that hosted our project and provided the necessary development environment, as part of a 7-month internship.

We would like to thank our supervisors, Marcin Żukowski and Peter Boncz, for their high involvement in the project, day-to-day guidance and productive discussion sessions. We also acknowledge the contribution of our second reader, Jacopo Urbani.

We worked in a very friendly environment and we owe this to the entire Vectorwise team in Amsterdam, especially to Bogdan and Julek, who always took the time to answer our questions - even the not-so-bright ones.

Finally, we would like to thank our families and friends for always having the patience to support and understand us, with all of our daily problems.

The A-Team

Thank you Alexandra for being by my side,

Andrei

Thank you, Sînzi, for supporting me through all this and constantly bringing joy to my life,

Adrian

Bibliography

[ABF$^+$92] Peter M. G. Apers, Care A. Van Den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 4:541–554, 1992.

[ADH02] A. Ailamaki, D.J. DeWitt, and M.D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, November 2002.

[Ani10] Kamil Anikiej. Multi-core parallelization of vectorized query execution, 2010.

[BAC$^+$90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, March 1990.

[BBS12] Stephen Baumanm, Peter Boncz, and Kai-Uwe Sattler. Bitwise dimensional co-clustering in column stores, 2012.

[BCL93] B. Bergsten, M. Couprie, and M. Lopez. Dbs3: a parallel database system for shared store. In *Proceedings of the second international conference on Parallel and distributed information systems*, PDIS '93, pages 260–263, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[BDW04] Rick Burns, Doug Drake, and Richard Winter. Scaling the Enteprise Data Warehouse, 2004. White Paper.

[BK99] Peter A. Boncz and Martin L. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, October 1999.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[CLRT00] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[DB210] IBM DB2. Partitioning and Clustering Guide, 2010.

[DGS$^+$90] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, March 1990.

[FD00] Graham E. Fagg and Jack J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world, 2000.

[FR98] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *In Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer-Verlag, 1998.

[GKP09] Tejus Gangadharappa, Matthew Koop, and Dhabaleswar K. Panda. Designing and evaluating mpi-2 dynamic process management support for infiniband. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 89–96, Washington, DC, USA, 2009. IEEE Computer Society.

[GL02] William Gropp and Ewing Lusk. Fault tolerance in mpi programs. *Special issue of the Journal High Performance Computing Applications (IJHPCA*, 18:363–372, 2002.

[Glu] GlusterFS. http://www.gluster.org/. Accessed: 10/08/2012.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2):102–111, May 1990.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[Gre10] Greenplum. Greenplum Database: Critical Mass Innovation, August 2010. Architecture White Paper.

[Heg06] D. A. Heger. An introduction to file system technologies in a cluster environment. Technical report, Austin, TX, 2006.

[HNZB08] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Positional delta trees to reconcile updates with read-optimized data storage. August 2008.

[Hog09] M. Hogan. Shared-Disk vs. Shared-Nothing Comparing Architectures for Clustered Databases, 2009.

[IMB] Intel MPI Benchmark User Guide and Methodology Description, Version 3.1. http://www.pccluster.org/score_doc/score-7-beta1/IMB/IMB_ug-3.1.pdf. Accessed: 10/08/2012.

[LM10] Barb Lundhild and Markus Michalewicz. Oracle Real Application Clusters (RAC) 11g Release 2, whitepaper, 2010.

[Lus11] Alicja Luszczak. Simple solutions for compressed execution in vectorized database system, 2011.

[MKAB04] M. W. Margo, P. A. Kovatch, P. Andrews, and B. Banister. An analysis of state-of-the-art parallel file systems for linux. In *The 5th International Conference on Linux Clusters: The HPC Revolution 2004*, Austin, TX, USA, May 2004. Hill.

[MRS02] Holger Märtens, Erhard Rahm, and Thomas Stöhr. Dynamic query scheduling in parallel data warehouses. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 321–331, London, UK, UK, 2002. Springer-Verlag.

[oEE90] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 1990.

[OV91] M. Tamer Ozsu and Patrick Valduriez. Distributed database systems: Where are we now. *IEEE Computer*, 24:68–78, 1991.

[OV96] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, March 1996.

[Par10] Paraccel. The Paraccel Analytic Database, December 2010. A Technical Overview.

[PS95] Eric Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, pages 127–145. Springer-Verlag, 1995.

[Rah93] Erhard Rahm. Parallel query processing in shared disk database systems, 1993.

[Rah95] Erhard Rahm. Dynamic load balancing in parallel database systems, 1995.

[Sch03] Philip Schwan. Lustre: Building a file system for 1,000-node clusters. In *PROCEEDINGS OF THE LINUX SYMPOSIUM*, page 9, 2003.

[Sev94] Kenneth C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *PERFORMANCE EVALUATION*, 19:107–140, 1994.

[SG95] J. Silcock and A. Goscinski. Message passing, remote procedure calls and distributed shared memory as communication paradigms for distributed systems. Technical report, Deakin University, 1995.

[SKPO88] Michael Stonebraker, Randy H. Katz, David A. Patterson, and John K. Ousterhout. The design of xprs. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 318–330, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[TH] TPC-H. The TPC-H Benchmark. http://www.tpc.org/tpch/. Accessed: 10/08/2012.

[THS10] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm grid file system. *New Generation Computing*, 28:257–275, 2010. 10.1007/s00354-009-0089-5.

[Top] Top500. Top500 supercomputer sites. http://www.top500.org. Accessed: 10/08/2012.

[Val93] Patrick Valduriez. Parallel database systems: open problems and new issues. *Distrib. Parallel Databases*, 1(2):137–165, April 1993.

[Ver10] Vertica. The Vertica Analytic Database Technical Overview White Paper, 2010.

[WA91] A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel main-memory environment. In *Distributed and Parallel Databases*, pages 68–77, 1991.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[WT91] Paul Watson and Paul Townsend. The EDS parallel relational database system. In Pierre America, editor, *Parallel Database Systems*, volume 503 of *Lecture Notes in Computer Science*, pages 149–166. Springer Berlin Heidelberg, 1991.

[ZaLC10] Jingren Zhou, Per ake Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the scope optimizer, 2010.

[Zuk09] M Zukowski. *Balancing vectorized executionwith bandwidth-optimized storage*. PhD thesis, Centrum Wiskunde en Informatica (CWI), 2009.